RICE UNIVERSITY

Portable High Performance and Scalability of Partitioned Global Address Space Languages

by

Cristian Coarfa

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dr. John Mellor-Crummey,
Associate Professor of
Computer Science

Dr. Ken Kennedy,
John and Ann Doerr University Professor
of Computational Engineering

Dr. Peter Joseph Varman,
Professor of
Electrical & Computer Engineering

HOUSTON, TEXAS

JANUARY, 2007

Portable High Performance and Scalability of Partitioned Global Address Space Languages

Cristian Coarfa

Abstract

Large scale parallel simulations are fundamental tools for engineers and scientists. Consequently, it is critical to develop both programming models and tools that enhance development time productivity, enable harnessing of massively-parallel systems, and to guide the diagnosis of poorly scaling programs. This thesis addresses this challenge in two ways. First, we show that Co-array Fortran (CAF), a shared-memory parallel programming model, can be used to write scientific codes that exhibit high performance on modern parallel systems. Second, we describe a novel technique for analyzing parallel program performance and identifying scalability bottlenecks, and apply it across multiple programming models.

Although the message passing parallel programming model provides both portability and high performance, it is cumbersome to program. CAF eases this burden by providing a partitioned global address space, but has before now only been implemented on shared-memory machines. To significantly broaden CAF's appeal, we show that CAF programs can deliver high-performance on commodity cluster platforms. We designed and implemented `cafc`, the first multiplatform CAF compiler, which transforms CAF programs into Fortran 90 plus communication primitives. Our studies show that CAF applications matched or exceeded the performance of the corresponding message passing programs. For good node performance, `cafc` employs an automatic transformation called procedure splitting; for high performance on clusters, we vectorize and aggregate communication at the source level. We extend CAF with hints enabling overlap of communication with com-

putation. Overall, our experiments show that CAF versions of NAS benchmarks match the performance of their MPI counterparts on multiple platforms.

The increasing scale of parallel systems makes it critical to pinpoint and fix scalability bottlenecks in parallel programs. To automatize this process, we present a novel analysis technique that uses parallel scaling expectations to compute scalability scores for calling contexts, and then guides an analyst to hot spots using an interactive viewer. Our technique is general and may thus be applied to several programming models; in particular, we used it to analyze CAF and MPI codes, among others. Applying our analysis to CAF programs highlighted the need for language-level collective operations which we both propose and evaluate.

Acknowledgments

I would like to express my deepest gratitude to my adviser, Dr. John Mellor-Crummey, for his invaluable guidance and assistance and for creating an extremely intellectually stimulating and challenging research environment. This thesis would not have been possible without him.

I want to thank Dr. Ken Kennedy for his insightful advice and suggestions. I would like to thank Dr. Peter Varman for his useful comments and discussion. Dr. Luay Nakhleh, Dr. Keith Cooper, and Dr. Moshe Vardi provided advice and encouragement when I needed them.

I want to thank my perennial co-author, Yuri Dotsenko, for a fruitful and rewarding collaboration.

Daniel Chavarria-Miranda was an excellent collaborator, mentor and friend. Timothy Harvey helped tremendously in preparing for my oral qualifying exam. William Scherer had useful suggestions for parts of my thesis.

The infrastructure I used for my experiments would not exist without the quality work of Nathan Tallent, Fengmei Zhao, Nathan Froyd, and Jason Eckhardt.

I was fortunate to work with great external collaborators. Robert Numrich and John Reid, who first designed CAF, provided invaluable advice. Jarek Nieplocha and Vinod Tipparaju helped tremendously to achieve performance using the ARMCI communication library. Kathy Yelick, Dan Bonachea, Christian Bell, Parry Husbands, Wei Chen, and Costin Iancu assisted in achieving high-performance using the GASNet communication library. Craig Rasmussen helped decipher the dope vector format for many native Fortran 95 compilers. Tarek El-Ghazawi, Francois Cantonnet, Ashrujit Mohanti, and Yiyi Yao provided benchmarks, insight, and quality work for a collaboration that yielded a joint article.

Leonikd Oliker and Jonathan Carter provided us with the LBMHD code.

During my stay at Rice I received help and assistance from a large number of the Compiler group members. I want to thank Robert Fowler, Chuck Koelbel, Zoran Budimlic, Mark Mazina, Arun Chauhan, Alex Grosul, Qing Yi, Guohua Jin, Todd Waterman, Anshuman Dasgupta, Anirban Mandal, Yuan Zhao, Apan Qasem, Cheryl McCosh, Rajarshi Bandyopadhyay, John Garvin, Mackale Joyner, and Rui Zhang.

I want to thank the Department of Energy's Office of Science for supporting this work through the Center for Programming Models for Scalable Parallel Computing. Early work on this project was supported by the Texas Advanced Technology Program.

To my wife Violeta, and my son Andrei, thank you for your love, patience, and support.

Contents

Abstract ii

Acknowledgments iv

List of Illustrations xi

1 Introduction 1

1.1 The Co-Array Fortran Programming Model 4

1.2 Thesis Statement 5

1.3 Joint Contributions 5

1.4 New Contributions 8

1.5 Thesis Overview 10

2 Related work 12

2.1 Library-based Parallel Programming Models 12

2.1.1 Message Passing Interface 12

2.1.2 One-sided Communication Libraries 16

2.2 Language-based Parallel Programming Models 18

2.2.1 Unified Parallel C 18

2.2.2 Titanium 21

2.2.3 High Performance Fortran 22

2.2.4 OpenMP 24

2.2.5 ZPL 27

2.2.6 SISAL 30

2.2.7 NESL 31

2.2.8 Single Assignment C (SAC) 33

2.2.9 The HPCS Languages 34
2.3 Implementations of Co-Array Fortran 34
2.4 Performance Analysis of Parallel Programs 35

3 Background 41
3.1 Refinements to the CAF Programming Model 41
3.2 Benchmarks 43
3.2.1 The NAS Parallel Benchmarks 43
3.2.2 LBMHD 45

4 A Source-to-source Compiler for Co-array Fortran 47
4.1 Memory Management 47
4.2 Local Co-Array Accesses 48
4.3 Remote Co-Array Accesses 49
4.4 Argument Passing 50
4.5 Synchronization 51
4.6 Communication Libraries 52
4.7 `cafc` Status 52

5 Optimizing the Performance of CAF Programs 54
5.1 Procedure Splitting 55
5.2 Representing Co-arrays for Efficient Local Computation 58
5.3 Evaluation of Representations for Local Accesses 61
5.4 Strided vs. Contiguous Transfers 65
5.5 Hints for Non-blocking Communication 66

6 An Experimental Evaluation of CAF Performance 69
6.1 Experimental Evaluation 69
6.2 NAS CG 70

6.3 NAS SP and BT . . . 73
6.4 NAS LU . . . 76
6.5 Impact of Optimizations . . . 80

7 Comparing the Performance of CAF and UPC Codes 82
7.1 Methodology . . . 82
7.2 Experimental Platforms . . . 83
7.3 Performance Metric . . . 84
7.4 NAS CG . . . 84
7.5 NAS BT . . . 92

I I 95

8 Analyzing the Effectiveness of CAF Optimizations 97
8.1 $2^k r$ Experimental Design Methodology . . . 98
8.2 Writing LBMHD in CAF . . . 98
8.3 Experimental Design . . . 99
8.4 Experimental Results . . . 104
8.5 Discussion . . . 110

9 Space-efficient Synchronization Extensions to CAF 114
9.1 Implementation of `sync_notify` and `sync_wait` . . . 114
9.2 Eventcounts . . . 116
9.3 Eventcounts Implementation Strategy . . . 118
9.4 Eventcounts in Action . . . 120
9.4.1 Jacobi Solver . . . 120
9.4.2 Conjugate Gradient . . . 121
9.4.3 An ADI Solver . . . 122
9.4.4 Generalized Wavefront Applications . . . 124

9.5 Summary . . . 127

10 Towards Communication Optimizations for CAF 128

10.1 A Memory Model for Co-Array Fortran . . . 128
10.2 Implications of the CAF Memory Model for Communication Optimization . 138
10.3 Dependence Analysis for Co-Array Fortran Codes . . . 140
10.3.1 Co-space Types and Co-spaces Operators . . . 140
10.3.2 Dependence Analysis Using Co-space Operators . . . 141
10.3.3 Discussion . . . 143
10.4 Dependence-based Vectorization of CAF Codes . . . 145
10.4.1 Dependence-based Vectorization Correctness . . . 153
10.4.2 Transformation Details . . . 153
10.4.3 Discussion . . . 156
10.5 Dependence-based Communication Optimizations of CAF . . . 158

11 Pinpointing Scalability Bottlenecks in Parallel Programs 164

11.1 Call Path Profiling and Analysis . . . 167
11.2 Automatic Scalability Analysis . . . 168
11.2.1 Call Path Profiles of Parallel Experiments . . . 169
11.2.2 Simple Strong Scaling . . . 170
11.2.3 Relative Strong Scaling . . . 171
11.2.4 Average Strong Scaling . . . 171
11.2.5 Weak Scaling for a Pair of Experiments . . . 172
11.2.6 Weak Scaling for an Ensemble of Experiments . . . 172
11.2.7 Analysis Using Excess Work . . . 173
11.2.8 Automating Scalability Analysis . . . 174
11.3 Experimental Methodology . . . 174
11.4 Experimental Results . . . 177
11.4.1 Analysis of LANL's POP Application . . . 177

11.4.2 Analysis of the NAS MG Benchmark . . . 182
11.4.3 Analysis of the NAS SP Benchmark . . . 187
11.4.4 Analysis of the NAS CG Benchmark . . . 190
11.4.5 Analysis of the LBMHD Benchmark . . . 191
11.4.6 Analysis of a MILC Application . . . 195
11.5 Discussion . . . 200

12 Conclusions 204

Bibliography 208

Appendices 226

A Scaling Analysis of Parallel Program Performance 226
A.1 Analysis of the NAS MG Benchmark . . . 226
A.2 Analysis of the NAS SP Benchmark . . . 227
A.3 Analysis of the NAS CG Benchmark . . . 229
A.4 Analysis of the NAS LU Benchmark . . . 238
A.5 Analysis of the NAS BT Benchmark . . . 241

B Extending CAF with collective operations 251
B.1 Reductions . . . 252
B.2 Broadcast . . . 255
B.3 Scatter/AllScatter . . . 255
B.4 Gather/AllGather . . . 256
B.5 All-to-all Communication . . . 257
B.6 Implementation Strategy . . . 257
B.7 Experimental Evaluation of Reductions . . . 258

Illustrations

1.1 Graphical representation of a co-array: every image has an instance of the array. 4

2.1 2D Jacobi relaxation example in MPI. 14
2.2 HPF multigrid method example [17]. 23
2.3 STREAM benchmark kernel fragment expressed in Fortran+OpenMP. . . . 26
2.4 Parallel 3-point stencil program expressed in ZPL. 28
2.5 A Jacobi solver fragment expressed in NESL [3]. 32
2.6 Fragment of a Jacobi solver written in SAC [169] 33

4.1 Examples of code generation for remote co-array accesses. 50
4.2 `cafc`-runtime data structure used to implement the `sync_notify`/`sync_wait` primitives. 52

5.1 Procedure splitting transformation. 57
5.2 Fortran 90 representations for co-array local data. 59
5.3 The STREAM benchmark kernels (F90 & CAF). 62

6.1 Comparison of MPI and CAF parallel efficiency for NAS CG on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters. . . . 71
6.2 A typical fragment of optimized CAF for NAS CG. 72

6.3 Comparison of MPI and CAF parallel efficiency for NAS BT on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters. . . . 73
6.4 Comparison of MPI and CAF parallel efficiency for NAS SP on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters. . . . 74
6.5 Forward sweep communication in NAS BT and NAS SP. 74
6.6 Comparison of MPI and CAF parallel efficiency for NAS LU on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters. . . . 76
6.7 Parallel efficiency for several CAF versions of NAS BT on an Alpha+Quadrics cluster. 77
6.8 Parallel efficiency for several CAF versions of NAS BT on an Itanium2+Myrinet cluster. 78
6.9 Parallel efficiency for several CAF versions of NAS BT on an Itanium2+Quadrics cluster. 78
6.10 Parallel efficiency for several CAF versions of NAS LU on an Alpha+Quadrics cluster. 79
6.11 Parallel efficiency for several CAF versions of NAS LU on an Itanium2+Myrinet cluster. 79
6.12 Parallel efficiency for several CAF versions of NAS LU on an Itanium2+Quadrics cluster. 80

7.1 Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class A on an Itanium2+Myrinet architecture. 85
7.2 Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class C on an Itanium2+Myrinet architecture. 86
7.3 Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class B on an Alpha+Quadrics architecture. 86
7.4 Comparison of MPI, CAF and UPC parallel efficiency for NAS CG on SGI Altix 3000 and SGI Origin 2000 shared memory architectures. 87

7.5 UPC and Fortran versions of a sparse matrix-vector product. 88
7.6 Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class A, on an Itanium2+Myrinet architecture. 89
7.7 Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class C, on an Itanium2+Myrinet architecture. 90
7.8 Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class B, on an Alpha+Quadrics architecture. 90
7.9 Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class B on an SGI Altix 3000 shared memory architecture. 91
7.10 Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class A on an SGI Origin 2000 shared memory architecture. 91

8.1 Visual tests for problem sizes 1024^2 and 2048^2 , 64 CPUs, on the SGI Altix 3000. 103
8.2 Visual tests for problem sizes 1024^2 and 2048^2, 64 CPUs, on the Itanium2+Quadrics architecture. 107
8.3 Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an SGI Altix 3000 system. 111
8.4 Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an Itanium2+Quadrics system. 113
8.5 Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an Itanium2+Myrinet system. 113

9.1 Current `cafc` data structure used for the implementation of the `sync_notify`/`sync_wait` primitives. 115
9.2 Graphical representation of an eventcount. Different process images can have different number of eventcount entries. 118

9.3 Steps taken in the execution of `advance_eventcount(evid, P, eidx, count)`. 119
9.4 Four-point stencil Jacobi solver pseudocode. 121
9.5 Four-point stencil Jacobi solver written using eventcounts. 121
9.6 A typical fragment of optimized CAF for NAS CG. 122
9.7 Fragment from the CAF SP `x_solve` routine, using `sync_notify`/`sync_wait`. 123
9.8 Fragment from the CAF SP `x_solve` routine, using eventcounts. 124
9.9 Graphical representation of progress in a generalized wavefront application. 125
9.10 Pseudocode variants for a generalized sweep application. 126

10.1 Relationship between `sync_notify`/`sync_wait` and remote accesses. . 130
10.2 Relationship between eventcounts and remote accesses. 131
10.3 Relationship between barriers and remote accesses. 132
10.4 Relationship between synchronization and remote accesses among multiple process images. 138
10.5 The driver procedure for the vectorization algorithm, *VectorizeComm*. . . . 146
10.6 The *VectorizeLoop* procedure. 147
10.7 The procedure *ClassifyCAFReference*. 147
10.8 The procedure *AllocateTemporariesAndRewriteReference*. 149
10.9 The procedure *GenerateRemoteAccessCode*. 151
10.10 Buffer management for remote writes subscripts and right-hand side data; padding is used so that the targets of subscript and data pointers each have a 64-bit alignment. 155
10.11 Code generation example for remote writes with subscripts using indirection arrays 160
10.12 Code generation example for remote writes with subscripts using multiple affine expressions of the loop index variables 161

10.13 Opportunities for dependence-based communication optimization of CAF codes . . . 162
10.14 Opportunities for optimization using combined dependence and synchronization analysis. . . . 163

11.1 Motivating example for parallel performance analysis using *calling contexts*: users are interested in the performance of communication routines called in the `solver` routine. . . . 165
11.2 Processes for computing and displaying the call path-based scalability information. . . . 175
11.3 Screenshot of strong scaling analysis results for POP, using relative excess work, on 4 and 64 CPUs. . . . 178
11.4 Screenshot of strong scaling analysis results for POP, for the baroclinic module, using relative excess work, on 4 and 64 CPUs. . . . 179
11.5 Screenshot of strong scaling analysis results for POP, using average excess work, for an ensemble of executions on 4, 8, 16, 24, 32, 36, 40, 48, and 64 CPUs. . . . 180
11.6 Screenshot of strong scaling analysis results for POP, for the baroclinic module, using average excess work, for an ensemble of executions on 4, 8, 16, 24, 32, 36, 40, 48, and 64 CPUs. . . . 181
11.7 Scalability of relative costs for communication primitives and computation for the CAF version of the NAS MG benchmark class A (size 256^3). . . . 183
11.8 Communication and synchronization volume for the CAF version of NAS MG, class A (size 256^3). . . . 183
11.9 Screenshot of strong scaling analysis results for CAF MG class A (size 256^3), using relative excess work on 1 and 64 processors. . . . 185

11.10 Screenshot of strong scaling analysis results for CAF MG class A (size 256^3), using relative excess work on 2 and 64 processors, for the routine `zran3`. 186
11.11 Scalability of relative costs for communication primitives and computation for the CAF version of the NAS SP benchmark class A (size 64^3). 187
11.12 Communication and synchronization volume for the CAF version of NAS SP, class A (size 64^3). 188
11.13 Basic pattern of conversion from two-sided message passing communication in MPI into one-sided communication in CAF. 189
11.14 Screenshot of strong scaling analysis results for the CAF version of NAS SP class A (size 64^3), using relative excess work on 4 and 64 CPUs. 190
11.15 Screenshot of strong scaling analysis results for the CAF version of NAS SP class A (size 64^3), using relative excess work on 4 and 64 CPUs, for the routine `copy_faces`. 191
11.16 Screenshot of strong scaling analysis for UPC NAS CG class A (size 14000), using relative excess work on 1 and 16 CPUs. 192
11.17 Scalability of relative costs for communication primitives and computation for the CAF version of the LBMHD kernel, size 1024^2. 193
11.18 Parallel efficiency for the timed phases of MPI and CAF variants of the LBMHD kernel on an Itanium2+Myrinet 2000 cluster. 194
11.19 Communication and synchronization volume for the CAF version of LBMHD (size 1024^2). 195
11.20 Screenshot of strong scaling analysis results for CAF LBMHD (size 1024^2), using relative excess work, on 4 and 64 CPUs. 196
11.21 Screenshot of strong scaling analysis results for CAF LBMHD (size 1024^2), using relative excess work, on 4 and 64 CPUs, for the routine `stream`. 197

11.22 Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors. 198
11.23 Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `ks_congrad_two_src`. 199
11.24 Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `grsource_imp`. . . . 200
11.25 Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `load_fatlinks`. . . 201
11.26 Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `ks_congrad`. 202

A.1 Scalability of relative costs for communication primitives and computation for the MPI version of the NAS MG benchmark class A (size 256^3). 227
A.2 Screenshot of strong scaling analysis results for MPI NAS MG class A (size 256^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 processors. 228
A.3 Screenshot of strong scaling analysis for MPI MG class A (size 256^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 processors, for the routine `mg3p`. 229
A.4 Scalability of relative costs for communication primitives and computation for the MPI version of the NAS SP benchmark class A (size 64^3). 230
A.5 Screenshot of strong scaling analysis results for MPI NAS SP class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. 231
A.6 Screenshot of strong scaling analysis results for MPI NAS SP class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs, for the routine `copy_faces`. 232

A.7 Scalability of relative costs for communication primitives and computation for the MPI version of the NAS CG benchmark class A (size 14000). 233
A.8 Scalability of relative costs for communication primitives and computation for the CAF version of the NAS CG benchmark class A (size 14000). . . . 233
A.9 Communication and synchronization volume for the CAF version of NAS CG, class A (size 14000). 234
A.10 Screenshot of strong scaling analysis results for MPI NAS CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs. 234
A.11 Screenshot of strong scaling analysis results for MPI NAS CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the routine `conj_grad`. 235
A.12 Screenshot of strong scaling analysis results for CAF CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs. . . . 236
A.13 Screenshot of strong scaling analysis results for CAF CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the routine `conj_grad_psbody`. 237
A.14 Scalability of relative costs for communication primitives and computation for the MPI version of the NAS LU benchmark class A (size 64^3). 240
A.15 Scalability of relative costs for communication primitives and computation for the CAF version of the NAS LU benchmark class A (size 64^3). 240
A.16 Communication and synchronization volume for the CAF version of NAS LU, class A (size 64^3). 241
A.17 Screenshot of strong scaling analysis results for MPI NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs. . . 242
A.18 Screenshot of strong scaling analysis results for the MPI version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the subroutine `ssor`. 243

A.19 Screenshot of strong scaling analysis results for the CAF version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs. 244
A.20 Screenshot of strong scaling analysis results for the CAF version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the function `ssor`. 245
A.21 Scalability of relative costs for communication primitives and computation for the MPI version of the NAS BT benchmark class A (size 64^3). 246
A.22 Scalability of relative costs for communication primitives and computation for the CAF version of the NAS BT benchmark class A (size 64^3), using the ARMCI communication library. 246
A.23 Communication and synchronization volume for the CAF version of NAS BT, class A (size 64^3). 247
A.24 Screenshot of strong scaling analysis results for MPI NAS BT class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. 247
A.25 Scalability of relative costs for communication primitives and computation for the CAF version of NAS BT class A (size 64^3), for the routine `x_solve`, using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. 248
A.26 Screenshot of strong scaling analysis results for the CAF version of NAS BT class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. 249
A.27 Screenshot of strong scaling analysis results for the CAF version of NAS BT class A (size 64^3), for the routine `y_solve`, using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. 250

B.1 Scalability of MPI and CAF variants of the LBMHD kernel on an Itanium2+Myrinet 2000 cluster. 259

Chapter 1

Introduction

Large scale parallel simulations are an essential tool for scientists and engineers. Providing scientific codes developers with parallel programming models that enable them to be productive and to effectively harness the power of current massively parallel systems has been a long standing challenge for the computer scientists in the high-performance scientific community. It is a hard reality that often parallel applications do not achieve the desired scalability, and programmers spend considerable effort tuning the applications to achieve high-performance. To direct and prioritize the optimization effort, it is important to have tools that enable programmers to quickly diagnose and find the parts of their codes that do not scale according to their expectations.

Recently, it has become clear that increasing processor clock frequency to build faster computers has reached fundamental physical barriers due to excessive power consumption and heat dissipation. Major computer vendors are therefore building multicore chips to increase the performance of computers for next generation designs of consumer market processors [18, 51, 121, 132]. As a result, parallel computing is moving into a high-profile, mainstream role, and the delivery of effective parallel programming models is a high priority task.

The desirable features for a parallel programming model are: i) *ease of use*, so users are productive; ii) *expressiveness*, so programmers can code a wide range of algorithms; iii) *high-performance*, so parallel codes utilize efficiently the capabilities of a parallel system of choice, and iv) *performance portability*, so programmers can write their code once and achieve good performance on the widest possible range of parallel architectures. Existing programming models, such as Message Passing Interface (MPI) [97], High-Performance

Fortran (HPF) [128], and OpenMP [133] have various drawbacks.

MPI is a library-based parallel programming model that relies on message passing communication. It is widely portable, and supported on practically every architecture of interest for parallel computing. Most large scale parallel codes are written using MPI, which has become the *de facto* standard for parallel computing. MPI 1.1 uses a *two-sided* (send and receive) communication model to communicate data between processes. With a two-sided communication model, both the sender and receiver explicitly participate in a communication event. As a consequence, both sender and receiver temporarily set aside their computation to communicate data. Note that having two processes complete a point-to-point communication explicitly synchronizes the sender and receiver. Years of experience with MPI have shown that while it enables achieving performance, it does so at a productivity cost. Writing MPI codes is difficult, error prone, and it demands that programmers select and employ the proper communication primitives to achieve high performance.

Language-based programming models offer an alternative to library-based programming models. In particular, compilers for parallel programming languages have an opportunity to deliver portable performance. HPF relies exclusively on capable compilers to generate efficient code, and a user has little control over the final performance of a HPF program. As of this writing, HPF has not delivered high performance for a wide range of codes. OpenMP enables a user to develop quickly a parallel application by specifying loop- and region-level parallelism; however, since users cannot specify affinity between data and processors, OpenMP programs have difficulties in scaling to large hardware shared memory systems. Also, OpenMP codes do not yield scalable performance on distributed memory systems.

Partitioned Global Address Space (PGAS) languages, such as Co-Array Fortran [155], Unified Parallel C [45], and Titanium [198], offers a pragmatic alternative to the HPF and OpenMP language models. They enable scientific programmers to write performance portable and scalable parallel codes using available compiler technology, whereas HPF and OpenMP require significant compiler technology improvements to enable developers to

achieve similar scalability and performance portability. The PGAS languages offer a partitioned global space view, with two-levels of memory: *local* and *remote*. Communication and synchronization are part of the language, and therefore are amenable to compiler optimization. Users retain control over performance-critical decisions such as data distribution and computation and communication placement.

In this thesis we present our experiences with Co-Array Fortran (CAF). CAF provides a SPMD programming model that consists of a set of parallel extensions to Fortran 95. The central concept of CAF is the *co-array*. At the language level, co-arrays are declared as regular Fortran 95 arrays, with a bracket notation at the end, as shown in Figure 1.1. The effect is that all process images contain an instance of the co-array; the co-array instance present on a process image is denoted the "local part" of the co-array for that process image, while the remaining instances are "remote co-array parts." Users can access both local and remote co-array memory by using subscripted references. One can express bulk communication at the source level by using Fortran 95 array section references. CAF uses *one-sided communication* (`PUT` or `GET`) to access remote data. When using one-sided communication, one process image specifies both the source and the destination of communicated data. From the programmer's perspective, the other process image is not aware of the communication. Thus, the one-sided model cleanly separates data movement from synchronization; this can be particularly useful for simplifying the coding of irregular applications.

Tuning of parallel applications is an important step on the road towards high-performance and scalability. To help users efficiently diagnose their scaling impediments, we describe and evaluate a novel scaling analysis technique that automatically quantifies how much calling contexts deviate from their expected scalability and that uses an interactive viewer to efficiently guide a user to scaling hot spots in the code. We demonstrate that our technique is effective when applied across multiple programming models, to a wide range of codes, and that it determines different causes of scalability problems.

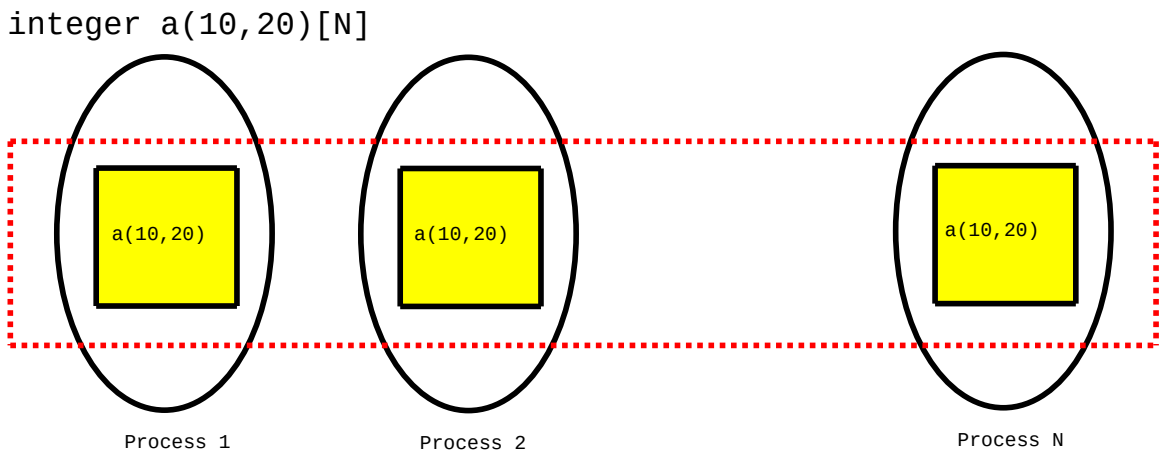


Figure 1.1: Graphical representation of a co-array: every image has an instance of the array.

1.1 The Co-Array Fortran Programming Model

Co-array Fortran supports SPMD parallel programming through a small set of language extensions to Fortran 95. An executing CAF program consists of a static collection of asynchronous process images. Similar to MPI, CAF programs explicitly distribute data and computation. However, CAF belongs to the family of Global Address Space programming languages and provides the abstraction of globally accessible memory for both distributed and shared memory architectures.

CAF supports distributed data using a natural extension to Fortran 95 syntax. For example, the declaration presented and graphically represented in Figure 1.1 creates a shared co-array `a` with 10×20 integers local to each process image. Dimensions inside square brackets are called co-dimensions. Co-arrays may be declared for user-defined types as well as primitive types. A local section of a co-array may be a singleton instance of a type rather than an array of type instances. Co-arrays can be static objects, such as COMMON or SAVE variables, or can be declared as ALLOCATABLE variables and allocated and deallocated dynamically during program execution, using collective calls. Co-arrays of

user-defined types may contain allocatable components, which can be allocated at runtime independently by each process image. Finally, co-array objects can be passed as procedure arguments.

Instead of explicitly coding message exchanges to access data belonging to other processes, a CAF program can directly reference non-local values using an extension to the Fortran 95 syntax for subscripted references. For instance, process `p` can read the first column of co-array `a` from process `p+1` referencing `a(:,1)[p+1]`.

CAF has several synchronization primitives. `sync_all` implements a synchronous barrier across all images; `sync_team` is used for barrier-style synchronization among dynamically-formed *teams* of two or more processes; and `sync_memory` implements a local memory fence and ensures the consistency of a process image's memory by completing all of the outstanding communication requests issued by this image.

Since both remote data access and synchronization are language primitives in CAF, communication and synchronization are amenable to compiler-based optimization. In contrast, communication in MPI programs is expressed in a more detailed form, which makes effective compiler transformations much more difficult.

A more complete description of the CAF language can be found in [154, 156].

1.2 Thesis Statement

Co-array Fortran codes can deliver high performance and scalability comparable to that of hand-tuned message-passing codes across a broad range of architectures. When CAF programs or other SPMD parallel codes do not achieve the desired performance and scalability, we can automatically diagnose impediments to their scalability.

1.3 Joint Contributions

Before this work, CAF was implemented only on Cray T3E and X1 systems. These machines support a global shared address space in hardware and provide efficient vector prim-

itives for remote memory accesses. For wide acceptance, CAF should ideally be implemented on a wide range of machines, including clusters that lack hardware support for a global address space. One could envision a user developing and testing a program on a multicore laptop, then deploying and running it on the largest parallel machine of choice. In joint work with Yuri Dotsenko at Rice University, we implemented `cafc`, the first multiplatform, open source CAF compiler, as a source-to-source translation system. We refined the CAF programming model to enable users to write performance portable codes, To demonstrate that CAF applications can achieve scalability and high-performance on a wide range of systems, we developed CAF codes, determined key optimizations necessary to achieve high-performance, and showed that the resulting codes matched the performance their of hand-tuned MPI counterparts.

`cafc` transforms CAF sources into Fortran 95 augmented with communication code, using a near-production-quality front-end Open64/SL [159]. We implemented the `cafc` runtime on top of one-sided communication libraries such as ARMCI [150] and GASNet [33]. `cafc` is capable of mapping CAF onto clusters that lack a shared memory fabric.

CAF is not yet a language standard. Our goal for `cafc` was to support sufficient CAF features so that users can write nontrivial and efficient parallel codes. In `cafc` we implemented declarations of COMMON, SAVE, ALLOCATABLE and parameter co-arrays, declarations of co-arrays of primitive and user-defined types with allocatable components, local and remote co-array accesses, and a subset of CAF intrinsics.

The original CAF programming model was implemented on Cray's systems with tightly-coupled hardware support for global address space. Architectural assumptions that came from these systems made their way into the programming model. To enable CAF programmers to write performance portable codes, we refined the CAF model by relaxing the requirement that each procedure call implies a fence — effectively ensuring that all communication issued before the procedure call completed — since it would limit the potential overlap of communication with computation. The CAF model initially contained only barrier synchronization, among all processes or among groups of processes. We ex-

tended the model with the point-to-point synchronization primitives `sync_notify` and `sync_wait`.

We demonstrated that CAF can match or exceed MPI performance for codes such as the NAS MG, CG, BT, SP and LU [24], the Sweep3D [6] neutron transport code, and the LBMHD kernel [157], on both cluster and shared memory architectures. This is an important scientific result, because the previous implementation of CAF enabled achieving high performance only on Cray global address space systems.

Since `cafc` performs source-to-source translation, to achieve *efficient node performance* it must generate code amenable to backend compiler analysis and optimization. For efficient communication, `cafc` relies on the underlying communication library (e.g. ARMCI or GASNet) to allocate data, separate from the memory managed by the Fortran 95 runtime system. The `cafc`-generated code uses Fortran 95 pointers to access local co-array data. This might lead backend compilers to make overly conservative assumptions regarding pointer aliasing and and inhibit important loop optimizations. To address this problem, we implemented an automatic transformation that we called procedure splitting [73]. If a CAF procedure performs local accesses to SAVE and COMMON co-arrays, then procedure splitting converts the procedure into an outer and inner pair of subroutines. The outer one passes the SAVE and COMMON co-arrays that are referenced as argument co-arrays to the inner subroutine, together with all the original arguments of the initial procedure. The inner subroutine performs the same computation as the original procedure, but with all the SAVE and COMMON co-array references converted into argument co-arrays. `cafc` transforms argument co-arrays into dummy array arguments. The overall effect for the `cafc`-generated code is transforming all the local co-array accesses Fortran 95 pointer references into array argument references. This conveys to a backend compiler the lack of aliasing between co-arrays, their memory contiguity and their dimensionality. We also evaluated multiple co-array representations [74].

Communication performance increases with communication granularity. For our CAF codes, we manually applied communication vectorization, communication packing and

aggregation at the source level [56, 73]. For asynchrony tolerance, we introduced and implemented extensions to the CAF language that enable use of non-blocking communication primitives.

To improve *synchronization performance*, we proposed and evaluated synchronization strength reduction, a source-level transformation replacing expensive barrier synchronization with lower-cost notify and wait primitives and showed its importance for both regular and irregular parallel codes. For producer-consumer communication patterns we discovered that insufficient buffer storage led to additional synchronization latency exposed on the execution critical path, which limited parallel performance. We showed that by using multiple communication buffers at source level we were able to match or exceed the performance of hand-tuned MPI versions for wavefront applications and line sweep computations [57, 73].

1.4 New Contributions

To improve the performance and scalability of parallel codes, it is crucial to correctly identify impediments to scalability. To enhance development productivity, it is desirable to pinpoint bottlenecks automatically and focus a programmer's attention on the parts of the code that are most responsible for loss of scalability. To address this need, we developed an automatic method of pinpointing and quantifying scalability impediments in parallel codes by determining where codes diverge from a user's expectations.

In general, users have well-defined performance expectations for their codes. For example, when attempting strong scaling of a parallel program, users expect that since the problem size and the work performed remain constant, the total execution time will decrease proportionally with the number of processors on which the parallel code is executed. When attempting weak scaling of a parallel program, users expect that since the problem size per processor remains constant and the number of processors increases, the overall execution time will remain constant. For sequential applications, users expect a certain time cost with respect to the input size; for example, a compiler writer might expect that an analysis

phase takes time linear with respect to the program size. In practice, it is often the case that programs do not perform according to the expectations of their developers; the challenge is then to identify which program components deviate the most from the expected behavior, in order to direct and prioritize the optimization efforts. We present and demonstrate the effectiveness of our analysis method for both strong scaling and weak scaling parallel programs.

Our analysis proceeds as follows. Once the expectations are formally defined, the program under analysis is executed on different number of processors. We use a profiling tool that collects calling context trees (CCTs) [19] for unmodified, optimized binaries,. In a CCT, each node corresponds to a procedure, such that the path from the root to each node reflects an actual call path during the program execution. The nodes of the CCT are annotated with the number of samples that were collected by the profiler in the procedure corresponding to that node. After running the parallel program and collecting the CCT for each execution, we analyze corresponding nodes in the CCT for different number of processors. Since our expectation is well-defined (e.g. linear scaling of running time or constant execution time), we can compute automatically how much each node deviates from our ideal scaling annotations. We denote this deviation *excess work*, and we normalize it by dividing by the total execution time for the parallel program; the resulting metric is denoted *relative excess work*. We compute this metric for both *inclusive* and *exclusive* costs; the exclusive costs represent the time spent within a particular procedure, while the inclusive costs correspond to the sum of the exclusive costs for that procedure and for all the routines called directly or indirectly by that procedure. Having metrics for both of these costs enables us to determine if the lack of scalability for a function's inclusive costs is due to inefficient work performed in that routine or to calls to routines with poor scaling. After computing this metric for all the nodes in the CCT, we use an interactive viewer to display the annotated CCT, sorting the nodes based on their value for the relative excess work. The viewer also displays the source code associated with the CCT nodes. Thus, the interactive viewer enables a user to quickly identify and navigate to the scaling trouble spots in the

code.

To validate the scaling analysis method, we used it to analyze the scalability of MPI, CAF, and UPC codes. The results highlighted the need for a non-blocking implementation of synchronization primitives for CAF, for language or library support of collective operations in both CAF and UPC, and for aggregation of collective calls in MPI codes. We demonstrated the power of our scaling analysis method by diagnosing scalability bottlenecks in multiple programming models and for diverse causes including non-scalable computation, inefficient use of well-implemented primitives, and inefficient implementation of other primitives.

Using lessons learned from the scalability analysis, we explored extending the CAF language with collective operations on groups of processors, including user-defined reduction operations on user-defined types. We designed an implementation strategy that leverages MPI collective operations and evaluated language-level collectives using several benchmarks.

Vectorization is an essential transformation for achieving communication granularity. We designed and proved the correctness of an algorithm for compiler-directed, dependence-based communication vectorization of CAF codes.

When scaling CAF to thousands of processors, it is important to have synchronization primitives that can be implemented efficiently, in terms of both time and space cost. The current CAF implementation of point-to-point synchronization primitives is not space efficient. To address this, we explored an extension of the CAF synchronization mechanism with eventcounts, which offer the same expressiveness and ease of use as the point-to-point primitives, but require less space.

1.5 Thesis Overview

This thesis is structured as follows. Chapter 2 describes the relationship to prior work. Chapter 3 presents the Co-Array Fortran language and our extensions, as well as the parallel benchmarks we used to evaluate the performance of CAF codes. Chapter 4 describes

the implementation strategy for `cafc`. Chapter 5 presents automatic and manual optimizations for improving the performance of local co-array accesses and of communication. Chapter 6 discusses CAF implementations of the NAS benchmarks [24] BT, CG, SP and LU and evaluates the impact of optimizations on scalar and parallel performance. Chapter 7 presents an evaluation of the impact of local performance and communication optimizations for UPC versions of the NAS benchmarks BT and CG. Chapter 8 uses a $2^k r$ full factorial design [123] to evaluate the impact of vectorization, aggregation, non blocking communication and synchronization strength reduction on the performance of the LBMHD benchmark. Chapter 9 explores space-efficient synchronization extensions to CAF. In chapter 10, we discuss the CAF memory model, sketch a strategy for performing dependence analysis on Co-Array Fortran codes, and describe a dependence-based algorithm for automatic communication vectorization of CAF codes. Chapter 11 describes our scaling analysis techniques and their validation through experiments with CAF, UPC, and MPI codes. Chapter 12 summarizes our contributions and findings and outlines future research directions.

Chapter 2

Related work

Technologies for parallel programming enabling users to achieve productivity, expressiveness, and scalability have been a longtime focus of research. It would be desirable for a user to write a parallel program once, then rely on the available tools to compile the program on any particular parallel architecture and achieve good scalability. In practice, parallel programming models range from library-based, such as Message Passing Interface (MPI), to language-based, such as High-Performance Fortran (HPF) and ZPL. Sections 2.1 and 2.2 discuss several programming models, focusing on their main features, ease of programming, expressiveness, availability, and documented performance. We also describe communication optimization techniques used for those programming models. Section 2.3 discusses other implementations of Co-Array Fortran.

Understanding the performance bottlenecks of parallel programs is a first important step on the way to achieving high-performance and scalability. It would be desirable to have tools that automatically analyze unmodified, optimized parallel codes, determine scaling impediments, and efficiently point a user to the scaling hot spots and associate them with the appropriate source code. Section 2.4 describes previous work in parallel programs performance analysis.

2.1 Library-based Parallel Programming Models

2.1.1 Message Passing Interface

Message Passing Interface (MPI) [97, 137, 138, 176] is a library-based parallel programming model based on the two-sided communication message-passing paradigm. MPI is a

single-program-multiple-data (SPMD) programming model, in which the users have a local view of computation. The MPI 1.2 [137, 138, 176] standard provides support for blocking and non-blocking point-to-point communication, barriers, collective routines such as reductions, broadcast, and scatter-gather, user-defined types and user-defined communicator groups. The MPI 2.0 [97, 137] standard adds support for one-sided communication, process creation and management, additional collective routines, and parallel IO. A precursor of MPI was PVM [181].

Even though the MPI 1.2 standard contains over 150 functions, studies of real applications have shown that the set of MPI primitives used in practice is smaller [189]. A study by Han and Jones [100] showed that the 12 applications they studied spend approximately 60% of their execution time in MPI calls; non-blocking point-to-point communication calls, such as `ISend`, `Irecv` and `Wait`, are much more commonly used than the blocking ones, such as `Send` and `Recv`. Among the collective operations, five of them are particularly common: barrier, allreduce, broadcast, gather and all-to-all.

Figure 2.1 presents an example of Jacobi 2D relaxation expressed in Fortran and MPI, omitting the declarations. Each processor packs the overlap regions for the east and west neighbors. Next, all processors posts non-blocking receives, by calling `MPI_Irecv`, for the north, south, west and east neighbors. The processors then perform blocking sends, by calling `MPI_Send`, to their neighbors, followed by potentially blocking checks that the non-blocking receives from their neighbors have completed — by using `MPI_Wait`. The received overlap regions are unpacked and the 5-point stencil is performed by every process. Finally, the maximum absolute difference between the previous temperature matrix and the new one is computed by using the collective call `MPI_All_Reduce`.

MPI has implementations on virtually every parallel system; they range from open-source ones [85, 94–96] to vendor versions [163, 174]. This ubiquitous availability has helped MPI become the *de facto* standard for parallel programming, and enable large groups of developers to write parallel programs and achieve relatively scalable performance. Carefully hand-tuned MPI codes, such as the NAS parallel benchmarks [22–24]

```
! update halo.
! pack
wSendBuf(1:MM) = ANS(1,1:MM)
eSendBuf(1:MM) = ANS(NN,1:MM)

! post receives
call MPI_IRECV(ANS(1,MM+1), NN,        &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(north), 99, MPI_COMM_WORLD,&
  recvNorth, ierr)
call MPI_IRECV(ANS(1,0), NN,           &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(south), 99, MPI_COMM_WORLD,&
  recvSouth, ierr)
call MPI_IRECV(eRecvBuf(1), MM,        &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(east), 99, MPI_COMM_WORLD, &
  recvEast, ierr)
call MPI_IRECV(wRecvBuf(1), MM,        &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(west), 99, MPI_COMM_WORLD, &
  recvWest, ierr)

! isend
call MPI_SEND(ANS(1,1), NN,            &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(south), 99, MPI_COMM_WORLD,&
  ierr)
call MPI_SEND(ANS(1,MM), NN,           &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(north), 99, MPI_COMM_WORLD,&
  ierr)
call MPI_SEND(wSendBuf(1), MM,         &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(west), 99, MPI_COMM_WORLD, &
  ierr)
call MPI_SEND(eSendBuf(1), MM,         &
  MPI_DOUBLE_PRECISION,                &
  NEIGHBORS(east), 99, MPI_COMM_WORLD, &
  ierr)
```

```
! check for completion
call MPI_WAIT(recvNorth, asynch_status, ierr)
call MPI_WAIT(recvSouth, asynch_status, ierr)
call MPI_WAIT(recvWest, asynch_status, ierr)
call MPI_WAIT(recvEast, asynch_status, ierr)

! unpack
ANS(NN+1,1:MM) = eRecvBuf(1:MM)
ANS(0,1:MM) = wRecvBuf(1:MM)

!   5-point stencil
do J= 1,MM
   do I= 1,N N
      WRK(I,J) = (1.0/6.0) * (RHS(I,J) +  &
           ANS(I-1,J  ) + &
           ANS(I+1,J  ) + &
           ANS(I,  J-1) + &
           ANS(I,  J+1)   )
   enddo
enddo

! calculate global maximum residual error.
PMAX = MAXVAL( ABS( WRK(1:NN,1:MM) -      &
  ANS(1:NN,1:MM) ) )
call MPI_ALLREDUCE(PMAX, RESID_MAX,       &
  1, MPI_DOUBLE_PRECISION,                &
  MPI_MAX, MPI_COMM_WORLD, ierr)
```

Figure 2.1: 2D Jacobi relaxation example in MPI.

became a yardstick against which any other parallel implementations, library-based or language-based, are compared and evaluated.

While MPI provides the means to write portable and efficient codes, it has a significant productivity drawback. The message passing programming model is difficult to use and error-prone. Programs based on library calls are traditionally difficult to optimize by compilers, and in practice, the responsibility for achieving high-performance code falls squarely on application developers. In their quest for higher performance, application de-

velopers often encode information about the target machine, such as the optimum message size for the interconnect, into the MPI code; this leads to hard-to-maintain code, since potentially one would need to have different versions of the communication code tuned for each architecture of interest. Another drawback is that the two-sided communication model might not be best suited for the capabilities of a particular architecture. In the case of hardware shared memory machines such as a SGI Altix 3000 [149, 175] and Cray X1 [58], MPI communication calls often introduce extra data copies between source and destination; on clusters having interconnects with RDMA capabilities, such as Myrinet [21, 145, 146], QSNet II [161, 164], MPI communication calls would perform extra data copies.

From the perspective of development time productivity, it would be desirable to use higher-level, language-based, parallel programming models, rather than the library-based message passing model. The arguments to move higher on the abstraction scale from MPI are that users manage less low-level details, becoming more productive; a compiler can help tailor a parallel program to perform well on a particular architectures, improving the performance portability of parallel codes and reducing their development and maintenance costs. These advantages have a strong appeal; however, the reason that MPI is still the most widely used parallel programming model is that higher level programming models have failed to deliver the high-performance and scalability for the range of algorithms of interest across the spectrum of available architectures, both shared-memory and cluster-based. Delivering both the performance and development time productivity is therefore a challenge for the parallel computing tools and technologies research community.

We show in this thesis that CAF codes can achieve performance comparable to that of corresponding MPI codes, for a range of applications including tightly-coupled codes based on dense matrices, such as NAS BT, NAS SP, NAS MG, and LBMHD, and for sparse irregular problems such as the NAS CG.

Communications optimizations such as vectorization, aggregation, and overlap of non-blocking communication with computation are widely used in MPI codes. Such optimizations are however expressed strictly at the source level in MPI, and we describe how a CAF

compiler could perform communication vectorization automatically.

MPI has a rich set of collective communication primitives, including support for broadcast, reductions, and scatter-gather operations. In this thesis we propose an implementation design of CAF collective operations extensions using the corresponding MPI primitives, and show that by using the language-level collective operations we are able to reduce the initialization time of NAS MG by up to 60% on 64 processors, and also improve the execution time of LBMHD by up to 25% on 64 processors.

2.1.2 One-sided Communication Libraries

Recent advances in high-performance interconnects made one-sided communication libraries attractive for parallel computing. On loosely-coupled architectures, an efficient one-sided communication library should take advantage of Remote Direct Memory Access (RDMA) capabilities of modern networks, such as Myrinet [21] and Quadrics [161]. During an RDMA data transfer, the Network Interface Chip (NIC) controls the data movement without interrupting the remote host Central Processing Unit (CPU). This enables the CPU to compute while communication is in progress. On many multiprocessor architectures, a cache coherence protocol is used to maintain consistency between CPU caches and memory that is the source or sink of communication. On shared memory platforms such as Altix 3000, one-sided communication is performed by the CPU using load/store instructions on globally addressable shared memory. The hardware uses directory-based cache coherence to provide fast data movement and to maintain consistency between CPU caches and (local or remote) shared memory. As the study [74] demonstrated, on shared-memory architectures fine-grain one-sided communication is fastest with compiler generated load/store instructions, while large contiguous transfers are faster when transmitted using a `memcpy` library function optimized for the target platform.

Two portable, one-sided, communication libraries are Aggregate Remote Memory Copy Interface (ARMCI) [150] and the GASNet [33] library.

ARMCI —a multi-platform library for high-performance one-sided communication—

as its implementation substrate for global address space communication. ARMCI provides both blocking and split-phase non-blocking primitives for one-sided data movement as well as primitives for efficient unidirectional synchronization. On some platforms, using split-phase primitives enables communication to be overlapped with computation. ARMCI provides an excellent implementation substrate for global address space languages making use of coarse-grain communication because it achieves high performance on a variety of networks (including Myrinet, Quadrics, and IBM's switch fabric for its SP systems) as well as shared memory platforms (Cray X1, SGI Altix3000, SGI Origin2000) while insulating its clients from platform-specific implementation issues such as shared memory, threads, and DMA engines. A notable feature of ARMCI is its support for non-contiguous data transfers [151].

GASNet is a language-independent low level networking layer that provides portable support for high-performance communication primitives needed for parallel global address space SPMD languages. GASNet is composed of two layers: the lower level is an interface termed the GASNet core API, based on active messages; the higher level is broader interface called the GASNet extended API, which provides one-sided remote memory operations and collective operations. GASNet is supported on high-performance network interconnects such as Infiniband, Quadrics, Myrinet, LAPI, on shared memory platforms such as the Cray X1 and SGI Altix 3000, and also has portable reference implementations on top of UDP and MPI. To communicate using Active Messages (AM) [190], each message sent between communicating processes contains two parts: one is a message handler identifier, and the other is the message payload. Upon receiving an Active Message, a dispatcher running on the receiving processor determines which Active Message handler should be invoked, invokes it and it passes it the AM payload.

Libraries such as ARMCI and GASNet could be used directly to develop parallel applications, but they are cumbersome to use by a programmer. Instead, they are usually used as communication layers by source-to-source compilers such as `cafc` and the Berkeley UPC compiler.

2.2 Language-based Parallel Programming Models

2.2.1 Unified Parallel C

Unified Parallel C (UPC) [45,78] is an explicitly parallel extension of ISO C that supports a global address space programming model for writing SPMD parallel programs. In the UPC model, SPMD threads share a part of their address space. The shared space is logically partitioned into fragments, each with a special association (affinity) to a given thread. UPC declarations give programmers control over the distribution of data across the threads; they enable a programmer to associate data with the thread primarily manipulating it. A thread and its associated data are typically mapped by the system into the same physical node. Being able to associate shared data with a thread makes it possible to exploit locality. In addition to shared data, UPC threads can have private data as well; private data is always co-located with its thread.

UPC's support for parallel programming consists of a few key constructs. UPC provides the `upc_forall` work-sharing construct. At run time, `upc_forall` is responsible for assigning independent loop iterations to threads so that iterations and the data they manipulate are assigned to the same thread. UPC adds several keywords to C that enable it to express a rich set of private and shared pointer concepts. UPC supports dynamic shared memory allocation. The language offers a range of synchronization and memory consistency control constructs. Among the most interesting synchronization concepts in UPC is the non-blocking barrier, which allows overlapping local computation and inter-thread synchronization. Parallel I/O [77] and collective operation library specifications [193] have been recently designed and will be soon integrated into the formal UPC language specifications. Also, [34] presented a set of UPC extensions that enables efficient strided data transfers and overlap of computation and communication.

UPC and CAF belong to the same family of partitioned global address space languages. Here, we mention some of the important differences between UPC and CAF. Based on Fortran 90, CAF contains multidimensional arrays; arrays and co-arrays can be passed as

procedure arguments, and can be declared with a different shape for the callee. Due to its C legacy, UPC cannot pass multidimensional arrays as arguments; for scientific codes which manipulate arrays, a UPC user has to resort to pointers and subscript linearization, often using macros. To access local co-array data, a CAF user relies on regular Fortran 90 array references, omitting the brackets; in UPC one performs array references using the MYTHREAD identifier or C pointers. To access remote elements, CAF uses array expressions with explicit bracket expressions, while UPC perform flat array accesses through shared pointers using linearized subscripts. For bulk and strided remote accesses, CAF uses Fortran 90 array sections, while UPC employs library functions. UPC provides two memory consistency models, strict and relaxed. Relaxed accesses performed by the same or different threads can be observed in any order; however, relaxed accesses executed by the same thread to the same memory location, with one access being a write, are observed by all threads in order. Strict accesses are observed by all threads in the same order, as if there was a global ordering of the strict accesses. If relaxed accesses occur before a strict access, the results of the relaxed accesses are observed by all threads before the results of the strict access; if a strict access is followed by relaxed accesses, then the results of the strict accesses are observed by all threads before the results of the relaxed accesses. For performance reasons, CAF provides a weak release consistency memory model. The UPC NAS benchmarks were written using the relaxed memory model, mainly for performance reasons. Having strict variables, however, is useful in enabling users to add synchronization primitives at the source level.

The Berkeley UPC (BUPC) compiler [54] performs source-to-source translation. It first converts UPC programs into platform-independent ANSI-C compliant code, tailors the generated code to the the target architecture (cluster or shared memory), and augments it with calls to the Berkeley UPC Runtime system, which in turn, invokes a lower level one-sided communication library called GASNet [33]. The GASNet library is optimized for a variety of target architectures and delivers high performance communication by applying communication optimizations such as message coalescing and aggregation as well

as optimizing accesses to local shared data. We used both the 2.0.1 and 2.1.0 versions of the Berkeley UPC compiler in our study.

The Intrepid UPC compiler [122] is based on the GCC compiler infrastructure and supports compilation to shared memory systems including the SGI Origin, Cray T3E and Linux SMPs. The GCC-UPC compiler used in our study is version 3.3.2.9, with the 64-bit extensions enabled. This version incorporates inlining optimizations and utilizes the GASNet communication library for distributed memory systems. Other UPC compilers are provided by HP [105] and by Cray [60].

Performance studies of UPC codes on multiple architectures [26, 27, 42–44, 54] identified as essential optimizations non-blocking communication and computation overlap, prefetching of remote data, message aggregation and privatization of local shared data, strip-mining of messages, and efficient address translation, performed either at source or runtime level.

Chen *et al* [53] present algorithms for enforcing sequential consistency for UPC programs by performing cycle detection. For Co-Array Fortran, we advocate a release consistency memory model for performance reasons.

Iancu *et al* [119] describe a method of automatically generating non-blocking communication at runtime level; their implementation is at the user level, above the level of the GASNet communication library. One interesting proposal is to complete remote communication on the first access to the remote data, by using the SIGSEV signal handler. Chen *et al* [52] discuss compiler optimizations for fine grain accesses by redundancy elimination, generation of non-blocking communication events for `GET`s and `PUT`s, and by coalescing communication events. To coalesce fine grain reads, the proposed technique is to prefetch locally the whole address range between the two reads, provided it is smaller than some machine-dependent threshold.

UPC was extended with collective operations for broadcast, reductions (including user-defined reductions), scatter-gather and general permutations [193]. While in our proposed collective primitives extensions to CAF the data arguments of collective operations can

be either private or shared data, in UPC arguments are required to reside in the shared space; this requires users that want to use collective operations on private data to either copy the arguments into shared space or to redeclare or allocate the private data as shared memory variables. A syntactic difference between UPC and CAF is that for UPC a user has to specify the appropriate UPC collective operation primitive, based on the type of the argument, while for CAF a compiler can infer the type and translate the collective operation accordingly using overloading.

2.2.2 Titanium

Titanium [198] is a parallel global address space language designed as a parallel extension of Java. Titanium provides a SPMD control model, flexible and efficient multi-dimensional arrays (potentially amenable to compiler optimizations), built-in types for representing multi-dimensional points, rectangles and general domains that are used to perform indexing of multidimensional arrays and to specify iteration spaces. Titanium supports unordered loop iteration spaces, which might be exploited by an optimizing compiler. Titanium enables memory management based on user controlled regions, besides regular Java garbage collection, and user-defined immutable classes. For synchronization, developers use textual barriers, which simplify compiler analysis of synchronization. Objects are shared by default, but users can control the sharing by using special qualifiers; Titanium possesses an augmented type system used to express or infer locality and sharing for distributed data structures. Titanium has a open-source multiplatform implementation, which is augmented with a library of useful parallel synchronization operations and collectives.

Su and Yelick [178, 179] describe an inspector executor method to optimize loops with irregular accesses. The method uses textual barriers to transform `GET`s into `PUT`s, and uses a hardware performance model to determine how the `GET`s or `PUT`s should be performed. They reuse a communication schedule if the loop performing the vectorization is enclosed into a separate loop and they can prove that the indirection array is not modified.

Titanium supports broadcast, exchange and reduction (including user-defined reduc-

tions) collective operations on teams of processors.

2.2.3 High Performance Fortran

High Performance Fortran (HPF) [80, 106, 107] is a high-level implicitly parallel programming language. HPF consists of extensions to Fortran 90; a user writes a sequential program in Fortran 90, then adds HPF directives to the Fortran code and then uses a HPF compiler to compile the code into an executable parallel program. From the Fortran 90 syntax perspective, HPF directives are simply comments, so in the absence of an HPF compiler a programmer can use a regular Fortran 90 compiler to compile an HPF program into a sequential executable program. In Figure 2.2, we present an HPF code fragment intended to model a multigrid method, from Allen and Kennedy [17]. The *TEMPLATE* directive declares a virtual processor array. The *ALIGN* directive specifies how an array is aligned with a certain template. The *DISTRIBUTE* directive specifies how a virtual processor array or a data array is distributed over the memories of a parallel machine. This specification is machine-independent. In the example presented in Figure 2.2, the template T is block-distributed on both dimensions.

One core property of HPF programs is that many performance critical decisions are made by the HPF compilers. To overcome potential limitations of HPF compilers, the HPF standard was extended with directives that enable the user to convey program properties to a compiler. The *INDEPENDENT* directive specifies that the following loop does not carry data dependencies and therefore can be safely parallelized; as shown in the example, this directive can also be used for nested loops. The *NEW* directive is used to specify variables that are replicated on each processor; in the example, the loop induction variable I is replicated. Due to the wide use of reductions, HPF enables the user to specify that a variable is collecting the result of a reduction, using the *REDUCTION* directive. HPF is supported by several commercial compilers [35, 37, 38, 99, 130, 135, 180].

From a productivity standpoint, HPF would be the ideal language for scientific code writers already proficient in Fortran. One of the often cited drawbacks of HPF was its

```
REAL A(1023, 1023), B(1023,1023), APRIME(511,511)
!HPF$ TEMPLATE T(1024, 1024)
!HPF$ ALIGN A(I,J) WITH T(I,J)
!HPF$ ALIGN B(I,J) WITH T(I,J)
!HPF$ APRIME(I,J) WITH T(2*I-1,2*J-1)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK)

!HPF$ INDEPENDENT, NEW(I)
DO J=2, 1022  ! Multigrid smoothing (Red-Black)
  !HPF$ INDEPENDENT
  DO I=MOD(J,2), 1022, 2
    A(I,J)=0.25*(A(I+1,J)+A(I+1,J)+A(I,J-1) &
      +A(I,J+1)+B(I,J))
  ENDDO
ENDDO

!HPF$ INDEPENDENT, NEW(I)
DO J=2, 510 ! Multigrid restriction
  !HPF$ INDEPENDENT
  DO I=2, 510
    APRIME(I,J) = 0.05*(A(2*I-2,2*J-2)+           &
                  4*A(2*I-2,2*J-1)+A(2*I-2,2*J)+  &
                  4*A(2*I-1,2*J-2)+ 4*A(2*I-1,2*J)+ &
                  A(2*I, 2*J-2)+4*A(2*I,2*J-1) +  &
                  A(2*I,2*J))
  ENDDO
ENDDO

! Multigrid convergence test
ERR = MAXVAL(ABS(A(:,:)-B(:,:)))
```

Figure 2.2: HPF multigrid method example [17].

inability to match MPI performance for a range of applications. Recently, Chavarria *et al* [49, 50, 62, 63] showed that HPF codes using multipartitioning, a block-cyclic distribution, were able to match MPI performance for challenging line-sweep applications such as the NAS benchmarks BT and SP.

A series of communication optimization techniques were developed for HPF: communication vectorization, communication coalescing, communication aggregation, support for accesses through indirection arrays, computation replication [9–13, 108–118, 126, 127].

HPF is an implicit parallel language, while CAF uses explicit parallelism. In HPF a programmer is dependent on the HPF compiler to achieve performance, while in CAF the user retains control over performance critical factors such as data distribution, communication and computation placement. CAF is more performance transparent than HPF: in CAF

a programmer knows that high costs are incurred by remote accesses (marked syntactically using brackets) and synchronization.

When performing communication optimizations for CAF, we have to take into account several factors. First, remote references are explicit in the code, using the bracket notation; second, we have to observe the memory consistency model, by paying attention to synchronization statements. When performing communication vectorization and communication aggregation for CAF, we first determine regions of code which are dominated and postdominated by synchronization statements. Finally, matching the location on the source and destination process images for PUTs or GETs is challenging; when we cannot do that, we need to rely on either expressing communication at language level through Fortran 90 array sections syntax, or to use active messages. Dotsenko [72] uses textual barriers to match communication endpoints between processes.

The vectorization algorithm we describe in chapter 10.4 hoists communication to the outermost possible level, and performs hoisting of communication using complex expressions as indices. We use a simplified version of the inspector-executor model, and do not optimize remote accesses when using indirection arrays as described in Das *et al* [65–67], and Hanxleden *et al* [101, 102, 184]. The CAF codes that we targeted used a single level of indirection so the algorithm presented in chapter 10.4 would suffice to optimize them.

The HPF library contains collective routines; also, since the program formulation is sequential, an implementation of the language would have to support all Fortran95 intrinsic functions that perform some operation on full arrays, such as sum, min, max, etc. In CAF a user has to code explicitly which collective operation he or she needs and specify the appropriate arguments.

2.2.4 OpenMP

OpenMP [133] is an implicit parallel programming model, based on directives, library support and environment variables, added to sequential languages such as Fortran or C/C++. OpenMP programs are a single thread — the master thread — at launch, but users can use

parallel regions to start new threads — slave threads; at the end of a parallel regions control returns to the master thread. Conceptually, OpenMP employs a fork-and-join parallelism model. By using a `OMP PARALLEL` directive, a programmer specifies a region of code that will be executed by all threads; the user can control the number of threads by using a library routine or an environment variable. Loop-level worksharing is achieved by using the `OMP PARALLEL DO` directive, which shares the iterations of a loop among the existing threads. To reduce the fork-and-join overhead, several parallel loops can be combined in a single parallel region. OpenMP provides several means of synchronization: barriers, critical sections, atomic updates at statement level, and code sections executed only by the master thread. User can specify both private and shared variables; global variables such as COMMON or SAVE in Fortran of static variables in C are by default shared, while stack variables in procedures called from parallel regions are private. OpenMP enables programmers to indicated that certain lines in a loop correspond to arithmetic reductions.

In Figure 2.3, we present a fragment from the STREAM benchmark [134] expressed using OpenMP. The example uses several loop-level parallelism constructs, uniformly distributing the loop iterations among the executing threads.

OpenMP relies on users to to specify directives correctly. For example, one can use `OMP PARALLEL DO` only if there are no loop-carried dependencies. OpenMP programmers can use the incremental parallelism approach, when only a couple of loops at a time are parallelized.

OpenMP is supported by a large number of commercial compilers for both Fortran 90 and C/C++ implementations. The biggest drawback to OpenMP is its lack of performance on distributed shared memory platforms. Programmers don't have syntactic means to indicate the affinity between data and particular threads, which leads to unnecessary communication at runtime. This affects even OpenMP performance on hardware shared-memory machines: a study by Dotsenko, Coarfa *et al* [74] showed that for NAS benchmarks such as SP class C and MG class C, the OpenMP versions are competitive only up to 9-16 processors, after which their efficiency degrades significantly with respect to the MPI and CAF

```
!$OMP PARALLEL DO
      DO 10 j = 1,n
          a(j) = 2.0d0
          b(j) = 0.5D0
          c(j) = 0.0D0
   10 CONTINUE
      t = mysecond()
!$OMP PARALLEL DO
      DO 20 j = 1,n
          a(j) = 0.5d0*a(j)
   20 CONTINUE
      t = mysecond() - t

*       --- MAIN LOOP ---
      scalar = 0.5d0*a(1)
      DO 70 k = 1,ntimes

          t = mysecond()
          a(1) = a(1) + t
!$OMP PARALLEL DO
          DO 30 j = 1,n
              c(j) = a(j)
   30     CONTINUE
          t = mysecond() - t
          c(n) = c(n) + t
          times(1,k) = t

          t = mysecond()
          c(1) = c(1) + t
```

```
!$OMP PARALLEL DO
          DO 40 j = 1,n
              b(j) = scalar*c(j)
   40     CONTINUE
          t = mysecond() - t
          b(n) = b(n) + t
          times(2,k) = t!$OMP PARALLEL DO
          t = mysecond()
          a(1) = a(1) + t
          DO 50 j = 1,n
              c(j) = a(j) + b(j)
   50     CONTINUE
          t = mysecond() - t
          c(n) = c(n) + t
          times(3,k) = t

          t = mysecond()
          b(1) = b(1) + t
!$OMP PARALLEL DO
          DO 60 j = 1,n
              a(j) = b(j) + scalar*c(j)
   60     CONTINUE
          t = mysecond() - t
          a(n) = a(n) + t
          times(4,k) = t
   70 CONTINUE
```

Figure 2.3: STREAM benchmark kernel fragment expressed in Fortran+OpenMP.

versions of those benchmarks.

A recent trend is to use a hybrid OpenMP/MPI programming model on clusters of SMPs, where one uses MPI to communicate among cluster nodes, but relies on OpenMP to achieve parallelism within one node.

OpenMP enables users to specify reductions operations withing a parallel region by indicating the reduction type and the argument; a compiler would then be responsible for implementing the reduction. OpenMP does not have support for broadcast; to achieve the same effect, a user would have to code an assignment to a shared variable and relay on the compiler to recognize this communication pattern and implement it efficiently. Also OpenMP does not allow users to specify their own reduction operators.

2.2.5 ZPL

ZPL [47, 70, 177] is a high-level, implicit parallel programming language, in which programmers have a global view of computation. We will give an overview of the language by using a simple three-point stencil program presented in Figure 2.4. ZPL contains both `parallel` and private arrays. Parallel arrays are declared using *regions*. The parallel array `A` has the indices sets specified by the region `BigR`, `[0..n+1, 0..n+1]`. The last row of `A` is initialized to 1; the rest of the array is initialized to 0. Accesses to parallel arrays are performed exclusively by using special operators. To perform the stencil computation, the *at operator*(`@`) is used; this operator shifts the values of the array `A` by an offset vector called a *direction* and specified using the keyword `direction`. In our example, the stencil computation involves the east, north-west and north-east neighboring cells. Shift references could potentially induce communication. The result of the stencil is assigned to the parallel array `Temp`. Next, the program computes the difference between the values of `A` and `Temp` by using the sum reduction operator `+<<` applied to the parallel array `A-Temp`; ZPL supports reductions for others operators such as multiplication, maximum and minimum. Notice that the index set for the `repeat` loop is specified by using the region `R`. Another operator for parallel arrays is the *remap* operator, which enables a programmer to specify data movement between parallel arrays using patterns more complicated than the shift operator.

A core feature of ZPL is its transparent performance model, known as *what-you-see-is-what-get* (WYSYWIG). A programmer is always aware of the places in the source code that can trigger communication events. For example, a shift operator will probably induce communication with certain neighbors of a processor. The reduce operator leads to a local reduction per processor and then to log-cost communication between processors. The remap operators causes potentially all-to-all communication, which is expensive.

An open-source ZPL compiler was developed at the Washington University. The compiler perform source-to-source translation from ZPL to C with calls to a runtime library; it can use the MPI, PVM [181] or SHMEM [61, 174] libraries as communication medium.

```
program three_pt_stencil
config var
  n :integer = 256;
region
  R = [1..n, 1..n];
  BigR = [0..n+1, 0..n+1];
direction
  east=[ 0, -1];
  nw = [-1,  1];
  ne = [-1,  1];
var
  A, Temp: [BigR] double;
constant
  epsilon: double = 0.00001

procedure three_pt_stencil
var
  nr_iters : integer
  err      : double
begin
  [BigR]       A := 0;
  [south of R] A := 1;
  nr_iters := 0;
  [R]  repeat
     nr_iters += 1;
     Temp := (A@east + A@nw + A@ne)/3.0;
     err := +<<abs(A-Temp);
  until err <= epsilon;
  writeln(''Iteration performed: %d\n'':nr_iters);
end;
```

Figure 2.4: Parallel 3-point stencil program expressed in ZPL.

The ZPL compiler is responsible for mapping a parallel array to the set of available processors; private scalars and arrays are replicated and kept consistent. Recent extensions to ZPL [71] enable the declaration of user-defined data distributions, improving the expressiveness of ZPL.

A study performed by Chamberlain *et al* [48] compares a ZPL version of MG, for classes B (size 256^3) and C (size 512^3), with corresponding version written in MPI, HPF and CAF. ZPL is able to match the performance of MPI on architectures such as a Linux cluster with Myrinet interconnect, up to 128 processors, and a Sun Enterprise, up to 8 processors; ZPL is slightly outperformed by the MPI one on a Linux cluster with Ethernet, on an IBM SP machine, and on an SGI Origin. On a Cray T3E, however, the ZPL version significantly outperforms MPI, up to 256 processors, due mainly to the ZPL compiler's

ability to harness the SHMEM [61, 174] library, leading to more efficient communication than that of MPI. The authors speculate that generating code for SHMEM on SGI and the IBM SP would enable the ZPL code to match the MPI performance. A study by Dietz *et al* [68] showed that a ZPL version of NAS CG using MPI as communication substrate was able to match the MPI performance for class C (size 150000) on an IBM SP2 for 128 processors and on a LinuxBios/BProc Cluster for up to 1024 processors. For FT, the MPI version outperforms the ZPL version on the IBM SP2 and LinuxBios/BProc Cluster due mainly to lower performance of the transposition phase of FT. The PhD thesis of Dietz [70] shows that a ZPL versions of IS class C (with 2^{27} keys and 2^{10} buckets) achieves performance comparable to that of the MPI version on a Cray T3E, up to 256 processors. We couldn't find documented performance results for ZPL versions of the SP, BT and LU NAS benchmarks.

Chamberlain *et al* [36] present communication optimizations performed by a ZPL compiler: message vectorization, message pipelining and redundant communicaiton removal. Dietz *et al* [68] determine optimizations necessary for the implementation of a remapping operator: using an inspector-executor schedule and saving it for multiple uses, computation/communication overlap, efficient schedule representation, dead source/destination reuse and RDMA `PUT`/`GET`. CAF can express array remapping at language level as a succession of remote assignments. In case of vectorizing array accesses with irregular accesses, we would compute the list of accessed array locations and pass it using an active message to a remote node. Finally, at the level of the CAF compiler, we determine when we perform co-array to co-array accesses and use direct sends/receives, effectively achieving zero-copy communication.

ZPL supports both full reductions and parallel prefix reductions, broadcast operations, applied to whole arrays or parts of an array. These operations are then translated by a ZPL compiler. ZPL also supports user-defined reductions. The remapping operator can be used to implement a collective exchange operation.

CAF is more performance transparent than ZPL: a CAF programmer has more control

over the final performance of his or her code compared to a ZPL programmer, who needs to rely on the ZPL compiler to generate efficient communication and computation.

2.2.6 SISAL

SISAL (Streams and Iterations in a Single Assignment Language) [79] is a general purpose functional language. The order of execution of the program is determined by availability of values for the operands of expressions rather than by static ordering in the program source, making SISAL a dataflow language. A compiler has the freedom to schedule expression evaluation in any order that satisfies data dependencies, even to schedule them in parallel. SISAL supports calling C and Fortran routines to perform efficient local computaton. A user can express parallelism in SISAL by using for loops, annotating loops for which all iterations are independent. To get parallel performance, a user has to rely on the quality of the SISAL compiler and runtime to achieve load balancing and manage the communication overhead.

osc [41] is an optimizing SISAL compiler that generates code for vector, sequential and shared memory machines. osc transforms SISAL into Fortran or C code, augmented with calls into the osc runtime system. osc performs optimizations such as update-in-place intended to reduce the amount of copying, and splits the code into parts that can be executed independently in parallel. The execution model relies on a shared queue that contains slices of work and on a server thread that distributes the slices onto the available processors. SISAL achieved comparable performace to hand tunded codes on shared memory machines for a series of benchmarks: for the Abingdon Cross image processing benchmark [5] and for several Livermore Loops [40].

Several efforts have been made to port SISAL to distributed memory machines. *fsc* [81] is a prototype SISAL compiler for distributed and shared memory systems. fsc is derived from the osc compiler, but modifies the code generation phase to use the Filaments library as a runtime system. Filaments is a library supporting fine-grained threads and shared memory on distributed memory systems. Using fine-grain threads enables the implemen-

tation of both recursive and loop-level parallelism, and it permits runtime load balancing. An fsc-compiled SISAL version of matrix multiply achieved a speedup of 2.88 on 4 processors, a Jacobi Iteration solver achieved 2.03 speedup on 4 processors, and a version of adaptive quadrature achieved a speedup of 3.59 on 4 CPUs.

D-OSC [86] extends osc to generate C code with calls to a message passing library. D-OSC parallelizes `for` loops; a master process determines slices of computation and distributes them to be executed in parallel by slave processes. If a slice contains other parallel loops, the slave executing it takes the role of the master process and further distributes its slices to other processors. D-OSC implements optimizations such as replacing multidimensional arrays with rectangular arrays, coalescing messages directed to the same processor, and using computation replication to reduce the need for communication. These optimizations reduce the number of messages and communication volume for benchmarks such as selected Livermore and Purdue loops, matrix multiply, Laplace, but no timing measurements were provided.

Pande *et al* [160] extended the osc compiler to work on distributed memory machines; they proposed a threshold scheduling algorithm for the SISAL tasks that trades off between parallel speedup and necessary number of processors. At runtime, message passing is used to communicate the necessary values between processors. The experiments showed speedups of up to 10 on 33 processors for various Livermore loops.

While SISAL codes showed good scalability on tightly coupled shared memory systems, achieving similar results on large scale distributed memory systems remains an open problem. Using CAF, users can get high-performance and scalability on both shared and distributed memory, by retaining explicit control of data decomposition and communication and computation placement.

2.2.7 NESL

NESL [2, 28–32] is a data-parallel programming language using functional semantics developed at Carnegie Mellon. NESL offered two new key concepts: nested data parallelism,

```
function sparse_mvmult(A,x) =
let ids,vals = unzip(flatten(A));
    newvals = {vals*g:vals;g in x->ids}
in {sum(row): row in partition(newvals,{#A:A})} $

% A sparse matrix and a vector %
function jacobi_loop(x,A,b,i) =
if (i == 0) then x
else let
    y = sparse_mvmult(A,x);
    x = {x + b - y:  x in x; b in b; y in y };
in jacobi_loop(x,A,b,i-1) $

function jacobi(A,b,n) =
  jacobi_loop(dist(0.,#a),a,b,n);

A = [[(0, 1.), (1, .2)         ],
     [(0, .2), (1, 1.), (2, .4)],
     [         (1, .4), (2, 1.)]];
b = [1.,1.,1.];

% Run jacobi for steps iterations %
x = jacobi(A,b,steps);

% Check how close the answer is -- it should equal [1,1,1] %
sparse_mvmult(A,x);
```

Figure 2.5: A Jacobi solver fragment expressed in NESL [3].

which makes it suitable for expressing irregular algorithms, and a language-based performance model, enabling a programmer to calculate the work and the depth of a program, metrics related to the program execution time. Functional semantics enables functions to be executed in parallel when there is no aliasing between sibling function calls. NESL enables these functions to spawn other parallel function calls. NESL also supports data parallelism using its sequence concept: a one dimensional distributed array consisting of data items or other sequences. NESL has a parallel apply-to-each construct that operates in parallel on the elements of a sequence. In Figure 2.5 we present a fragment from a Jacobi solver expressed in NESL, that executes `steps` iterations.

Although the performance model gives users an estimate of the running time of a NESL program, issues such as data locality and interprocessor communication are completely under a NESL compiler's control. In CAF, a programmer retains control over such performance critical decisions.

```
inline double[] onestep(double[] B) {
  A = with ( . < x < . )
  modarray(B, x, 0.25*(B[x+[1,0]]
       + B[x-[1,0]]
       + B[x+[0,1]]
       + B[x-[0,1]]) );
  return(A);
}

inline double[] relax(double[] A, int steps) {
  for (k=0; k<steps; k++) {
    A = onestep(A);
  }
  return(A);
}

int main () {
  A = with( . <= x <= .)
  genarray([SIZE1, SIZE2], 0.0d);

  A = modarray(A, [0,1], 500.0d);

  A = relax( A, LOOP);

  z = with( 0*shape(A) <= x < shape(A))
         fold(+, A[x]);

  printf("%.10g\n", z);

  return(0);
}
```

Figure 2.6: Fragment of a Jacobi solver written in SAC [169]

.

2.2.8 Single Assignment C (SAC)

Single Assignment C (SAC) [170,171] is a functional parallel programming language based on ANSI C. It supports multidimensional C arrays, array properties query operators, and it contains the operator `with-loop`, which can be used for array creation, operations that modify array elements, or to fold array elements into one value using binary operators. In Figure 2.6 we present a Jacobi relaxation solver written in SAC that uses a five point stencil.

Performance-critical decisions for SAC programs, such as interprocessor communication, are left at the compiler's discretion, as opposed to CAF programs, where communication is syntactically marked. SAC is implemented as of this writing on shared-memory systems only, while our CAF compiler works on a wide range of systems. Performance

studies [48, 92, 93] showed that while SAC displayed good scaling, they suffered from scalar performance problems compared to their Fortran 77 counterparts for NAS FT, for which it was slower by a factor of 2.8x, and is within 20% from the serial performance of NAS MG for class A (size 256^3).

2.2.9 The HPCS Languages

As part of the DARPA High Productivity Computing Systems (HPCS) [1] effort to realize efficient parallel architectures and productive programming models, several vendors proposed new language-based parallel programming model. Cray introduced the Chapel language [59], IBM proposed the X10 language [120], and Sun designed the Fortress language [16]. While these languages have generated significant commercial and academic interest, as of the writing of this document they only have prototype implementations, and published performance results on masivelly parallel systems are not available yet.

2.3 Implementations of Co-Array Fortran

Before our work, the only available implementation of the Co-Array Fortran language was the one provided by Cray [173], only on Cray X1 and Cray T3E machines. It used the native Fortran 90 vectorizing compiler to perform transformations such as communication vectorization and strip-mining, streaming remote data into local computation and making efficient use of the vector processing capabilities of the machines. Our compiler is multiplatform, which should help broaden the acceptance of the CAF model. A study by Chamberlain *et al* [48] showcased the capability of CAF of delivering parallel performance superior to that of MPI on hardware shared memory Cray platforms. We show in this thesis that CAF can match or exceed MPI performance on a range of architectures, both cluster and shared memory. To achieve performance portability for CAF, essential optimizations are procedure splitting, communication vectorization, communication packing and communication aggregation, and synchronization strength reduction. We have not ported `cafc` to Cray platforms yet.

Wallcraft has developed a translator [192] from Co-Array Fortran to OpenMP, which works only for a subset of CAF and targets shared-memory architectures. Wallcraft performed a study of the CAF potential compared to MPI for the HALO benchmark [191], showing that CAF can deliver good latency on hardware shared-memory architectures.

Eleftheriou *et al* implemented a co-array style C++ library for the Blue Gene/L supercomputer, rather than as a language, for the purpose of rapid prototyping and deployment. Two threads are defined for each process image, one performing the local computation, the other one servicing communication requests. We believe that a library-based implementation, while rapid to develop and useful for performance potential evaluation, lacks the automatic optimizations that a compiler-based language implementation can offer. `cafc` is not implemented on Blue Gene/L at the moment; as of this writing, the ARMCI and GASNet communication libraries are emerging on this platform.

Dotsenko [72] proposed, implemented and evaluated several language extensions for CAF. Co-functions, which enable computation shipping, simplify the writing of parallel search operations and enabled a CAF version of the RandomAccess benchmark to outperform the MPI implementation. Co-spaces, textual barriers and single-value variables enabled an automatic implementation of synchronization strength reduction, which converts barriers into notify-wait synchronization. Finally, multiversion variables extend the CAF language with two-sided communication and yielded performance comparable to that of hand-coded versions for NAS SP and the Sweep3D benchmark.

2.4 Performance Analysis of Parallel Programs

There are many approaches to analyzing the scalability of parallel programs. We can separate the analysis problem into several subproblems: *acquiring* the performance data, *analyzing* it and *presenting* it in a form useful to application developers. Our automatic scaling analysis based on expectations collects performance data on unmodified, fully-optimized binaries using sampling-based callstack profiling implemented by `csprof`, independent of the parallel programming model. Next, it perform a scaling analysis after the program

execution during which associates scalability information with calling context tree nodes. Finally, it uses `hpcviewer` to display this information to an application developer.

Vampir [147], MPE and Jumpshot [197, 199], MPICL [196] and ParaGraph [103, 104] are toolsets that perform tracing of MPI calls; they use instrumented versions of the MPI library. They build and display time-space diagrams of the communication activity. Such tools enable users to visually determine inefficient communication patterns and map them back to source code. They are complementary to the call-stack profiling analysis and visualization provided by `csprof`, the source correlation module and `hpcviewer`. The trace size collected by such tools is proportional to the number of communication calls, while for `csprof` the performance data size is proportional to the size of the call tree. Our scaling analysis method is also able to determine scaling inefficiencies due to non-scaling computation, and attributes scaling impediments to all nodes in the calling context trees.

The Pablo performance analysis environment [166] records and analyzes user specified events. It collects event traces, event counters, and time intervals. It requires instrumentation of the source code to insert calls to the data tracing library; this is achieved through means of a graphical interface. Pablo incorporates several strategies to control the amount of trace data. First, it monitors the frequency of events, and if the frequency of an event exceeds a threshold, then it records only the event count, but not a trace of the event. Second, it performs dynamic statistical clustering of trace data. Our analysis strategy works on unmodified optimized binaries, and the user control the performance data size by controlling the sampling frequencey.

OMPtrace [46] is a trace-based system used for profiling of MPI codes. It performs binary instrumentation of calls into the OpenMP runtime, and can collect metrics from hardware counters to measure events such as cycles, cache misses, floating point instructions and memory loads. OMPtrace also has the ability to collect user-specified events. Traces are then analyzed and displayed by Paraver [162]; a user can instruct Paraver to present both raw and user-defined metrics. We used `csprof` to profile MPI, CAF and UPC programs, and we have no experience with using `csprof` to analyze OpenMP pro-

grams; however, our method should apply to analyze the scaling of SPMD-style OpenMP programs. Our infrastructure also supports the measurement of user-defined events, enabling scaling analysis for them as well.

Falcon [98] is a trace-based online parallel program steering system. Users define "sensors" that are application specific and rely on an instrumentation tool to incorporate them into the executable program. At runtime, trace information collected by these sensors is sent to a central server and analyzed; as a result of this analysis the system or a user can recommend then enforce changes in the program (such as changing an underlying algorithm or replacing global computations with less precise local computations). In this respect it represents also an infrastructure for adaptive improvement of parallel programs. Active Harmony [55, 183, 185] is a software architecture that supports automated runtime tuning of applications. Applications export a set of tuning parameters to the system; an adaptation server would use a search-based strategy to select the set of parameters that yields the best results, i.e. running time or memory usage. Our method perform post-mortem analysis of program scaling analysis, but its results can be used as well to improve program performance. We present in this thesis its applicability to strong scaling analysis, but our method could be applied to analyze scaling with respect to any parameters, such as input size, and could be used to evaluate for example the benefits of using different scalar and parallel algorithms. Also, a steering-based system could use our method to analyze online the benefits of changing aspects of the program execution. By not making any prior assumptions regarding the lack of scaling causes, our method can be used to discover potential scaling parameters, acting as a complement to such online performance tuning systems.

Vetter [186] describes an assisted learning based system that analyzes MPI traces and automatically classifies communication inefficiencies, based on the duration of such communication operations as blocking and nonblocking send/receive. Our analysis method is generally applicable, without looking for particular inefficient performance patterns. We have not explored using learning strategies to analyze the performance data; when analyzing large programs, with numerous subroutine exhibiting various degrees of scaling loss,

we believe that learning and data minining strategies might be necessary to point a user to scaling hotspots.

Wu *et al* [197] present a strategy of performing trace-based analysis of multithreaded MPI implementations running on SMP clusters. A challenge is to account for thread scheduling within an SMP nodes. Their system infers interval records from tracing events; this is then used to generate multiple views such as thread activity view, processor activity view, and thread-processor view, which tracks thread scheduling among different processors on the same node. Interval record data is then visualized using Jumpshot. While our method is applicable independent of the programming model, we do not analyze thread migration; all our experiments used processes bound to processors for the duration of the program.

mpiP [188] uses an instrumented MPI library to record calls to MPI primitives and performs call-stack unwinding of user-selectable depth. Vetter *et al* described a strategy [188] they call rank-based correlation to evaluate the scalability of MPI communication primitives. Their notion of scalability is different than ours: an MPI communication routine does not scale if its rank among other MPI calls performed by the application increases significantly when the number of processors increases. Because `csprof` collects profile data for the whole application automatically, we can compute, associate and display scalability information for all the calling context tree nodes, not just with those associated with MPI calls. Moreover, we can descend inside MPI calls, and analyze if their implementation shows lack of scaling. An important advantage of our method is that it gives a user *quantitative* information regarding the lack of scaling, while the rank-correlation method yields only *qualitative* information. The overhead of mpiP is proportional to the number of MPI calls, while the overhead of `csprof` is proportional to the sampling frequency.

PHOTON MPI [187] uses an instrumented MPI profiling layer and a modified MPI library to implement communication sampling: only some of the MPI blocking and non-blocking communication events are considered according to one of multiple sample strategies. The data gathered can be analyzed at runtime in the profiling layer and only summary

information needs to be kept around and later written to a file. This approach reduces dramatically the size of trace files and also reduces and controls the profiling overhead. However, at the moment this approach does not uses callstack information for data analysis. Our scaling analysis method does not generate a statistical classification of communication without communication library instrumentation. However, the calling context trees for a particular parallel execution could be used to present and classify the communication calls based on their cost, rather than their size. We haven't explored program characterization based on CCTs.

Quartz [20] aims to determine the causes for loss of parallelism for applications running on a multiprocessors system. Quartz can detect causes such as load imbalance, contention on synchronization objects, excessive time spent in serial parts of a code. The main metric of Quartz is normalized processor time, defined as processor time divided by concurrent parallelism. Quartz works by periodically checkpointing to memory the number of busy processors and the state of each processor, and using a dedicated processor to analyze this data. Quartz displays the costs it found in a top-down fashion according to the call graph. Our approach of using `csprof` enables profiling of both shared-memory and distributed memory applications, without dedicated processors; our top-down and bottom-up views enable a user to determine the cost of spin-waiting or communication and assign them to nodes in the calltree. If an application exhibits systematic load imbalance, synchronization objects contention, or serialization, then our method would pinpoint their effects on scaling. However, if the goal is analyzing parallel performance problems based on a single parallel run, then we could use the CCTs collected on different nodes to determine load imbalance by employing the expectation of equal execution times for the CCT nodes of the two performance profiles.

Paradyn [139] is a parallel performance analysis infrastructure that relies on dynamic instrumentation of binaries. Since instrumented program parts exhibit significant execution time overheads, to make this analysis method feasible for long-running parallel programs Paradyn needs to be parsimonious with which program segments are instrumented. The

approach is to use a performance problem search strategy to identify apriori known inefficiency causes, which program parts lead to loss of performance, and at which point in the program execution. The analysis results are used to instrument only those program parts, rather then the whole program. Our scaling analysis method doesn't make any assumption about the scalability impdiments, identifying all non-scaling calling context tree nodes. Such a method could be a complement to Paradyn, by discovering causes of lack of scaling. Our performance data collection is extremely efficient, compared to using instrumentation of binaries; however, at the moment we do not exclude performance data from the final calling context tree, whereas after Paradyn would determine that a program part performs well, it would ignore it in further analysis.

KOJAK [143] is a software system aiming to automatically detect communication bottlenecks. It works with C, C++, and Fortran source code, for the MPI, OpenMP and SHMEM programming models. The approach requires instrumentation of the application. The source code is processed by OPARI [140, 142] which instruments OpenMP constructs and generates calls to the POMP [141] API. Functions can be instrumented at source level using TAU [172] or at binary level using DPCL [69]. MPI calls are instrumented using the PMPI library [137, 138]. The performance traces are produced using the EPILOG [194] library. The resulting traces can be analyzed by the EXPERT [195] analyzer, which attempts to determine patterns that correspond to known inefficiencies, and are then dislayed using the EXPERT presenter. Additionally, the EPILOG traces can be converted to VAMPIR format and visualized with the VAMPIR event trace analysis tool. The execution time overhead is proportional to the number of instrumented functions called and can lead to large output trace sizes. Our method has a controlable overhead, by setting the sampling frequency, and it works on unmodified, fully optimized binaries, being thus easier to use. Our scaling analysis is also independent of the programming model. EXPERT looks for several performance problems categories, which might be more useful for an application developer, while our method determines CCT nodes that exhibit poor scaling, and then relies on the user to identify and address the source of the scaling problems.

Chapter 3

Background

We have introduced the Co-array Fortran programming model in Chapter 1. This chapter describes refinements to CAF aimed towards writing high-performance, scalable and performance portable codes, and the parallel benchmarks used in this thesis.

3.1 Refinements to the CAF Programming Model

Our previous studies [56, 73] identified a few weaknesses of the original CAF language specification that reduce the performance of CAF codes and proposed extensions to CAF to avoid these sources of performance degradation. First, the original CAF specification [156] requires programs to have implicit memory fences before and after each procedure call to ensure that the state of memory is consistent before and after each procedure invocation. This guarantees that each array accessed within a subroutine is in consistent state upon entry and exit from the subroutine. In many cases, an invoked procedure does not access co-array data at all or accesses only co-array data that does not overlap with co-array data accessed by the caller. As a consequence, it is not possible to overlap communication with a procedure's computation with memory fences around the procedure's call sites.

Second, CAF's original team-based synchronization required using collective synchronization even in cases when it is not necessary. In [56], we propose augmenting CAF with unidirectional, point-to-point synchronization primitives: `sync_notify` and `sync_wait`. `sync_notify(q)` sends a notify to process image `q`; this notification is guaranteed to be seen by image `q` only after all communication events previously issued by the notifier to image `q` have been completed. `sync_wait(p)` blocks its caller until it receives a matching notification message from the process image `p`. Communication events for CAF remote

data accesses are blocking. While it is possible to exploit non-blocking communication in some cases, automatically replacing blocking communication with its non-blocking counterpart and overlapping communication with computation requires sophisticated compiler analysis. To enable savvy application developers to overlap communication and computation in cases where compiler analysis cannot do so automatically, it is useful for CAF to provide a user-level mechanism for exploiting non-blocking communication. To address that, we proposed a small set of primitives that enable application developers to delay the completion of communication events, presented in more detail in section 5.5.

Collective communication calls are important building blocks for many parallel algorithms [91], so supporting them efficiently in CAF codes is paramount. There are several alternatives:

1. Users must write their own reductions: this leads to applications that are performance portable.

2. CAF should be extended with collective operations as language primitives. While a recent revision of the CAF standard [154] proposes a small set of collective operations, we believe that CAF users should be able to express complex collective operations such as all-to-all, scatter-gather, and reductions with both traditional operators —sum, product, max, min — and user-defined operations. CAF would be then extended with the corresponding primitives.

3. Collective operations should be provided as part of the standard library, and let the vendors be responsible for the most efficient implementation on a certain platform. This alternative is also pragmatic, but in long term we might prefer to have a CAF compiler analyze the collective operations and perhaps optimize them; this might be more difficult with collectives implemented as library calls.

Algorithms for efficient collective operations use different approaches for different machines and different interconnects; if sophisticated reductions are part of the language or of the standard library, then a CAF compiler could select the appropriate collective operation

implementation for the target architecture at build time, as part of a autotuning step. In Chapter 11 I present and evaluate a set of collective operations extensions to CAF and an implementation strategy based on MPI.

3.2 Benchmarks

3.2.1 The NAS Parallel Benchmarks

The NAS parallel benchmarks [24] are widely used to evaluate the performance of parallel programming models. In this thesis I used several of them: SP, BT, MG, CG, and LU.

NAS SP and BT. As described in a NASA Ames technical report [24], the NAS benchmarks BT and SP are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar pentadiagonal systems resulting from full diagonalization of the approximately factored scheme.

Both SP and BT consist of an initialization phase followed by iterative computations over time steps. Each time step first calculates boundary conditions, then calculates the right hand sides of the equations. Next, it solves banded systems in three computationally intensive bi-directional sweeps along each of the x, y, and z directions. Finally, it updates flow variables. Each time step requires loosely-synchronous communication before the boundary computation, and employs tightly-coupled communication during the forward and backward line sweeps along each dimension.

Because of the line sweeps along each of the spatial dimensions, traditional block distributions in one or more dimensions would not yield good parallelism. For this reason, SP and BT use a skewed block cyclic distribution called multipartitioning [24, 148]. A fundamental property of multipartitioning distributions is that a single physical processor owns all of the tiles that are neighbors of a particular processor's tiles along any given direction. Consequently, if a processor's tiles need to shift data to their right neighbor along a particu-

lar dimension, the processor needs to send values to only one other processor. This property is exploited to achieve scalable performance. With this distribution, each processor handles several disjoint blocks in the data domain. Blocks are assigned to the processors so that there is an even distribution of work for each directional sweep and each processor has a block on which it can compute in each step of every sweep. Using multipartitioning yields full parallelism with even load balance while requiring only coarse-grain communication.

The MPI implementation of NAS BT and SP attempts to hide communication latency by overlapping communication with computation, using non-blocking communication primitives. For example, in the forward sweep, except for the last tile, non-blocking sends are initiated to update the ghost region on its neighbor's next tile. Afterwards, each process advances to its own next tile, posts a non-blocking receive, performs some local computation, then waits for the completion of both its non-blocking send and receive. The same pattern is present in the backward sweep.

NAS MG. The MG multigrid benchmark computes an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $n \times n \times n$ grid with periodic boundary conditions [24].

In the NAS MG benchmark, for each level of the grid, there are periodic updates of the border region of a three-dimensional rectangular data volume from neighboring processors in each of six spatial directions. The MPI implementation uses four buffers, two for receiving and two for sending data. For each of the three spatial axes, two messages (except for the corner cases) are sent using blocking MPI send to update the border regions on the left and right neighbors.

NAS CG. In the NAS CG parallel benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [24]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The irregular communication requirement of this benchmark is a challenge for most systems.

On each iteration of loops involving communication, the MPI version initiates a non-blocking receive to wait for data from the processor specified by `reduce_exch_proc(i)`, followed by an MPI send to the same processor. After the send, the process waits until its MPI receive completes. Thus, there is no overlap of communication and computation.

NAS LU. The NAS LU parallel benchmark solves the 3D Navier-Stokes equation as do SP and BT. LU implements the solution by using a Successive Over-Relaxation (SSOR) algorithm which splits the operator of the Navier-Stokes equation into a product of lower-triangular and upper-triangular matrices (see [24] and [84])). The algorithm solves five coupled nonlinear partial differential equations on a 3D logically structured grid using an implicit pseudo-time marching scheme. It is a challenging application to parallelize effectively due to the potential for generating many small messages between processors. Computationally, the application is structured by computing the elements of the triangular matrices in the subroutines `jacld` and `jacu` respectively. The next step is to solve the lower and upper triangular systems, using subroutines `blts` and `buts`. After these steps, the variables are updated, a new right-hand side is computed and the process repeats inside a time-step loop. The MPI code requires a power-of-two number of processors. The problem is partitioned on processors by repeatedly halving the grid in the dimensions x and y, alternately, until all power-of-two processors are assigned. This results in vertical pencil-like grid partitions on processors.

For each z plane, the computations proceeds as a sweep starting with one corner in a z plane to the opposite corner of the same z-plane; the computation is structured as a wave-front. The communication of partition boundaries occurs after the computation is complete on all diagonals that contact an adjacent partition. This has the potential of generating a relatively large number of small messages of 5 words each.

3.2.2 LBMHD

LBMHD [157] simulates a charged fluid moving in a magnetic field using a Lattice Boltzmann formulation of the magnetohydrodynamics equations. The benchmark performs sim-

ulations for a 2D spatial grid, which is coupled to an octagonal streaming lattice and block distributed over a 2D processor grid. The simulation consists of a sequence of collision and stream steps. A collision step performs computation only on local data. A stream step requires both contiguous and strided communication between processors for grid points at the boundaries of each block, and third degree polynomial evaluation.

Chapter 4

A Source-to-source Compiler for Co-array Fortran

We designed the `cafc` compiler for Co-array Fortran with the major goals of being portable and delivering high-performance on many platforms. Ideally, a programmer would write a CAF program once in a natural style and `cafc` would adapt it for high performance on the target platform of choice.

To achieve this goal, `cafc` performs source-to-source transformation of CAF code into Fortran 95 code augmented with communication operations. By employing source-to-source translation, `cafc` aims to leverage the best Fortran 95 compiler available on the target platform to optimize local computation. We chose to generate Fortran 95 code rather than C code because for scientific programs Fortran 95 compilers tend to generate more efficient code than C compilers, on equivalent codes. For communication, `cafc` typically generates calls to one-sided communication library primitives, such as ARMCI or GASNet; however. For shared memory systems `cafc` can also generate code that employs load and store operations for communication. `cafc` is based on OPEN64/SL [159], a version of the OPEN64 [158] compiler infrastructure that we modified to support source-to-source transformation of Fortran 95 and CAF. This chapter describes joint work with Yuri Dotsenko.

4.1 Memory Management

To support efficient access to remote co-array data on the broadest range of platforms, memory for co-arrays must be managed by the communication substrate; typically, this memory is managed separately from memory managed conventionally by a Fortran 95 compiler's runtime system. Currently, co-array memory is allocated and managed by un-

derlying one-sided communication libraries such as ARMCI and GASNet, for the sake of communication efficiency. For ARMCI, on cluster systems with RDMA capabilities, co-arrays are allocated in memory that is registered and pinned, which enables data transfers to be performed directly using the DMA engine of the NIC. For GASNet, the allocated memory is used with an efficient protocol named Firehose, that register with the NIC and pins the memory pages actually used in communication.

`cafc` has to manage memory for static co-arrays, such as SAVE and COMMON, and for dynamic co-arrays, such as ALLOCATABLE.

- The memory management strategy implemented by `cafc` for SAVE and COMMON co-arrays has three components. At compile time, `cafc` generates procedure view initializers, which are responsible for allocating the proper storage and setting up the co-array representation for local accesses. At link time, `cafc` collects all the initializers and synthesizes a global startup procedure that calls them. Finally, on program launch, the global startup procedure is called and it performs co-array memory allocation and initialization of co-array representation for local access.

- For ALLOCATABLE co-arrays, `cafc` transforms allocation statements into a call to the runtime library that collectively allocates co-array memory and sets the co-array views. On deallocation, `cafc` issues a call to a collective routine that frees the co-array storage.

4.2 Local Co-Array Accesses

For CAF programs to perform well, access to local co-array data must be efficient. Since co-arrays are not supported in Fortran 95, we need to translate references to the local portion of a co-array into valid Fortran 95 syntax. For performance, our generated code must be amenable to back-end compiler optimization. In chapter 5 we describe several alternative representations for co-arrays. Our current strategy is to use a Fortran 95 pointer to access local co-array data. Because the `cafc` runtime system must allocate co-array data

outside the region of memory managed by the Fortran 95 runtime system, we need the ability to initialize and manipulate compiler-dependent representations of Fortran 95 array descriptors. A Fortran 95 pointer consists of an array descriptor known as a dope vectors We leverage code from the CHASM library [165] from Los Alamos National Laboratory to enable `cafc` to be usable with multiple compilers on a range of platforms.

4.3 Remote Co-Array Accesses

Co-array accesses to remote data must be converted into Fortran 95; however, this is not straightforward because the remote memory may be in a different address space. Although the CAF language provides shared-memory semantics, the target architecture may not; a CAF compiler must perform transformations to bridge this gap. On a hardware shared memory platform, the transformation is relatively straightforward since references to remote memory in CAF can be expressed as loads and stores to shared locations; in previous work [74] we explored alternative strategies for performing communication on hardware shared memory systems. The situation is more complicated for cluster-based systems with distributed memory.

To perform data movement on clusters, `cafc` must generate calls to a communication library to access data on a remote node. Moreover, `cafc` must manage storage to temporarily hold remote data needed for a computation. For example, in the case of a read reference of a co-array on another image, as shown in Figure 4.1(a) a temporary, `temp`, is allocated just prior to the statement to hold the value of the `coarr(:)` array section from image p. Then, a call to get data from image p is issued to the runtime library. The statement is rewritten as shown in Figure 4.1(b). The temporary is deallocated immediately after the statement. For a write to a remote image, such as the one in Figure 4.1(c), a temporary `temp` is allocated prior to the remote write statement; the result of the evaluation of the right-hand side is stored in the temporary; a call to a communication library is issued to perform the write; and finally, the temporary is deallocated, as shown in Figure 4.1(d). When possible, the generated code avoids using unneeded temporary buffers. For example,

```
arr(:)=coarr(:)[p] + ...
```

(a) Remote read example

```
... allocate temp ...
... remote read call ...
arr(:) = temp(:) + ...
... deallocate temp ...
```

(b) `cafc`-generated code

```
coarr(:)[p1,p2]= ...
```

(c) Remote write example

```
... allocate temp ...
temp(:)=...
... remote write call ...
... deallocate temp ...
```

(d) `cafc`-generated code

```
coarr(:)[p1,p2]= coarr(:)
```

(e) Co-array to co-array communication

Figure 4.1: Examples of code generation for remote co-array accesses.

for an assignment performing a co-array to co-array copy, such as shown in Figure 4.1(e), `cafc` generates code to move the data directly from the source into the destination. In general, `cafc` generates blocking communication operations. However, user directives [73] enable `cafc` to exploit non-blocking communication.

4.4 Argument Passing

CAF provides two methods of passing co-array data: *by value* and *by co-array*. To pass co-array data by value, one uses parantheses around a co-array reference, as one would do to pass by value in Fortran 95. To pass co-array data by co-array, the programming model requires that an interface always be used for the called subroutine. The shape of

an argument co-array must be defined in the callee; this enables reshaping of co-array arguments. Future work aims to support inference of the interface for functions defined in the same file.

`cafc` replaces a co-array argument passed by co-array by two arguments: one is an array argument *coArrayLocal*, corresponding to the local co-array data; the other, *coArrayHandle*, corresponds to an opaque co-array handle. For accesses to local data, *coArrayLocal* is used; for communication, *coArrayHandle* is passed as an argument to runtime calls. Future work is aimed at removing the *coArrayHandle* and have the runtime determine the co-array memory based on the address of the co-array local part; this would simplify the interoperability of `cafc`-compiled CAF code with other SPMD parallel programming models, such as MPI and UPC.

4.5 Synchronization

To support point-to-point synchronization in CAF (`sync_notify` and `sync_wait`) using the ARMCI runtime library, we collaborated with the developers of ARMCI on the design of suitable `armci_notify` and `armci_wait` primitives. ARMCI ensures that if a blocking or non-blocking `PUT` to a remote process image is followed by a notify to the same process image, then the destination image receives the notification after the `PUT` operation has completed. While ARMCI supports non-blocking communication, on some architectures, the implementation of `armci_notify` itself is blocking. This limits the overlap of communication and computation if a CAF programmer writes a non-blocking write to a remote co-array and notifies the destination process image immediately thereafter.

To support `sync_notify` and `sync_wait` in CAF using the GASNet library, while ensuring the communication completion semantics, we implemented support for this primitives in the `cafc` runtime system. For a parallel execution of a CAF program on P images, `cafc` uses three arrays, as shown in Figure 9.1

The location `sent[p]` stores the number of notifies *sent* to processor p; `received[p]`

```
long sent[P];
long received[P];
long waited[P];
```

Figure 4.2: `cafc`-runtime data structure used to implement the `sync_notify`/`sync_wait` primitives.

stores the notifies count *received* by the current process image from p, while `waited[p]` stores the number of notifies *expected* by the current processor from image p. The `cafc` runtime collects a list of all outstanding communication requests. Upon the execution of a `sync_notify(p)` by processor q, the `cafc` runtime enforces the completion of all outstanding requests to processor p, after which it increments `sent[p]` on q and then copies its contents into `received[q]` on processor p. Upon the execution of a `sync_wait(q)` by processor p, the executing process image increments `waited[q]`, then spin waits until `received[q]` exceeds `waited[q]`.

To maximize the overlap of communication and computation, `sync_notify` should have a non-blocking implementation as well. In chapter 11 we show that blocking notifies constitute a scalability impediment.

4.6 Communication Libraries

For performance portability reasons, we chose to engineer `cafc` on top of portable, one-sided communication libraries. In Section 2.1.2 we presented the capabilities of one-sided communication libraries such as ARMCI and GASNet. The `cafc` runtime can utilize effectively either of the two communication libraries.

4.7 `cafc` Status

At the time of this writing, `cafc` supports COMMON, SAVE, and ALLOCATABLE co-arrays of primitive and user-defined types, passing of co-array arguments, co-arrays with multiple co-dimensions, co-array communication using array sections, the CAF synchro-

nization primitives and most of the CAF intrinsic functions. The following features of CAF are currently not supported: triplets in co-dimensions, and parallel I/O. Ongoing work is aimed at removing these limitations. `cafc` compiles natively and runs on the following architectures: Pentium clusters with Ethernet interconnect, Itanium2 clusters with Myrinet or Quadrics interconnect, Alpha clusters with Quadrics interconnect, SGI Origin 2000 and SGI Altix 3000, Opteron clusters with Infiniband interconnect. Future work aims to port `cafc` onto very large scale systems including BlueGene/L and Cray XT3.

Chapter 5

Optimizing the Performance of CAF Programs

To harness the power of existing parallel machines, one needs to achieve both scalar performance and communication performance.

To achieve high scalar performance when employing source-to-source translation, we need to generate local code amenable to optimization by a backend Fortran compiler. During experiments with `cafc`-compiled codes, we refined the co-array representation for local accesses and designed a transformation, procedure splitting, necessary to achieve good local performance. In this chapter we describe procedure splitting, a transformation necessary to achieve good scalar performance, then compare Fortran 90 representations of COMMON block and SAVE co-arrays on scalable shared-memory multiprocessors to find the one that yields superior performance for local computation. We report our findings for two NUMA SGI platforms (Altix 3000 and Origin 2000) and their corresponding compilers (Intel and SGI MIPSPro Fortran compilers). An important finding is that no single Fortran 90 co-array representation and code generation strategy yields the best performance across all architectures and Fortran 90 compilers.

To obtain communication performance, we need to increase communication granularity and overlap computation and communication. Communication vectorization in CAF codes can be expressed at source level, using the Fortran 95 array section syntax. Another optimization is communication packing, and we present several alternatives for performing it. To achieve communication and computation overlap, we use hints for issuing of non-blocking communication.

An appealing characteristic of CAF is that a CAF compiler can automatically tailor code to a particular architecture and use whatever co-array representations, local data access

methods, and communication strategies are needed to deliver the best performance.

5.1 Procedure Splitting

In early experiments comparing the performance of CAF programs compiled by `cafc` with the performance of Fortran+MPI versions of the same programs, we observed that loops accessing local co-array data in the CAF programs were often significantly slower than the corresponding loops in the Fortran+MPI code, even though the source code for the computational loops were identical. Consider the following lines that are common to both the CAF and Fortran+MPI versions of the `compute_rhs` subroutine of the NAS BT benchmark. (NAS BT is described in Section 6.3.)

```
rhs(1,i,j,k,c) = rhs(1,i,j,k,c) + dx1tx1 *  &
   (u(1,i+1,j,k,c) - 2.0d0*u(1,i,j,k,c) +   &
   u(1,i-1,j,k,c)) -                        &
   tx2 * (u(2,i+1,j,k,c) - u(2,i-1,j,k,c))
```

In both the CAF and Fortran+MPI sources, `u` and `rhs` reside in a single COMMON block. The CAF and Fortran+MPI versions of the program declare identical data dimensions for these variables, except that the CAF code adds a single co-dimension to `u` and `rhs` by appending a "`[*]`" to the end of its declaration. As described in Section 4.2, `cafc` rewrites the declarations of the `u` and `rhs` co-arrays with co-array descriptors that use a deferred-shape representation for co-array data. References to `u` and `rhs` are rewritten to use Fortran 90 pointer notation as shown here:

```
rhs%ptr(1,i,j,k,c) = rhs%ptr(1,i,j,k,c) + dx1tx1 * &
   (u%ptr(1,i+1,j,k,c) - 2.0d0*u%ptr(1,i,j,k,c) +  &
   u%ptr(1,i-1,j,k,c)) -                           &
   tx2 * (u%ptr(2,i+1,j,k,c) - u%ptr(2,i-1,j,k,c))
```

Our experiments showed that the performance differences we observed between the `cafc`-generated code and its Fortran+MPI counterpart result in part from the fact that the Fortran 90 compilers we use to compile `cafc`'s generated code conservatively assume

that the pointers `rhs%ptr` and `u%ptr` might alias one another.* Overly conservative assumptions about aliasing inhibit optimizations.

We addressed this performance problem by introducing an automatic, demand-driven procedure-splitting transformation. We split each procedure that accesses SAVE or COMMON co-array variables into a pair of outer and inner procedures†. We apply this transformation prior to any compilation of co-array features. Pseudo-code in Figure 5.1 illustrates the effect of the procedure-splitting transformation.

The outer procedure retains the same procedure interface as the original procedure. The outer procedure's body contains solely its data declarations, an interface block describing the inner procedure, and a call to the inner procedure. The inner procedure is created by applying three changes to the original procedure. First, its argument list is extended to account for the SAVE and COMMON co-arrays that are now received as arguments. Second, explicit-shape co-array declarations are added for each additional co-array received as an argument. Third, each reference to any SAVE or COMMON co-array now also available as a dummy argument is replaced to use the dummy argument version instead. In Figure 5.1, this has the effect of rewriting the reference to `c(50)` in `f` with a reference to `c_arg(50)` in `f_inner`.

After procedure splitting, the translation process for implementing co-arrays, as described in chapter 4, is performed. The net result after splitting and translation is that within the inner procedure, SAVE and COMMON co-arrays that are now handled as dummy arguments are represented using explicit-shape arrays rather than deferred-shape arrays. Passing these co-arrays as arguments to the inner procedure to avoid accessing SAVE and COMMON co-arrays using Fortran 90 pointers has several benefits. First, Fortran compilers may assume that dummy arguments to a procedure do not alias one another; thus,

*Compiling the `cafc`-generated code for the Itanium2 using Intel's `ifort` compiler (version 8.0) with the `-fno-alias` flag removed some of performance difference in computational loops between the CAF and Fortran+MPI codes.

†Our prototype currently supports procedure splitting only for subroutines; splitting for functions will be added soon.

```
subroutine f(a,b)
real a(10)[*], b(100), c(200)[*]
save c
... = c(50) ...
end subroutine f
```

(a) Original procedure

```
subroutine f(a,b)
real a(10)[*], b(100), c(200)[*]
save c
interface
  subroutine f_inner(a,b,c_arg)
    real a[*], b, c_arg[*]
  end subroutine f_inner
end interface
call f_inner(a,b,c)
end subroutine f

subroutine f_inner(a,b,c_arg)
real a(10)[*], b(100), c_arg(200)[*]
... = c_arg(50) ...
end subroutine f_inner
```

(b) Outer and inner procedures after splitting.

Figure 5.1: Procedure splitting transformation.

these co-arrays are no longer assumed to alias one another. Second, within the inner procedure, the explicit-shape declarations for co-array dummy arguments retain explicit bounds that are otherwise obscured when using the deferred-shape representation for co-arrays in the generated code that was described in Section 4.2. Third, since local co-array data is referenced in the inner procedure as an explicit-shape array, it is known to be contiguous, whereas co-arrays referenced through Fortran 90 pointers may be strided. Our experiments also showed that knowing that data is contiguous improves software prefetching (as well as write hinting in Compaq's Fortran 90 compiler). The overall performance benefits of this

transformation are evaluated in Section 6.1.

5.2 Representing Co-arrays for Efficient Local Computation

To achieve the best performance for CAF applications, it is critical to support efficient computation on co-array data. Because `cafc` uses source-to-source translation into Fortran 90, this leads to the question of what is the best set of Fortran 90 constructs for representing and referencing co-array data. There are two major factors affecting the decision: (i) how well a particular back-end Fortran 90 compiler optimizes different kinds of data references, and (ii) hardware and operating system capabilities of the target architecture.

Most Fortran compilers effectively optimize references to COMMON block and SAVE variables, but fall short optimizing the same computation when data is accessed using Cray or Fortran 90 pointers. The principal stumbling block is alias analysis in the presence of pointers. COMMON block and SAVE variables as well as subroutine formal arguments in Fortran 90 cannot alias, while Cray and Fortran 90 pointers can. When compiling a CAF program, `cafc` knows that in the absence of Fortran EQUIVALENCE statements COMMON block and SAVE co-arrays occupy non-overlapping regions of memory; however, this information is not conveyed to a back-end compiler if `cafc` generates code to access local co-array data through pointers. Conservative assumptions about aliases cause back-end compilers to forgo critical performance optimizations such as software pipelining and unroll-and-jam, among others. Some, but not all, Fortran 90 compilers have flags that enable users to specify that pointers do not alias, which can ameliorate the effects of analysis imprecision.

Besides the aliasing problem, using Fortran 90 pointers to access data can increase register pressure and inhibit software prefetching. The shape of a Fortran 90 pointer is not known at compile time; therefore, bounds and strides are not constant and thus occupy extra registers, increasing register pressure. Also a compiler has no knowledge whether the memory pointed to by a Fortran 90 pointer is contiguous or strided, which complicates generation of software prefetch instructions.

```
type t1
  real, pointer ::  local(:,:)
end type t1
type (t1) ca
```

(a) Fortran 90 pointer representation.

```
type t2
  real ::  local(10,20)
end type t2
type (t2), pointer ::  ca
```

(b) Pointer to structure representation.

```
real ::  a_local(10,20)
pointer (a_ptr, a_local)
```

(c) Cray pointer representation.

```
real ::  ca(10,20)
common /ca_cb/ ca
```

(d) COMMON block representation.

```
subroutine foo(...)
  real a(10,20)[*]
  common /a_cb/ a
  ...
end subroutine foo
```

(e) Original subroutine.

```
!  subroutine-wrapper
subroutine foo(...)
  !  F90 pointer representation of
a
  ...
  call foo_body(ca%local(1,1),...)
end subroutine foo

!  subroutine-body
subroutine foo_body(a_local,...)
  real ::  a_local(10,20)
  ...
end subroutine foo_body
```

(f) Parameter representation.

Figure 5.2: Fortran 90 representations for co-array local data.

The hardware and the operating system impose extra constraints on whether a particular co-array representation is appropriate. For example, on a shared-memory system a co-array should not be represented as a Fortran 90 COMMON variable if a COMMON block cannot be mapped into multiple process images. Below we discuss five possible Fortran 90 representations for the local part of a co-array variable `real a(10,20)[*]`.

Fortran 90 pointer. Figure 5.2(a) shows the representation of co-array data first used by `cafc`. At program launch, `cafc`'s run-time system allocates memory to hold `10` $\times$ `20` array of double precision numbers and initializes the `ca%local` field to point to it.

This approach enabled us to achieve performance roughly equal to that of MPI on an Itanium2 cluster with a Myrinet2000 interconnect using the Intel Fortran compiler v7.0 (using a "no-aliasing" compiler flag) to compile `cafc`'s generated code [56]. Other compilers do not optimize Fortran 90 pointers as effectively. Potential aliasing of Fortran 90 or Cray pointers inhibits some high-level loop transformations in the HP Fortran compiler for the Alpha architecture. The absence of a flag to signal the HP Alpha Fortran compiler that pointers don't alias forced us to explore alternative strategies for representing and ref-

erencing co-arrays. Similarly, on the SGI Origin 2000, the MIPSPro Fortran 90 compiler does not optimize Fortran 90 pointer references effectively.

Fortran 90 pointer to structure. In contrast to the Fortran 90 pointer representation shown in Figure 5.2(a), the *pointer-to-structure* shown in Figure 5.2(b) conveys constant array bounds and contiguity to the back-end compiler.

Cray pointer. Figure 5.2(c) shows how a Cray pointer can be used to represent the local portion of a co-array. This representation has similar properties to the pointer-to-structure representation. Though the Cray pointer is not a standard Fortran 90 construct, many Fortran 90 compilers support it.

COMMON block. On the SGI Altix and Origin architectures, the local part of a co-array can be represented as a COMMON variable in each SPMD process image (as shown in Figure 5.2(d)) and mapped into remote images as symmetric data objects using SHMEM library primitives. References to local co-array data are expressed as references to COMMON block variables. This code shape is the most amenable to back-end compiler optimizations and results in the best performance for local computation on COMMON and SAVE co-array variables (see Section 5.3).

Subroutine parameter representation. To avoid pessimistic assumptions about aliasing, a *procedure splitting* technique can be used. If one or more COMMON block or SAVE co-arrays are accessed intensively within a procedure, the procedure can be split into wrapper and body procedures (see Figures 5.2(e) and 5.2(f)). The wrapper procedure passes all (non-EQUIVALENCEd) COMMON block and SAVE co-arrays used in the original subroutine to the body procedure as explicit-shape arguments[‡]; within the body procedure, these variables are then referenced as routine arguments. This representation enables `cafc` to pass bounds and contiguity information to the back-end compiler. The

[‡]Fortran 90 argument passing styles are described in detail elsewhere [7].

procedure splitting technique proved effective for both the HP Alpha Fortran compiler and the Intel Fortran compiler.

5.3 Evaluation of Representations for Local Accesses

Currently, `cafc` generates code that uses Fortran 90 pointers for references to local co-array data. To access remote co-array elements, `cafc` can either generate ARMCI calls or initialize Fortran 90 pointers for fine-grain load/store communication. Initialization of pointers to remote co-array data occurs immediately prior to statements referencing non-local data; pointer initialization is not yet automatically hoisted out of loops. To evaluate the performance of alternate co-array representations and communication strategies, we hand-modified code generated by `cafc` or hand-coded them. For instance, to evaluate the efficiency of using SHMEM instead of ARMCI for communication, we hand-modified `cafc`-generated code to use `shmem_put`/`shmem_get` for both fine-grain and coarse-grain accesses.

We used two NUMA platforms for our experiments: an SGI Altix 3000[§] and an SGI Origin 2000[¶]. We used the STREAM benchmark to determine the best co-array representation for local and remote accesses. To determine the highest-performing representation for fine-grain remote accesses we studied the Random Access and Spark98 benchmarks. To investigate the scalability of CAF codes with coarse-grain communication, we show results for the NPB benchmarks SP and MG.

The STREAM [134] benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth in MB/s (10^6 bytes/s) and the corresponding computation rate for simple vector kernels. The top half of Figure 5.3 shows vector kernels for a Fortran 90 version of the benchmark. The size of each array should exceed the capacity

[§]Altix 3000: 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128 GB RAM, running the Linux64 OS with the 2.4.21 kernel and the 8.0 Intel compilers

[¶]Origin 2000: 16 MIPS R12000 processors with 8MB L2 cache and 10 GB RAM, running IRIX 6.5 and the MIPSpro Compilers version 7.3.1.3m

of the last level of cache. The performance of compiled code for the STREAM benchmark also depends upon the quality of the code's instruction stream[‖].

```
DO J=1, N
 C(J)=A(J)
END DO
```

(a) Copy

```
DO J=1, N
 B(J)=s*C(J)
END DO
```

(b) Scale

```
DO J=1, N
 C(J)=A(J)+B(J)
END DO
```

(c) Add

```
DO J=1, N
 A(J)=B(J)+s*C(J)
END DO
```

(d) Triad

```
DO J=1, N
 C(J)=A(J)[p]
END DO
```

(e) CAF Copy

```
DO J=1, N
 B(J)=s*C(J)[p]
END DO
```

(f) CAF Scale

```
DO J=1, N
 C(J)=A(J)[p]+B(J)[p]
END DO
```

(g) CAF Add

```
DO J=1, N
 A(J)=B(J)[p]+s*C(J)[p]
END DO
```

(h) CAF Triad

Figure 5.3: The STREAM benchmark kernels (F90 & CAF).

We designed two CAF versions of the STREAM benchmark: one to evaluate the representations for local co-array accesses, and a second to evaluate the remote access code for both fine-grain accesses and bulk communication. Table 5.1 presents STREAM bandwidth measurements on the SGI Altix 3000 and the SGI Origin 2000 platforms.

Evaluation of local co-array access performance. To evaluate the performance of local co-array accesses, we adapted the STREAM benchmark by declaring `A`, `B` and `C` as co-arrays and keeping the kernels from the top half of Figure 5.3 intact. We used the Fortran 90 version of STREAM with the arrays A, B and C in a COMMON block as a baseline for comparison The results are shown in the local access part of the Table 5.1. The results for the COMMON block representation are the same as the results of the original Fortran 90. The Fortran 90 pointer representation without the "no-aliasing" compiler flag yields only 30% of the best performance for local access; it is not always possible to use no-aliasing

[‖]On an SGI Altix, we use `-override_limits -O3 -tpp2 -fnoalias` for the Intel 8.0 compiler. On the Origin, we use `-64 -O3` for the MIPSpro compiler.

flags because user programs might have aliasing unrelated to co-array usage. On both architectures, the results show that the most efficient representation for co-array local accesses is as COMMON block variables. This representation enables the most effective optimization by the back-end Fortran 90 compiler; however, it can be used only for COMMON and SAVE co-arrays; a different representation is necessary for allocatable co-arrays.

Evaluation of remote co-array access performance. We evaluated the performance of remote reads by modifying the STREAM kernels so that A,B,C are co-arrays, and the references on the right-hand side are all remote. The resulting code is shown in the bottom half of Figure 5.3. We also experimented with a bulk version, in which the kernel loops are written in Fortran 90 array section notation. The results presented in the Table 5.1 correspond to the following code generation options (for both fine-grain and bulk accesses): the library-based communication with temporary buffers using ARMCI calls, Fortran 90 pointers, Fortran 90 pointers with the initialization hoisted out of the kernel loops, library-based communication using SHMEM primitives, Cray pointers, Cray pointers with hoisted initialization without the no-aliasing flag, Cray pointers with hoisted initialization, and a vector of Fortran 90 pointers to remote data. The next result corresponds to a hybrid representation: using the COMMON block representation for co-array local accesses and Cray pointers for remote accesses. The last result corresponds to an OpenMP implementation of the STREAM benchmark coded in a similar style to the CAF versions; this is provided to compare the CAF versions against an established shared memory programming model.

The best performance for fine-grain remote accesses is achieved by the versions that use Cray pointers or Fortran 90 pointers to access remote data with the initialization of the pointers hoisted outside loops. This shows that hoisting initialization of pointers to remote data is imperative for both Fortran 90 pointers and Cray pointers. Using the vector of Fortran 90 pointers representation uses a simpler strategy to hoist pointer initialization that requires no analysis, yet achieves acceptable performance. Using a function call per each fine-grain access incurs a factor of 24 performance degradation on Altix and a factor of five on the Origin.

	SGI Altix 3000				SGI Origin 2000			
Program representation	Copy	Scale	Add	Triad	Copy	Scale	Add	Triad
Fortran, COMMON block arrays	3284	3144	3628	3802	334	293	353	335
Local access, F90 pointer, w/o no-aliasing flag	1009	929	1332	1345	323	276	311	299
Local access, F90 pointer	3327	3128	3612	3804	323	277	312	298
Local access, F90 pointer to structure	3209	3107	3629	3824	334	293	354	335
Local access, Cray pointer	3254	3061	3567	3716	334	293	354	335
Local access, split procedure	3322	3158	3611	3808	334	288	354	332
Local access, vector of F90 pointers	3277	3106	3616	3802	319	288	312	302
Remote access, general strategy	33	32	24	24	11	11	8	8
Remote access bulk, general strategy	2392	1328	1163	1177	273	115	99	98
Remote access, F90 pointer	44	44	34	35	10	10	7	7
Remote access bulk, F90 pointer	1980	2286	1997	2004	138	153	182	188
Remote access, hoisted F90 pointer	1979	2290	2004	2010	294	268	293	282
Remote access, shmem_get	104	102	77	77	72	70	57	56
Remote access, Cray pointer	71	69	60	60	26	26	19	19
Remote access bulk, Cray ptr	2313	2497	2078	2102	346	294	346	332
Remote access, hoisted Cray pointer, w/o no-aliasing flag	2310	2231	2059	2066	286	255	283	275
Remote access, hoisted Cray pointer	2349	2233	2057	2073	346	295	347	332
Remote access, vector of F90 pointers	2280	2498	2073	2105	316	291	306	280
Remote access, hybrid representation	2417	2579	2049	2062	350	295	347	333
Remote access, OpenMP	2397	2307	2033	2052	312	301	317	287

Table 5.1: Bandwidth for STREAM in MB/s on the SGI Altix 3000 and the SGI Origin 2000.

For bulk access, the versions that use Fortran 90 pointers or Cray pointers perform better for the kernels Scale, Add and Triad than the general version (1.5-2 times better on an SGI Altix and 2.5-3 times better on an SGI Origin), which uses buffers for non-local data. Copying into buffers degrades performance significantly for these kernels. For Copy, the general version does not use an intermediate buffer; instead, it uses `memcpy` to transfer the data directly into the `C` array and thus achieves high performance.

We implemented an OpenMP version of STREAM that performs similar remote data accesses. On and SGI Altix, the OpenMP version delivered performance similar to the

CAF implementation for the Copy, Add, and Triad kernels, and 90% for the Scale kernel. On an SGI Origin, the OpenMP version achieved 86-90% of the performance of the CAF version.

In conclusion, for top performance on the Altix and Origin platforms, we need distinct representations for co-array local and remote accesses. For COMMON and SAVE variables, local co-array data should reside in COMMON blocks or be represented as subroutine dummy arguments; for remote accesses, `cafc` should generate communication code based on Cray pointers with hoisted initialization.

5.4 Strided vs. Contiguous Transfers

It is well-known that transferring one large message instead of many small messages in general is much cheaper on loosely-coupled architectures. With the column-major layout of co-arrays, one language-level communication event, such as `a(i,1:n)[p]=b(j,1:n)`, might lead to `n` one-element transfers, which can be very costly. To overcome this performance hurdle, an effective solution is to pack strided data on the source, and unpack it on the destination. For example, for a PUT of a strided co-array section, which is non-contiguous in memory, it may be beneficial to pack the section on the sender and unpack it in the corresponding memory locations on the receiver. There can be several levels in the runtime environment where the data can be packed and unpacked to ensure efficient transfers.

In the CAF program This approach requires some effort on the programmer's side and can preclude CAF compiler from optimizing code for tightly-coupled architectures, such as the Cray X1.

By the CAF compiler In a one-sided communication programming paradigm, a major difficulty to pack / unpack data on this level is to transform one-sided communication into two-sided. For a PUT, the CAF compiler can easily generate packing code, but it is difficult to infer where in the program to insert the unpacking code so the receiving image unpacks data correctly. Similar complications arise for GETs. If Active Messages [76] are supported

on a target platform, `cafc` could potentially generate packing code for the source process and an unpacking code snippet to execute on the destination.

In the runtime library This is the most convenient level in the runtime environment to perform packing / unpacking of strided communication. An optimized runtime library can use a cost model to decide if it is beneficial to pack data for a strided transfer. It also knows how to unpack data on the remote image, and it can take advantage of hardware specific features, e.g., RDMA transfers. The ARMCI library used by our CAF compiler runtime library already performs packing/unpacking of data for Myrinet. However, we discovered that it does not currently do packing for Quadrics. Instead, ARMCI relies on the Quadrics driver support for strided transfers, which deliver poor performance.

On a Myrinet network, we determined that the ARMCI packing/unpacking of strided transfers outperforms a strategy based solely on active messages. The explanation for this is that for large messages ARMCI packs chunks of the transfer, sends them to the destination, where it executes unpacking code. By performing effective pipelining of message chunks, ARMCI overlaps packing, communication and unpacking for different chunks. An active-message based solution will not benefit of this overlap and thus lose in performance to ARMCI.

5.5 Hints for Non-blocking Communication

Overlapping communication and computation is an important technique for hiding interconnect latency as well as a means for tolerating asynchrony between communication partners. However, as CAF was originally described [156], all communication must complete before each procedure call in a CAF program. In a study of our initial implementation of `cafc`, we found that obeying this constraint and failing to overlap communication with independent computation hurt performance [56].

Ideally, a CAF compiler could always determine when it is safe to overlap communication and computation and to generate code automatically that does so. However, it is not always possible to determine at compile time whether a communication and a computation

may legally be overlapped. For instance, if the computation and/or the communication use indexed subscripts, making a conservative assumption about the values of indexed subscripts may unnecessarily eliminate the possibility of communication/computation overlap. Also, without whole-program analysis in a CAF compiler, in the presence of separate compilation one cannot determine whether it is legal to overlap communication with a called procedure.

To address this issue, we believe it is useful to provide a mechanism to enable knowledgeable CAF programmers to provide hints as to when communication may be overlapped with computation. Such a mechanism serves two purposes: it enables overlap when conservative analysis would not, and it enables overlap in `cafc`-generated code today before `cafc` supports static analysis of potential communication/computation overlap. While exposing the complexity of non-blocking communication to users is not ideal, we believe it is pragmatic to offer a mechanism to avoid performance bottlenecks rather than forcing users to settle for lower performance.

To support communication/computation overlap in code generated by `cafc`, we implemented support for three intrinsic procedures that enable programmers to demarcate the initiation and signal the completion of non-blocking PUTs. We use a pair of intrinsic calls to instruct the `cafc` run-time system to treat all PUT operations initiated between them as non-blocking. We show this schematically below.

```
region_id =  open_nb_put_region()
...
Put_Stmt_1
...
Put_Stmt_N
...
call close_nb_put_region(region_id)
```

In our current implementation of the `cafc` runtime, only one non-blocking region may be open at any particular point in a process image's execution. Each PUT operation that executes when a non-blocking region is open is associated with the `region_id` of the open non-blocking region. It is a run-time error to close any region other than the one

currently open. Eventually, each non-blocking region initiated must be completed with the call shown below.

```
call complete_nb_put_region(region_id)
```

The completion intrinsic causes a process image to wait at this point until the completion of all non-blocking PUT operations associated with `region_id` that the process image initiated. It is a run-time error to complete a non-blocking region that is not currently pending completion.

Using these hints, the `cafc` run-time system can readily exploit non-blocking communication for PUTs and overlap communication with computation. Overlapping GET communication associated with reads of non-local co-array data with computation would also be useful. We are currently exploring how one might sensibly implement support for overlapping GET communication with computation, either by initiating GETs early or delaying computation that depends upon them.

Chapter 6

An Experimental Evaluation of CAF Performance

In this chapter we describe our implementation strategy for NAS CG, BT, SP and LU, and present performance results on multiple architectures. A major result is that CAF codes can match the performance of hand-tuned MPI benchmarks on multiple platforms. We also evaluate the impact of the scalar and communication performance optimizations described in Chapter 5.

6.1 Experimental Evaluation

We compare the performance of the code `cafc` generates from CAF with hand-coded MPI implementations of the NAS MG, CG, BT, SP and LU parallel benchmark codes. The NPB codes are widely regarded as useful for evaluating the performance of compilers on parallel systems. For our study, we used MPI versions from the NPB 2.3 release. Sequential performance measurements used as a baseline were performed using the NPB 2.3-serial release.

For each benchmark, we compare the parallel efficiency of MPI and `cafc`-generated code for each benchmark. We compute parallel efficiency as follows. For each parallel version ρ, the efficiency metric is computed as $\frac{t_s}{P \times t_p(P,\rho)}$. In this equation, t_s is the execution time of the original sequential version implemented by the NAS group at the NASA Ames Research Laboratory; P is the number of processors; $t_p(P, \rho)$ is the time for the parallel execution on P processors using parallelization ρ. Using this metric, perfect speedup would yield efficiency 1.0 for each processor configuration. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gouge the relative performance of multiple benchmark implementations across the *entire* range of

processor counts.

To evaluate the performance of CAF programs optimized by `cafc` we performed experiments on three cluster platforms. The first platform we used was the Alpha cluster at the Pittsburgh Supercomputing Center. Each node is an SMP with four 1GHz processors and 4GB of memory. The operating system is OSF1 Tru64 v5.1A. The cluster nodes are connected with a Quadrics interconnect (Elan3). We used the Compaq Fortran 90 compiler V5.5. The second platform was a cluster of HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB of L1 cache, 256KB of L2 cache, and 1.5MB of L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel Fortran compiler version 8.0 for Itanium as our Fortran 90 back-end compiler. The third platform was a cluster of HP Long's Peak dual-CPU workstations at the Pacific Northwest National Laboratory. The nodes are connected with Quadrics QSNet II (Elan 4). Each node contains two 1.5GHz Itanium2 processors with 32KB/256KB/6MB L1/L2/L3 cache and 4GB of RAM. The operating system is Red Hat Linux (kernel version 2.4.20). The back-end compiler is the Intel Fortran compiler version 8.0. For all three platforms we used only one CPU per node to avoid memory contention.

In the following sections, we briefly describe the NAS benchmarks used in our evaluation, the key features of their MPI and CAF parallelizations and compare the performance of the CAF and MPI implementations on both architectures studied.

6.2 NAS CG

The MPI version of NAS CG is described in section 3.2.1. Our tuned CAF version of NAS CG does not differ much from the MPI hand-coded version. In fact, we directly converted two-sided MPI communication into equivalent calls to notify/wait and a vectorized one-sided get communication event. Figure 6.2 shows a typical fragment of our CAF parallelization using notify/wait synchronization. Our experiments showed that for this code,

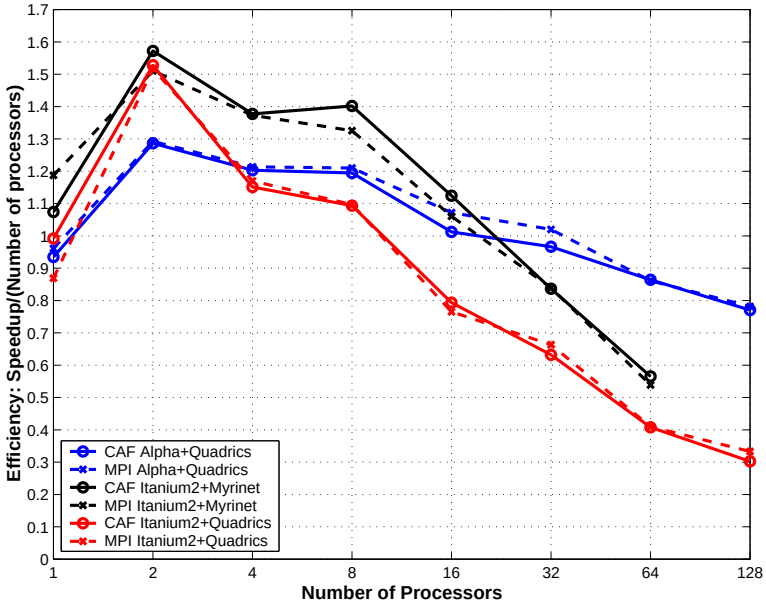


Figure 6.1: Comparison of MPI and CAF parallel efficiency for NAS CG on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

replacing the co-array remote read (`get`) operation with a co-array remote write (PUT) had a negligible effect on performance because of the amount of synchronization necessary to preserve data dependences.

In initial experimentation with our CAF version of CG on various numbers of processors, we found that on less than eight processors, performance was significantly lower than its MPI counterpart. In our first CAF implementation of CG, the receive array q was a common block variable, allocated in the static data by the compiler and linker. To perform the communication shown in Figure 6.2 our CAF compiler prototype allocated a temporary buffer in memory registered with ARMCI so that the Myrinet hardware could initiate a DMA transfer. After the `get` was performed, data was copied from the temporary buffer into the q array. For runs on a small number of processors, the buffers are large. Moreover, the registered memory pool has the starting address independent of the addresses of

```
! notify our partner that we are here and wait for
! him to notify us that the data we need is ready
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! get data from our partner
  q(n1:n2) = w(m1:m1+n2-n1)[reduce_exch_proc(i)]
! synchronize again with our partner to
! indicate that we have completed our exchange
! so that we can safely modify our part of w
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! local computation
  ... use q, modify w ...
```

Figure 6.2: A typical fragment of optimized CAF for NAS CG.

the common blocks. Using this layout of memory and a temporary communication buffer caused the number of L3 cache misses in our CAF code to be up to a factor of three larger than for the corresponding MPI code, resulting in performance that was slower by a factor of five. By converting q (and other arrays used in co-array expressions) to co-arrays, it moved their storage allocation into the segment with co-array data (reducing the potential for conflict misses) and avoided the need for the temporary buffer. Overall, this change greatly reduced L3 cache misses and brought the performance of the CAF version back to level of the MPI code. Our lesson from this experience is that memory layout of communication buffers, co-arrays, and common block/save arrays might require thorough analysis and optimization.

To summarize, the important CAF optimizations for CG are: communication vectorization, synchronization strength-reduction and data layout management for co-array and non-coarray data. Here we describe experiments with NAS CG class C (size 150000, 75 iterations). Figure 6.1 shows that on the Alpha+Quadrics and the Itanium2+Quadrics clusters our CAF version of CG achieves comparable performance to that of the MPI version. The CAF version of CG consistently outperforms the MPI version for all the parallel runs

on Itanium2+Myrinet.

Experiments with CG have showed that using PUTs instead of GETs on the Quadrics platforms yields performance improvements of up to 8% for large scale jobs on the Alpha + Quadrics platform and up to 3% on the Itanium2+Quadrics platform.

6.3 NAS SP and BT

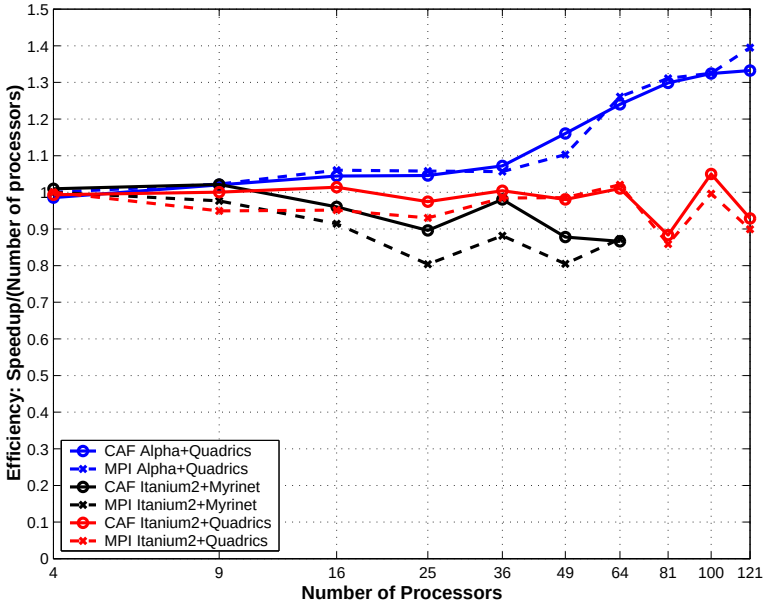


Figure 6.3: Comparison of MPI and CAF parallel efficiency for NAS BT on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

An overview of the MPI versions of NAS BT and SP is described in section 3.2.1. Our CAF implementations of the BT and SP benchmarks was inspired by the MPI version. When converting from MPI-based communication to co-arrays, two major design choices were investigated. First, we could use the same data distribution (same data structures) as the MPI version, but use co-arrays instead of regular MPI buffers. The communication is then expressed naturally in co-array syntax by describing the data movement from the co-array buffer on the sender to the co-array buffer on the receiver. The second alter-

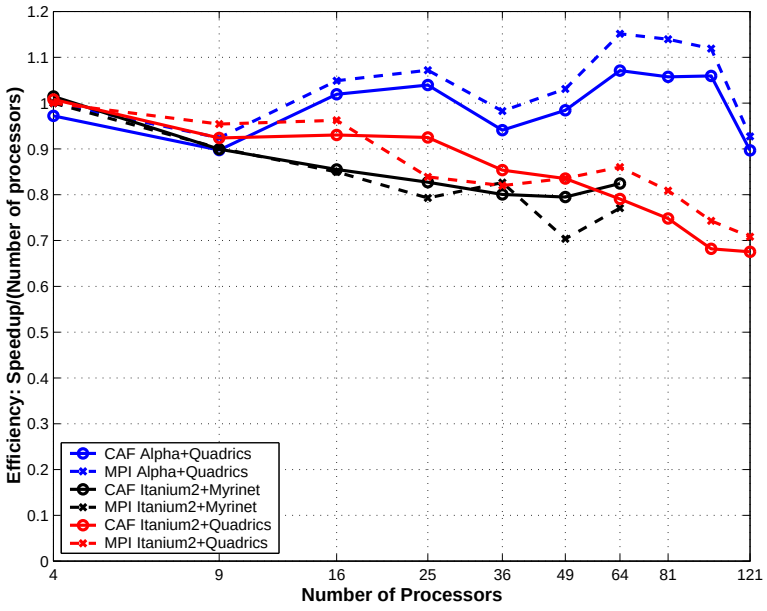


Figure 6.4: Comparison of MPI and CAF parallel efficiency for NAS SP on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

```
lhs( 1:BLOCK_SIZE, 1:BLOCK_SIZE,
     cc, -1,
     0:JMAX-1, 0:KMAX-1,
     cr) [successor(1)] =
lhs( 1:BLOCK_SIZE, 1:BLOCK_SIZE,
     cc, cell_size(1,c)-1,
     0:JMAX-1, 0:KMAX-1, c)
```

(a) NAS BT

```
.... pack into out_buffer_local......

out_buffer(1:p, stage+1:stage+1)
     [successor(1)] =
out_buffer_local(1:p, 0:0)

.... unpack from out_buffer.........
```

(b) NAS SP

Figure 6.5: Forward sweep communication in NAS BT and NAS SP.

native follows more closely the spirit of the language. The working data itself is stored into co-arrays, and then the communication is expressed using co-array syntax, without any intermediate buffers for packing and unpacking. Each design choice influences the synchronization required to achieve correct results.

The CAF implementation for BT and SP inherits the multipartitioning scheme used by

the MPI version. In BT, the main working data resides in co-arrays, while in SP it resides in non-shared arrays. For BT, during the boundary condition computation and during the forward sweep for each of the axes, in the initial version no buffers were used for packing and unpacking, as shown in Figure 6.5(a); however we had to follow PUTs with notifies, to let the other side know the data is available. A second version performed source-level communication packing. On the contrary, in SP all the communication is performed via co-array buffers (see Figure 6.5(b)). In the backward sweep, both BT and SP use auxiliary co-array buffers to communicate data.

In our CAF implementation of BT, we had to consider the trade-off between the amount of memory used for buffers and the amount of necessary synchronization. By using more buffer storage we were able to eliminate both output and anti-dependences due to buffer reuse, thus obviating the need for extra synchronization. We used a dedicated buffer for each communication event during the sweeps, for a total buffer size increase by a factor of square root of the number of processors. Experimentally we found that this was beneficial for performance while the memory increase was acceptable. To yield better performance on cluster architectures, we manually converted co-array GETs into PUTs. Another issued we faced was determining the correct offset in the remote co-array buffer where to put the data. In order to avoid extra communication necessary to retrieve the offsets, our CAF version exchanged this information during the program initialization stage. This stage does not appear in the time measurements, which only consider the time-steps.

It is worth mentioning that the initial version of CAF benchmark was developed on a Cray T3E, and our intended platform was an Itanium2 cluster with Myrinet interconnect. Several features available on the Cray T3E, such as efficient fine-grain communication and efficient global synchronization, were not present on clusters. In order to obtain high-performance, we had to apply by hand the transformations such as: communication vectorization, conversion of barriers into notifies, get to put conversion.

The performance achieved by the CAF versions of BT class C (size 162^3, 200 iterations) and SP class C (size 162^3, 400 iterations) are presented in Figures 6.3 and 6.4. On the

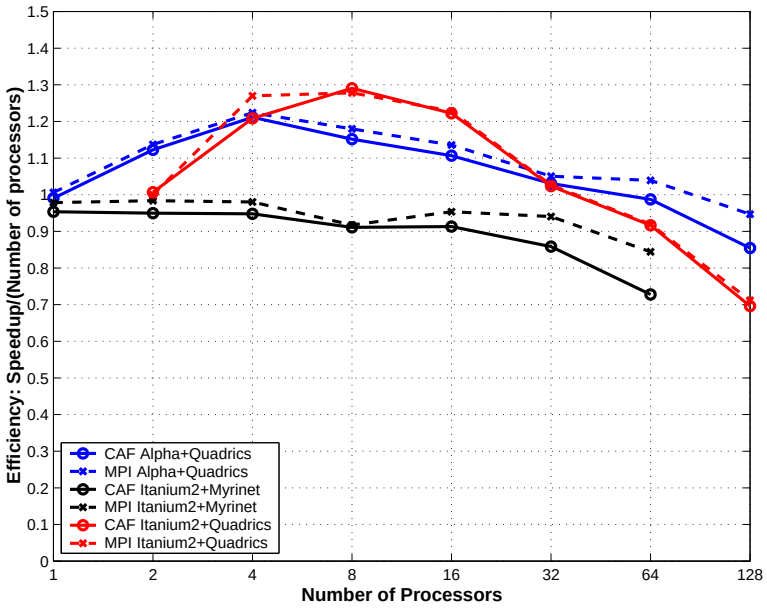


Figure 6.6: Comparison of MPI and CAF parallel efficiency for NAS LU on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

Alpha+Quadrics cluster, the performance of the CAF version of BT is comparable to that of the MPI version. On the Itanium2+Myrinet cluster, CAF BT outperforms the MPI versions by as much as 8% (and is comparable for 64 processors); on the Itanium2+Quadrics cluster, our CAF version of BT exceeds the MPI performance by up to 6% (3% on 121 processors). The CAF versions of SP is outperformed by MPI on the Alpha+Quadrics cluster by up to 8% and Itanium2+Quadrics clusters by up to 9%. On the Itanium2+Myrinet cluster, SP CAF exceeds the performance of MPI CAF by up to 7% (7% on 64 processors). The best performing CAF versions of BT and SP use procedure splitting, packed PUTs and non-blocking communication generation.

6.4 NAS LU

The MPI version of NAS LU is described in section 3.2.1. Our CAF implementation follows closely the MPI implementation. We have transformed into co-arrays the grid pa-

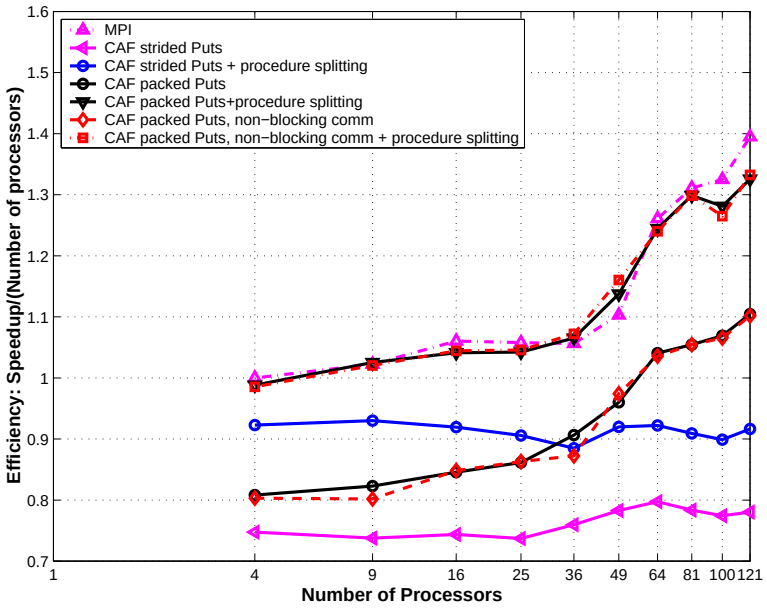


Figure 6.7: Parallel efficiency for several CAF versions of NAS BT on an Alpha+Quadrics cluster.

rameters, the field variables and residuals, the output control parameters and the Newton-Raphson iteration control parameters. Local computation is similar to that of MPI. The various exchange procedures use co-arrays with two co-dimensions in order to naturally express communication with neighbors in four directions: north, east, south and west. For example, a processor with the co-indices `[row,col]` will send data to `[row+1,col]` when it needs to communicate to the south neighbor and to `[row,col-1]` for the west neighbor.

The experimental results for the MPI and CAF versions of LU class C (162^3, 250 iterations) on all platforms are presented in Figure 6.6. On the Alpha+Quadrics cluster the MPI version outperforms the CAF version by up to 9%; on the Itanium2+Myrinet cluster, MPI LU exceeds the performance of CAF LU by as much as 13%. On the Itanium2+Quadrics cluster, the CAF and MPI versions of LU achieve comparable performance. The best performing CAF version of LU uses packed PUTs and procedure splitting.

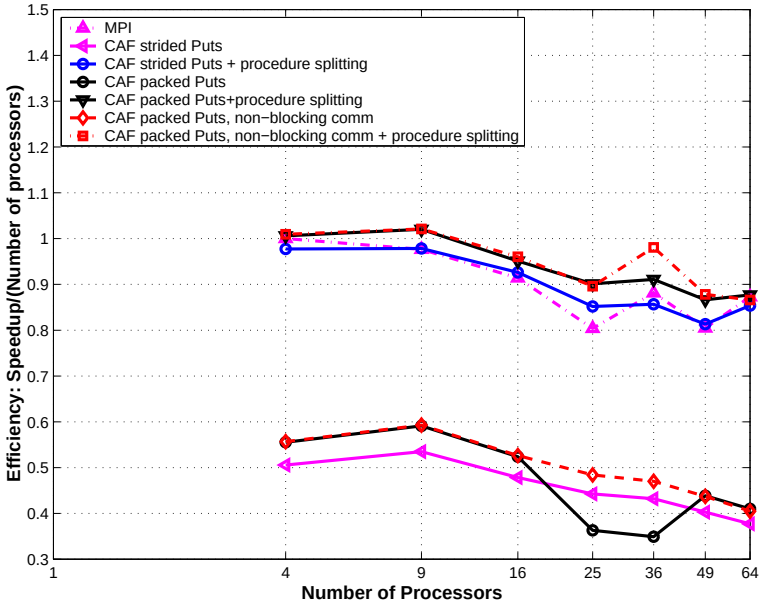


Figure 6.8: Parallel efficiency for several CAF versions of NAS BT on an Itanium2+Myrinet cluster.

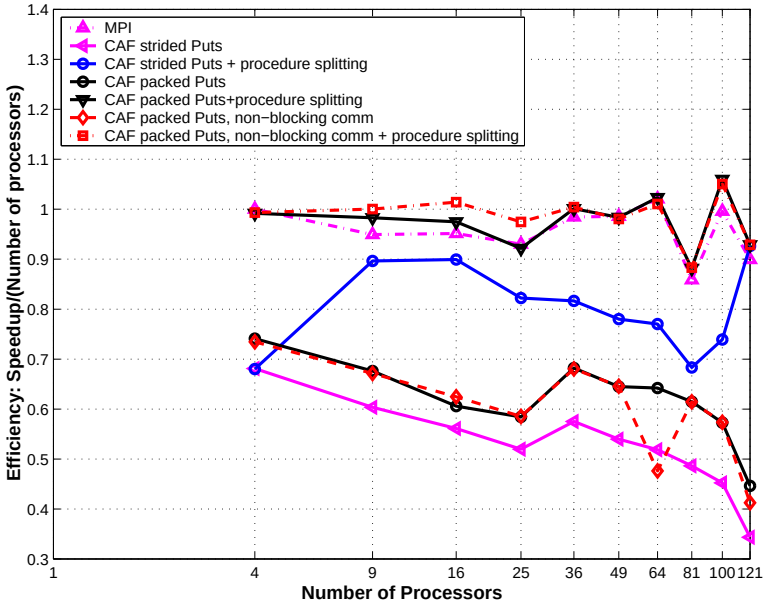


Figure 6.9: Parallel efficiency for several CAF versions of NAS BT on an Itanium2+Quadrics cluster.

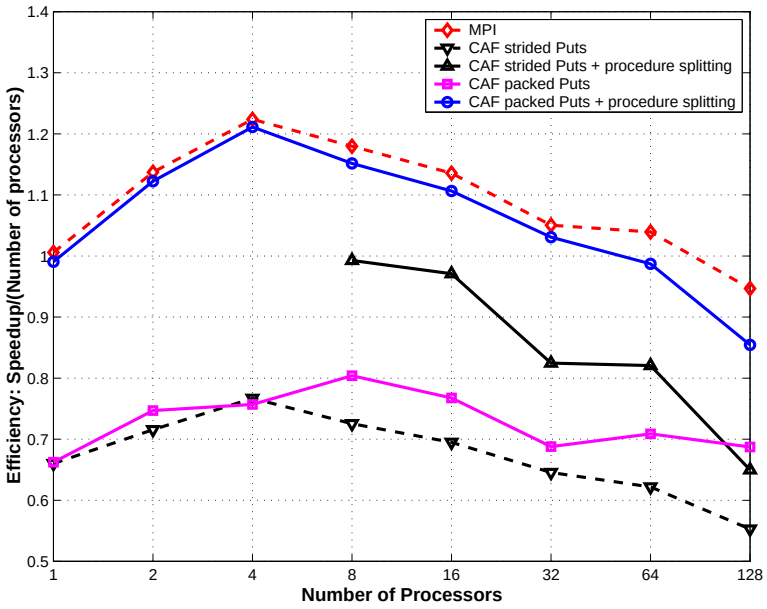


Figure 6.10: Parallel efficiency for several CAF versions of NAS LU on an Alpha+Quadrics cluster.

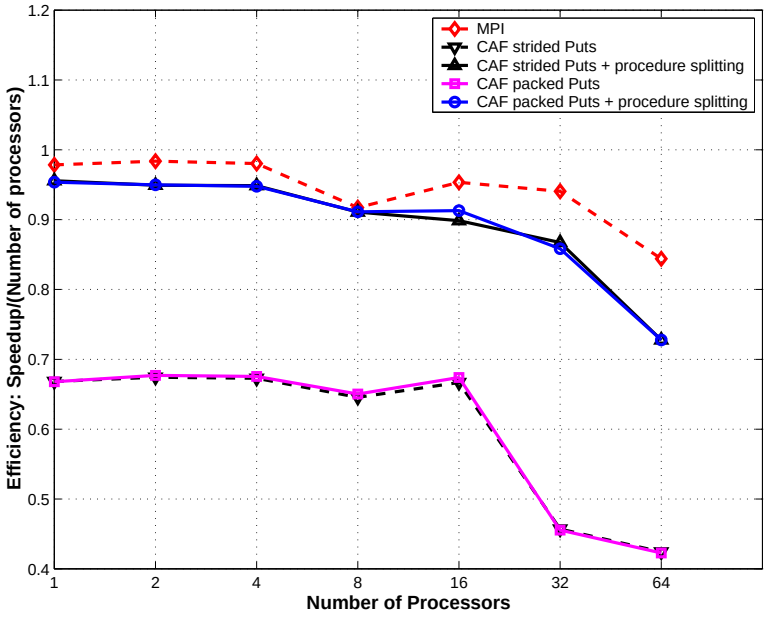


Figure 6.11: Parallel efficiency for several CAF versions of NAS LU on an Itanium2+Myrinet cluster.

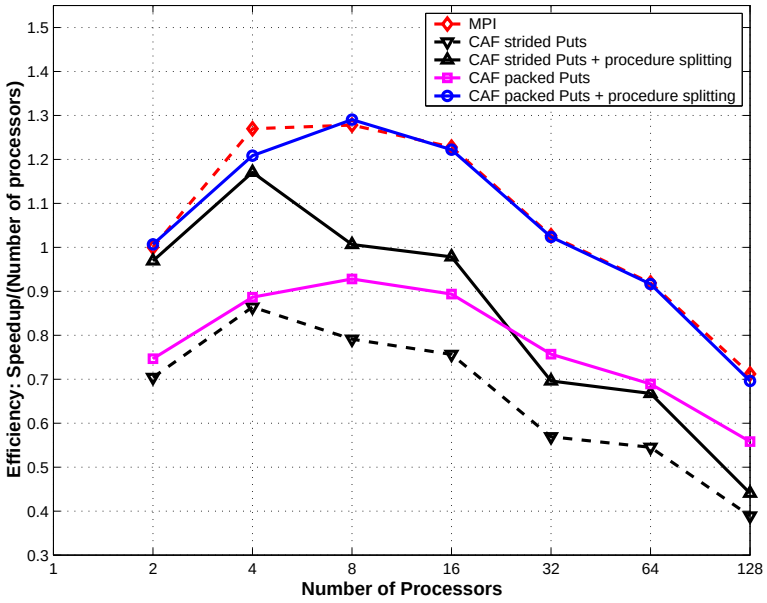


Figure 6.12: Parallel efficiency for several CAF versions of NAS LU on an Itanium2+Quadrics cluster.

6.5 Impact of Optimizations

In Chapter 5, we described several optimizations to improve the performance of CAF programs: procedure splitting, issuing of non-blocking communication and communication packing. To experimentally evaluate the impact of each optimization, we implemented several versions of each of the NPB benchmarks presented above. In Figures 6.7, 6.9, and 6.7 we present results on the Alpha+Quadrics, the Itanium2+Myrinet and the Itanium2+Quadrics clusters for the MPI version of BT and the following BT CAF versions: strided PUTs, strided PUTs with procedure splitting, packed PUTs, packed PUTs with procedure splitting, packed non-blocking PUTs and packed non-blocking PUTs with procedure splitting. In Figures 6.10, 6.11, and 6.12 we present results on the Alpha+Quadrics, the Itanium2+Myrinet and the Itanium+Quadrics clusters for the MPI version of LU and the following CAF versions: strided PUTs, strided PUTs with procedure splitting, packed PUTs and packed PUTs with procedure splitting. For both BT and LU the communication

packing is performed at source level.

For BT, procedure splitting is a high-impact transformation: it improves the performance by 13–20% on the Alpha+Quadrics cluster, by 25–55% on the Itanium2+Quadrics cluster, and by 42–60% on the Itanium2 + Myrinet cluster. For LU, procedure splitting yields an improvement of 15–33% on Alpha+Quadrics and 29–42% on Itanium2 + Myrinet. The CAF versions of BT and LU benefit significantly from the procedure splitting optimization because SAVE and COMMON co-arrays are heavily used in local computations. For benchmarks such as CG, MG and SP, where co-arrays are used solely for data movement (by packing data, sending it and unpacking it on the destination) the benefits of the procedure splitting are modest. In addition, procedure splitting doesn't degrade performance for any of the programs we used in our experiments.

For BT, non-blocking PUTs improved performance by up to 2% on the Alpha+Quadrics platform, by up to 7% on the Itanium2+Myrinet platform and by up to 5% on the Itanium2+Quadrics platform. For MG, non-blocking PUTs improved performance by up to 3% on all platforms. For SP, non-blocking communication improved performance as much as 8% on Itanium2+Myrinet, though only up to 2% on the Quadrics clusters.

Packing data and performing contiguous rather than strided PUTs yields a performance improvement on both Quadrics platforms, on which the ARMCI library does not provide automatic packing. On the Myrinet platform, ARMCI supports data packing for communication, and thus there is no improvement from packing data at source level in CAF applications. For BT CAF, the execution time is improved up to 31% on the Alpha+Quadrics cluster and up to 30% on the Itanium2+Quadrics cluster. For LU CAF, the improvement is up to 24% on the Alpha+Quadrics cluster and up to 37% on the Itanium2+Quadrics cluster.

Chapter 7

Comparing the Performance of CAF and UPC Codes

In chapter 6 we have presented the impact of communication and synchronization optimizations on CAF implementations of the NAS benchmarks. Communication aggregation and generating code amenable to backend compiler optimizations are important concerns for other PGAS languages as well. In this chapter we evaluate the UPC implementations of the NAS benchmarks CG and BT and show how applying source level optimizations can improve their scalar and communication performance. The UPC programming model and UPC compilers were reviewed in Section 2.2.1.

7.1 Methodology

To assess the ability of PGAS language implementations to deliver performance, we compare the performance of CAF, UPC and Fortran+MPI implementations of the NAS Parallel Benchmarks (NPB) CG and BT. The NPB codes are widely used for evaluating the performance of parallel compilers and parallel systems. For our study, we used MPI codes from the NPB 2.3 release. Sequential performance measurements used as a baseline were performed using the Fortran-based NPB 2.3-serial release. The CAF and UPC benchmarks were derived from the corresponding NPB-2.3 MPI implementations; they use essentially the same algorithms as the corresponding MPI versions.

MPI versions of the NAS CG and BT were described in section 3.2.1. We presented the CAF versions of NAS CG and BT in sections 6.2 and 6.3.

7.2 Experimental Platforms

Our experiments studied the performance of the NAS CG and BT benchmarks on four architectures.

The first platform is a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB of L1 cache, 256KB of L2 cache, and 1.5MB L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel compilers V8.0 as our back-end compiler and the Berkeley UPC compiler V2.1.0* with the `gm` conduit.

The second platform was the Lemieux Alpha cluster at the Pittsburgh Supercomputing Center. Each node is an SMP with four 1GHz processors and 4GB of memory. The operating system is OSF1 Tru64 v5.1A. The cluster nodes are connected with a Quadrics interconnect (Elan3). We used the Compaq Fortran 90 compiler V5.5 and Compaq C/C++ compiler V6.5 as well as the Berkeley UPC compiler V2.0.1† using the `elan` conduit.

The other two platforms are non-uniform memory access (NUMA) architectures: an SGI Altix 3000 and an SGI Origin 2000. The Altix 3000 has 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128GB RAM, running the Linux64 OS with the 2.4.21 kernel, Intel compilers V8.0, and the Berkeley UPC compiler V2.1.0‡ using the `shmem` conduit. The Origin 2000 has 32 MIPS R10000 processors with 4MB L2 cache and 16 GB RAM, running IRIX64 V6.5, the MIPSpro Compilers V7.4 and the Berkeley UPC compiler V2.0.1§ using the `smp` conduit.

---

*back-end compiler options: -override_limits -O3 -g -tpp2

†back-end compiler options: -fast -O5 -tune host -intrinsics

‡back-end compiler options: -override_limits -O3 -g -tpp2

§back-end compiler options: -64 -mips4 -DMPI -O3

7.3 Performance Metric

For each application and platform, we selected the largest problem size for which all the MPI, CAF, and UPC versions ran and verified within the architecture constraints (mainly memory).

For each benchmark, we compare the parallel efficiencies of the CAF, UPC and MPI versions. We compute parallel efficiency as follows. For each parallel version ρ, the efficiency metric is computed as $\frac{t_s}{P \times t_p(P,\rho)}$. In this equation, t_s is the execution time of the original Fortran sequential version implemented by the NAS group at the NASA Ames Research Laboratory; P is the number of processors; $t_p(P, \rho)$ is the time for the parallel execution on P processors using parallelization ρ. Using this metric, perfect speedup would yield efficiency 1.0. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts. There are also sequential C implementations of the NAS CG and BT benchmarks that employ the same algorithms as the original Fortran versions. The performance of the C version of CG is similar to that of the original Fortran versions. The C version of BT is up to two times slower than its Fortran variant.

7.4 NAS CG

Figures 7.1 and 7.2 show the parallel efficiency of NAS CG classes A (problem size 14000) and C (problem size 150000) on an Itanium2+Myrinet 2000 cluster. In the figure, MPI represents the NPB-2.3 MPI version, CAF represents the fastest CAF version, *BUPC* represents a UPC implementation of CG compiled with the Berkeley UPC compiler, *CAF-barrier* represents a CAF version using barrier synchronization, and *BUPC-reduction* represents an optimized UPC version.

The CAF version of CG was derived from the MPI version by converting two-sided MPI communication into equivalent calls to notify/wait and vectorized one-sided communica-

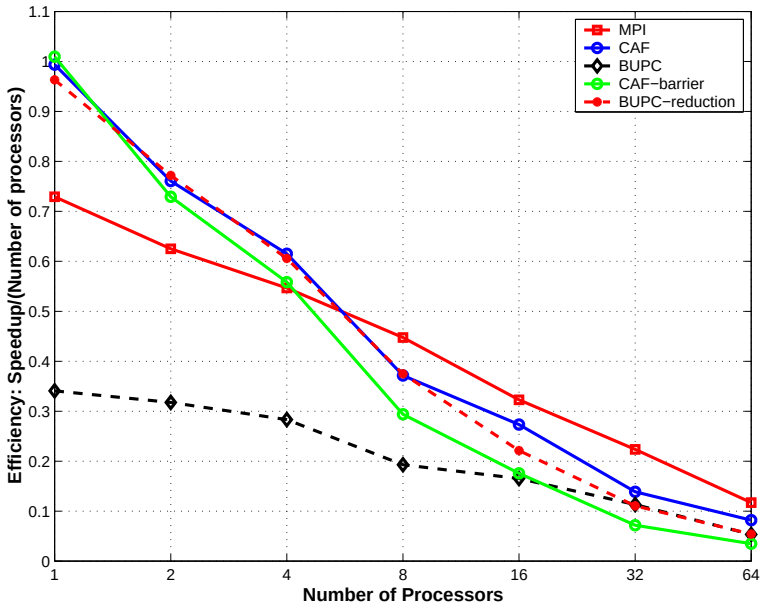


Figure 7.1: Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class A on an Itanium2+Myrinet architecture.

tion [56]. The *BUPC* version is also based on the MPI version; it uses UPC shared arrays for communication and split-phase barriers and employs thread-privatization [42] (using regular pointers to access shared data available locally) for improved scalar performance.

The performance of the MPI and CAF versions is comparable for class C, consistent with our previous studies [56, 73]. The performance of *BUPC* was up to a factor of 2.5 slower than that of MPI. By using HPCToolkit, we determined that for one CPU, both the MPI and the *BUPC* versions spend most of their time in a loop that performs a sparse vector-matrix product; however, the *BUPC* version spent over twice as many cycles in the loop as the Fortran version. The UPC and the Fortran versions of the loop are shown in Figure 7.5. By inspecting the Intel C and Fortran compilers optimization report, we determined that the Fortran compiler recognizes that the loop performs a sum reduction and unrolls it, while the C compiler does not unroll it. We manually modified the UPC version of the loop to compute the sum using two partial sums, as shown in Figure 7.5(c); we denote this version *BUPC-reduction*. On Itanium processors, this leads to a more efficient

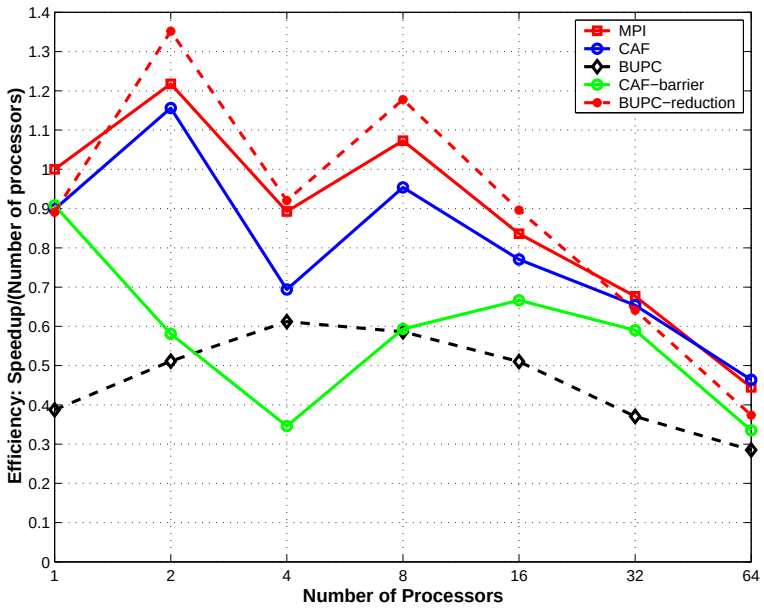


Figure 7.2: Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class C on an Itanium2+Myrinet architecture.

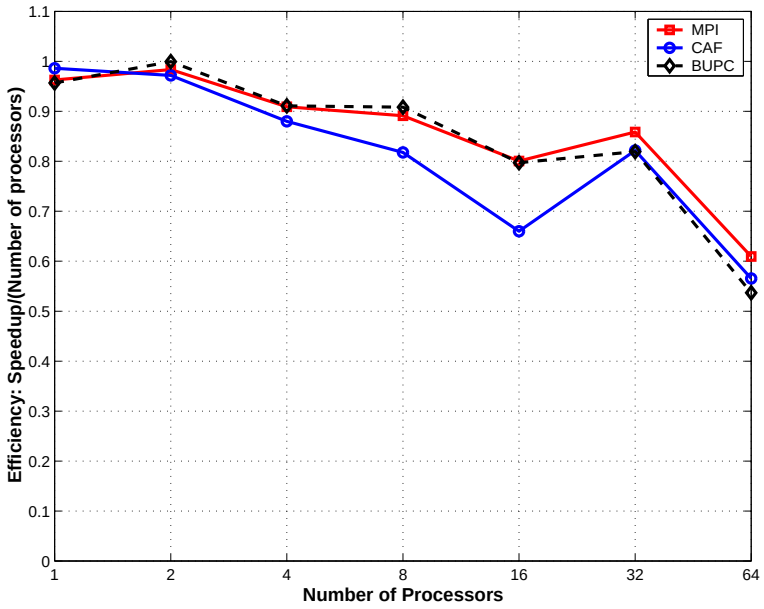
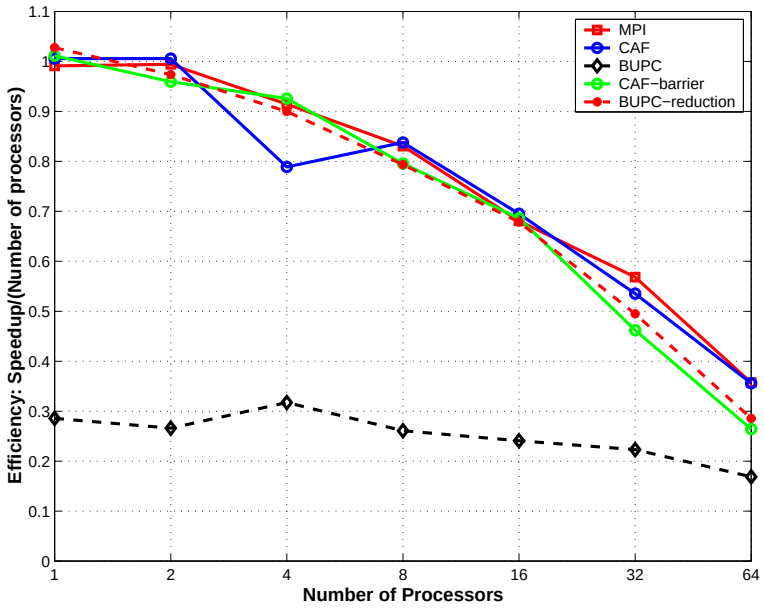
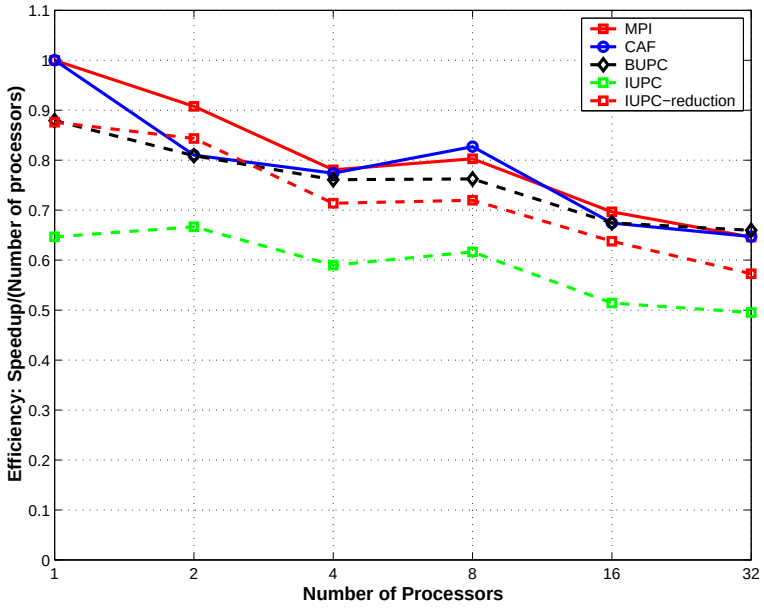


Figure 7.3: Comparison of MPI, CAF and UPC parallel efficiency for NAS CG class B on an Alpha+Quadrics architecture.



(a) CG class C on Altix 3000



(b) CG class B on Origin 2000

Figure 7.4: Comparison of MPI, CAF and UPC parallel efficiency for NAS CG on SGI Altix 3000 and SGI Origin 2000 shared memory architectures.

```
sum = 0.0;
for (k = rowstr[j];
     k < rowstr[j+1];
     k++) {
  sum +=
    a[k-1]*p[colidx[k-1]-1];
}
```

(a) UPC

```
sum = 0.d0
do k=rowstr(j),rowstr(j+1)-1
  sum = sum + a(k)*p(colidx(k))
end do
```

(b) Fortran

```
t1 = t2 = 0
for (...; k+=2 ) {
  t1 += a[k-1] * p[colindex[k-1]-1]
  t2 += a[k] * p[colindex[k]-1]
}
// + fixup code if the range of k isn't even
sum = t1 + t2
```

(c) UPC with sum reduction

Figure 7.5: UPC and Fortran versions of a sparse matrix-vector product.

instruction schedule.

For one CPU, *BUPC-reduction* achieved the same performance as MPI. The graph in Figure 7.2 shows that *BUPC-reduction* is up to 2.6 times faster then *BUPC*. On up to 32 CPUs, *BUPC-reduction* is comparable in performance to MPI. On 64 CPUs, *BUPC-reduction* is slower by 20% than the MPI version. To explore the remaining differences, we investigated the impact of synchronization. We implemented a CAF version that uses barriers for synchronization to mimic the synchronization present in *BUPC-reduction*. As shown in Figure 7.2, the performance of *CAF-barrier* closely matches that of *BUPC-reduction* for large numbers of CPUs; it also experiences a 38% slowdown compared to the CAF version.

Figure 7.3 shows the parallel efficiency of NAS CG class B (problem size 75000) on

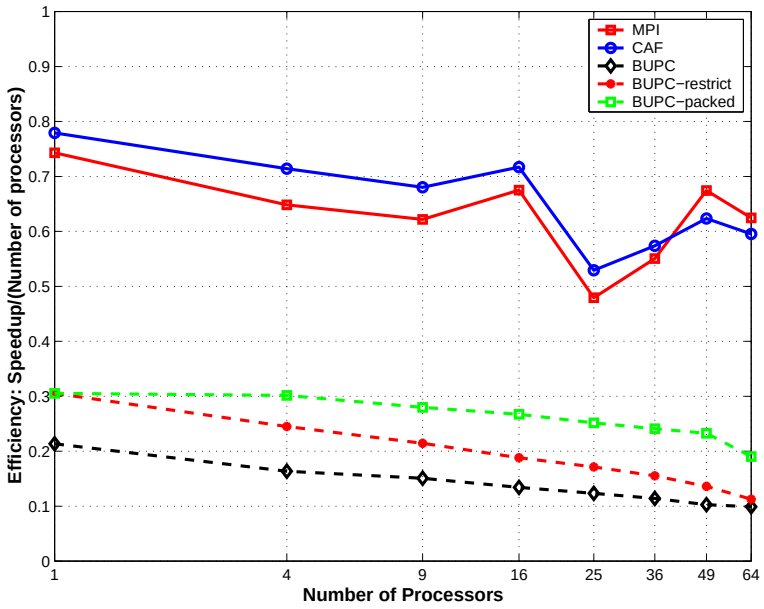


Figure 7.6: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class A, on an Itanium2+Myrinet architecture.

an Alpha+Quadrics cluster. This study evaluated the same versions of the MPI, CAF and *BUPC* codes as on the Itanium2+Myrinet 2000 cluster. On this platform, the three versions of NAS CG achieve comparable performance. The Compaq compiler was able to optimize the non-unrolled C version of the sparse matrix-vector product loop; for this reason *BUPC* and *BUPC-reduction* yield similar performance.

Figure 7.4(a) shows the parallel efficiency of NAS CG class C (problem size 150000) on an SGI Altix 3000. This study evaluates the same versions of NAS CG as those used on the Itanium2+Myrinet 2000 cluster. The CAF and MPI versions have similar performance. *BUPC* is up to a factor of 3.4 slower than MPI. *BUPC-reduction* performs comparably to MPI on up to 32 CPUs and it is 14% slower on 64 CPUs. The *CAF-barrier* version experiences a slowdown of 19% relative to CAF. Notice also that while the performance degradation due to the use of barrier-only synchronization is smaller on the SGI Altix 3000 than on the Itanium2+Myrinet 2000 cluster, it prevents achieving high-performance on large number of CPUs on both architectures.

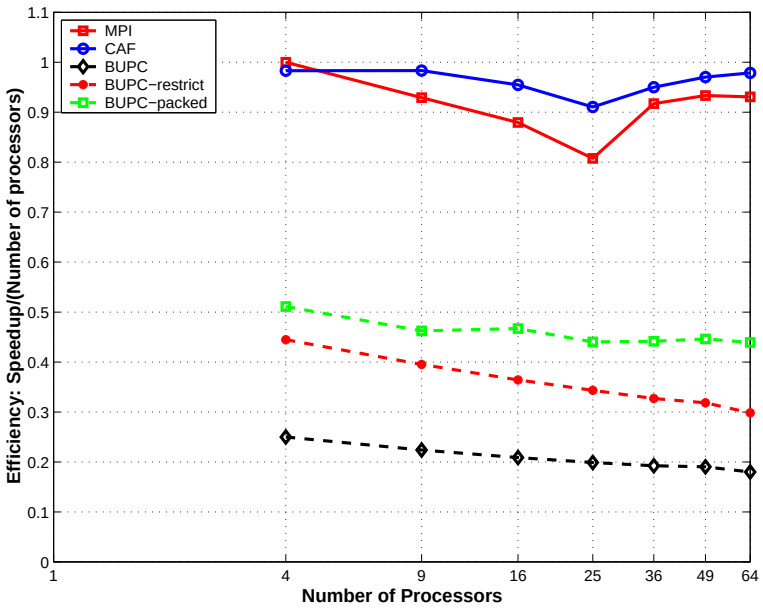


Figure 7.7: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class C, on an Itanium2+Myrinet architecture.

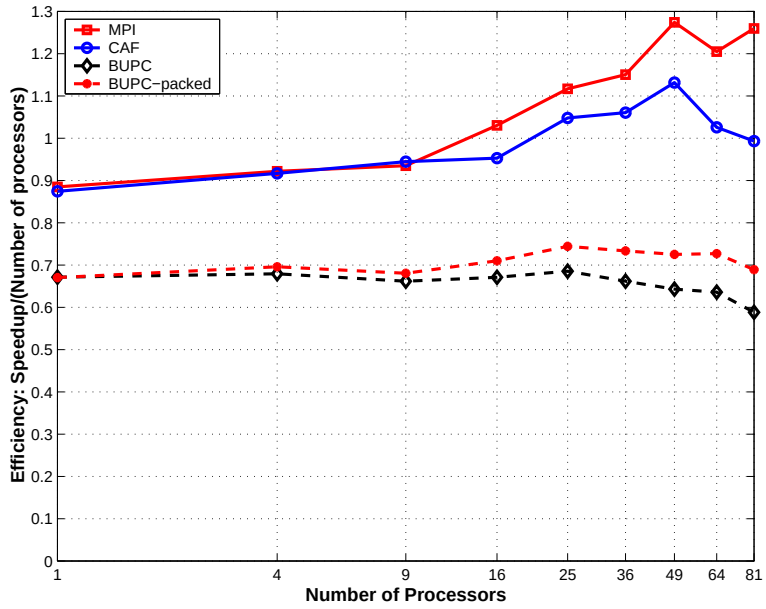


Figure 7.8: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class B, on an Alpha+Quadrics architecture.

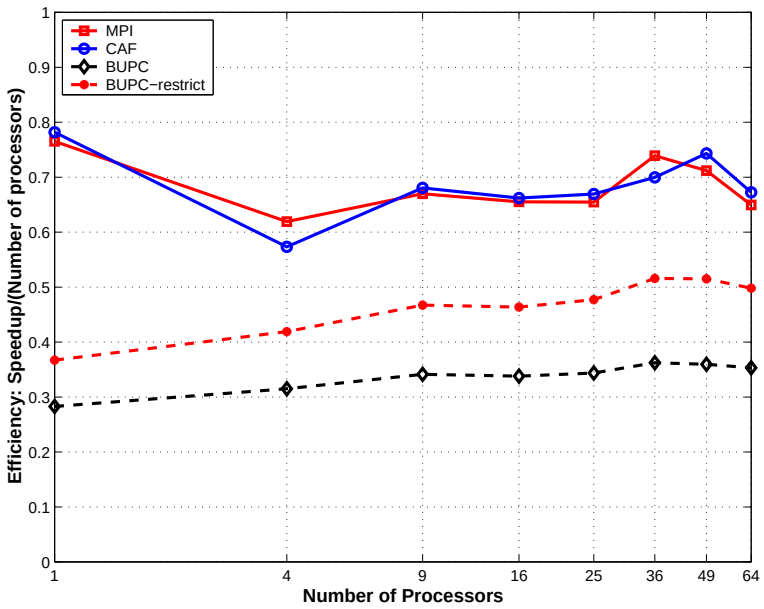


Figure 7.9: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class B on an SGI Altix 3000 shared memory architecture.

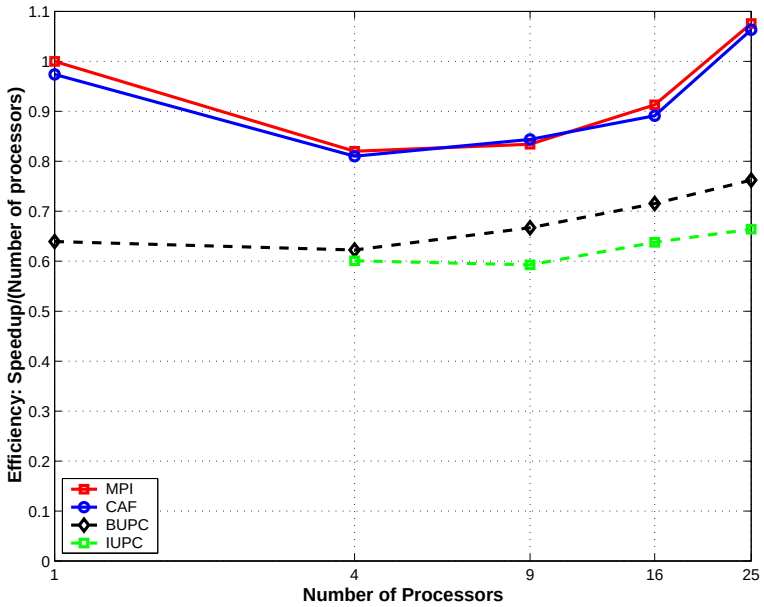


Figure 7.10: Comparison of MPI, CAF and UPC parallel efficiency for NAS BT class A on an SGI Origin 2000 shared memory architecture.

The parallel efficiency of NAS CG class B (problem size 75000) on the SGI Origin 2000 is shown in Figure 7.4(b). We used the same MPI and CAF versions as for the previous three platforms. We used the Berkeley UPC and the Intrepid UPC compilers to build the UPC codes; the corresponding versions are *BUPC* and *IUPC*. On this platform, MPI, CAF and *BUPC* have comparable performance across the range of CPUs. In each case, the MIPSPro compilers were able to optimize the sparse matrix-vector product loop automatically and effectively; consequently, using the partial sums version didn't boost performance. We also didn't notice a performance difference between CAF and *CAF-barrier*. The *IUPC* version is up to 50% slower than the other three versions. The principal loss of performance stems from ineffective optimization of the sparse-matrix vector product computation. *IUPC-reduction* represents an IUPC-compiled version of UPC CG with the sparse matrix-vector product loop unrolled; this version is only 12% slower than MPI.

7.5 NAS BT

In Figures 7.6 and 7.7, we present parallel efficiency results of NAS BT classes A (problem size 64^3) and C (problem size 162^3) on an Itanium2+Myrinet 2000 cluster. We used the NPB-2.3 MPI version, MPI, the most efficient CAF version, CAF, a UPC implementation similar to MPI and compiled with the Berkeley UPC compiler, *BUPC*, and two optimized UPC versions, *BUPC-restrict* and *BUPC-packed*. Due to memory constraints, we couldn't run the sequential Fortran version of BT for class C; to compute parallel efficiency we assume that the efficiency of MPI on four CPUs is one, and compute the rest of the efficiencies relative to that baseline performance.

The CAF implementation of BT is described in more detail in chapter 6. It uses communication vectorization, a trade-off between communication buffer space and amount of necessary synchronization, procedure splitting and non-blocking communication. It also uses the packing of strided `PUTS`, due to inefficient multi-platform support of strided `PUTS` by the CAF runtime. The initial UPC version was also derived from the MPI version.

The performance of the CAF version is better than or equal to that of MPI. The per-

formance of the initial UPC version, *BUPC*, was up to a factor of five slower than that of the MPI version. By using HPCToolkit, we determined that several routines that perform computation on the local part of shared data, namely `matmul_sub`, `matmul_vec`, `binvrhs`, `binvcrhs` and `compute_rhs`, are considerably slower in *BUPC* compared to the MPI version. To reduce overly conservative assumption about aliasing, we added the `restrict` keyword to the declarations of all the pointer arguments of the subroutines `matmul_sub`, `matmul_vec`, `binvrhs`, and `binvcrhs`. The modified UPC version of NAS BT is *BUPC-restrict*; it is up to 42% faster than *BUPC*.

To investigate the impact of communication performance on parallel efficiency, we instrumented all NAS BT versions to record the times spent in communication and synchronization. We found that *BUPC-restrict* spent about 50-100 times more in communication on the Itanium2+Myrinet 2000 cluster because the communication in the sweeps was not fully vectorized; it transfers a large number of messages of 25 double precision numbers. In chapter 6 we show that, in the absence of efficient runtime support for strided communication, packing for the CAF version of BT can improve performance by as much as 30% on cluster platforms.

We transformed the *BUPC-restrict* version to perform packing and unpacking and used the UPC `upc_memget` primitive to communicate the packed data; the resulting version with packed communication is denoted *BUPC-packed*. This version is up to 32% faster than *BUPC-restrict*. Overall, *BUPC-packed* yields a factor of 2.44 improvement over *BUPC*.

In Figure 7.8 we present the results for NAS BT class B[¶](problem size 102^3) on an Alpha+Quadrics cluster. The MPI version yields the best performance; CAF is up to 26% slower than MPI, and *BUPC* is up to two times slower than MPI. On the Alpha+Quadrics cluster, using the `restrict` keyword didn't have an effect; consequently, *BUPC* and *BUPC-restrict* have similar performance. This shows that even though the back-end C

[¶]We used class B due to limitations encountered for class C for the CAF and *BUPC* versions. CAF could not allocate the large data size required for BT class C on small number of processors, while *BUPC* could not allocate memory for a number of threads larger than 100.

compiler can optimize routines such as `matmul_sub`, `matmul_vec`, `binvrhs`, and `binvcrhs`, which contain at most one loop or just straight-line code, it has difficulties optimizing `compute_rhs`. This subroutine contains several complex loop nests and performs references to the local parts of multiple shared arrays using private pointers; this poses a challenge to the back-end C compiler. In the CAF version, `compute_rhs` performs the same computations on local parts of co-arrays; to convey the lack of aliasing to the back-end Fortran compiler we use procedure splitting. Packing of communication led to a performance gain: *BUPC-packed* is up to 14% faster than *BUPC*, although it is still up to 82% faster than MPI.

In Figure 7.9 we present the results for NAS BT class B (problem size 102^3) on an SGI Altix 3000 platform. We studied class B, due to memory and time constraints on the machine. The MPI and CAF versions have similar performance, while *BUPC* is up to two times slower than MPI. *BUPC-restrict* is up to 30% faster than *BUPC* and up to 43% slower than MPI. *BUPC-packed* has the same performance as *BUPC-restrict*. Packing didn't improve the performance because fine-grain data transfers are efficiently supported in hardware.

Finally, in Figure 7.10 we present results on the SGI Origin 2000 machine. We studied class A (problem size 64^3) of NAS BT due to memory and time constraints. The CAF and MPI versions perform comparably, while *BUPC* performs 40% slower than the MPI version. Similar to our experiences with the other benchmarks, using `restrict` didn't improve the performance of *BUPC-restrict*, and similar to the SGI Altix 3000, communication packing didn't improve the performance of *BUPC-packed*.

Part I

I

I

Chapter 8

Analyzing the Effectiveness of CAF Optimizations

An application compiled by an optimizing compiler usually undergoes several transformations and optimizations with the goal of increasing the application's performance. It is often desired to quantify how much each optimization contributes to performance; it is also important to understand how optimizations interact with each other, e.g., one optimization might be an enabling transformation for another optimization, or might inhibit it. Due to the complex nature of the transformations, one needs to have a rigorous methodology to estimate these effects. Such a methodology is the the $2^k r$ factorial design [123]. In this chapter we will use the $2^k r$ full factorial design with r replications to assess the impact of compiler optimizations and their interactions on application performance.

In previous chapters, we identified several important source-to-source code transformations to increase the performance of parallel CAF codes. Understanding how transformations affect performance helps to prioritize their implementation. For this study, we selected the LBMHD [157] application, described in Section 3.2.2, coded several Co-Array Fortran versions, and analyzed it using the $2^k r$ experimental design methodology. Since our ultimate goal is to achieve portable and scalable high performance, we also present a comparison of the best-performing CAF version of LBMHD with its MPI counterpart.

In section 8.1 we present an overview of the $2^k r$ experimental design methodology and we describe a CAF implementation of LBMHD in section 3.2.2. We describe our experimental approach in section 8.3, and present our results and analysis in section 8.4. Finally, we discuss our results in section 8.5.

8.1 $2^k r$ Experimental Design Methodology

The $2^k r$ experimental design is used to determine the effect of k factors, each of which has two levels, and r replications are used to estimate the experimental error. The design consists of determining factors and the model, constructing the corresponding sign table, collecting experimental data, finally, determining the model coefficients and their confidence intervals as well as running visual tests to verify the model assumptions. The interesting factors and interactions should be statistically significant (the confidence interval does not include zero), and practically significant (the percentage of variation explained is larger than 0.05% according to Jain). The details of the $2^k r$ experimental design can be found in Jain's performance analysis book [123] chapters 17-18.

8.2 Writing LBMHD in CAF

We obtained both MPI and CAF versions of LBMHD from Jonathan Carter from Lawrence Berkeley National Laboratory. The original CAF version of LBMHD was developed for the Cray X1 architecture. It uses allocatable co-arrays and partially vectorized remote co-array reads (GETs) to communicate data between processors. We converted remote co-array reads into remote co-array assignments (PUTs) to enable the use of non-blocking communication hints. For the problem sizes of 1024^2 and 2048^2, which we used in our experiments, communication is a significant portion of program execution time. Thus, we tested transformations that optimize communication, in particular, communication vectorization, communication packing and aggregation, synchronization strength reduction, and use of non-blocking communication hints. The LBMHD code does not offer opportunities to evaluate the procedure splitting optimization because no computation is performed using local co-array data.

Symbol	Factor	Level -1	Level +1
A	Comm. vectorization	unvectorized comm	vectorized comm
B	Sync. strength-reduction	group sync	point-to-point sync
C	Comm. packing	unpacked comm	packed comm
D	Non-blocking comm.	blocking comm	non-blocking comm
E	Architecture type	cluster	smp
F	Number of CPUs	4	64
G	Problem size	1024^2	2048^2

Table 8.1: Factors and levels for the CAF implementations of LBMHD.

8.3 Experimental Design

Before using the $2^k r$ experimental design methodology, we had to carefully choose the relevant factors. We looked at a total of seven factors, $A - G$, out of which four represent the presence or absence of optimizations, while the remaining three include problem size, number of CPUs and architecture type; the meaning attached to each factor levels is described in Table 8.1.

We analyzed our data with both additive and multiplicative models. For the additive model, the model equations for a $2^4 r$ experiment (for a particular choice of platform, number of CPUs and problem size) is

$$\begin{aligned} y &= q_0 + q_A x_A + q_B x_B + q_C x_C + q_D x_D + q_{AB} x_{AB} + q_{AC} x_{AC} + q_{AD} x_{AD} + \\ & \quad q_{BC} x_{BC} + q_{BD} x_{BD} + q_{CD} x_{CD} + q_{ABC} x_{ABC} + q_{ABD} x_{ABD} + q_{ACD} x_{ACD} \end{aligned}$$

For the multiplicative model, the model equation is

$$\begin{aligned} y &= 10_0^q 10^{q_A x_A} 10^{q_B x_B} 10^{q_C x_C} 10^{q_D x_D} 10^{q_{AB} x_{AB}} 10^{q_{AC} x_{AC}} 10^{q_{AD} x_{AD}} \\ & \quad 10^{q_{BC} x_{BC}} 10^{q_{BD} x_{BD}} 10^{q_{CD} x_{CD}} 10^{q_{ABC} x_{ABC}} 10^{q_{ABD} x_{ABD}} \\ & \quad 10^{q_{ACD} x_{ACD}} 10^{q_{BCD} x_{BCD}} 10^{q_{ABCD} x_{ABCD}} \end{aligned}$$

For the 2^4k experiment with the factors A, B, C, and D, we hand-coded 16 versions of the LBMHD benchmark. The versions were termed `mhd-caf-xyzw`, where x, y, z, and w have the following meaning:

$$x = \begin{cases} 0 & x_A = -1 \\ 1 & x_A = +1 \end{cases} \qquad y = \begin{cases} 0 & x_B = -1 \\ 1 & x_B = +1 \end{cases}$$

$$z = \begin{cases} 0 & x_C = -1 \\ 1 & x_C = +1 \end{cases} \qquad w = \begin{cases} 0 & x_D = -1 \\ 1 & x_D = +1 \end{cases}$$

When implementing a version with synchronization strength reduction, `mhd-x1zw`, the communication is essentially the same as in the version `mhd-x0zw`, but the synchronization primitives are interspersed with the communication code; a `sync_notify` to an image P is issued as soon as the communication events to P have been issued.

When implementing a version that employs communication packing, `mhd-xy1w`, communication to a image P is issued as soon as packing for that image is ready; taking this one step further, we have reordered the packing and the communication steps such that a image packs and communicates all necessary data for one neighbor at a time. Correspondingly, on the destination side, a image waits for a notification, then unpacks the data, for one source at a time. When using the non-blocking communication, this provides more opportunities to overlap communication with packing and unpacking. It is important to mention that communication packing superseeds communication vectorization; for this reason, a version `mhd-caf-1y1w` is identical to the version `mhd-caf-0y1w`.

To perform experiments considering any of the remaining factors, E, F, or G, one simply changes the submission parameters such as problem size of number or CPUs, or the target machine.

We performed two sets of experiments: 2^4r and 2^5r, measuring the running time as the

Factor	Effect	% of Variation	Confidence Interval	Stat. Imp.
I	-1.0167	0.0000	(-1.0428 , -0.99)	x
A	0.0029	0.0061	(-0.0233 , 0.03)	
B	-0.0255	0.4840	(-0.0516 , 0)	
C	-0.3324	82.3032	(-0.3585 , -0.31)	x
D	-0.0185	0.2556	(-0.0446 , 0.01)	
AB	-0.0109	0.0885	(-0.0370 , 0.02)	
AC	-0.0029	0.0061	(-0.0290 , 0.02)	
AD	0.0024	0.0042	(-0.0237 , 0.03)	
BC	-0.0429	1.3688	(-0.0690 , -0.02)	x
BD	0.0078	0.0450	(-0.0183 , 0.03)	
CD	-0.0186	0.2590	(-0.0448 , 0.01)	
ABC	0.0109	0.0885	(-0.0152 , 0.04)	
ABD	-0.0043	0.0138	(-0.0304 , 0.02)	
ACD	-0.0024	0.0042	(-0.0285 , 0.02)	
BCD	0.0225	0.3778	(-0.0036 , 0.05)	
ABCD	0.0043	0.0138	(-0.0218 , 0.03)	

Table 8.2: Effects and variation explained for LBMHD, for problem size 1024^2 and 64 CPUs, on the SGI Altix 3000 platform.

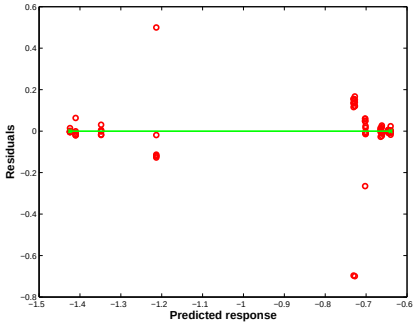
response variable. The first set analyzes four factors (A-D) under either a multiplicative or an additive model. The second set is an attempt to add a fifth factor – the architecture type. The fifth factor compares a cluster-based architecture (Itanium2+Quadrics) and a hardware shared-memory architecture (Altix 3000). Because the runtime difference between equivalent runs on different architectures is significantly large than runtime variation due to optimizations on either of the platforms, we normalize the time of each run by dividing it by the average time among all runs on the corresponding architecture. While this might in-

Factor	Effect	% of Variation	Confidence Interval	Stat. Imp.
I	3.5580	0.0000	(3.5572 , 3.5587)	x
A	-0.0069	0.4200	(-0.0076 , -0.0061)	x
B	-0.0213	4.0500	(-0.0221 , -0.0206)	x
C	-0.1030	94.7400	(-0.1038 , -0.1023)	x
D	0.0015	0.0200	(0.0008 , 0.0023)	x
AB	0.0025	0.0600	(0.0018 , 0.0033)	x
AC	0.0069	0.4200	(0.0061 , 0.0076)	x
AD	0.0010	0.0100	(0.0003 , 0.0018)	x
BC	0.0018	0.0300	(0.0011 , 0.0026)	x
BD	0.0017	0.0300	(0.0010 , 0.0025)	x
CD	-0.0017	0.0300	(-0.0024 , -0.0009)	x
ABC	-0.0025	0.0600	(-0.0033 , -0.0018)	x
ABD	0.0007	0.0000	(-0.0001 , 0.0014)	
ACD	-0.0010	0.0100	(-0.0018 , -0.0003)	x
BCD	-0.0010	0.0100	(-0.0018 , -0.0003)	x
ABCD	-0.0007	0.0000	(-0.0014 , 0.0001)	

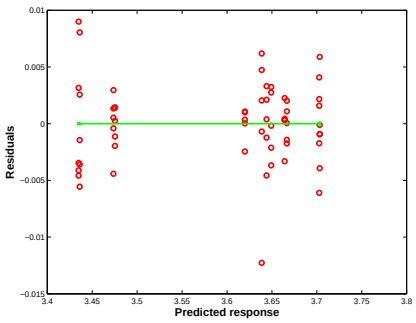
Table 8.3: Effects and variation explained for LBMHD, for problem size 2048^2 and 64 CPUs, on the SGI Altix 3000 platform.

troduce some inaccuracy into the analysis, without it, the architecture type factor dominates the analysis making other factor and interaction contributions essentially irrelevant.

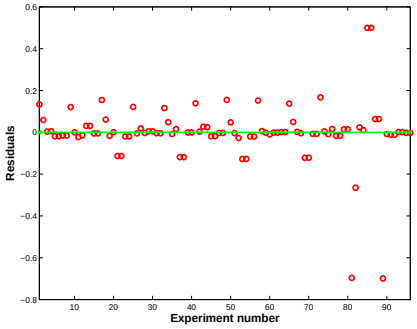
We have also tried to have the problem size as a factor. However, the problem size dominates all other factors and interactions, making the analysis not interesting. Similarly, if we use the number of CPUs as a factor (e.g., 4 and 64), it dominates the analysis. It might be possible to successfully use this factor for weak scaling experiments, in which one expects the running time not to depend so much on the number of CPUs as it does for



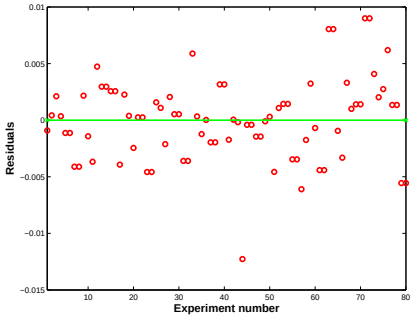
(a) Residuals vs predicted value, 1024^2



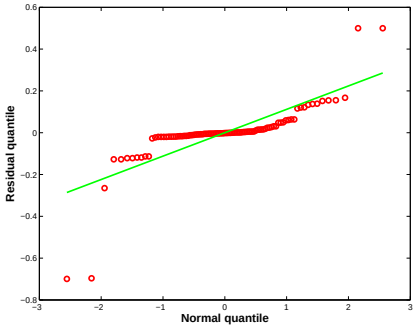
(b) Residuals vs predicted value, 2048^2



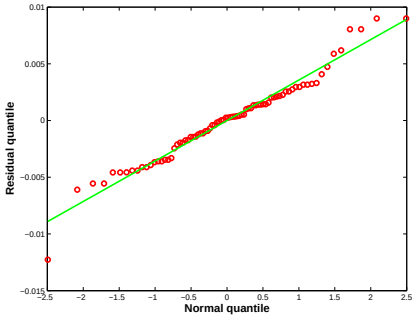
(c) Residuals vs exp. number, 1024^2



(d) Residuals vs exp. number, 2048^2



(e) Resid. quant.-quant. plot, 1024^2



(f) Resid. quant.-quant. plot, 2048^2

Figure 8.1: Visual tests for problem sizes 1024^2 and 2048^2 , 64 CPUs, on the SGI Altix 3000.

strong scaling experiments.

Our final goal is to achieve high parallel performance. Since the gold standard of parallel programming is still MPI, it is usual for the performance parallel languages benchmarks to be compared against that of their MPI counterparts. We compare the best-performing

CAF version of LBMHD with the equivalent MPI version over a large span of CPU numbers.

8.4 Experimental Results

We evaluated the impact and interactions of CAF optimizations on three platforms.

The first platform used was a cluster of 2000 HP Long's Peak dual-CPU workstations at the Pacific Northwest National Laboratory. The nodes are connected with Quadrics QSNet II (Elan 4). Each node contains two 1.5GHz Itanium2 processors with 32KB/256KB/6MB L1/L2/L3 cache and 4GB of RAM. The operating system is Red Hat Linux (kernel version 2.4.20). The back-end compiler is the Intel Fortran compiler version 8.0.

The second platform is an SGI Altix 3000, with 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128 GB RAM, running the Linux64 OS with the 2.4.21 kernel and the Intel Fortran compiler version 8.0.

The third platform we used for experiments was a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB/256KB/1.5MB of L1/L2/L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel Fortran compiler version 8.0 for Itanium as our Fortran 90 back-end compiler.

On the SGI Altix 3000 system, we performed $2^4 r$ full-factorial experiments for sizes 1024^2 and 2048^2, on 4, 16, and 64 CPUs. We performed experiments for both the additive and the multiplicative model; the percentage of variation explained by the major factors are similar, and the visual tests are similar for both models. We will present the results for the multiplicative model for the problem sizes 1024^2 and 2048^2, on 64 CPUs.

In Tables 8.2 and 8.3 we present the coefficient for the multiplicative model, the percentage of variation explained by each factor and the confidence intervals for each factor for problem sizes of 1024^2 and 2048^2. For a problem size of 1024^2, the factors that explain the largest percentage of variation and are statistically significant at the 90% confidence

level are C, the communication packing optimization, which explains 82% of variation, followed by BC, the interaction between synchronization strength-reduction and communication packing. Statistically insignificant factors are A, B, D, AB, AC, AD, BD, CD, ABC, ABD, ACD, BCD and $ABCD$. The results are surprising, showing that only one factor and one interaction are simultaneously practically significant and statistically significant. Overall, the chosen factors and their interactions explain 85% of total variation. For the problem size of 2048^2, the major factors and interactions are C, communication packing, B, synchronization strength-reduction, A, communication vectorization, AC, AB, and ABC. The factors D, AD, BC, CD, ACD, and BCD are practically insignificant (their percentage of variation explained is less than 0.05). The only statistically insignificant interactions are ABD and $ABCD$.

In Figure 8.1 we present the visual tests recommended by Jain. The visual tests don't show any trend of residuals vs the predicted value or the experiment number; the quantile-quantile plots of the residuals are reasonably linear.

The factors that explain the largest percentage of variation and are statistically significant at the 90% confidence level are C, the communication packing optimization, which explains 82% of variation, followed by BC, the interaction between synchronization strength-reduction and communication packing. Statistically insignificant factors are A, B, D, AB, AC, AD, BD, CD, ABC, ABD, ACD, BCD and $ABCD$. The results are surprising, showing that only one factor and one interaction are simultaneously practically significant and statistically significant. Overall, the chosen factors and their interactions explain 85% of total variation.

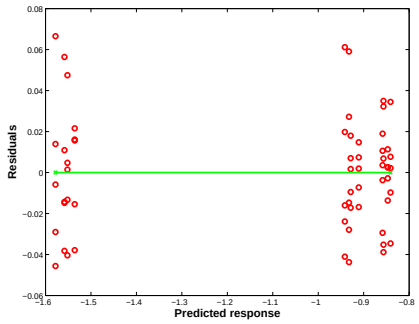
In Table 8.4 we present the percentage of variation explained by the practically and statistically significant factors for LBMHD, for problem sizes 1024^2 and 2048^2 on 4, 16 and 64 CPUs. The dominant factor is communication packing, explaining 82-99% of variation. Synchronization strength-reduction explains 4% of variation for problem size 2048^2 on 64 CPUs, but is statistically insignificant for problem size 1024^2, contrary to our expectations; we explain this by the fact that the 1024^2 and 2048^2 problem size experiments

Factor	1024^2			2048^2		
	4	16	64	4	16	64
I	0	0	0	0	0	0
A	1.482	0.898	0.334	0.329	1.581	0.832
B	0.303	0.059	0.050	0.497	1.109	0.270
C	95.701	97.316	98.488	92.349	93.368	91.969
D		0.091			0.231	
AC	1.482	0.898	0.334	0.329	1.581	0.832
AD				0.510		
BC	0.240			0.108	0.127	
BD	0.113	0.118		0.528	0.883	
CD	0.181	0.147		3.114	0.667	0.975
ACD				0.510		
BCD	0.063			0.868	0.194	

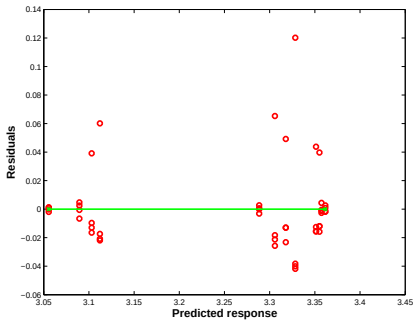
Table 8.4: Practically significant factors at 90% confidence for LBMHD, for problem sizes 1024^2 and 2048^2 and for 4, 16, and 64 CPUs, on the Itanium2+Quadrics platform.

were performed on different CPU sets and under different system loads. Communication vectorization, A, and the interaction AC explain up to 2.27% for four CPUs, and less for a larger number of CPUs; this shows that as we increase the number of CPUs, packing is more important for achieving high-performance. Finally, non-blocking communication has a insignificant impact on performance; this is expected since the SGI Altix 3000 system doesn't provide hardware support for non-blocking communication.

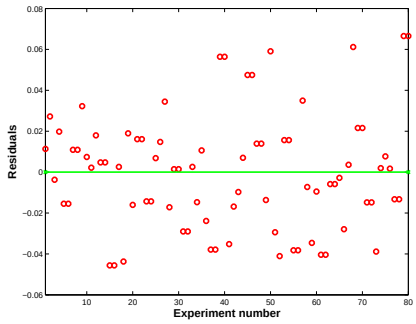
Similarly to the SGI Altix platform, tables 8.5 and 8.6 present factor and interaction coefficients, percentage of variation explained by them and their confidence intervals at the 90% significance level under a multiplicative model for 1024^2 and 2048^2 problem sizes on an Itanium2 cluster with a Quadrics interconnect (mpp2 PNNL cluster). For the 1024^2



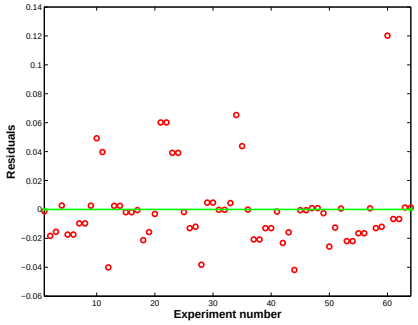
(a) Residuals vs predicted value, 1024^2



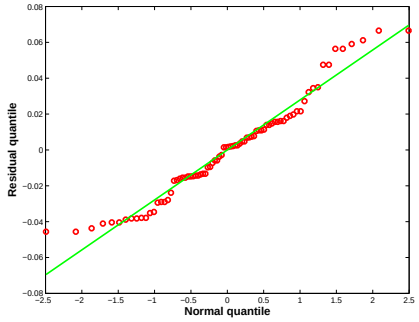
(b) Residuals vs predicted value, 2048^2



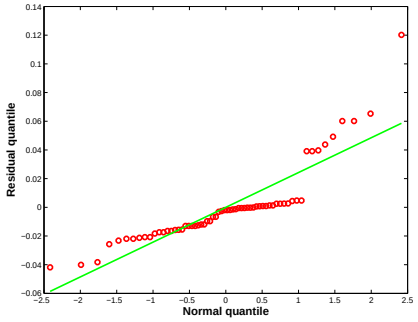
(c) Residuals vs exp. number, 1024^2



(d) Residuals vs exp. number, 2048^2



(e) Resid. quant.-quant. plot, 1024^2



(f) Resid. quan.-quant. plot, 2048^2

Figure 8.2: Visual tests for problem sizes 1024^2 and 2048^2, 64 CPUs, on the Itanium2+Quadrics architecture.

problem size, the most significant factor is C, communication packing and aggregation, explains 98.5% of variation. Other significant factors are A, B, D, AC, and CD. For 2048^2 problem size, again communication packing is the most significant factor explaining 92% of variation; other significant factors are A, AC, and CD.

Figure 8.2 present Jain-recommended visual tests to verify the model. The residuals seem not to depend on the predicted response and experiment number. The quantile-quantile plots are reasonable close to linear indicating distributions of residuals close to the normal distribution.

Table 8.7 presents statistically and practically significant factors and interactions for 1024^2 and 2048^2 problem sizes on 4, 16 and 64 CPUs. The major factor is communication packing and aggregation (C) for all experiment configurations. To our surprise, the contribution of the communication vectorization factor was barely noticeable (0.3-1.5%) indicating that there exists an inefficiency in the ARMCI for strided transfers.

We performed 2^5r experiments on all three platforms, choosing the factors A, B, C, D, and adding F, the number of CPUs; the response was the total execution time. The percentage of variation explained by the number of CPUs is very high: 96-99% on the Itanium2+Quadrics cluster, 96-99% on the SGI Altix 3000 system, and 99.6% on the Itanium2+Myrinet 2000 cluster. We noticed similar results when using the parallel efficiency as response variable. This results are due to the fact that LBMHD exhibits strong scaling (i.e. the problem size is the same for an increasing number of CPUs). The conclusion is that we cannot use the number of CPUs as a factor, because it would completely dominate the remaining factors.

Figures 8.3, 8.4, and 8.5 present the parallel efficiency for MPI and fastest CAF versions over a large range of CPUs. The plots show that on the SGI Altix 3000 and Itanium2+Quadrics platforms the CAF version significantly outperforms the MPI version. MPI outperforms CAF for the 2048^2 size on the Itanium2+Myrinet cluster, while for the 1024^2 the MPI and the CAF version achieve comparable performance.

Table 8.8 presents statistically and practically significant factors and interactions in a 2^5r cross-platform experimental design. The fifth factor, E, stands for the architecture type: cluster (mpp2) or hardware shared memory (Altix). The running times were normalized as explained in 8.3 to accomodate for differences in serial performance due to different host CPUs and memory controllers. While the normalization might introduce errors into

Factor	Effect	% of Var.	Confidence Interval	Stat. Imp.
I	-1.223	0.000	(-1.23 , -1.22)	x
A	-0.019	0.334	(-0.03 , -0.01)	x
B	-0.008	0.050	(-0.01 , 0.00)	x
C	-0.333	98.488	(-0.34 , -0.33)	x
D	-0.002	0.003	(-0.01 , 0.00)	
AB	-0.002	0.003	(-0.01 , 0.00)	
AC	0.019	0.334	(0.01 , 0.03)	x
AD	0.002	0.002	(0.00 , 0.01)	
BC	-0.005	0.020	(-0.01 , 0.00)	
BD	0.001	0.000	(-0.01 , 0.01)	
CD	-0.007	0.045	(-0.01 , 0.00)	x
ABC	0.002	0.003	(0.00 , 0.01)	
ABD	-0.002	0.004	(-0.01 , 0.00)	
ACD	-0.002	0.002	(-0.01 , 0.00)	
BCD	-0.002	0.002	(-0.01 , 0.00)	
ABCD	0.002	0.004	(0.00 , 0.01)	

Table 8.5: Effects and variation explained for LBMHD (size 1024^2, 64 CPUs) on the Itanium2+Quadrics platform.

the model, for 1024^2 problem size the total percentage of explained variation is 99.6%; however, it is only 59.12% for 2048^2 problem size. The most dominant factor is again communication packing and aggregation. The architecture type factor is also significant: 8.3% for 1024^2 problem size and 5.1% for 2048^2 problem size.

Factor	Effect	% of Var.	Confidence Interval	Stat. Imp.
I	3.212	0.000	(3.21 , 3.22)	x
A	-0.012	0.832	(-0.02 , -0.01)	x
B	-0.007	0.270	(-0.01 , 0.00)	
C	-0.122	91.969	(-0.13 , -0.12)	x
D	-0.005	0.157	(-0.01 , 0.00)	
AB	0.000	0.001	(-0.01 , 0.01)	
AC	0.012	0.832	(0.01 , 0.02)	x
AD	0.003	0.047	(0.00 , 0.01)	
BC	-0.004	0.108	(-0.01 , 0.00)	
BD	-0.001	0.012	(-0.01 , 0.01)	
CD	-0.013	0.975	(-0.02 , -0.01)	x
ABC	0.000	0.001	(-0.01 , 0.01)	
ABD	0.002	0.020	(0.00 , 0.01)	
ACD	-0.003	0.047	(-0.01 , 0.00)	
BCD	-0.005	0.144	(-0.01 , 0.00)	
ABCD	-0.002	0.020	(-0.01 , 0.00)	

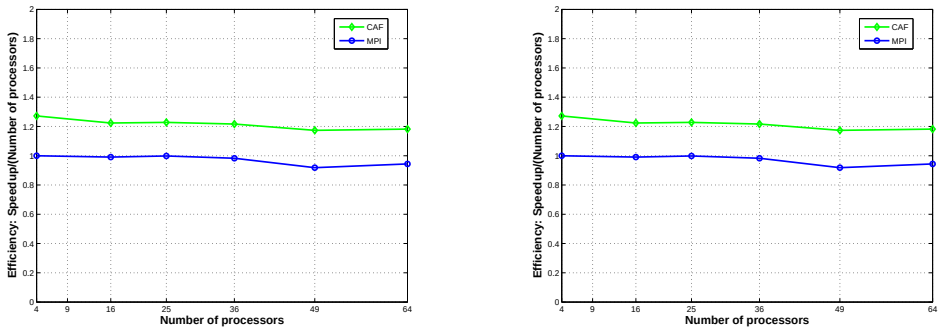
Table 8.6: Effects and variation explained for LBMHD (size 2048^2, 64 CPUs) on the Itanium2+Quadrics platform.

8.5 Discussion

Our 2^5r experiments showed that communication packing and aggregation is a crucial transformation for achieving high performance over multiple architecture types. After using the 2^kr experimental design methodology to analyze the impact and interactins of CAF versions of LBMHD, we think that this methodology has only a limited applicability. It is of most use when prioritizing the implementation of such optimizations in a compiler; one can implement first the most important optimizations, followed by optimizations which

Factor	% of Variation for 1024^2			% of Variation for 2048^2		
	4	16	64	4	16	64
A	2.271	0.092		1.174	0.640	0.420
B	0.820	0.056			0.234	4.050
C	93.021	99.363	82.303	83.588	98.032	94.740
AB						0.060
AC	2.271	0.092		1.174	0.640	0.420
BC			1.369			
BD	0.187					
ABC						0.060

Table 8.7: Practically and statistically significant factors for LBMHD, for problem sizes 1024^2 and 2048^2 and for 4, 16, and 64 CPUs, on an SGI Altix 3000.



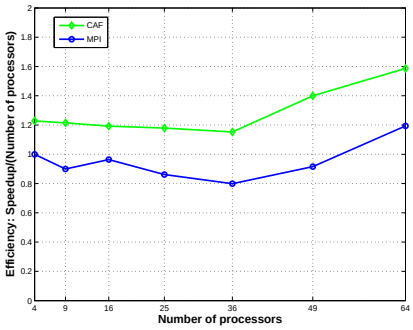
(a) Problem size 1024^2 (b) Problem size 2048^2

Figure 8.3: Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an SGI Altix 3000 system.

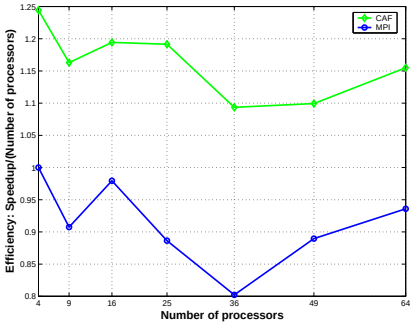
are part of important interactions. However, the methodology might be too coarse, considering that a certain optimization might be implemented in multiple ways; for example, communication packing also required a careful reordering of packing, communication and synchronization events.

Factor	% of Var. (1024^2)	% of Var. (2048^2)
I	0.0000	0.0000
A	0.1698	
B	0.3395	
C	90.0816	51.3322
D	0.0137	
E	8.3072	5.1890
AC	0.1698	
AE	0.0130	
BC	0.0445	
BE	0.1436	
CD	0.0154	
CE	0.2114	
ACE	0.0130	
BCDE	0.0162	
Total	99.5809	59.1244

Table 8.8: Statistically significant effects and variation explained for LBMHD (64 CPUs) on the Itanium2+Quadrics and SGI Altix 3000 platforms for 1024^2 and 2048^2 problem sizes.

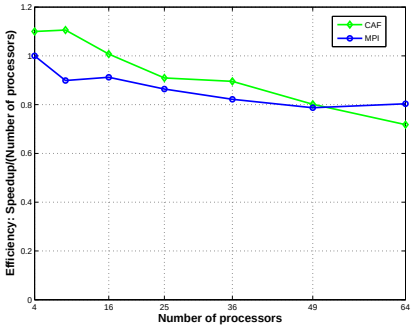


(a) Problem size 1024^2

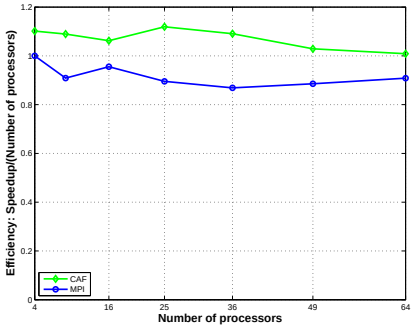


(b) Problem size 2048^2

Figure 8.4: Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an Itanium2+Quadrics system.



(a) Problem size 1024^2



(b) Problem size 2048^2

Figure 8.5: Parallel efficiency of LBMHD for problem sizes 1024^2 and 2048^2, on an Itanium2+Myrinet system.

Chapter 9

Space-efficient Synchronization Extensions to CAF

When crafting new language features, the performance-minded designer should consider whether the new features lead themselves to efficient implementations on multiple platforms. On the emerging petascale systems, both space and time must be considered as measures of efficiency. In Section 3.1 we presented the `sync_notify`/`sync_wait` synchronization extensions, that enabled us to move away from using costly barrier synchronization where lightweight point-to-point synchronization suffices. However, these primitives require $O(P^2)$ space for a P-processor parallel execution. In this chapter we propose *eventcounts* as an alternative, space-efficient synchronization mechanism, sketch an implementation using an Active Messages underlying layer, and explore how several classes of application would be written using this primitive.

9.1 Implementation of `sync_notify` and `sync_wait`

There are multiple possible implementations for `sync_notify`/`sync_wait` primitives; we will discuss several of them and point their shortcomings.

One implementation would be to queue up notifies on the remote processors, and to have each remote process image dequeue its notifies as it performs `sync_wait`s. The space requirement would be bounded by the total number of outstanding notifies. For well-written programs, we would expect the number of outstanding notifies to be reasonably small. However, misbehaving or incorrect programs might just issue notifies continuously and not consume them, depleting the memory resources. It would be desirable to have an implementation for which the space requirement would be bounded independent of the program behavior.

```
long sent[P];
long received[P];
long waited[P];
```

Figure 9.1: Current `cafc` data structure used for the implementation of the `sync_notify`/`sync_wait` primitives.

An alternative implementation would be to use a hash table of notify counters per process. The key in the hash table would be the image number of the sender, and the values cached would correspond to notify counts. This approach would leave to a space requirement proportional to the number of neighbors that an image communicates with over the program execution. A scenario for which this approach would be suboptimal is when an image communicates with a small group during some program phase, then with some other group in a different phase; the hash table size would keep increasing, even the space requirements for synchronization would not.

The current implementation of the `sync_notify` and `sync_wait` primitives in the `cafc` runtime uses an amount of space bounded at program launch. Three arrays are used, as shown in Figure 9.1

The location `sent[p]` stores the number of notifies *sent* to processor p; `received[p]` stores the number of notifies *received* by the current process image from p, while `waited[p]` stores the number of notifies *expected* by the current processor from image p. Upon the execution of a `sync_notify(p)` by processor q, the `cafc` runtime enforces the completion of all outstanding requests to processor p, after which it increments `sent[p]` on q and then copies its contents into `received[q]` on processor p. Upon the execution of a `sync_wait(q)` by processor p, the executing process image increments `waited[q]`, then spin waits until `received[q]` exceeds `waited[q]`.

While this implementation of `sync_notify` and `sync_wait` enables us to overcome the performance limitations of barrier-only synchronization, it has two significant drawbacks.

1. the *space cost* on P process images is $O(P^2)$; when using systems such as Blue Gene/L, with as much as 131072 processors, the quadratic space cost might become problematic.

2. *composability*: a programmer attempting to overlap synchronization with local computation might issue a `sync_notify` in one routine and issue a `sync_wait` in a different routine, and would have to track the choreography of synchronization events interprocedurally. However, modern codes are highly modular, and composing various routines, each of which would do its own synchronization, might result in incorrect behavior of the program.

9.2 Eventcounts

To scale to petascale systems, it would be desirable to have *space-efficient*, *composable* synchronization primitives. A mechanism that caught our attention was that of *eventcounts and sequencers*, proposed by Reed and Kanodia in [167]. We proposes an adaptation of that mechanism for CAF, by providing the following eventcount interface:

- ```
  integer function allocate_eventcount(size)
    integer size
  ```

  This function is collective and has the effect of allocating a distributed eventcount; on a particular image the eventcount has `size` entries. The eventcount allocation routine returns a eventcount identifier, which can be further used to operate on the allocated eventcount. Our interface proposes eventcounts that are global objects, working on the group of all the images of a running CAF program. In [72], Dotsenko proposed an extension of CAF with co-spaces, which are groups with well-defined topologies and created with a hierarchical structure. Eventcounts can be extended from global objects to object to objects associated with co-spaces; an eventcount identifier will then be unique within its associated co-space. A graphical representation of an eventcount is given in Figure 9.2; we emphasize that the eventcounts don't need to have the same number of entries on each image.
  ```

- `subroutine reset_eventcount(evid)`
  `integer evid`

  This function is collective and resets the eventcount to 0 on all images and for all entries on each image. The initial allocation of eventcounts performs an implicit reset.

- `subroutine advance_eventcount(evid, proc, index, count)`
  `integer evid, proc, index, count`

  This primitive has the effect of advancing the eventcount `evid` on process image `proc`, entry `index` by `count`. Similar to a `sync_notify`, it also means that all communication events between the current process and processor p have completed upon completion of the advance primitive on p.

- `subroutine wait_eventcount(evid, index, count)`
  `integer evid, index, count`

  This primitive checks if the local entry `index` on the eventcount `evid` on the current process image has advanced by `count` from the last wait primitive; if the condition is not met, the current processor's execution is suspended until the eventcount has advanced the required number of units.

- `logical function test_eventcount(evid, index, count)`
  `integer evid, index, count`

  This primitive checks if the local entry `index` on the eventcount `evid` on the current process image has advanced by `count` from the last wait primitive; if the condition is met, the primitive returns `true`, otherwise it returns `false`.

- `subroutine release_eventcount(evid)`
  `integer evid`

  This primitive frees the resources used by the eventcount `evid`.

Operations specified using an invalid `evid` are incorrect and might trigger exceptions. Eventcounts identifier can be passed as procedure arguments, enabling overlap of synchronization with computation. Since eventcount are allocated on demand, different solvers

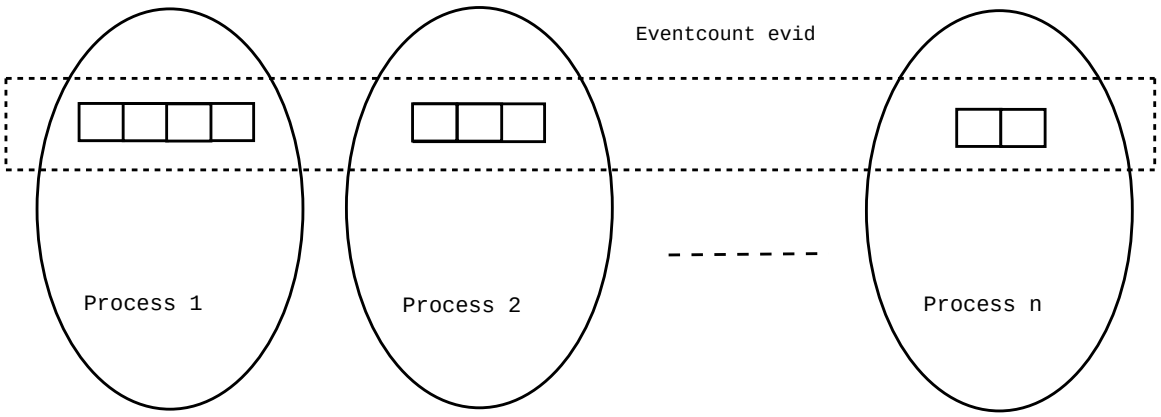


Figure 9.2: Graphical representation of an eventcount. Different process images can have different number of eventcount entries.

can get different eventcounts and operate independently on them; this in effect ensures composability with respect to synchronization of CAF routines using eventcounts as their synchronization mechanism.

9.3 Eventcounts Implementation Strategy

By providing access to eventcounts by means of an API, we can support them in the CAF runtime in a portable fashion as a CAF extension, without modifying the `cafc` front-end. A practical solution for eventcounts representation on each image is a hash tables of arrays. For each eventcount, we need to two arrays: one corresponding to the current values of the eventcounts, and one corresponding to the last value checked by a wait operation.

```
struct EventCount {
   integer eventCountId;
   long* received;
   long* waited;
}
```

On allocation, we could use a global eventcount counter which contains the next unused eventcount id value; a CAF runtime would increment it then use its value as the next

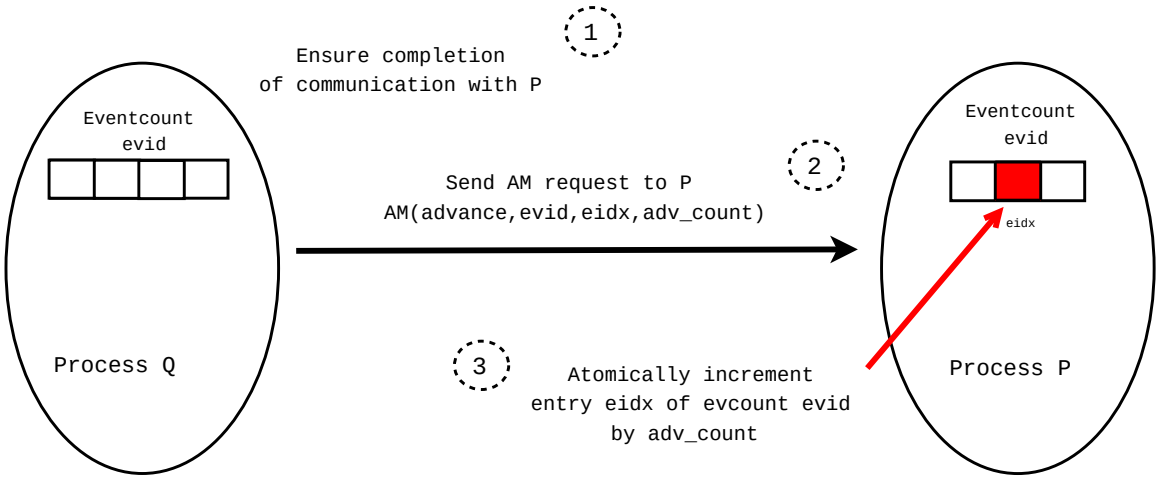


Figure 9.3: Steps taken in the execution of `advance_eventcount(evid, P, eidx, count)`.

eventcount id. Next, each image would allocate a EventCount structure with the required number of entries — the argument `size` given to `allocate_eventcount`, and would initialize the received and waited values to 0. A pointer to the structure would then be inserted into the hash table, using the eventcount id modulo a maximum hash table size as key. The `eventCountId` field should be added to the eventcount representation to resolve conflicts in the hash table.

In Figure 9.3 we present a strategy of implementing `advance_eventcount` using Active Messages (AM). GASNet provides a robust and portable support for active messages, while ARMCI has only fledgling support. The first step is to ensure that the communication events between the current processor q and p have completed. A simple, but inefficient way of achieving this is to force completion of outstanding `PUT` requests from q to p. The next step is to send an active message request for the AM handler `advance`, with the arguments `evid` – the eventcount id, `eidx` — the eventcount entry index, `count` — the amount by which the eventcount entry will be incremented. Once the AM handler gets scheduled for execution on p, it looks up in the hash table the entry corresponding to

the eventcount `evid`, and then atomically updates the entry `eidx`, using primitives such as fetch-and-add or load-link/store-conditional. For AM libraries which ensure atomicity at handle level by executing the AM handles until completion within the same tread, it is not required to use the atomic update primitives, and simple read/writes to the eventcount memory location suffice; GASNet is such an AM library.

On the execution of a `wait_eventcount`, the process p first updates the value for the `waited` array by adding the increment it is waiting for, then spinwaits as long as the `received` value for the entry of interest is strictly smaller than the waited value.

To execute the `reset_eventcount` primitive, each image looks up the eventcount entry in the event count hash table, after which it zeroes the `received` and the `waited` arrays. To deallocate an eventcount, each image looks up the eventcount entry in the event count hash table, after which it deallocates the `received` and `waited` arrays, followed by deallocating the eventcount entry.

9.4 Eventcounts in Action

In this section we will present examples of eventcount usage for synchronization in several common data exchange patterns.

9.4.1 Jacobi Solver

In Figure 9.4 we present the main loop of a Jacobi four point stencil solver, and in Figure 9.5 we present the same loop, written using eventcounts for synchronization. We need to signal the following facts: the remote overlap regions are available to be written, and the buffer writing from all four neighbors completed. We need to use an eventcount with five entries per image, one entry per neighbor to allow remote writing to the neighbor, and one entry to signal write completion from all four neighbors. Overall, the space requirement is $O(5P)$, compared to $O(P^2)$ for the `sync_notify` and `sync_wait` primitives.

```
do step = 1, nsteps
.... fill in remote overlap region for north neighbor ...
.... fill in remote overlap region for south neighbor ...
.... fill in remote overlap region for east neighbor ...
.... fill in remote overlap region for west neighbor ...
.... perform stencil computation ....
enddo
```

Figure 9.4: Four-point stencil Jacobi solver pseudocode.

```
evid = allocate_eventcount(5)
north_index = 1
south_index = 2
east_index = 3
west_index = 4

do step = 1,nstep
advance_eventcount(evid, north_processor, south_index, 1)
advance_eventcount(evid, south_processor, north_index, 1)
advance_eventcount(evid, west_processor, east_index, 1)
advance_eventcount(evid, east_processor, west_index, 1)

wait_eventcount(evid,  north_index, 1)
.... fill in remote overlap region for north neighbor ...
  advance_eventcount(evid, north_processor, 5, 1)

wait_eventcount(evid,  south_index, 1)
.... fill in remote overlap region for south neighbor ...
advance_eventcount(evid, south_processor, 5, 1)

wait_eventcount(evid,  east_index, 1)
.... fill in remote overlap region for east neighbor ...
advance_eventcount(evid, east_processor, 5, 1)

  wait_eventcount(evid,  west_index, 1)
.... fill in remote overlap region for west neighbor ...
advance_eventcount(evid, west_processor, 5, 1)

wait_eventcount(evid, 5, 4)
.... perform stencil computation ....
enddo
```

Figure 9.5: Four-point stencil Jacobi solver written using eventcounts.

9.4.2 Conjugate Gradient

In Section 6.2, we presented a CAF implementation of the NAS CG benchmark; we present a fragment of CG in Figure 9.6(a). Each processor needs to synchronize with $\lceil log(P) \rceil$ processors; this shows that we can implement the same synchronization using eventcounts, each process having $\lceil log(P) \rceil$ eventcount entries, which makes the overall space requirement $O(Plog(P))$ vs $O(P^2)$. In Figure 9.6(b) we present the same CAF NAS CG fragment

```
! notify our partner that we are here and wait for
! him to notify us that the data we need is ready
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! get data from our partner
  q(n1:n2) = w(m1:m1+n2-n1)[reduce_exch_proc(i)]
! synchronize again with our partner to
! indicate that we have completed our exchange
! so that we can safely modify our part of w
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! local computation
  ... use q, modify w ...
```

(a) `sync_notify`/`sync_wait` implementation

```
 evid = allocate_eventcount(ceil(log(num_images())))
! notify our partner that we are here and wait for
! him to notify us that the data we need is ready
  call advance_eventcount(evid, reduce_exch_proc(i)+1, i, 1)
  call wait_eventcount(evid,i,1)
! get data from our partner
  q(n1:n2) = w(m1:m1+n2-n1)[reduce_exch_proc(i)]
! synchronize again with our partner to
! indicate that we have completed our exchange
! so that we can safely modify our part of w
  call advance_eventcount(evid, reduce_exch_proc(i)+1, i, 1)
  call wait_eventcount(evid,i,1)
! local computation
  ... use q, modify w ...
```

(b) Eventcount-based implementation

Figure 9.6: A typical fragment of optimized CAF for NAS CG.

as in Figure 9.6(a) implemented using eventcounts:

9.4.3 An ADI Solver

In Section 6.3, we presented an optimized CAF implementation of NAS SP; in Figure 9.7 we show the communication, synchronization and computation structure for the `x_solve` routine, using `sync_notify`/`sync_wait` primitives. Since each process image synchronizes with only two neighbors in both the forward and the backward sweep phase, we can use an eventcount with two entries for each of the `x_solve`, `y_solve` and `z_solve` routines; the first eventcount entry will be used to signal that the remote buffer is available to be written, and the second eventcount entry will be advanced to indicate the completion of

```
! forward substitution
do  stage = 1, ncells
  if ( stage .ne. 1) then
    call sync_wait(predecessor(1)+1)
    ... unpack buffer ...
    if (stage .ne. ncells) then
      call sync_notify(predecessor(1)+1)
    endif
  endif
  ... perform forward sweep computation ..
  if (stage .ne. ncells) then
    ... pack data for successor ...
    if (stage .ne. 1) then
      call sync_wait(successor(1)+1)
    endif
    ... perform PUT ..
    call sync_notify(successor(1)+1)
  endif
enddo

! backsubstitution
call sync_notify(successor(1)+1)
call sync_wait(predecessor(1)+1)
do    stage = ncells, 1, -1
  if (stage .ne. ncells) then
    call sync_wait(successor(1)+1)
    ... unpack buffer ..
    if (stage .ne. 1) then
      call sync_notify(successor(1)+1)
    endif
  else
    ... computation ...
  endif
  ... perform backsubstitution ...
  if (stage .ne.  1) then
    ... pack buffer ...
    if (stage .ne. ncells) then
      call sync_wait(predecessor(1)+1)
    endif
    ... perform PUT to predecessor ...
    call sync_notify(predecessor(1)+1)
  endif
enddo
```

Figure 9.7: Fragment from the CAF SP `x_solve` routine, using `sync_notify`/`sync_wait`.

communication. The overall space cost for the sweeps along x, y, and z-directions will then be $O(6P)$. The version of `x_solve` that uses eventcounts is displayed in Figure 9.8.

```
evidx = allocate_eventcount(2)
! forward substitution
do  stage = 1, ncells
  if ( stage .ne. 1) then
    .. perform local computation w/o remote data ...
    call wait_eventcount(evidx,2,1)
     ... unpack buffer ...
     if (stage .ne. ncells) then
       call advance_eventcount(evidx, predecessor(1)+1, 1, 1)
     endif
  else
    .. perform local computation w/o remote data ...
  endif
  ... perform local computation ...
  if (stage .ne. ncells) then
    ... pack data for successor ...
    if (stage .ne. 1) then
      call wait_eventcount(evidx, 1, 1)
    endif
    ... perform PUT ..
    call advance_eventcount(evidx, successor(1)+1, 2, 1)
  endif
enddo

! backsubstitution
call advance_eventcount(evidx, successor(1)+1, 1, 1)
call wait_eventcount(evidx, 1, 1)
do stage = ncells, 1, -1
  if (stage .ne. ncells) then
    call wait_eventcount(evidx, 2, 1)
    ... unpack buffer ..
    if (stage .ne. 1) then
      call advance_eventcount(evidx,successor(1)+1,1,1)
    endif
  else
    ... computation ...
  endif
  ... perform backsubstitution ...
  if (stage .ne.  1) then
    ... pack buffer ...
    if (stage .ne. ncells) then
      call wait_eventcount(evidx,1,1)
    endif
    ... perform PUT to predecessor ...
    call advance_eventcount(evidx,predecessor(1)+1,2,1)
  endif
enddo
```

Figure 9.8: Fragment from the CAF SP `x_solve` routine, using eventcounts.

9.4.4 Generalized Wavefront Applications

Let's consider a generalized multiphase wavefront application, in which the dependency structure is given by a directed acyclic graph G_ϕ for every phase ϕ in the set of phases Φ. Each node executes the processing described in

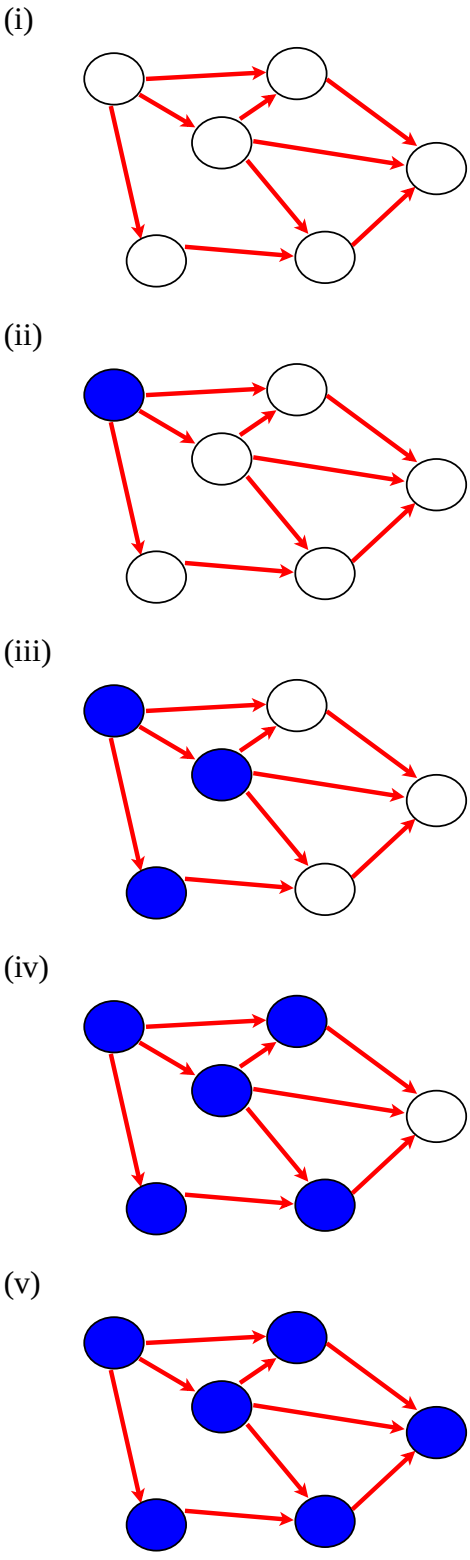


Figure 9.9: Graphical representation of progress in a generalized wavefront application.

```
p = this_image()
foreach phase φ ∈ Φ
    wait for data from all nodes in predecessors(p, φ)
   ... perform local computation ....
    send data to all nodes in successors(p, φ)
end
```

(a) Pseudocode for a generalized sweep application

```
p = this_image()
foreach phase φ ∈ Φ
    ... fill index(p, q,φ), position of p
      among successors of q ...
    evid_φ = allocate_eventcount(—successors(p, φ)—)
    foreach q in predecessors(p,φ)
        advance_eventcount(evid_φ,q,1+index(p,q,φ),1)
    end
    wait_eventcount (evid_φ,1,—predecessors(p,φ)—)
    foreach r in successors(p,φ)
        wait_eventcount(evid_φ,1+index(r,p,φ),1)
        ... send data to r ...
        advance_eventcount(evid_φ,r,1,1)
    end
end
```

(b) Pseudocode for a generalized sweep application using eventcounts

Figure 9.10: Pseudocode variants for a generalized sweep application.

A graphical representation of the application progress is given in Figure 9.9. To implement the synchronization, we need to use $|\Phi|$ eventcounts. The size of eventcount ϕ on node p is $1+|successors(p,\phi)|$, for a total space cost of $\Sigma_{\phi\in\Phi}\Sigma_{p=1}^{P}(1+|successors(p,\phi)|$. Notice that we could reuse some of the individual phase eventcounts (for example using only two) if we could prove that by the time we want to reuse an eventcount ϕ all the synchronization performed with ϕ in a prior phase completed on all images. Each node p will then execute the pseudocode presented in Figure 9.10(b).

9.5 Summary

In this chapter, we presented an extension to the CAF synchronization model, eventcounts, aimed at addressing space efficiency on petascale machines and synchronization composability for modular software. We described the API for eventcounts, an implementation strategy using active messages, and showed how they can be used to support data movement patterns common in scientific applications. Generally, `PUT`-based synchronization requires two phases: obtaining permission to write the remote buffer, then performing the remote write followed by notifying the remote process image. The eventcounts are as difficult to use for the first synchronization phase as the `sync_notify`/`sync_wait` mechanism. They can be easier to use for the second part, especially if we need notifications from several images before proceeding. The advantages of eventcounts over notifies are reduced space cost, in most of the examples we showed, and composability, enabling users to integrate seamlessly modular CAF solvers developed by different parties.

Chapter 10

Towards Communication Optimizations for CAF

A major appeal of a language-based programming model over library-based models such as MPI is that a compiler can more readily assist a programmer in tailoring the code to get high performance on the desired platform. It would be desirable to have a CAF compiler perform automatic communication optimization of CAF programs; however, we first need to create a framework that will guarantee the correctness of such transformations. In this chapter, we start by describing a memory consistency model for CAF and its implications on statement reordering, followed by a dependence analysis strategy in the presence of co-array accesses. In Chapter 6 we mentioned that communication vectorization for CAF codes such as NAS CG led to a performance improvements of up to 30%; in this chapter we present a dependence-based communication vectorization algorithm, followed by a proof of correctness and transformation details. We conclude the chapter by presenting the challenges of performing vectorization in the presence of resource constraints, and discuss future profitable dependence-based CAF optimizations.

10.1 A Memory Model for Co-Array Fortran

Having a well-defined memory model for Co-Array Fortran is of utmost importance: CAF users must know what is the expected behavior of their programs, and compiler writers must understand the safety conditions for automatic transformation of CAF codes. For parallel languages, the memory model has to take into account communication and synchronization.

As described in Section 1.1, CAF users can express remote reads (or GETs) and remote writes (or PUTs) at language level, using the bracket notation for remote references. The

CAF language, including our extensions described in Section 3.1, provides several synchronization primitives: `sync_all`, `sync_notify` and `sync_wait`. In Chapter 9, we proposed eventcounts as a space-efficient extension to the CAF synchronization mechanism.

In Section 3.1 we specified the semantics of `sync_notify` and `sync_wait` with respect to PUTs. Next, we describe in more detail the relation between synchronization and communication. For the purpose of exposition, we will define the function $version(x, P)$ for each co-array variable x and every process image P, using the following rules:

1. for every co-array x, on every process image P, $version(x, P) = 0$ at the start of program execution.

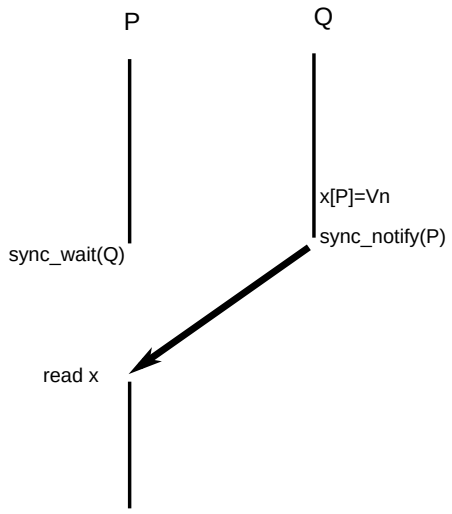
2. for each local write performed by a process image P to its local part of co-array x, $version(x, P) = version(x, P) + 1$.

3. for every remote write performed by a process image P to the local part of co-array x on image Q, $version(x, Q) = version(x, Q) + 1$.

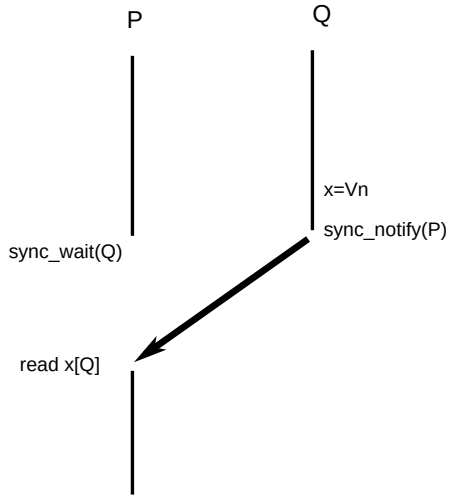
The function $version(x, P)$ denotes the version number (or version) of the variable x on P. To indicate that a local write to co-array x on image P has the effect $version(x, P) = n$, we will the notation $x = Vn$. To indicate that a remote write performed by process image P to process image Q has the effect $version(x, Q) = n$, we use the notation $x[Q] = Vn$.

Figure 10.1(a) shows the ordering between notifies and PUTs. If process image Q writes the co-array x on P with version number n, then sends a notify to P; after P executes a matching wait it can only read from its local portion of x a version k with $k \geq n$. k might be greater than n because Q or some other process image might subsequently perform one or more writes to x on P after the synchronization point, that increase the version number of x observed by P.

Figure 10.1(b) shows the ordering between notifies and GETs. Process image Q writes its local part of the co-array x with version number n, and then sends a notify to P; after executing a matching wait, P will read from Q the value of x and is guaranteed to get a



(a) `sync_notify` and `PUT`s

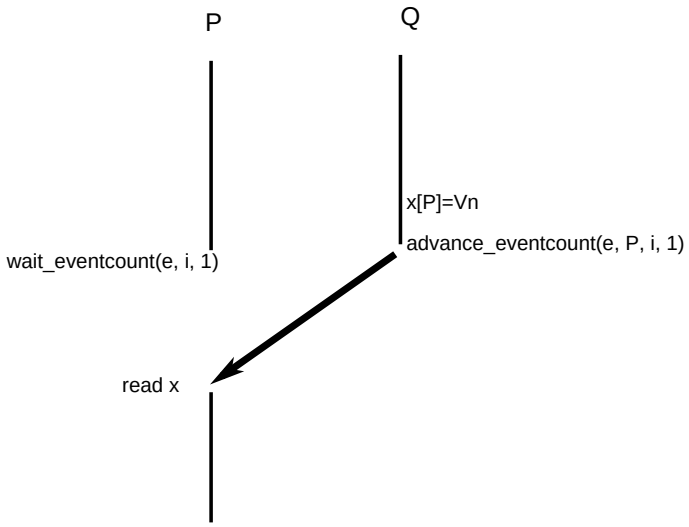


(b) `sync_notify` and `GET`s

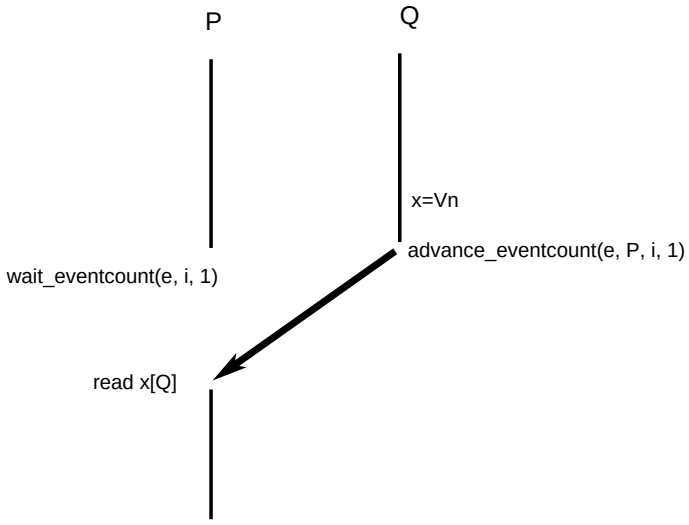
Figure 10.1: Relationship between `sync_notify`/`sync_wait` and remote accesses.

version k with $k \geq n$. k might be greater than n because Q or some other process image might subsequently perform one or more local writes to `x` after the synchronization point, writes that will increase the version number of `x` on Q observed by P.

In both `sync_notify`/`sync_wait` cases (a) and (b), P is guaranteed that Q has finished its local computation before the synchronization point and has finished all its `GET`s



(a) eventcounts and PUTs

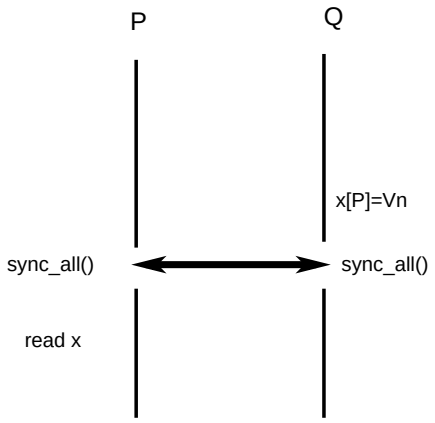


(b) eventcounts and GETs

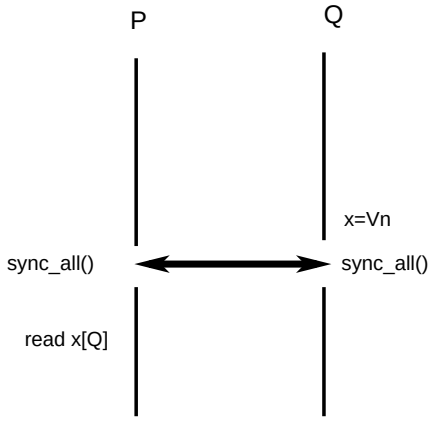
Figure 10.2: Relationship between eventcounts and remote accesses.

issued before calling `sync_notify`. However, P is not guaranteed that PUTs issued by Q to other process images have completed.

Figure 10.2(a) shows the ordering between eventcount operations and PUTs. If process image Q writes the co-array x on P with version number n, then advances by 1 the entry i



(a) barriers and PUTs



(b) barriers and GETs

Figure 10.3: Relationship between barriers and remote accesses.

of eventcount e on P; after P executes a matching wait it can only read from its local portion of x a version k with $k \geq n$. k might be greater than n because Q or some other process image might subsequently perform one or more writes to x on P after the synchronization point, that increase the version number of x observed by P.

Figure 10.1(b) shows the ordering between notifies and GETs. Process image Q writes its local part of the co-array x with version number n, and then advances by 1 the entry i of eventcount e on P; after executing a matching wait, P will read from Q the value of x and is guaranteed to get a version k with $k \geq n$. k might be greater than n because Q or

some other process image might subsequently perform one or more local writes to x after the synchronization point, writes that will increase the version number of x on Q observed by P.

In both eventcount cases (a) and (b), P is guaranteed that Q has finished its local computation before the synchronization point and has finished all its GETs issued before advancing the eventcount. However, P is not guaranteed that PUTs issued by Q to other process images have completed.

Figure 10.3(a) shows the ordering between barriers and PUTs. If process image Q writes the co-array x on P with version number n, and then sends a notify to P, P will then read from its local portion of x a version k with $k \geq n$. k might be greater than n because Q or some other process image might subsequently perform one or more writes to x on P after the synchronization point, writes that will increase the local version number of x observed by P.

Figure 10.3(b) shows the ordering between barriers and GETs. Process image Q writes its local part of the co-array x with version number n, and then synchronizes using a barrier with P; P will then read from Q a version k of x and is guaranteed that $k \geq n$. k might be greater than n because Q or some other process image might subsequently perform one or more local writes to x after the synchronization point, writes that will increase the version number of x on Q observed by P.

In both barrier cases (a) and (b), P is guaranteed that Q has finished its local computation and remote reads before the barrier. P is also guaranteed that remote writes issued by Q to other process images have completed.

There are several excellent reviews of memory consistency models [8, 87, 88, 144]. Pervasive throughout memory consistency model research is a tension between the *constraints* imposed by any particular memory model and the *performance* of programs written using it. More constraints make programming easier, but generally hurt performance. Fewer constraints means that a programmer has to be more careful when writing code and using the available communication and synchronization mechanisms, but the benefit is that of

increased performance. We review several memory consistency models and then discuss the memory model we propose for CAF.

Definition 9.1 In a *strict consistency* model, any read to a memory location X returns the values stored by the most recent write operation to X [182].

Definition 9.2. In a *sequentially consistent* model, the result of any execution is the same as if the reads and writes were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [129].

Definition 9.3 In a *processor consistency* model, writes done by a single processor are received by all other processors in the order in which they were issued, but writes from different processors may be seen in a different order by different processors [14, 89].

In the presence of synchronization variables, two more memory consistency models are defined.

Definition 9.4 In a *weak consistency* model [75], the following properties hold:

1. Accesses to synchronization variables are sequentially consistent.

2. No access to a synchronization variable may be performed until all previous writes have completed everywhere.

3. No data access (read or write) may be performed until all previous accesses to synchronization variables have been performed.

Definition 9.5 A *release consistency* model [88] uses locks on areas of memory, and propagates only locked memory as necessary. The basic operations *acquire* and *release* can be performed on locks. Release consistency is defined as follows:

1. Before accessing a shared variable, all previous acquires done by the process must have completed successfully.

2. Before a release is performed, all previous reads and writes done by the process must have completed.

3. The acquire and release accesses must be sequentially consistent.

We formally define a memory consistency model for CAF as follows:

Definition 9.6 CAF has the following synchronization mechanisms: `sync_all`, `sync_team`, `sync_notify`, `sync_wait` and eventcounts. Data movement and synchronization interact in the following ways:

1. Writes performed by a process image to overlapping sections of its local co-array parts are observed by that process image in the order in which they were issued.

2. Writes performed by a process image to overlapping sections of remote co-array parts are observed by the destination process image in the order in which they were issued.

3. If a process image P sends a `sync_notify` to process image Q, then upon completion on Q of the matching `sync_wait`, all `PUT`s to co-array parts on Q and all `GET`s of co-array parts on Q issued by P before issuing the `sync_notify` are complete.

4. If a process image P advances an eventcount on process image Q, then upon completion on Q of the matching `wait_eventcount` all `PUT`s to co-array parts on Q and all `GET`s of co-array parts on Q issued by P before advancing the eventcount are complete.

5. After execution of a `sync_all`, for any process image P, any `PUT`s or `GET`s issued by P before the `sync_all` are complete.

6. After execution of a `sync_team`, for any process image P, any `PUT`s or `GET`s issued by P before the `sync_team` are complete.

This memory consistency model is weaker than that proposed in the latest CAF draft [154]. The main difference is that in the proposed CAF standard any synchronization operation implies that all previous `PUT`s and `GET`s have completed, while in the memory model that

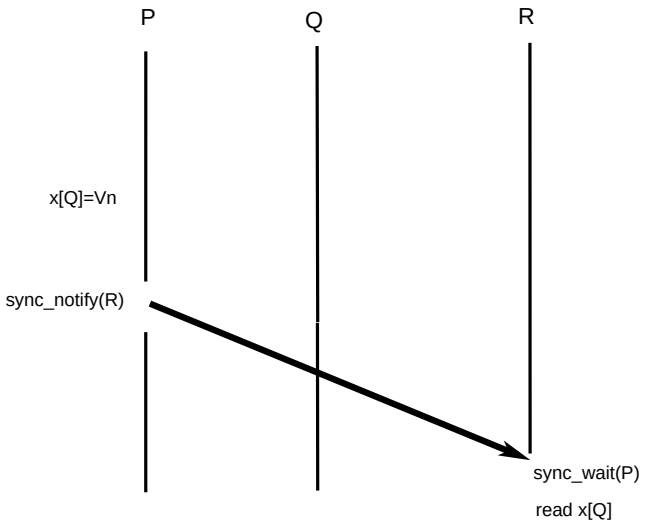
we propose the primitives `sync_notify` and `sync_wait` lead only to pairwise communication completion. Our model enables the overlap of communication issued by a process image P with different process images, thus decreasing exposed data transfer latency. The original CAF model contains critical sections; however, parallel programs using critical sections will not achieve scalable performance due to the serialization that critical sections require.

We can view the `sync_notify`, `sync_wait` and `sync_all` primitives as performing accesses to synchronization objects. Considering the ordering and constraints we described for `PUT`/`GET` and synchronization in CAF, the memory consistency model we propose for Co-Array Fortran is weaker than both the weak and release consistency models. For weak consistency, an access to a synchronization variables implies that all previous writes have completed. For release consistency, before performing a release, all previous reads and writes done by the process must complete. In both cases, the achieved effect of a synchronization operation by a process is that of a fence, which completes all writes performed by the process. In the case of a distributed memory system such as a cluster, with shared memory located on several cluster nodes, this might unnecessarily expose data transfer latencies. Consider the case where a process image p initiates a bulk remote write to shared data residing on a remote node, then initiates a bulk remote write to shared data residing on a second node, after which p invokes a synchronization operation. Upon the execution of the synchronization, both writes must complete, when it might more profitable to wait first for the completion of one of the writes, perform some computation, then wait for the completion of the second write. For CAF, we propose that pairwise synchronization operations `sync_notify` and `sync_wait` have the effect of pairwise completion of communication. We chose this memory model because it is conducive to high-performance, so CAF programs can overlap data transfers to different images and thus reduce exposed data transfer latency.

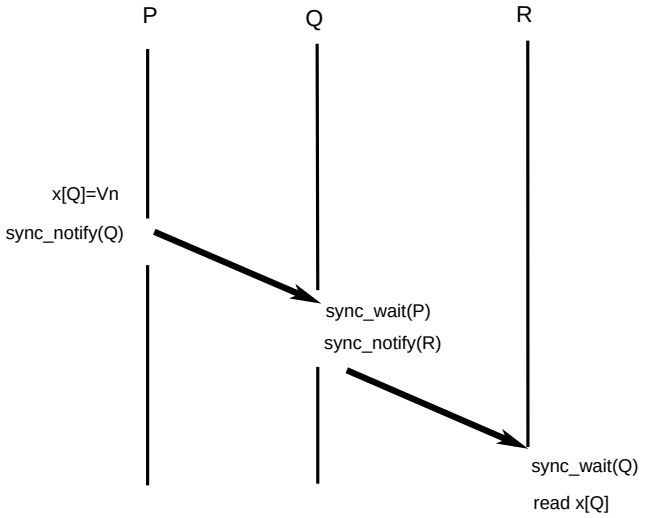
The CAF memory model is weaker than the Java memory consistency model [90, 131]. We do not provide any guarantees for CAF programs that contain data races. In Java, ac-

cesses to shared variables can be protected with locks, using `synchronized` methods. The synchronization model of CAF does not contain locks, and it only enables trivial sharing of data. Dotsenko [72] considers locks for more general coordination. A benefit of the Java memory model is that a programmer can control accesses to shared data at a finer granularity level in Java than in CAF, by choosing on which shared object to operate. In CAF, a call to `sync_notify` from process image P to process image Q would lead to pairwise completion of all `PUT`s issued by P to Q, even if the `PUT`s write to separate co-arrays. A recent refinement of the Java memory model [131] provides new guarantees for operations using `volatile` variables: when thread A writes to a volatile variable V, and thread B reads from V, any variable values that were visible to A at the time that V was written are guaranteed now to be visible to B. The CAF model is weaker than the Java memory model. If process image P writes the co-array `x` on Q with version number n, and then sends a `sync_notify` to process image R, then process image R is not guaranteed to read a version k of `x` on Q such that $k \geq n$, as shown in Figure 10.4. However, if process image P writes the co-array `x` on Q with version number n, then sends a `sync_notify` to process image Q, Q performs a matching `sync_wait`, followed by a `sync_notify` to R, then R, upon execution of a matching `sync_wait` from Q, is guaranteed to read a version k of `x` on Q such that $k \geq n$. k might be greater than n because Q or some other process image might have subsequently performed one or more writes to `x` on Q. A graphical representation of this scenario is shown in Figure 10.4(b).

For the CAF memory model that we propose to enable data transfer latency hiding, it is crucial that the `sync_notify` primitive be non-blocking. If `sync_notify` is non-blocking then a process image P can issue a non-blocking `PUT` to process image Q, followed by a `sync_notify` to Q, and immediately afterwards issue a non-blocking `PUT` to R, followed by a `sync_notify` to R. The net effect is that the `PUT`s to Q and R may overlap, which reduces the exposed data transfer latency. If `sync_notify` were blocking, then it would make it harder to hide data transfer and synchronization latency.



(a) R is not guaranteed to read a version k of $x[Q]$ with $k \geq n$.



(b) R is guaranteed to read a version k of $x[Q]$ with $k \geq n$.

Figure 10.4: Relationship between synchronization and remote accesses among multiple process images.

10.2 Implications of the CAF Memory Model for Communication Optimization

Based on the CAF memory model that we described, we can infer several rules limiting compiler-performed motion of code performing remote accesses *unless analysis proves*

that the code motion does not result in conflicting concurrent operations on shared data.

- For any process image P, `PUT`s to remote co-array data associated with P cannot move after a `sync_notify` to P unless it can be proven it is safe; otherwise, the destination process image might read a value older than the one written by the `PUT`.

- `PUT`s and `GET`s cannot move before a barrier. In the case of a `PUT` from process image P to co-array `x` on process image Q, the barrier completion might indicate to P that it is safe to perform the `PUT`, e.g., Q is done reading its local part of co-array `x`. Moving the `PUT` before the barrier would then lead to a race condition. In the case of a `GET` by process image P of co-array `x` on process image Q, the barrier completion might indicate to P that it is safe to perform the `GET`, e.g., Q is done writing its local part of co-array `x`. Moving the `GET` before the barrier would then lead to a race condition.

- `PUT`s and `GET`s cannot move after a barrier. In the case of a `PUT` from process image P to co-array `x` on Q, the barrier would indicate to Q that the `PUT` issued before the barrier has completed, and it is safe to read its local part of `x`. Moving the `PUT` after the barrier might lead to a situation where the `PUT` is not completed, but Q assumes that it is completed, accesses its local co-array part of `x` and reads a value older than the one it it supposed to read. In the case of a `GET` by process image P of co-array `x` on Q, the barrier would indicate to Q that the `GET` issued before the barrier has completed, and it is safe to write its local part of `x`. Moving the `GET` after the barrier might lead to a situation where the `GET` is not completed, but Q assumes that it is completed, writes its local co-array part of `x` and creates a race condition.

- For a co-array `x`, an access written as `x[p]`, even if `p` corresponds to the local image, is treated as communication. However, a CAF runtime library is free to recognize this case and implement it using a memory copy. In either case, completion is enforceable through synchronization statements.

Based on the following observations, we can make a conservative requirement for correctness of CAF transformations: in the absence of detailed analysis of local and remote data accesses both before and after communication, for a transformation of data race free programs to be correct, it should not move communication before or after synchronization points, and it should not reorder remote accesses to the same memory locations.

10.3 Dependence Analysis for Co-Array Fortran Codes

In this section, we present a strategy for performing dependence analysis of Co-Array Fortran codes. Dependence analysis for two Fortran 95 array references involves analyzing the set of pairs of corresponding subscripts, generating a set of constraints that all need to be satisfied in order to have a dependence, and analyzing if that constraint set can be solved within the context of the pair of references [17] (for example, when one or multiple loop indices are involved in the subscript pairs, an additional constraint is that each loop index can have only the values specified by its corresponding loop header). To perform dependence analysis for local and remote co-array references, we propose to consider the set of corresponding pairs of subscripts for local dimensions, but to augment that set with corresponding pairs of co-subscripts when present. This approach enables CAF compiler writers to leverage existing dependence analysis techniques for sets of subscript pairs. Once dependence analysis results are available, we present a correctness theorem for remote access reordering transformations. Our strategy works in the presence of the user-defined co-space extension to CAF proposed by Dotsenko [72]. We review the co-space extension in Section 10.3.1, describe our dependence analysis strategy in Section 10.3.2, and present a correctness theorem for dependence-based communication transformations in Section 10.3.3.

10.3.1 Co-space Types and Co-spaces Operators

To aid compiler analysis and enable users to organize process images, Dotsenko [72] proposed extending CAF with *co-spaces*. The co-space concept was inspired by MPI commu-

nicators; it enables users to organize process images into groups, with each group potentially having its own topology. There are three types of co-spaces:

- *Cartesian* co-spaces correspond to MPI Cartesian communicators; process images are organized into a Cartesian multi-dimensional grid. Neighbors are referenced using the `neighbor` operator. Consider a Cartesian co-space c with k dimensions, and a process image in c with the Cartesian coordinates $(p_1, p_2, ..., p_k)$: then $neighbor(c, i_1, i_2, ..., i_k)$, where i_j, $j = 1, k$ are integer expressions, refers to the process image with the coordinates $(p_1 + i_1, p_2 + i_2, ..., p_k + i_k)$ within the Cartesian co-space c.

- *graph* co-spaces correspond to the MPI graph communicators. Each process image has a list of successor process images, specified at co-space creation, such that there is a directed edge from the current process image to each successor process image in the list in the graph co-space. Consider a graph co-space c and a process image P within c; to refer to its k-th neighbor in the list of adjacent process images, P uses the operator $neighbor(c, k)$, where k is an integer expression.

- *group* co-spaces simply impose an order relation on a set of process images; to refer to the k-th process image in the group co-space c, one uses the operator $neighbor(c, k)$.

10.3.2 Dependence Analysis Using Co-space Operators

Let's consider two co-array references for which we want to perform dependence analysis. Each reference can be one of the following:

- local co-array reference

- co-array reference to a remote image specified using the $neighbor$ operator within a Cartesian, graph, or group co-space

We need to consider six cases. For brevity, we use $\overrightarrow{i}$ to refer to $< i_1, i_2, ..., i_k >$, $\overrightarrow{j}$ to refer to $< j_1, j_2, ..., j_k >$, $\overrightarrow{r}$ to refer to $< r_1, r_2, ..., r_m >$, and $\overrightarrow{q}$ to refer to $< q_1, q_2, ..., q_n >$.

1. a local reference $a(\overrightarrow{i})$ and a local reference $a(\overrightarrow{j})$. We consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$ for dependence analysis.

2. a local reference $a(\overrightarrow{i})$ and a remote reference $a(\overrightarrow{j})[neighbor(c, \overrightarrow{r})]$ where c corresponds to a Cartesian co-space. We consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$ and $< 0, r_s >$, $s = 1, m$ for dependence analysis.

3. a local reference $a(\overrightarrow{i})$ and a remote reference $a(\overrightarrow{j})[neighbor(c, r)]$ where c corresponds to a graph or group co-space. We consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$ for dependence analysis, and assume that the $neighbor$ operator can induce a dependence in the processor space.

4. two remote references using Cartesian co-spaces $a(\overrightarrow{i})[neighbor(c_1, \overrightarrow{q})]$ and $a(\overrightarrow{j})[neighbor(c_2, \overrightarrow{r})]$. If $c_1 \neq c_2$, then we consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$, for dependence analysis, and assume that there is a dependence within the processor space. If $c_1 = c_2$, then we consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$ and $< q_s, r_s >$, $s = 1, m$ for dependence analysis.

5. two remote references using graph or group co-spaces $a(\overrightarrow{i})[neighbor(c_1, q)]$ and $a(\overrightarrow{j})[neighbor(c_2, r)]$. If $c_1 \neq c_2$, then we consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$, for dependence analysis, and assume that there is a dependence within the processor space. If $c_1 = c_2$, then we consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$ and $< q, r >$ for dependence analysis.

6. a remote reference using a Cartesian co-space $a(\overrightarrow{i})[...]$ and a remote reference $a(\overrightarrow{j})[...]$ using a graph or group co-space. We consider the set of subscript pairs $< i_l, j_l >$, $l = 1, k$, for dependence analysis, and assume that there is a dependence within the processor space.

With these rules, we define the following types of dependences involving co-array accesses:

Definition 9.7 Dependences between local co-array accesses are *local dependences*.

Definition 9.8 Dependences involving at least one remote co-array reference are *cross-processor dependences*.

10.3.3 Discussion

If dependence analysis determines that remote co-array references are engaged in true, anti-, or output dependences, any code transformation must preserve those dependences to avoid violating the CAF memory consistency model.

One special case is when the local co-array image is referenced using bracket notation with an expression that cannot be analyzed at compile time. We have two options: either consider the possibility of dependences between local accesses and remote accesses, or pass the compiler a special flag to inform it that references to local parts of co-arrays are always specified with bracket notation. Such a requirement is not an unreasonable one, since the two-level memory feature of CAF leads to the users explicitly differentiate between local and remote accesses.

We recommend that CAF users employ the neighbor operator with constant arguments whenever possible when referring to remote co-arrays. This leads to code that a CAF compiler can more readily analyze and optimize than the one that uses general expressions for co-subscripts.

We can use dependences and the proposed CAF memory consistency model to guide `cafc` automatic transformations. Allen and Kennedy [17] define reordering transformations as follows:

Definition 9.9 A *reordering transformation* is any program transformation that merely changes the order of execution of the code, without adding or deleting any effects of execution of statements.

A CAF compiler can perform reordering transformation, but also remote reference re-

ordering transformations, defined as follows:

Definition 9.10 A *remote reference reordering transformation* reorders remote accesses with respect to their original statements. In the case of a remote read, the remote read is performed *before* the original statement, the off-processor values are saved in a temporary, and the temporary is used instead of the original remote read reference. In the case of a remote write, the value to be written is saved in a temporary, and the remote write is performed *after* the original statement.

Theorem 9.1 A CAF transformation that performs statement reordering and remote reference reordering does not change the meaning of a program without data races if it does not move remote accesses before or after synchronization statements and if it preserves local and cross-processor dependences.

Allen and Kennedy [17] prove by contradiction that transformations that perform statement reordering without changing dependences preserve the meaning of a program. Consider a program with the statements S_1, S_2, ..., S_n, such that each statement reads values produced by previous statements and in turn outputs new values. Consider a permutation S'_1, S'_2, ..., S'_n of the program statements induced by a reordering transformation. Assume that the meaning of the program after reordering is changed, and let S'_k be the first statement which produces a different output. This is due to S'_k reading a different input value x than in the original program execution. This can happen in three cases:

1. A statement S'_i writes x with the value that S'_k was supposed to read *after* S'_k reads it. This violates a true dependence, and contradicts the assumption that no dependence is violated.

2. A statement S'_i that in the original program execution was writing x after S'_k now writes x *before* S'_k reads it. This violates an anti-dependence, and contradicts the assumption that no dependence is violated.

3. A statement S'_i writes x before S'_k with the value that S'_k is supposed to read, but a statement S'_j that in the original program execution was writing x before S'_i now

writes it after S_i'. This violates an output dependence, and contradicts the assumption that no dependence is violated.

To extend that result to CAF, notice that each processor's dependences are preserved, and we are left to prove that after transformations, each processor reads/writes the same data when executing remote accesses. By performing remote reference reordering without crossing synchronization statements, we are guaranteed to perform the same remote accesses as in the original program. For a remote read in a program free of data races, the remote data is already available after some prior synchronization point, otherwise the original program contained a race condition; this implies that after remote read reordering the local process fetches the same remote value. For a remote write to process image P, note that there must be a synchronization statement *S* that followed the remote write and guaranteed that the write was delivered to P, because in a data race free program all conflicting accesses are separated by synchronization. Since after the reordering of the remote write no synchronization statements are crossed, the same synchronization statement *S* signals the completion of the remote write to P, so P reads the same result after the execution of its matching synchronization statement. Therefore, the statement and remote reference reordering transformation preserves the meaning of a data race free program.

10.4 Dependence-based Vectorization of CAF Codes

CAF codes with remote accesses can be analyzed using extensions of existing dependence analysis techniques and optimized by a CAF compiler. In this section, we describe a simple dependence-based vectorization algorithm, prove its correctness, present transformation details, and then discuss what steps are necessary to further tailor communication vectorization to various target architectures.

We review several terms used in the algorithm.

Definition 9.11 A *control-flow graph* (CFG) is a directed graph representation of all possible paths that can be taken during program execution. The graph nodes correspond to *basic blocks*, which are straight line sequences of code without any jumps. The graph

procedure *VectorizeComm* (procedure P)
scalarize array sections references [17]
assemble the set of subscript pairs for dependence analysis (see Section 10.3)
perform dependence analysis [17]
determine the set of outermost loops $LoopSet$
that do not contain synchronization statements.
foreach loop L_{out} in $LoopSet$
VectorizeLoop(L_{out},L_{out})
end
perform procedure splitting for all temporaries created during the vectorization process
and used with CAF array syntax expressions (see Section 5.1)

Figure 10.5: The driver procedure for the vectorization algorithm, *VectorizeComm*.

edges correspond to jumps in the program. The CFG has two special nodes, the *entry* node, through which all control flow enters the graph, and the *exit* node, through which all control flow exits the graph.

Definition 9.12 A CFG node y postdominates a CFG node x if every path from x to the exit node passes through y.

Definition 9.13 A statement y is said to be *control dependent* on another statement x if

1. there exists a nontrivial path from x to y such that every statement $z \neq x$ in the path is postdominated by y.

2. x is not postdominated by y.

Definition 9.14 A *control dependence graph* is graph that represents the control dependences between CFG blocks.

Definition 9.15 For each loop L, we define its *loop nesting level*, $level(L)$, as follows

1. $level(L) = 0$ iff $\neg\exists L'$ such that $L \subset L'$.

2. $level(L) = n + 1$ iff $\exists L'$ such that $level(L') = n$ and $L \subset L'$ and $\neg\exists L''$ such that $L \subset L'' \subset L'$.

procedure *VectorizeLoop*(L, L_{out})
  foreach outer loop L_i inside L itself
    VectorizeLoop(L_i, L_{out})
  foreach remote reference Ref in the body of L
    $L_{maxdep} = max\{level(L')|L'$ carries a dependence on the statement
      containing $Ref\}$
    if (Ref is a remote read)
      L_{vect}(Ref) $= max(L_{maxdep} + 1, level(L_{out}))$
    else
      $L_{minctrldep} = min\{level(L')|L'$ such that $Ref \in L'$ and the statement
        containing Ref is not control dependent on any non-loop header statement in $L'\}$
      $L_{vect}(Ref) = max(L_{maxdep} + 1, level(L_{out}), L_{minctrldep})$
    end
  end
  foreach reference Ref such that $L_{vect}(Ref) = level(L)$
    call *AllocateTemporariesAndRewriteReference*(L, $L_{vect}(Ref)$, Ref)
    call *GenerateRemoteAccessCode*(L, $L_{vect}(Ref)$, Ref)
  end

Figure 10.6: The *VectorizeLoop* procedure.

function *ClassifyCAFReference*(L, L_{vect}, Ref)
  Let L'_1, ..., L'_k be the loops containing Ref, such that $level(L'_i) \geq L_{vect}$, for $i = 1, k$
  Let L_1, L_2, ..., L_p be the loops with index variables used
    in the subscript expressions for Ref.
  Let $L_{i,lb}$, $L_{i,ub}$ be the lower bound, upper bound for loop L_i
  Let $L_{i,stride}$, $L_{i,idx}$ be the stride, loop index variable for loop L_i
  if each $L_{i,idx}$ is used in exactly one affine expression subscript $\alpha_i L_{i,idx} + \beta_i$
      and each α_i and β_i are constant w.r.t. $L'_1, L'_2, ..., L'_k$
    return AFFINE
  else
    return NON_AFFINE
  end

Figure 10.7: The procedure *ClassifyCAFReference*.

We present the driver procedure for our dependence-based communication vectorization algorithm, *VectorizeComm*, in Figure 10.5. The algorithm first scalarizes array section references, e.g. transforms Fortran 95 array section references into loop nests, as described in Allen and Kennedy [17]. Next, it assembles a set of subscript pairs, for both local di-

mensions and for co-dimensions, as described in Section 10.3. After that, it performs data dependence analysis of corresponding subscript pairs using techniques described in Allen and Kennedy [17].

To compute $level(L)$ for every loop, we would perform a recursive preorder traversal of the control dependence graph, and assign to each loop either the nesting level 0, if it does not have any loop ancestors, or the nesting level of the nearest loop ancestor plus 1.

The vectorization algorithm determines the set of outermost loops that do not contain synchronization statements. Formally,

$LoopSet = \{L | L$ does not contain any synchronization statements and $\neg\exists L'$ such that $L \subset L'$ and L' does not contain any synchronization statements $\}$

To determine $LoopSet$, we would construct the control dependence graph, perform a postorder traversal of the graph, and mark all the loops that contain synchronization statements. Next, we would perform a preorder traversal of the graph, and upon encountering a loop which is not marked we would add it to $LoopSet$ and stop traversing the successors of that loop in the control dependence graph.

Next, *VectorizeComm* invokes the routine *VectorizeLoop* for each loop in $LoopSet$. The procedure *VectorizeLoop* for a loop L is presented in Figure 10.6. For each remote co-array reference Ref in the statements immediately inside L, we first determine the loop nesting level where it can be vectorized. For both read and write references, we define L_{maxdep} as the maximum nesting level of a loop carrying a dependence on the statement containing Ref. For remote reads, the loop nesting level at which vectorization can be performed is $max(L_{maxdep}, level(L_{out})$, where L_{out} is the nesting level of the loop in $LoopSet$ that contains L. For remote write accesses, we also determine $L_{minctrldep}$, the minimum nesting level of a loop such that the statement containing Ref is not control dependent on any non loop header statement inside L'. The loop nesting level at which vectorization can be performed is then $max(L_{maxdep}, level(L_{out}), L_{minctrldep})$. *VectorizeLoop* invokes the procedure *AllocateTemporariesAndRewriteReference*, described in Figure 10.8, to allocate temporaries for data and possibly indices and to rewrite the reference. Finally, it invokes

procedure $AllocateTemporariesAndRewriteReference$(L,$L_{vect}$, Ref)
Let L'_1, ..., L'_k be the loops containing Ref, such that $level(L'_i) \geq L_{vect}$, for $i = 1, k$
if the co-array variable and all the subscripts of Ref are not written inside L'_1, ..., L'_k
declare a buffer $temp$ and replace Ref with a reference to the buffer,
normalizing the indices
else
Let L_1, L_2, ..., L_p be the loops with index variables used
in the subscript expressions for Ref.
Let $L_{i,lb}$, $L_{i,ub}$ be the lower bound, upper bound for loop L_i
Let $L_{i,stride}$, $L_{i,idx}$ be the stride, loop index variable for loop L_i
switch *ClassifyCAFReference*(L, L_{vect}, Ref)
case AFFINE
declare a temporary buffer $temp$ of shape
$(1 : (L_{1,ub} - L_{1,lb})/L_{1,stride} + 1, ..., 1 : (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$
replace the reference Ref with $temp$
replace each subscript $\alpha_i L_{i,idx} + \beta_i$ with $(L_{i,idx} - L_{i,lb})/L_{i,stride} + 1$
case NON_AFFINE
n_s=number of subscript expressions using index variables of the loops L_1,...,L_p
declare $itemp$ with shape $1 : n_s, ..., (L_{i,ub} - L_{i,lb})/L_{i,stride} + 1, ...$
insert a loop nest L_{itemp} immediately before L_{vect}, duplicating
the loop headers of L_1, ..., L_p, to fill in $itemp$
for $s = 1, n_s$
synthesize assignment in the innermost loop of L_{itemp}
to $itemp$(s,..., $(L_{i,idx} - L_{i,lb})/L_{i,stride} + 1, ...)$ of
subscript expression number s from Ref
end
declare a temporary buffer $temp$ of shape
$(1 : (L_{1,ub} - L_{1,lb})/L_{1,stride} + 1, ..., 1 : (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$
replace Ref by $temp((L_{1,idx} - L_{1,lb})/L_{1,stride} + 1, ..., (L_{p,idx} - L_{p,lb})/L_{p,stride} + 1)$
end switch
end if

Figure 10.8: The procedure *AllocateTemporariesAndRewriteReference*.

the procedure *GenerateRemoteAccessCode*, shown in Figure 10.9, to synthesize code that accesses the remote data.

The procedure *ClassifyCAFReference*, presented in Figure 10.7, determine whether the reference is affine or non affine. Consider the loops L_1, L_2,.., L_p with indices $L_{1,idx}$, $L_{2,idx}$, ..., $L_{p,idx}$ used in the subscript expressions for the remote reference Ref. For Ref to be affine, each subscript must be an affine expression of exactly one of loop index of the

enclosing loops, such as $\alpha_i L_{i,idx} + \beta_i$, where all variables used in the expressions of α_i and β_i are not written inside any of the loops L_1, L_2, .., L_p, for $i = 1, p$.

The procedure *AllocateTemporariesAndRewriteReference,* shown in Figure 10.8, allocates temporaries for data and possibly indices and rewrites the reference. Consider L'_1, L'_2, ..., L'_k the loops that contain the reference Ref, with a nesting level greater or equal than the level at which vectorization can be performed. If the co-array variable and all the variables used for subscript expressions for Ref are not written inside the loops L'_i, for $i = 1, k$, then we declare a buffer $temp$ and replace Ref with a reference to $temp$. The procedure would also normalize the indices of $temp$. Otherwise, we consider the loops L_1, L_2, ..., L_p such that their indices $L_{1,idx}$, $L_{2,idx}$, ..., $L_{p,idx}$ are used in the subscript expression for Ref. We denote the lower bounds of the loops by $L_{i,lb}$, for $i = 1, p$, the upper bounds by $L_{i,ub}$, for $i = 1, p$, and the loop strides by $L_{i,stride}$, for $i = 1, p$.

AllocateTemporariesAndRewriteReference invokes *ClassifyCAFReference* to determine if Ref is affine or non affine. If the reference is affine, then the vectorization algorithm will use a regular CAF array section remote reference to access the remote data, that will be converted into communication code as described in Section 4.3. The procedure *AllocateTemporariesAndRewriteReference* declares $temp$, a temporary buffer for the off-processor data, of shape $(1 : (L_{1,ub} - L_{1,lb})/L_{1,stride} + 1, 1 : (L_{2,ub} - L_{2,lb})/L_{2,stride} + 1, ..., 1 : (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$. Next, the reference Ref is replaced with a reference to $temp$, and each affine subscript $\alpha_i L_{i,idx} + \beta_i$ will be replaced with its correspondent subscript within $temp$, $(L_{i,idx} - L_{i,lb})/L_{i,stride} + 1$.

If the reference is non affine, then we will use an Active Message to perform the remote access. Active Messages [190] (abbreviated AM) were reviewed in Section 2.1.2; a sender issues a message containing an AM handler identifier and data. On the receiving side, an AM dispatcher first determines the AM handler responsible for processing the message, then invokes the handler and passes it the message data. In the case Ref is non affine, the current process image would collect the local indices for the remote co-array data and send them in an AM. *AllocateTemporariesAndRewriteReference*

procedure $GenerateRemoteAccessCode(L, L_{vect}, Ref)$
Let L'_1, ..., L'_k be the loops containing Ref, such that $level(L'_i) \geq L_{vect}$, for $i = 1, k$
if (Ref is a remote read reference)
if the co-array variable and all the subscripts of Ref are not written inside L'_1, ..., L'_k
insert a Co-Array Fortran statement to assign the remote value
to $temp$ immediately before loop L_{vect}
else
Let L_1, L_2, ..., L_p be the loops with index variables used
in the subscript expressions for Ref.
Let $L_{i,lb}$, $L_{i,ub}$ be the lower bound, upper bound for loop L_i
Let $L_{i,stride}$, $L_{i,idx}$ be the stride, loop index variable for loop L_i
switch *ClassifyCAFReference*(L, L_{vect}, Ref)
case AFFINE
insert assignment of the remote reference
$Ref(..., \alpha_i L_{i,lb} + \beta_i : \alpha_i L_{i,ub} + \beta_i : \alpha_i L_{i,stride}, ...)$
into $temp(..., 1 : (L_{i,ub} - L_{i,lb}/L_{i,stride} + 1, ...)$
immediately before the loop L_{vect}
case NON_AFFINE
generate an AM handler to pack the remote reference Ref into $temp$
insert AM handler invocation before L_{vect}, passing $itemp$
end switch
else
if the co-array variable and all the subscripts of Ref are not written inside L'_1, ..., L'_k
insert a CAF statement that assigns $temp$ to the remote section immediately after L
else
switch *ClassifyCAFReference*(L, L_{vect}, Ref)
case AFFINE
insert remote assignment into $Ref(..., \alpha_i L_{i,lb} + \beta_i : \alpha_i L_{i,ub} + \beta_i : \alpha_i L_{i,stride}, ...)$
from $temp(..., 1 : (L_{i,ub} - L_{i,lb})/L_{i,stride} + 1, ...)$ immediately after L_{vect}
case NON_AFFINE
generate an AM handler to unpack $temp$ into the remote reference Ref
insert AM handler invocation passing $itemp$ and $temp$ immediately after L_{vect}
end switch
end if

Figure 10.9: The procedure *GenerateRemoteAccessCode*.

determines the number of subscript expressions that use the loop index variables $L_{1,idx}$, ..., $L_{p,idx}$, denoted by n_s. Next, it allocates a temporary to hold the indices, $itemp$, of shape $(1 : n_s, (1 : (L_{1,ub} - L_{1,lb})/L_{1,stride} + 1, 1 : (L_{2,ub} - L_{2,lb})/L_{2,stride} + 1, ..., 1 : (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$. To fill $itemp$ with the values of the local indices for Ref,

a loop nest L_{itemp} is inserted before the loop at level L_{vect}, duplicating the loop headers of the loops L_1, ..., L_p. In the innermost loop of L_{itemp}, we synthesize an assignment for each of the n_s subscript expression, assigning the value of the s-th subscript expression to $itemp(s, (L_{1,idx} - L_{1,lb})/L_{1,stride} + 1, (L_{2,idx} - L_{2,lb})/L_{2,stride} + 1, ..., (L_{p,idx} - L_{p,lb})/L_{p,stride} + 1)$. Next, the algorithm declares and allocates a temporary buffer $temp$ of shape $(1 : (L_{1,ub} - L_{1,lb})/L_{1,stride} + 1, ..., 1 : (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$, and replaces Ref by $temp((L_{1,idx} - L_{1,lb})/L_{1,stride} + 1, ..., (L_{p,idx} - L_{p,lb})/L_{p,stride} + 1)$.

The procedure *GenerateRemoteAccessCode*, shown in Figure 10.9, synthesizes code that accesses the remote data. To perform the remote accesses, we can use either array section CAF references, for which `cafc` generates communication code as explained in Section 4.3, or use Active Messages. For an active message, the compiler needs to synthesize the handler of the AM and to insert an invocation of the AM into the generated code. Consider L'_1, L'_2, ..., L'_k the loops that contain the reference Ref, with a nesting level greater or equal than the level at which vectorization can be performed.

For remote read references, if the co-array variable and all the variables used for subscript expressions for Ref are not written inside the loops L'_i, for $i = 1, k$, then we insert a CAF `GET` of the remote data into $temp$, immediately before the enclosing loop at nesting level L_{vect}. Otherwise, similar to the procedure *AllocateTemporariesAndRewriteReference*, we consider separately the cases of affine and non affine references. If a reference is affine, then we synthesize a CAF remote read reference from $Ref(\alpha_1 L_{1,lb} + \beta_1 : \alpha_1 L_{1,ub} + \beta_1 : \alpha_1 L_{1,stride}, ..., \alpha_p L_{p,lb} + \beta_p : \alpha_p L_{p,ub} + \beta_p : \alpha_p L_{p,stride}$ into $temp((L_{1,ub} - L_{1,lb})/L_{1,stride} + 1), ..., (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$ immediately before the enclosing loop at nesting level L_{vect}. If the reference is non affine, then we synthesize an AM handler to pack the remote reference Ref into $temp$, then insert an invocation of the AM handler immediately before the enclosing loop at nesting level L_{vect}.

For remote write references, if the co-array variable and all the variables used for subscript expressions for Ref are not written inside the loops L'_i, for $i = 1, k$, then we insert a CAF `PUT` into the remote data from $temp$, immediately after the enclosing loop at nesting

level L_{vect}. Otherwise, similar to the procedure *AllocateTemporariesAndRewriteReference*, we consider separately the cases of affine and non affine references. If a reference is affine, then we synthesize a remote write CAF reference to $Ref(\alpha_1 L_{1,lb} + \beta_1 : \alpha_1 L_{1,ub} + \beta_1 : \alpha_1 L_{1,stride}, ..., \alpha_p L_{p,lb} + \beta_p : \alpha_p L_{p,ub} + \beta_p : \alpha_p L_{p,stride}$ from $temp((L_{1,ub} - L_{1,lb})/L_{1,stride} + 1), ..., (L_{p,ub} - L_{p,lb})/L_{p,stride} + 1)$ immediately before the enclosing loop at nesting level L_{vect}. If the reference is non affine, then we synthesize an AM handler to unpack $temp$ into the remote reference Ref, then insert an invocation of the AM handler immediately after the enclosing loop at nesting level L_{vect}.

10.4.1 Dependence-based Vectorization Correctness

Theorem 9.2 The transformation performed by the routine *VectorizeComm* is correct for data race free programs.

Proof: Any remote access references introduced by the *VectorizeLoop* transformations would be inserted inside a loop L from $LoopSet$ or immediately before L or immediately after L. Since L does not contain any synchronization statements, in each case no communication statements would be moved past synchronization points.

The routine *VectorizeLoop* does not hoist a remote read or write reference Ref past the level of a loop which carries a dependence on the statement containing Ref, so it does not reverse any dependence.

We have proven that the transformation *VectorizeComm* doesn't move any remote accesses past synchronization points and that it preserves dependences. According to Theorem 9.1, the transformation *VectorizeComm* does not change the meaning of the code for a data race free program. Thus, the transformation described by *VectorizeComm* is correct.

10.4.2 Transformation Details

Temporary buffer management. A question relevant for performance is how temporary buffers are to be allocated and managed. For performance, the memory for temporary

buffers should be allocated by the communication library as it sees fit, e.g. perhaps in pinned physical pages on a Myrinet cluster. The natural language level representation for temporary buffers is Fortran 95 pointers. However, the use of Fortran 95 pointers might degrade local performance, because Fortran 95 compilers might conservatively assume pointer aliasing, and inhibit key optimizations for scalar performance.

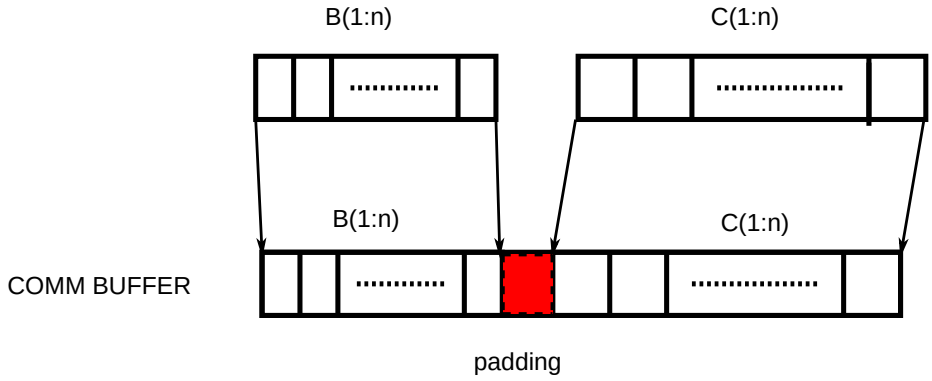
As shown in Chapter 6, procedure splitting is an important optimization for local performance. It transforms pointer references into array arguments, which conveys to the back-end compiler the lack of aliasing, the fact that the array is contiguous, and the shape of the local co-array. To get the same benefits for vectorization-introduced temporaries, after applying vectorization, we could perform procedure splitting and pass array temporaries as arguments to an inner routine as well.

Vectorization temporaries suitable for procedure splitting are those used in Co-Array Fortran array section assignment; their shape should also be expressed only by means of specification expressions with regard to the current procedure. A more aggressive transformation would be to outline the code between the allocation and deallocation of a temporary into a procedure, invoke that procedure and pass it the temporaries as array arguments.

Active Messages buffer management. For efficiency reasons, we need to pass to an AM a vector of indices, in the case of subscripts using array references, or perhaps coefficients, in the case of multiple affine expressions with respect to the loop variables. Clearly we want to avoid performing unnecessary data copying and using multiple messages. The solution we propose is to determine the size of the storage necessary to hold the subscript information, allocate a vector of length sufficient to hold the subscripts, and then collect the necessary subscript values in that vector. For a `GET`, it suffices to send the vector of subscript triplets for the local dimensions of Ref, in order to to collect the remote data into a return buffer. For a `PUT`, we need to transmit both the subscript values and the right-hand side data for the remote write statement. One alternative is to allocate two separate buffers, one for subscript values and one for off-processor data; the runtime layer would then copy them into contiguous storage and then invoke one active message; this leads to extra data

```
double precision A(1:100)[*]
integer B(1:100)
double precision C(1:100)
A(B(1:n))[p]=C(1:n)
```

(a) Example of PUT using array references for subscripts.



(b) Storage management for subscript values and right-hand side data.

Figure 10.10: Buffer management for remote writes subscripts and right-hand side data; padding is used so that the targets of subscript and data pointers each have a 64-bit alignment.

copying. A more effective solution is to determine the size of the storage necessary to hold the subscripts and the right-hand side data, then allocate a buffer large enough to hold both the indices and the off-processor data, and set up the pointers for indices and data to use this common storage. To preserve data alignment we must allocate a padding zone between the subscripts and the data, so that the targets of index and data pointers each have a 64-bit alignment. For the code fragment presented in Figure 10.10(a), the storage for the indices `B(1:n)` and the right-hand side `C(1:n)` would be managed as shown in Figure 10.10(b).

Active Message Handlers. Active Messages are flexible means of realizing communication, however they might be less efficient than Remote Data Memory Access (RDMA) on certain communication fabrics. For performance, it is preferable to express the vectorization using Fortran 95 array sections without indirection arrays for subscript expressions,

since then a communication library can use an RDMA transfer, and only when this is not possible we would use Active Messages to perform communication vectorization. For each vectorized communication event we would generate an AM invocation, inserted before the loop L_{vect} for remote reads and after the loop L_{vect} for remote writes. We must also generate an active message handler.

Next, we present the AM handler generation examples for two types of communication patterns.

Subscripts using indirection arrays. Consider the following loop:

The co-array A on the remote image p is accessed using the subscript vector B, as shown in Figure 10.11(a). We present the code we would generate on the source process image in Figure 10.11(b). To finalize the put, the remote process image would invoke the active message handler shown in Figure 10.11(c).

Subscripts using multiple affine expressions of the loop index variable. Consider the loop presented in Figure 10.12(a). The co-array A on the remote image p is accessed on a diagonal subsection. After vectorization, we would generate the code presented in Figure 10.12(b). To finalize the PUT, the remote process the image would invoke the active message handler shown in Figure 10.12(c).

10.4.3 Discussion

The procedure *VectorizeComm* might not efficiently handle control flow inside a loop body. For remote reads which are control-dependent on a conditional expression, *VectorizeComm* would prefetch a remote section of size proportional to the loop trip count. This might lead to unnecessary communication. However, *VectorizeComm* inhibits vectorization at the level of a loop L for remote writes which are control dependent on statements inside L. Another potential solution would be to detect computation slices necessary to determine the remote elements which are accessed, as described in Das *et al* [64]. The best choice is application dependent.

Our vectorization algorithm can be extended immediately to work correctly in the pres-

ence of natural loops, e.g., loops written using `IF-THEN-ELSE` and `GOTO` statements instead of using structured programming constructs such as `DO`-loops or `FOR`-loops. Natural loops can be identified by analyzing the control-flow graph [15], and we would not vectorize a reference `Ref` past a natural loop if it carries a dependence on the statement containing `Ref`.

For the sake of exposition, we presented an algorithm that performed vectorization of remote accesses for co-arrays of primitive types. The algorithm extends immediately to co-arrays of user-defined types without pointer fields, and to allocatable co-arrays of both primitive types and user-defined types without pointer fields. The algorithm can also be applied to co-arrays with allocatable components, where the target of the vectorization is represented by multiple references to a structure field of primitive type or user-defined type without pointer fields. For example, references to `a%b(1)`,`a%b(2)`,...,`a%b(n)` could be vectorized into `a%b(1:n)`, where `a` is a co-array and `b` is an allocatable component.

The communication vectorization algorithm would further need to address architectural constraints. One constraint is buffer size on nodes with limited memory. If full communication hoisting requires more memory than it is available, than instead of full hoisting of communication we need to first strip mine the loop which induces the vectorization, then perform full communication hoisting in the newly created inner loop. A similar method addresses another architectural constraint: the maximum size of the message that can be injected into the communication network. If we try to send very large messages, then the software communication layer will send the message in pieces, with a delay between each piece. This would expose communication latency, which is not desirable. The solution is again to strip mine the loop inducing communication vectorization, such that the hoisted communication size is smaller than the maximum message size accepted by the communication interconnect. In both cases, determining the appropriate size of the communication granularity would be machine-dependent and could be performed in a self-tuning step upon installation of the compiler on that system.

10.5 Dependence-based Communication Optimizations of CAF

In this chapter, we presented a memory consistency model for CAF, extended existing dependence analysis techniques to work with both local and remote co-array references, and presented a dependence-based communication vectorization algorithm. As explained, this transformation does not move communication statements past synchronization points.

Dependence-based communication optimization include more than just vectorization. We mention next several cases where dependence analysis can be used to improve the performance of CAF codes without moving communication past synchronization points.

One opportunity is to reorder communication statements so that both `PUT`s and `GET`s are initiated early. One such example is in Figure 10.13(a): the `GET` from process image 2 can be initiated before the I loop nest, and it can be checked for completion after the loop nest, as shown in Figure 10.13(b). This reordering would exploit non-blocking primitives to enable hiding the latency of the `GET` from process image 2 with the computation performed in the I loop. A CAF compiler could do this automatically by using a simple list-scheduling algorithm on the dependence graph that issues `GET`s as early as possible and delays their completion as late as possible.

Let's consider the code fragment presented in Figure 10.13(c). We have an opportunity to pipeline `PUT`s to the neighbors `up` and `down` and thus overlap communication with local computation, with the possible transformation presented in Figure 10.13(d). A CAF compiler could achieve this effect automatically by first scalarizing the two `PUT`s, fusing the J loop with the loops generated by the `PUT`s, strip mining the resulting loop, and then performing vectorization over the inner loop.

Let's consider the code fragment presented in Figure 10.13(e), we have the potential of waiting for `GET` completion right before the data obtained is used, as shown in Figure 10.13(f).

The largest benefit of CAF transformations should be achieved when dependence analysis and synchronization analysis are combined. In Figure 10.14(a) we present a code fragment that contains an opportunity to issue a `GET` earlier, before an unrelated `sync_notify`,

provided we can prove that $P \neq Q$. The transformed code is shown in Figure 10.14(b). In Figure 10.14 we present a code fragment that contain an opportunity to issue a `GET` before a barrier: since co-array `x` is written before the first barrier, and no process image accesses `x` before the first and the second barrier, it it safe to move the `GET` before the second barrier (but not before the first barrier).

```
double precision A(1:100)[*]
integer B(1:100)
double precision C(1:100)
integer i
do i=1,n
  A(B(i))[p]=C(i)
end do
```

(a) Example of code using indirection arrays for subscripts

```
amStorageAllocate(8*n+4*n, transfer_ptr)
SetStoragePointer(ptrIndex, transfer_ptr, 0)
SetStoragePointer(bufferPtr, transfer_ptr, 4*n+paddingSize)
ptrIndex(1:n) = B(1:n)
do i=1,n
  bufferPtr(i) = C(i)
end do
invoke AM to perform the remote PUT
```

(b) Code generation on the source process image

```
subroutine am_put(A, aShape, indexVector, buffer)
double precision A(1:100)
integer aShape(1)
integer indexVector( aShape(1))
integer i
double precision buffer(1:aShape(1))
do i=1, n
  A(indexVector(i))=buffer(i)
end do
```

(c) Corresponding AM handler

Figure 10.11: Code generation example for remote writes with subscripts using indirection arrays

```
double precision A(1:100, 1:100)[*]
double precision C(1:100)
integer i
k1= ... ! non-constant expression
k2= ... ! non-constant expression
do i=1, n
   A(i+k1,i+k2)[p]=C(i)
end do
```

(a) Example of code using multiple affine expression of loop index variables for subscripts

```
amStorageAllocate(8*n+4*n, transfer_ptr)
SetStoragePointer(ptrIndex, transfer_ptr, 0)
SetStoragePointer(bufferPtr, transfer_ptr, 4*n+paddingSize)
ptrIndex(1) = k1
ptrIndex(2) = k2
do i=1,n
   bufferPtr(i) = C(i)
end do
invoke AM to perform the remote PUT
```

(b) Code generation on the source process image

```
subroutine am_put(A, aShape, indexVector, buffer)
double precision A(1:100, 1:100)
integer aShape(1)
integer indexVector(2)
integer i
double precision buffer(1:aShape(1))
do i=1, n
  A(i+indexVector(1), i+indexVector(2))=buffer(i)
end do
```

(c) Corresponding AM handler

Figure 10.12: Code generation example for remote writes with subscripts using multiple affine expressions of the loop index variables

```
DO I=1,N
 ... compute on A, B ...
END DO

TEMP(1:N) = D(1:N)[2]
```

(a) Opportunity to initiate a GET earlier

```
... initiate GET of D(1:N)[2]
 into TEMP(1:N) ...
DO I=1,N
 ... compute on  A, B
END DO
... wait for GET completion ...
```

(b) Non-blocking GET

```
DO J=1,N
 ... compute A(:,J) ...
END DO

A(0,:)[down]=A(N,:)
A(N+1,:)[up]=A(1,:)
```

(c) Opportunity to pipeline PUTs

```
DO I=1,N,S
  ... compute A(:,I:I+S-1)
  start PUT to
      A(0,I:I+S-1)[down]
  start PUT to
      A(N+1,I:I+S-1)[up]
END DO
... wait for PUTs completion ...
```

(d) Pipelined PUTs

```
A(:,N)=B(1:N)[left]
DO J=1,N
 ... compute with A(:,J) ...
END DO
```

(e) GET/computation overlap opportunity

```
DO I=1,N,S
  initiate A(:,I:I+S-1)=
    B(I:I+S-1)[left]
END DO

DO I=1,N,S
  ... wait for completion of
   GET into A(:,I:I+S-1)
  ... compute with A(:,I:I+S-1)
END DO
```

(f) Pipelined GET

Figure 10.13: Opportunities for dependence-based communication optimization of CAF codes

```
x[P]=...
sync_notify(P)
... = x[Q]
```

(a) Opportunity to initiate a GET earlier

```
initiate GET of x[Q]
x[P]=...
call sync_notify(P)
... wait for GET completion ...
```

(b) Non-blocking GET before notify

```
... some process images
  write x ...
call sync_all()
... no process images
  accesses x ...
call sync_all()
...=x[Q]
```

(c) Opportunity to initiate a GET earlier

```
... some process images
  write x ...
call sync_all()
initiate GET of x[Q]
... no process image
  accesses x ...
call sync_all()
... GET must be completed ...
```

(d) Non-blocking GET before barrier

Figure 10.14: Opportunities for optimization using combined dependence and synchronization analysis.

Chapter 11

Pinpointing Scalability Bottlenecks in Parallel Programs

To exploit the power of petascale systems composed of tens of thousands of processors, parallel applications must scale efficiently. However, writing and tuning complex applications to achieve scalable parallel performance is hard.

Understanding a parallel code's impediments to scalability is necessary step for improving performance. Often, an application's scalability bottlenecks are not obvious. They can arise from a range of causes including replicated work, data movement, synchronization, load imbalance, serialization, and algorithmic scaling issues. Having an automatic technique for identifying scalability problems would boost development-time productivity.

When analyzing an application's scaling bottlenecks, one should focus on those that are the most significant. An application's components with the worst scaling behavior are often not the most significant scaling bottlenecks for the application as a whole. For instance, a routine that displays abysmal scaling but consumes only a fraction of a percent of the total execution time is less important than a routine that is only a factor of two from ideal scaling but accounts for nearly 50% of the total execution time on large numbers of processors. For developers to tune applications for scalable performance, effective tools for pinpointing scalability bottlenecks and quantifying their importance are essential.

In our early efforts to understand the performance problems of parallel codes, we used HPCToolkit [136], an efficient performance analysis toolchain. HPCToolkit enables the analysis of running program by performing sampling of various hardware counters. HPCToolkit associates the performance data with application source code and presents to a user via a graphical interface. HPCToolkit generates a *flat* performance profile; a user would know how much time is spent in a certain procedure, such a communication routine, over

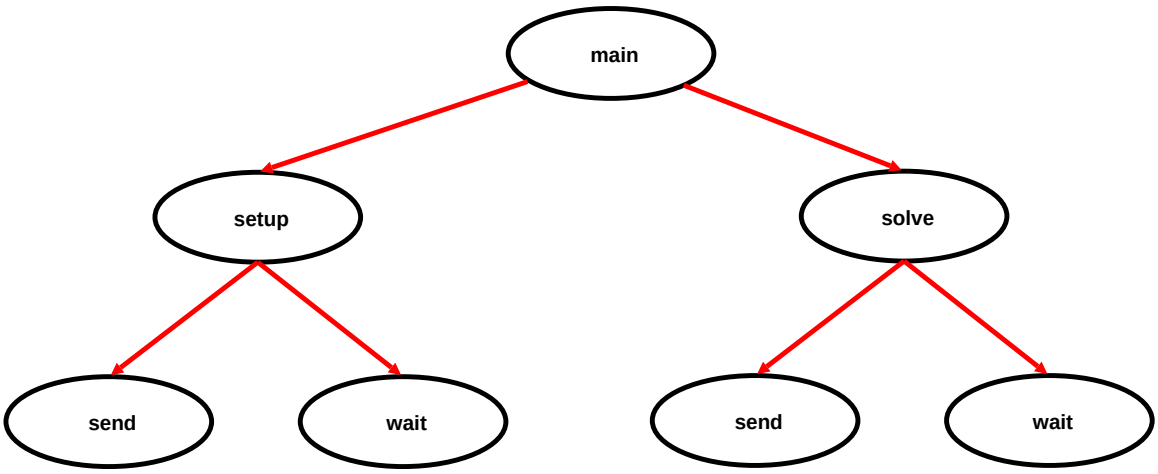


Figure 11.1: Motivating example for parallel performance analysis using *calling contexts*: users are interested in the performance of communication routines called in the `solver` routine.

the entire execution of a program. However, in our experience using HPCToolkit to profile parallel codes, we discovered that such information is not sufficient. If the goal is modifying the application to improve the parallel performance, then it is extremely useful to know the *calling context* of the communication routine, em on which call chain it occurred and with what frequency. A motivating example is given in Figure 11.1: let's consider a parallel application in which the `main` routine invokes a `setup` routine, followed by the actual `solver` routine. Both `setup` and `solver` routines invoke communication routines such as `send` and `wait`. A flat performance profile would tell us how much total time is spent in the communication routines; however, a user would be more interested in how much time is spent in the communication routines called from the solver.

This chapter describes a new approach for identifying scalability bottlenecks in executions of SPMD parallel programs, quantifying their impact on performance, and associating this information with the program source code. Our analysis technique and our tools that apply it are independent of the parallel programming model, underlying processor architecture, communication interconnect, and application characteristics. Our approach involves

three steps.

First, we collect call path profiles for two or more executions of unmodified, fully-optimized application binaries on different numbers of processors. Call path profiles capture the costs of the various calling paths during the execution of a program. We represent concisely a call path profile as a calling context tree (CCT) [19]. In a CCT, each node corresponds to a procedure, such that every path from the root to each node reflects an actual call path realized during the program execution. The nodes of the CCT are annotated with the number of samples that were collected by the profiler in the procedure corresponding to that node, which approximates the execution cost of the node.

Second, we use our expectations about how costs should differ among an ensemble of executions to compute scalability at each point in a program's execution. We assess each component's deviation from scalable performance by computing its cost in excess of its expected value. We report this cost normalized as a fraction of overall program execution time. To help developers understand how performance bottlenecks arise, we attribute scalability metrics to each node in an execution's calling context tree.

Third, with the aid of an interactive browser, an application developer can explore a calling context tree top-down fashion, see the contexts in which poor scaling behavior arises, see the source code lines that fail to deliver scalable performance, and understand exactly how much each scalability bottleneck dilates execution time.

In this chapter we evaluate the applicability of call path based profiling for parallel codes. We used a toolchain containing `csprof`, and `hpcviewer` to evaluate the scalability bottlenecks for a series of CAF applications such as the NAS benchmarks MG, CG, SP, and the LBMHD kernel, for a UPC version of NAS CG, and for a MPI version of the Parallel Ocean Program (POP), and for a MILC benchmark. In Appendix A, we present scaling analysis results for MPI and CAF versions of NAS MG, CG, SP, BT and LU. We determine which communication and synchronization primitives do not scale, and rely on the call path information to determine which code fragments and programming idioms are responsible for the non-scalable use of communication primitives.

11.1 Call Path Profiling and Analysis

The `csprof` profiler [82, 83], developed as part of the HPCToolkit project [168] at Rice University, profiles unmodified, fully-optimized executables without prior arrangement. `csprof` uses event-based sampling in conjunction with a novel call stack unwinding technique to attribute execution costs to calling contexts and associate frequency counts with call graph edges.

`csprof` stores sample counts and their associated calling contexts in a *calling context tree* (CCT) [19]. In a CCT, the path from each node to the root of the tree represents a distinct calling context. A calling context is represented by a list of instruction pointers, one for each procedure frame active at the time the event occurred. Sample counts attached to each node in the tree associate execution costs with the calling context in which they were recorded.

After post-mortem processing, `csprof`'s CCTs contain three types of nodes: procedure frames, call sites and simple statements. A procedure frame can have call sites and simple statements as children. A call site can have one or more procedure frames as children. Simple statements don't have any children.

In this chapter, we use `csprof`'s CCTs as the basis for analyzing an ensemble of executions using performance expectations. `csprof` supports measurement of both synchronous and asynchronous events. For each event, `csprof` records the calling context in which the event occurred. `csprof` has low overhead (2-7%) and has one order of magnitude lower overhead than instrumentation based profilers such as `gprof` for call intensive programs.

We co-designed an API for user defined synchronous metrics support in `csprof`. An application can check if it is being run with `csprof` by querying a function *csprofIsActive*. If the result is true, then the application can register metrics for synchronous profiling. Using the API, one first acquires a handle for a metric from from `csprof`, then specifies a string name for the metric along with a sampling frequency. Finally, the application can record metric events by calling a `csprof` API function. At that point, `csprof` will

unwind the stack and record the calling context for the event.

Mellor-Crummey, Tallent and Zhao developed a source correlation mechanism for call path profiles and an interactive viewer. The source correlation module takes as input the performance data collected by `csprof`, and converts it into an XML file containing the calling context tree associated with the sample events. I extended the source correlation to group call sites and line samples in the same function under the same procedure frame, and extended the XML output format to represent the procedure frames. The interactive viewer, `hpcviewer`, is a Java-based viewer of the XML file produced by the source correlation phase; it displays a top-down view of the call tree, together with the metrics collected by `csprof` (cycles or user-defined metrics), and enables a user to navigate the call tree. The metrics values for the tree nodes are inclusive: the metric value for call tree node corresponding to function `foo` is the sum of the metrics for all the functions called directly or indirectly by `foo` and the metric values collected in the body of `foo`.

We extended a prototype of `hpcviewer` for analysis of call path profiles with a bottom-up view of the call tree. The bottom-up view sorts all procedures by their inclusive metric value. For a given procedure, the bottom-up view enables a user to navigate up the call tree for that procedure and also attributes how much of the procedure's cost comes from different calling contexts. For example, a procedure `foo` might be called by `A`, `B`, and `C`, with 10% of the costs attributed to calls from `A`, 20% to calls from `B`, and 70% to calls from `C`. The bottom-up view displays this kind of information and enables a user to navigate from `foo` to its calling contexts corresponding to `A`, `B`, and `C`.

11.2 Automatic Scalability Analysis

Users have specific expectations about how the performance of their code should differ among an ensemble of executions. This is true for both serial and parallel executions.

Consider an ensemble of parallel executions. When different numbers of processors are used to solve the same problem (strong scaling), we expect an execution's speedup with respect to a serial execution to be linearly proportional to the number of processors

used. When different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), we expect the execution time for all executions in an ensemble to be the same. Both types of scaling have relevant practical applications. When time to solution is critical, such as when forecasting next week's weather, then strong scaling is preferred. When fine resolution modeling is necessary, then a common practice is to choose a problem size that can be run on a single node, and then increase the number of processors while keeping the problem size on each node constant.

In each of these situations, we can put our expectations to work for analyzing application performance. In particular, we use our expectations about how overall application performance will scale under different conditions to analyze how well computation performed in each calling context scales with respect to our expectations.

To apply our approach, we first use `csprof` to profile a program under different conditions (*e.g.*, on a different number of processors or using different input sizes). Second, we clearly define our expectations and compute how much performance deviates from our expectations in each calling context in an execution's CCT recorded by `csprof`. Finally, we use an interactive viewer to explore the CCT whose nodes are annotated with the scalability metrics that we compute. The interactive viewer enables developers to quickly identify trouble spots.

While the principle of performance analysis using expectations applies broadly, in this chapter we focus on using expectations to pinpoint scalability bottlenecks in an ensemble of executions used to study strong scaling or weak scaling of a parallel application.

11.2.1 Call Path Profiles of Parallel Experiments

We analyze strong or weak scaling for an ensemble of parallel executions $E = \{E_1, E_2, ..., E_n\}$, where E_i represents an execution on p_i processors, $i = 1, n$. Let T_i be the running time of the experiment E_i.

Our calling context trees contain three types of nodes: *procedure frames*, *call sites* and *statements*. A procedure frame node can have call sites and statements as children, and it

corresponds to invoked procedures. A call site can have procedure frames as children, and corresponds to source code locations where other procedures are invoked. Statement nodes don't have any children nodes, and they correspond to samples taken during computation performed in the various procedures. The analysis we present relies on CCTs to have the same structure in parallel executions on varying number of processors. For every node m in a CCT, let $C_{p_i}(m)$ be its cost on p_i processors. In our analysis, we consider both *inclusive* and *exclusive* costs. The inclusive cost at m represents the sum of all costs attributed to m and any of its descendants in the CCT. If m is an interior node in the CCT, it represents an invocation of a function f. Its inclusive cost represents the cost of the call to f itself along with the inclusive cost of any functions it calls. If m is a leaf in the CCT, it represents a statement instance inside a call to some function. If m is a procedure frame for f, its exclusive cost includes the cost incurred in statements in f, which are its children. If m is a call site, or a statement, its exclusive cost represents the cost attributed to m alone. For a leaf procedure the inclusive cost equals the exclusive cost. It is useful to perform scalability analysis for both inclusive and exclusive costs; if the loss of scalability attributed to the inclusive costs of a function invocation is roughly equal to the loss of scalability due to its exclusive costs, then we know that the computation in that function invocation doesn't scale. However, if the loss of scalability attributed to a function invocation's inclusive costs outweighs the loss of scalability accounted for by exclusive costs, we need to explore the scalability of the function's callees.

We introduce our scalability analysis by describing scalability metrics of increasing complexity, considering the cases of strong scaling and weak scaling.

11.2.2 Simple Strong Scaling

Consider two strong scaling experiments running on 1 and p processors, respectively. Let m a node in the CCT. In the ideal case, we would expect that $C_p(m) = \frac{1}{p}C_1(m)$, or equivalently that $pC_p(m) = C_1(m)$. Often, this will not be the case, and we can measure how far we are from our expectation of ideal scaling by computing the excess work amount for

node m in the p-processor execution as $pC_p(m) - C_1(m)$. To normalize this value, we divide the excess work by the total work performed in experiment E_p, to obtain

$$SEW(m) = \frac{pC_p(m) - C_1(m)}{pT_p}$$

the fraction of the execution time that represents excess work attributed to node m.

11.2.3 Relative Strong Scaling

Consider two strong scaling experiments executed on p and q processors, respectively, $p < q$. The expected behavior in the case of ideal relative scaling would be $qC_q(m) = pC_p(m)$. To capture the departure from the expectation, we compute the excess work in the q-processor execution as $qC_q(m) - pC_p(m)$. To normalize this previous value, as before, we divide it by the total work performed in experiment E_q, to obtain

$$REW(m) = \frac{qC_q(m) - pC_p(m)}{qT_q}$$

the fraction of the execution time that represents excess work attributed to node m.

11.2.4 Average Strong Scaling

Consider an ensemble of strong scaling experiments E_1, ..., E_n. We define the fraction of execution time that represents the average excess work attributed to CCT node m as follows:

$$AEW(m) = \frac{\Sigma_{i=2}^{n}(p_iC_{p_i}(m) - p_1C_{p_1}(m))}{\Sigma_{i=2}^{n}p_iT_i}$$

Notice that for $AEW(m)$, the numerator computes excess work relative to the work performed on the smallest number of processors. We use the cost on p_1 processors rather than the cost on one processor for the following reason: for large problems, it might not be possible to solve the whole problem on a single processor. In this case, we evaluate relative scaling with respect to the execution time on the smallest number of processors on which the chosen problem size runs. The average excess work metrics are intuitive; perfect

scaling corresponds to a value of 0, sublinear scaling yields positive values, and superlinear scaling yields negative values.

When analyzing scaling, we have a choice between using average scalability over an ensemble of experiments versus using relative scalability between the parallel runs on the smallest and the largest number of processors. The advantage of the average scalability metric is that it smoothes over the performance data noise between parallel runs on different number of processors. In contrast, using relative scaling with the largest number of processors provides a quantitative explanation of all of the parallel overhead incurred. Typically, both methods provide qualitatively similar results.

Note that the simple and relative excess work metrics described in the preceding sections are simply special cases of the more general average excess work metric that we describe here.

11.2.5 Weak Scaling for a Pair of Experiments

Consider two weak scaling experiments executed on p and q processors, respectively, $p < q$. The expectation is that $C_q(m) = C_p(m)$, and the deviation from the expectation is $C_q(m) - C_p(m)$. We normalize this value by dividing it by the total work performed in experiment E_q, and define the fraction of the execution time representing excess work attributed to node m as follows

$$REW(m) = \frac{C_q(m) - C_p(m)}{T_q}$$

11.2.6 Weak Scaling for an Ensemble of Experiments

Consider an ensemble of weak scaling experiments E_1, ..., E_n. We define the fraction of execution time that represents the average excess work attributed to CCT node m as follows:

$$AEW(m) = \frac{\Sigma_{i=2}^{n}(C_{p_i}(m) - C_{p_1}(m))}{\Sigma_{i=2}^{n} T_i}$$

The same argument for relative strong scaling vs average strong scaling apply when choosing between weak scaling between a pair of experiments vs weak scaling for an ensemble of experiments.

11.2.7 Analysis Using Excess Work

The excess work metrics that we described can be computed for both inclusive and exclusive execution costs. We define $IAEW(m)$ as the *inclusive average excess work* at node m; this represents the fraction of execution time corresponding to inclusive excess work attributed to CCT node m. We define $EAEW(m)$ as the *exclusive average excess work* at node m; this represents the fraction of execution time corresponding to exclusive excess work attributed to CCT node m. Similarly, we define $IREW(m)$ as the *inclusive relative excess work* at the node m and $EREW(m)$ as the *exclusive relative excess work* at node m.

$IREW(m)$ and $EREW(m)$ serve as complementary measures of scalability of CCT node m. By using both metrics, one can determine whether the application scales well or not at node m, and also pinpoint the cause of any lack of scaling. If a function invocation m has comparable positive values for $IREW(m)$ and $EREW(m)$, then the loss of scalability attributed to the inclusive costs of m is roughly equal to the loss of scalability due to its exclusive costs and we know that the cost of m doesn't scale. However, if the loss of scalability attributed to m's inclusive costs outweighs the loss of scalability accounted for by its exclusive costs, we need to explore the scalability of m's callees. To isolate code that is an impediment to scalable performance, one simply navigates down CCT edges from the root of the tree to trace down the root cause of positive $IREW$ values. A strength of this approach is that it enables one to pinpoint impediments to scalability, whatever their underlying cause (*e.g.*, replicated work, communication, etc.). We can perform a similar analysis using the $IAEW(m)$ and $EAEW(m)$ metrics.

11.2.8 Automating Scalability Analysis

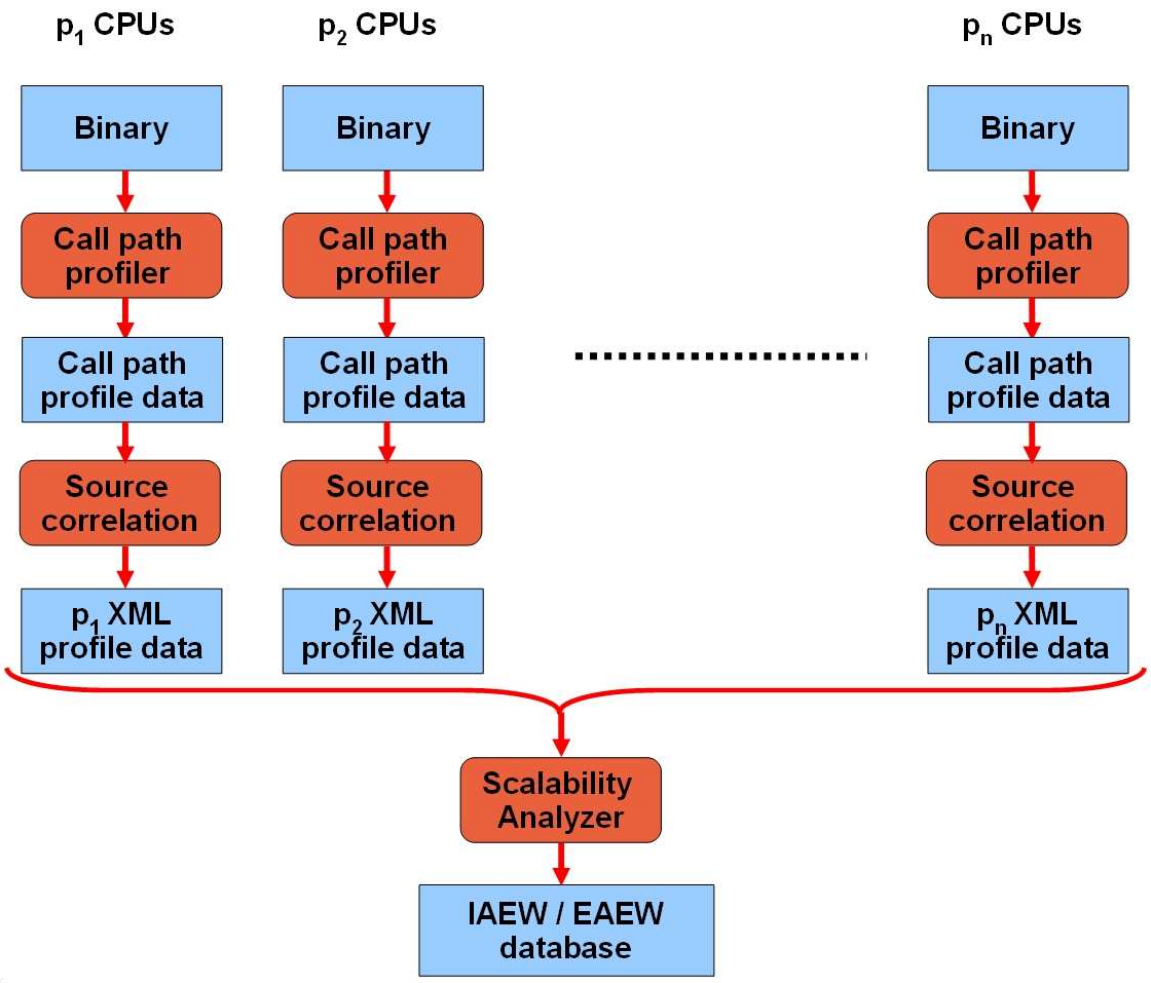
We prototyped tools to support automatic scalability analysis by building upon components of Rice University's HPCToolkit performance analysis tools [82, 136, 168] `csprof`, `xcsprof`, and `hpcviewer`. `csprof` was designed as a profiler for node programs; for parallel programs, we use `csprof` to collect a node profile for each process in a parallel execution. `xcsprof` is used to post-process a raw call path profile collected by `csprof`, correlate it with the application's source code, and produce an XML representation of a calling context tree annotated with performance metrics. `hpcviewer` is Java-based user interface that provides users with a top-down interactive and navigable view of a calling context tree, along with associated performance metrics and program source code.

In Figure 11.2, we present the process by which the $IAEW$ and $EAEW$ metrics are computed: call path profiles are collected for each process of a parallel execution using `csprof` on p_1, p_2, ..., p_n processors. The resulting profile data is then correlated with the source code and converted to XML format using `xcsprof`. Next, we collate the XML data from all experiments and compute the $IAEW$ and $EAEW$ scalability scores. Finally, a performance analyst can use `hpcviewer` to interactively explore a calling context tree annotated with both measured execution costs and the scalability metrics we compute. The $IREW(m)$ and $EREW(m)$ metrics are computed using a similar process.

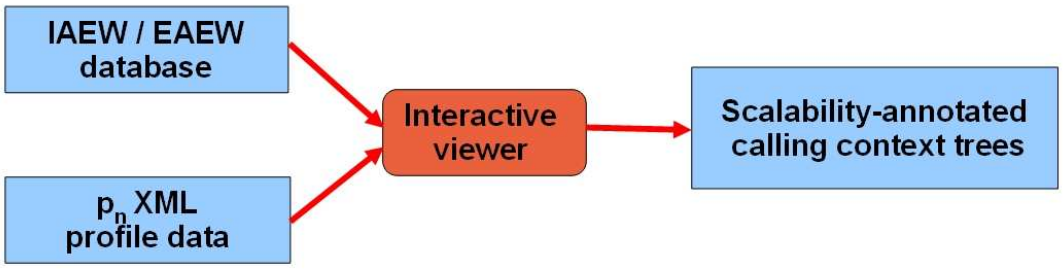
11.3 Experimental Methodology

For the analysis performed in this chapter, we used two types of metrics. One was the sampling-based number of cycles metric. The other consisted of user defined metrics; we instrumented `cafc`'s runtime using `csprof`'s API for monitoring synchronous events to register and then record the following metrics:

- number and volume of `PUTs`
- number and volume of `GETs`



(a) Process for performing scalability analysis using call path profiles.



(b) Process for visualizing call path based scalability analysis.

Figure 11.2: Processes for computing and displaying the call path-based scalability information.

- number of notify and waits
- number of barriers

We analyzed the parallel scalability for MPI and CAF versions of several benchmarks using two analysis techniques. The first type of analysis was semi-automatic and focused on understanding the impact of scalability of particular communication primitives on strong scaling. We determined the total time spent in each communication primitive of interest, then plotted the relative cost of communication and computation time as a function of the number of processors. The computation cost was computed as the difference between the total execution time and the total communication time. If the time spent in a particular communication primitives does not decrease proportional to the increase in number of processors, the performance of primitive is non-scalable. We leveraged `hpcviewer`'s bottom-up view to determine which call site or programming idiom was responsible. We were inspired by Bridges *et al* [39] to use stacked charts of relative costs to evaluate scalability of communication primitives and communication. Communication primitives whose relative cost increases with a growing number of processors point to scaling problems. Note that if computation scales ideally, then the relative cost of communication indicated by the layered charts would coincide with the excess work for each parallel experiment. However, in practice, for strong scaling applications the computation cost does not scale linearly with the number of processors, so the total cost of communication as indicated by the layered charts is usually an underestimation of the excess work.

The second type of analysis was the automatic expectations-based scaling analysis, which computed the excess work metrics for all nodes in the calling context tree of an application. Using `hpcviewer` we determined which functions were responsible for the lack of scalability, and whether any non-scalability was due to communication or computation.

Typically, parallel scientific codes include a initialization phase, a timed phase for which results are reported and which is the target of optimization, and a reporting and clean-up phase. It is important to note that our scaling analysis methods operate on the

complete application execution, and we report lack of scalability that could be part of any phase. Our scaling results cannot always be used for a straightforward quantitative performance comparison of different versions of the same algorithm, e.g. an MPI version vs a CAF version, but could be used to provide insight into scaling problems of each application under consideration.

The CAF codes we study were compiled and run with `cafc` using the ARMCI library. For the MPI codes we study, we analyze the cost of computation and that of the MPI primitives. For the CAF experiments, we focus on the cost of ARMCI primitives. We are also interested in determining the overhead incurred by using `csprof` to profile the parallel codes.

The experiments presented in this section were performed on a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900 MHz Intel Itanium 2 processors with 32KB/256KB/1.5MB of L1/L2/L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel compilers V9.0 as our back-end compiler. We used one CPU per node for our experiments.

For all the benchmarks analyzed we focused on small problem sizes, which tend to expose lack of scalability due to communication and synchronization inefficiencies on small number of processors. In the remaining of this chapter we present experimental results for which we gain insight using our scaling analysis. A comprehensive description of the rest of our scaling analysis experiments is given in Appendix A.

11.4 Experimental Results

11.4.1 Analysis of LANL's POP Application

An attractive scaling analysis target is represented by the vast amount of MPI applications. We analyzed the version 2.0.1 of the Parallel Ocean Program (POP) [124, 125], which uses MPI to communicate data. POP is an ocean circulation model in which depth is used as the

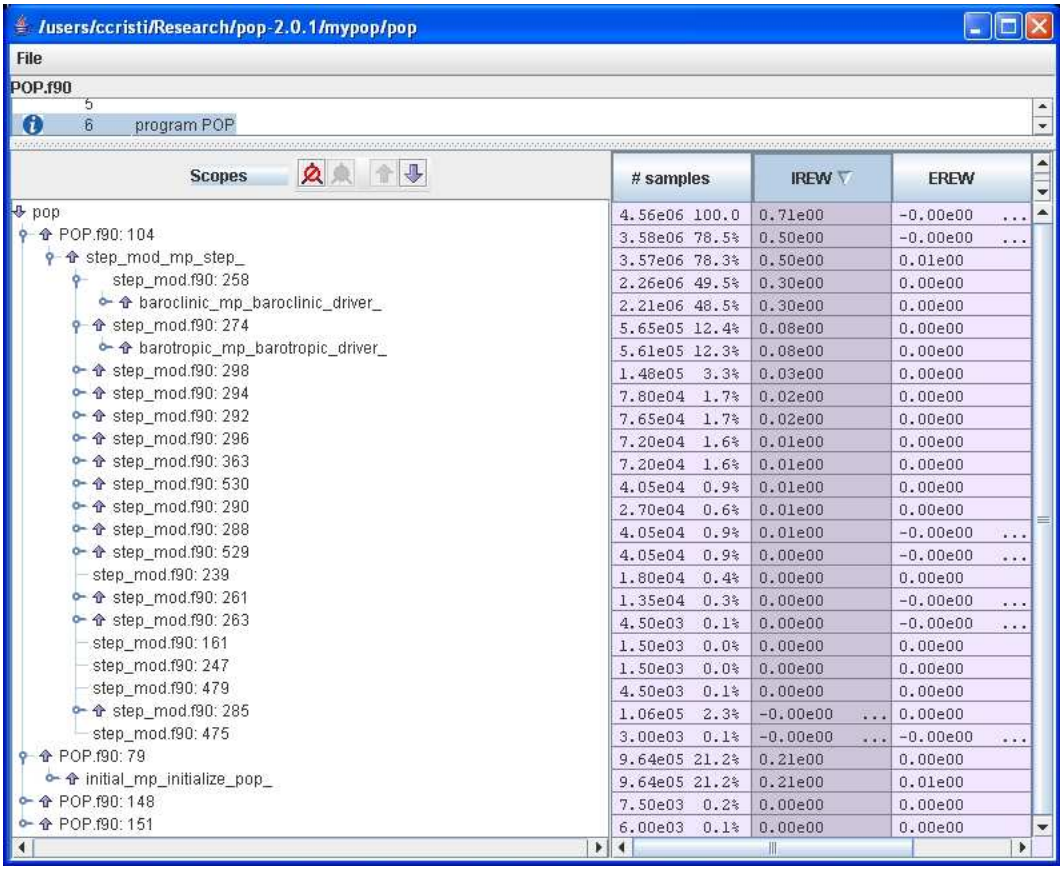


Figure 11.3: Screenshot of strong scaling analysis results for POP, using relative excess work, on 4 and 64 CPUs.

vertical coordinate. The model solves the three-dimensional primitive equations for fluid motions on the sphere under hydrostatic and Boussinesq approximations. Spatial derivatives are computed using finite-difference discretizations which are formulated to handle any generalized orthogonal grid on a sphere, including dipole and tripole grids which shift the North Pole singularity into land masses to avoid time step constraints due to grid convergence.

We analyzed POP for a “large” test domain, with 384x288 domain size, 32 vertical levels, and 2 tracers. We present scaling analysis results using relative excess work on 4 and 64 CPUs in Figures 11.3 and 11.4; we present the scaling analysis results using

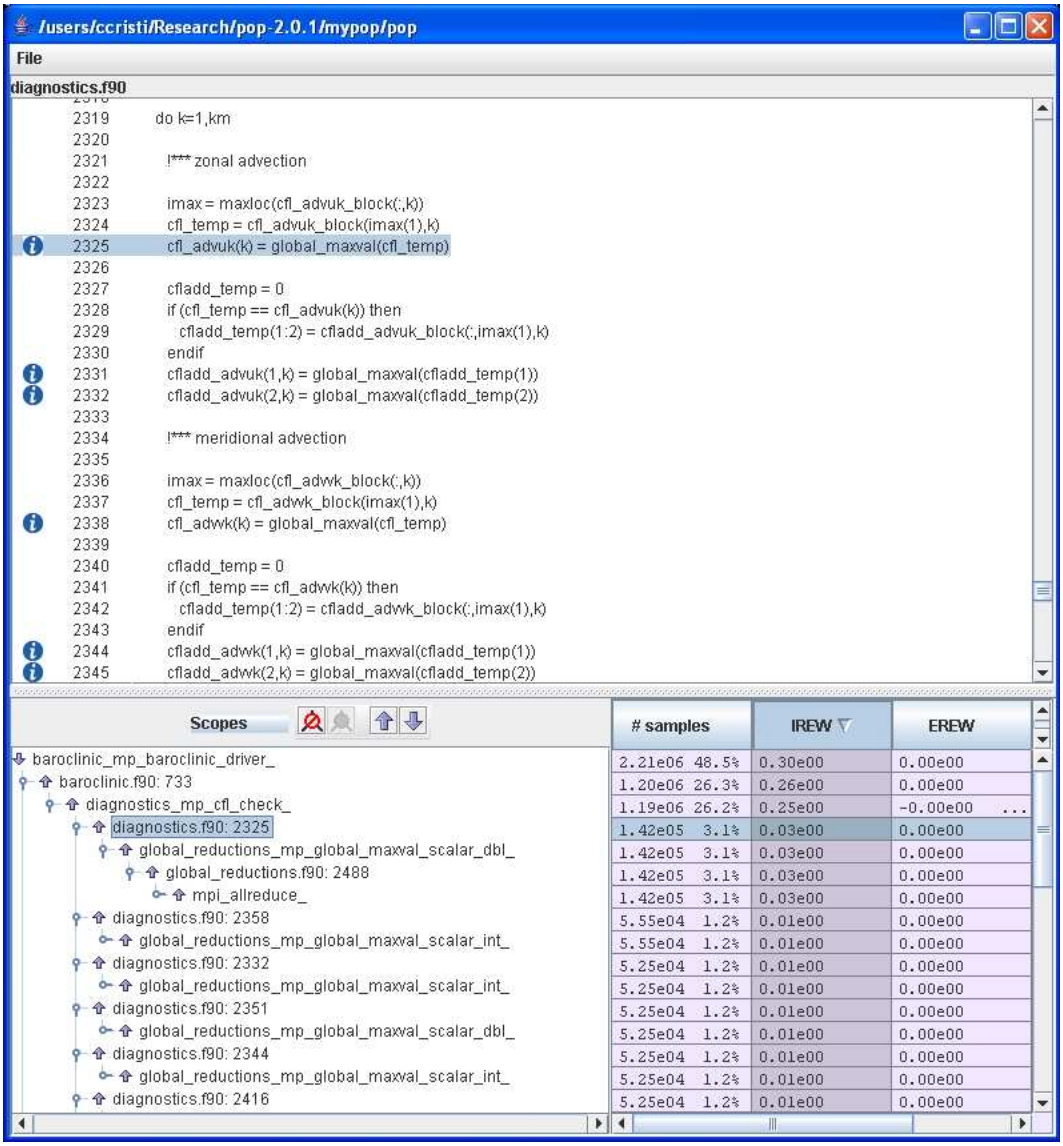


Figure 11.4: Screenshot of strong scaling analysis results for POP, for the baroclinic module, using relative excess work, on 4 and 64 CPUs.

average excess work for an ensemble of executions on 4, 8, 16, 24, 32, 36, 40 , 48, and 64 CPUs in Figures 11.5 and 11.6. The results obtained with the relative excess work are qualitatively similar to those obtained using the average excess work; however, the relative excess work obtained using the minimum and maximum number of CPUs emphasizes the program behavior on the largest number of CPUs.

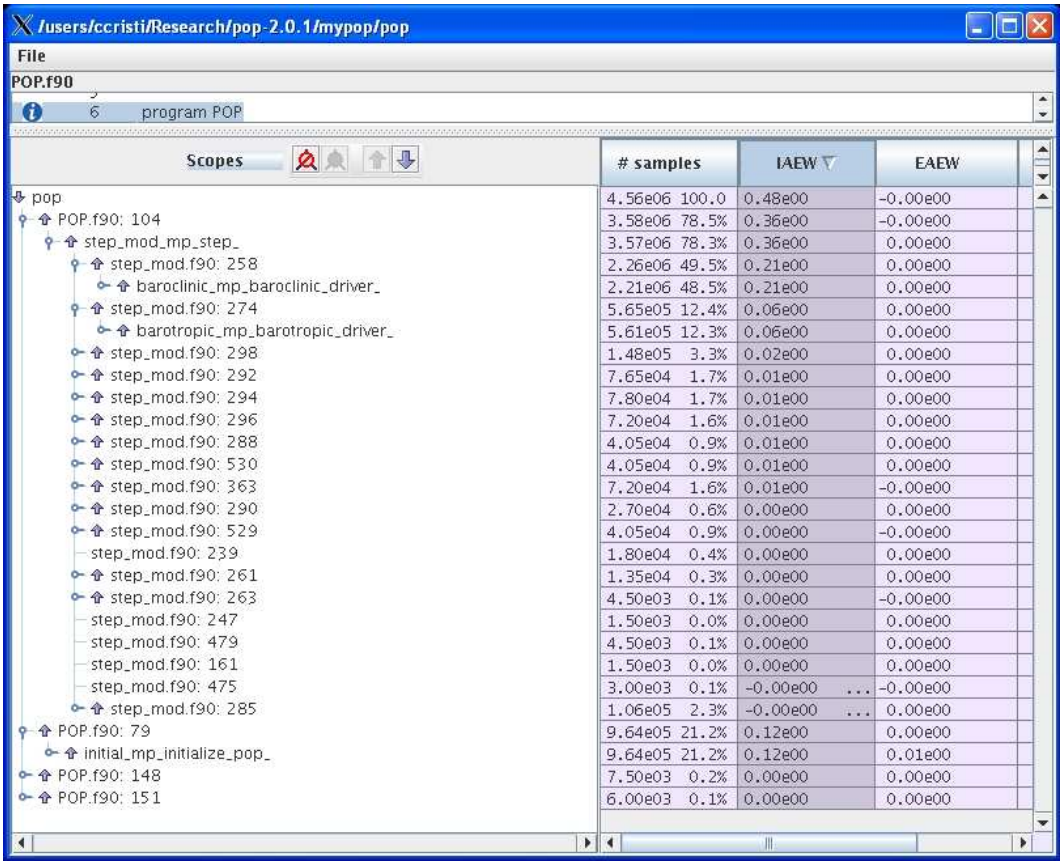


Figure 11.5: Screenshot of strong scaling analysis results for POP, using average excess work, for an ensemble of executions on 4, 8, 16, 24, 32, 36, 40, 48, and 64 CPUs.

The relative excess work results for 4 and 64 CPUs show that the the main program loses 71% efficiency, with 53% due to the time step routine, and 21% due to the initialization routine. The time step costs are further discriminated as 33% due to the baroclinic module, 8% due to the barotropic module, and other functions with smaller costs. Within the baroclinic driver, the routine `diagnostics_mp_cfl_check` is responsible for 25% loss of scalability; we show the scaling analysis for this routine in Figure 11.4.

The average excess work results on 4, 8, 16, 24, 32, 36, 40 , 48, and 64 CPUs showed that the main program displays 48% loss of scaling, out of which 36% are due to the time step routine, and the remaining 12% are due to initialization routine. The time step costs

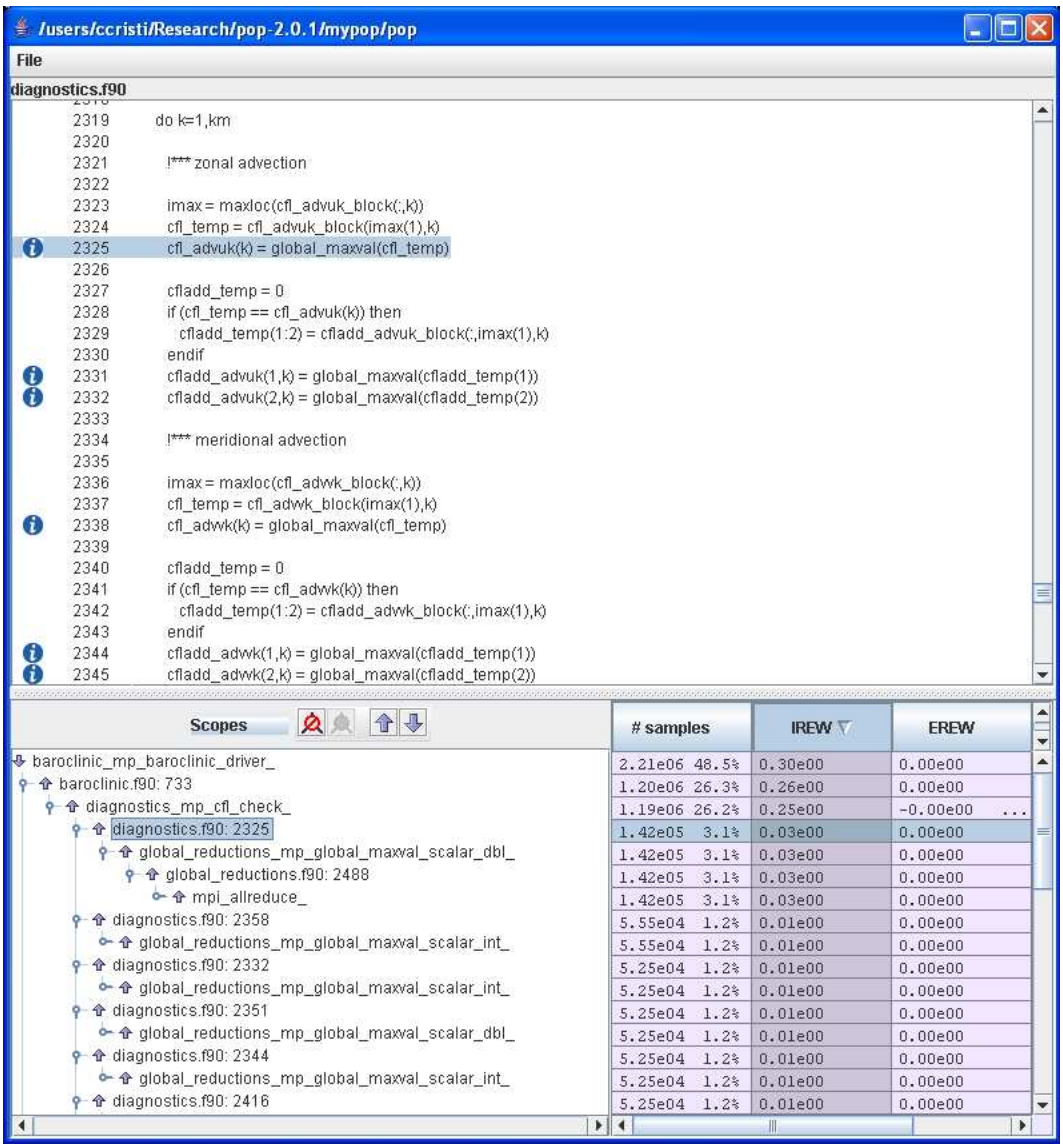


Figure 11.6: Screenshot of strong scaling analysis results for POP, for the baroclinic module, using average excess work, for an ensemble of executions on 4, 8, 16, 24, 32, 36, 40, 48, and 64 CPUs.

are split between the baroclinic module, with 21%, the barotropic module, with 6%, and other functions with smaller costs. Within the baroclinic driver, we observed an 18% loss of scalability due the routine `diagnostics_mp_cfl_check`; we present the scaling analysis results for this routine in Figure 11.6.

For both sets of results, we can notice that lack of scaling is due to multiple calls to the routine `global_reduction_maxval_scalar_dbi`. By using source code correlation, we discovered that for each of the vertical levels, POP performs multiple scalar reductions. This deficiency can be addressed by aggregating the reductions, and we found the interactive viewer of the annotated call tree to be extremely effective in pinpointing this scaling bottleneck quickly.

11.4.2 Analysis of the NAS MG Benchmark

The MG multigrid kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [24]. The CAF version of NAS MG is described elsewhere [73]. In Figure 11.7 we present the scalability of relative cost of communication primitives and computation for the CAF version of NAS MG; the overall excess work indicated by the layered chart is 82%. In Figure 11.8 we present a summary of the user-defined metrics for the volume of communication and synchronization. The profiling overhead was of 4-7% for the CAF MG experiments.

By analyzing the scalability of communication primitives and computation for CAF NAS MG, we determined that the relative cost of the `ARMCI_Get` and `ARMCI_Barrier` primitives increases as the number of processors increases. By using the bottom-up view, we determined that both primitives are used in inefficient user-written implementation of reductions such as sum and maximum. Original CAF source-level implementations of collective operations, which we received from Robert Numrich, were developed on Cray systems and used barriers. For example, a sum reduction for double precision vectors was implemented as follows. Let $m = \lceil \log(num\_images()) \rceil$. Next, a barrier is performed, after which each of the process images i, with $i = 1, m$ computes the partial sum reduction by getting and adding the corresponding vector of elements from process images i, $i + m$, $i + 2m$, ... A barrier is called again, after which process image i, with $i = 1, m$, gets the partial sums from process images 1, 2, ..., $i - 1$, $i + 1$, ..., m. A barrier is called again,

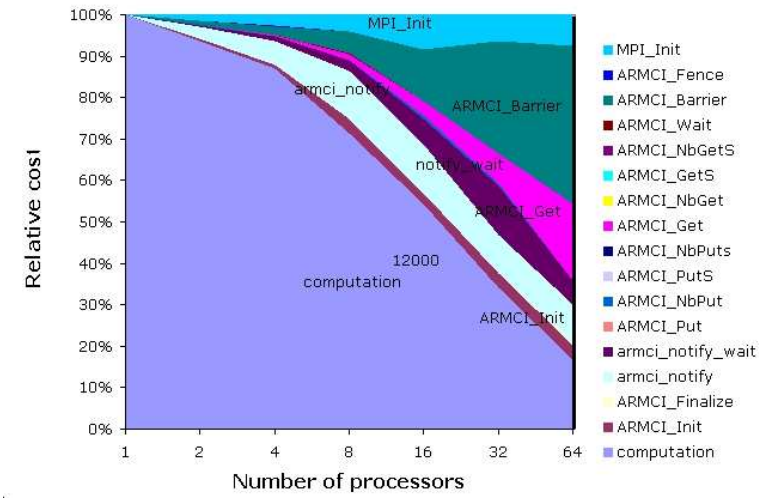


Figure 11.7: Scalability of relative costs for communication primitives and computation for the CAF version of the NAS MG benchmark class A (size 256^3).

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barriers
1	0	0	93	1064	0	0	292
2	714	51031104	95	1084	1428	1428	292
4	714	32018592	95	1084	1428	1428	292
8	714	19297376	95	1084	1428	1428	292
16	724	12939008	95	1084	1438	1438	292
32	734	8152464	95	1084	1448	1448	292
64	744	4938104	95	1084	1458	1458	292

Figure 11.8: Communication and synchronization volume for the CAF version of NAS MG, class A (size 256^3).

after which the remaining process images read the overall sum from one of the first m process images, such that process image j reads the sum from process image $1+mod(j, m)$, for $j = m + 1$,*num_images()*. These reductions implementations do not yield portable performance, since they are not efficient on clusters.

Even though for MG these reductions occur in the initialization phase, which is not

measured and reported in the timed phase of the benchmark, it points to a problem: the lack of collective communication at the language level leads users to write an implementation of these primitives that does not deliver portable high-performance.

By inspecting the communication and synchronization volume results presented in Figure 11.8, we noticed that the number of barriers is constant from 2 to 64 processors. However, since the computation volume decreases, it means that the relative importance of barriers (and reductions using them) increases. It is therefore crucial to have good support for collective operations.

Figures 11.9 and 11.10 show screenshots with results of strong scaling analysis using relative excess work for the CAF version of NAS MG, on 1 and 64 processors. The results in Figure 11.9 show that the $IREW$ for the main routine `rice_mg_caf` is 82%, out of which 44% is due to calls to `zran3`, 16% to the central timed routine, `mg3p`, 12% is due to a call to `cafinit_`, 9% to calls to `resid`, 4% to a call to `mg3p` in the initialization phase, 3% to the routine `cafglobalstartupinit_` , 3% to a call to `caf_all_max_dp`, 2% to calls to `norm2u3`, and 1% to a call to `caf_bcast_i` in the initialization phase. For `cafinit`, which is called when launching CAF programs, 10% of $IREW$ is due to a call to `MPI_Init` and 2% to a call to `ARMCI_Init`; these routines initialize the MPI and the ARMCI libraries, respectively. In Figure 11.10 we analyze top-down the routine `zran3`. By explaining why 44% of IREW is attributed to `zran3`, which randomly initializes the work array with a combination of zero and one values, we find that the loss of scalability is due to calls to `caf_allmax_i_psbody`, `caf_allmax_dp_psbody`, `caf_allmin_dp_psbody`, which are suboptimal implementations of minimum and maximum element reductions for integer and for double precision arrays. Within the implementation of these reductions, we find that the largest excess work amount is due to the use of `ARMCI_Barrier`. The routine `zran3` is called only in the initialization part of the benchmark and is not part of the benchmark's timed result; however, poor scaling for this routine hurts the scalability of the NAS MG program and consequently yields an inefficient use of the target parallel system.

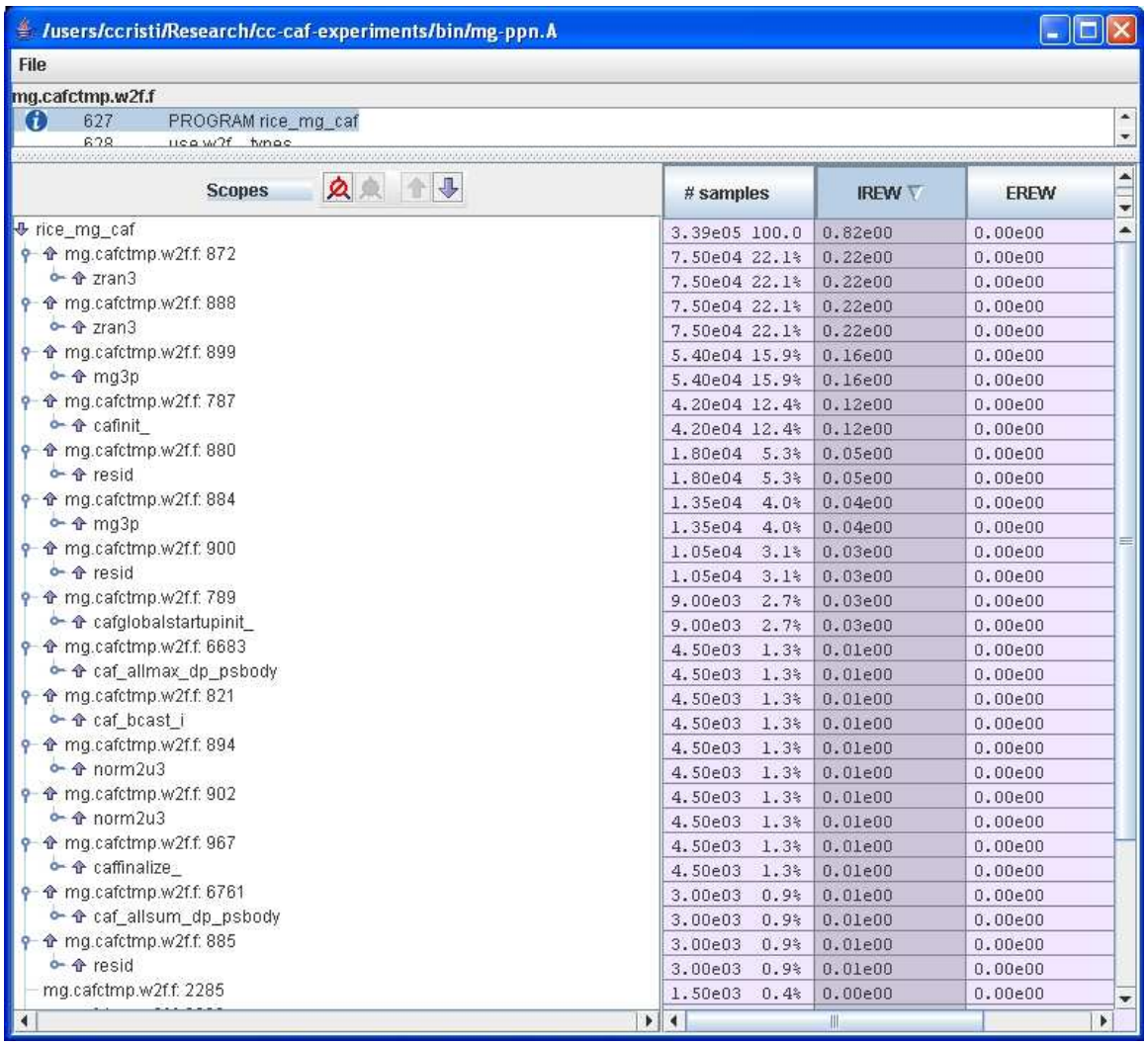


Figure 11.9: Screenshot of strong scaling analysis results for CAF MG class A (size 256^3), using relative excess work on 1 and 64 processors.

By using the top-down analysis with scalability information attributed to the calltree nodes, we obtained information similar to what we obtained using the communication primitives plots and the bottom-up view. Our scaling analysis based on expectations is also more detailed, since it can display calling contexts and it can also show line level detail.

In Appendix B, we present a proposed extension of CAF with collective operations

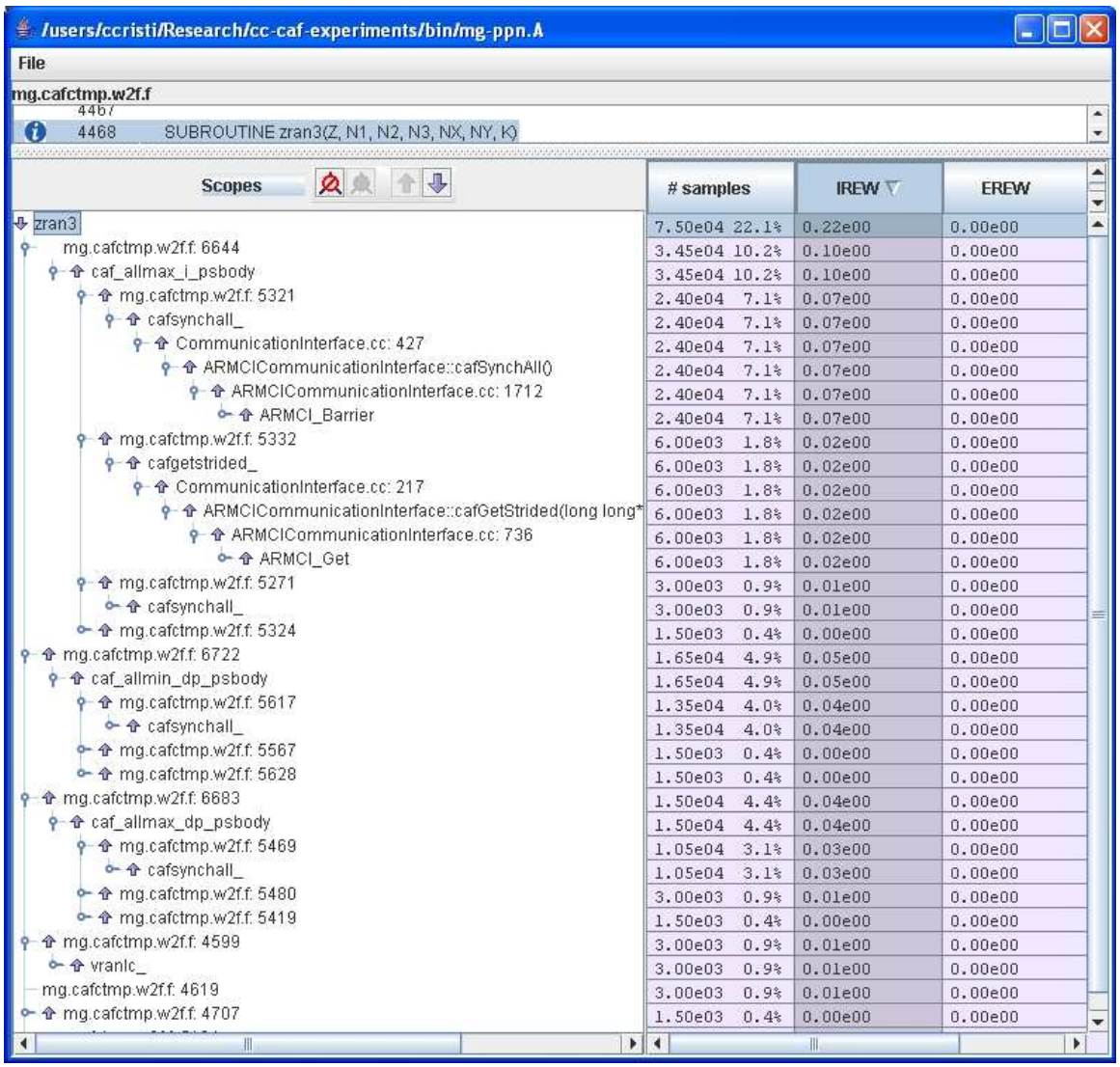


Figure 11.10: Screenshot of strong scaling analysis results for CAF MG class A (size 256^3), using relative excess work on 2 and 64 processors, for the routine `zran3`.

at language level and an MPI implementation strategy. For CAF MG we were able to reduce the initialization time by 40% on 64 CPUs by using our collective operations CAF extensions.

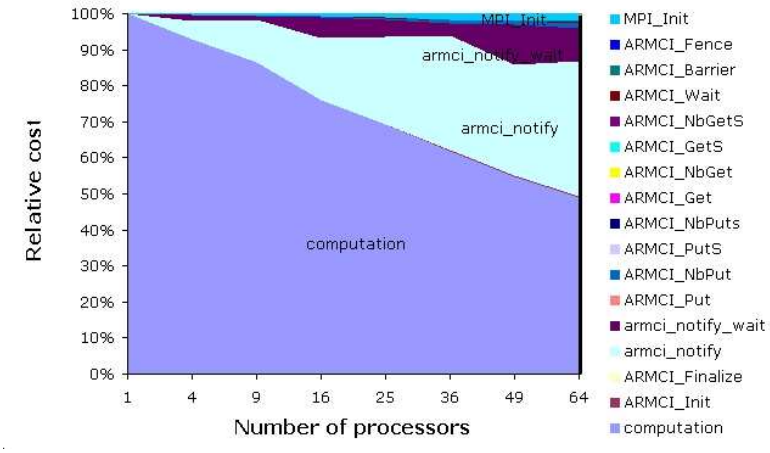


Figure 11.11: Scalability of relative costs for communication primitives and computation for the CAF version of the NAS SP benchmark class A (size 64^3).

11.4.3 Analysis of the NAS SP Benchmark

NAS SP is a simulated CFD application that solves systems of equations resulting from an approximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [24]. SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [24]. The CAF version of SP was described in Section 6.3. In Figure 11.11 we present the scalability of relative costs for communication primitives and computation for the CAF version of NAS SP; the excess work indicated by the layered chart is 51% on 64 CPUs. The profiling overhead was 1-8% for CAF NAS SP.

The results in Figure 11.11 show that as the number of processors grows, the cost of `sync_notify` becomes significant. Using the bottom-up view of `hpcviewer` we determined that 27% of the `sync_notify` cost on 64 CPUs is due to the calls in the routine `copy_faces`. The cause for this cost is the implementation of the `sync_notify` semantics: a notify to an image Q from P is received by Q only after all communication initiated by P to Q has completed. In practice, this means that before issuing the notify, image P polls until all `PUT`s from P to Q have completed, thus exposing the latency of commu-

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barriers
1	0	0	8	188	4423	2412	25
4	4818	493548288	5	104	9643	9642	27
9	7224	440017888	5	104	14455	14454	27
16	9630	372426064	5	104	19267	19266	27
25	12036	321252016	5	104	24079	24078	27
36	14442	280289968	5	104	28891	28890	27
49	15120	216247680	51	212	30252	30246	17
64	19254	218391120	5	104	38515	38514	27

Figure 11.12: Communication and synchronization volume for the CAF version of NAS SP, class A (size 64^3).

nicating the data. A solution to this problem would be to have support for non-blocking synchronization while maintaining the same semantics, e.g. after issuing a `sync_notify` the sender process images continues execution; however, the destination image would still receive the notify after the completion of communication issued by the sender process. Currently such support is missing from both ARMCI and GASNet libraries. 47% of the `sync_notify` cost on 64 CPUs is due to the "handshakes" necessary for communication in the sweep routines, `x_solve`, `y_solve` and `z_solve`. Notice that the number of `sync_notify`s and `sync_wait`s is slightly more than double the number of `PUT`s. This is due to the fact that the CAF version was adapted from the MPI version, and we used a basic pattern of conversion from 2-sided communication in the MPI version to the one-sided programming model of CAF. An MPI send/receive communication pair such as that presented in Figure 11.13(a) is replaced with the code shown in Figure 11.13(b).

In Figure 11.12, we present a summary of the user-defined metrics for the volume of communication and synchronization. The communication volume summary results show that the number of `PUT`s increases as $P^{\frac{3}{2}}$, where P is the number of processors. This is due to the multipartitioning distribution. The number of handshakes increases with the number

```
P:
 call MPI_send

Q:
 call MPI_recv
```

a) Two-sided communication

```
P:
   call sync_wait(Q)
   ... put to Q ...
   call sync_notify(Q)

Q:

   call sync_notify(P)
   call sync_wait(P)
   ... consume the data Q ...
```

b) One-sided communication

Figure 11.13: Basic pattern of conversion from two-sided message passing communication in MPI into one-sided communication in CAF.

of processors by the same function, with a multiplicative factor of two. The majority of the notifies' cost is due to the blocking implementation of notifies used to signal the completion of the PUTs from P to Q.

A previous study [57] identified the conversion of MPI 2 sided communication into one-sided communication as a problem, and suggested the use of multiversion buffers as a solution for the exposed latency while waiting for the remote buffer to become available for remote write. For NAS SP, during the forward substitution phase, the waiting for the buffer on Q to become available for writing represents only 3% of the forward substitution time. For the backward sweep, the waiting for the remote buffer to become available takes up to 17% of the backward sweep time. This suggests that using multiversion buffers might benefit more the backward substitution phase.

Figures 11.14 and 11.15 show screenshots of the strong scaling analysis results for the CAF version of NAS SP on 4 and 64 CPUs. The results in Figure 11.14 show that the value of $IREW$ for the main routine `mpsp` is 53%; this is slightly higer than the excess work of 51% indicated by the layered chart in Figure 11.11, due to poor scaling of local computation. The non-scalability is explained by the `adi` routine, which performs alternate direction integration, with a metric of 51%. In Figure 11.15, we analyze the scalability of `copy_faces`, which exchanges overlap values between cells. `copy_faces` has a $IREW$ score of 16% and an $EREW$ score of 4%; this means that the computation

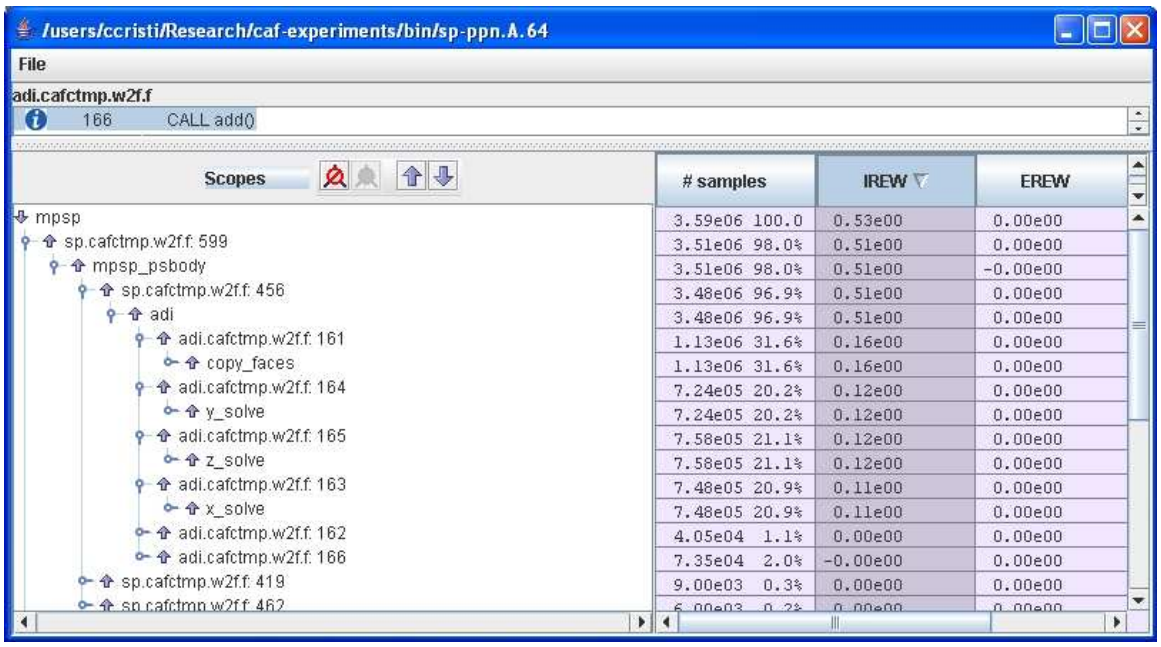


Figure 11.14: Screenshot of strong scaling analysis results for the CAF version of NAS SP class A (size 64^3), using relative excess work on 4 and 64 CPUs.

in `copy_faces` is also responsible for scalability loss. By investigating the call sites in `copy_faces`, we notice that a call to `notify` has an $IREW$ scores of 9%. This is consistent with the communication primitives scalability results, which pointed to `notify` as a major non-scalable part of the communication costs.

11.4.4 Analysis of the NAS CG Benchmark

To evaluate the applicability of the expectations-based scaling analysis to UPC codes, we analyzed the UPC version of NAS CG. The CG benchmark uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [24]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The UPC version of NAS CG was described in Section 7.4.

In Figure 11.16, we present a screenshot of the scaling analysis results of UPC NAS

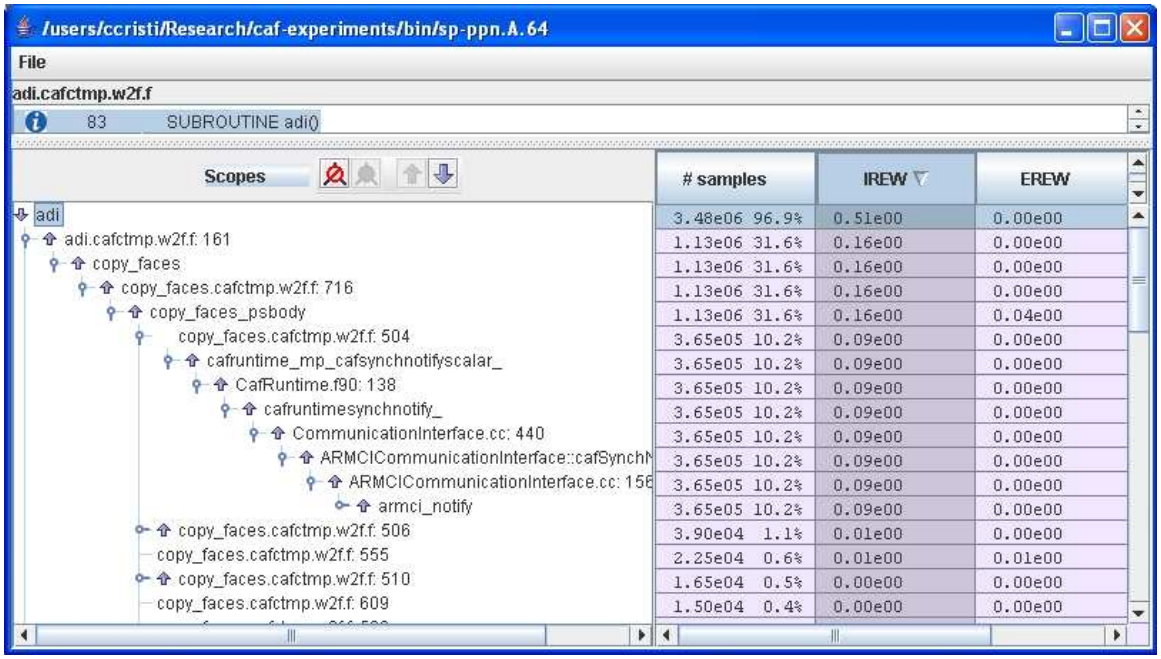


Figure 11.15: Screenshot of strong scaling analysis results for the CAF version of NAS SP class A (size 64^3), using relative excess work on 4 and 64 CPUs, for the routine `copy_faces`.

CG, using relative excess work on 1 and 16 CPUs. The main program loses 44% efficiency, out of which the `conj_grad` routine, which performs conjugate gradient computation, accounts for 37% loss of scalability. By further analyzing the calling context tree, we determined that two calls to `reduce_sum` costed 15% and 5%, respectively, and that a call to `upcr_wait` accounted for 6% loss. The source code correlation showed that `reduce_sum` has a suboptimal implementation, using barriers; a solution would be to employ one of the UPC collective operations.

11.4.5 Analysis of the LBMHD Benchmark

We described the MPI implementation of LBMHD in Section 3.2.2 and described our CAF implementation decisions in Section 8.2. In Figure 11.17 we present the scaling of relative cost for communication primitives and for computation scalability for the CAF version of

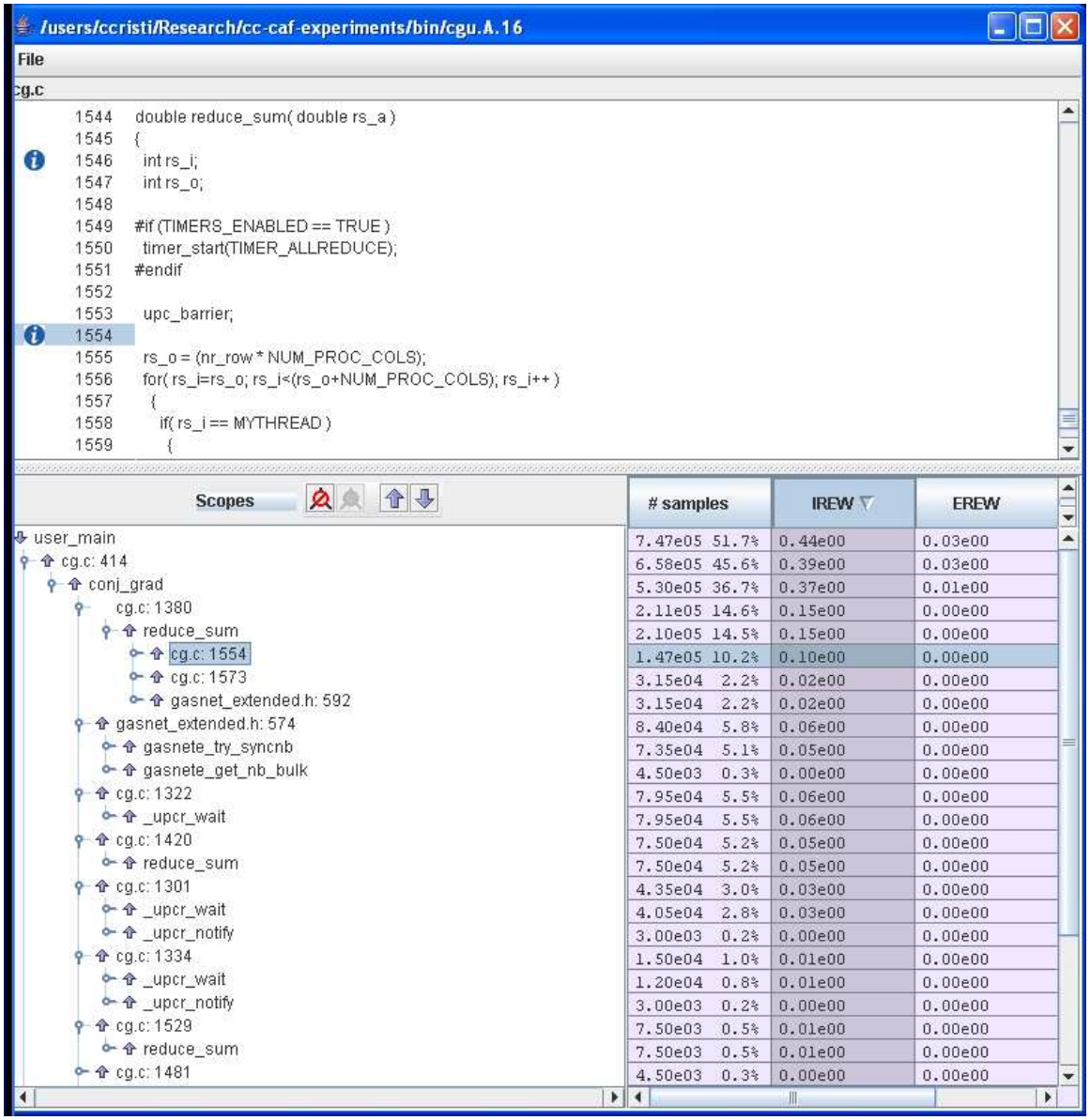


Figure 11.16: Screenshot of strong scaling analysis for UPC NAS CG class A (size 14000), using relative excess work on 1 and 16 CPUs.

LBMHD. The overall loss of efficiency on 64 CPUs indicated by the layered chart is of 39%. In Figure 11.19 we present a summary of the user-defined metrics for the volume of communication and synchronization. The profiling overhead was of 7-16% for CAF LBMHD.

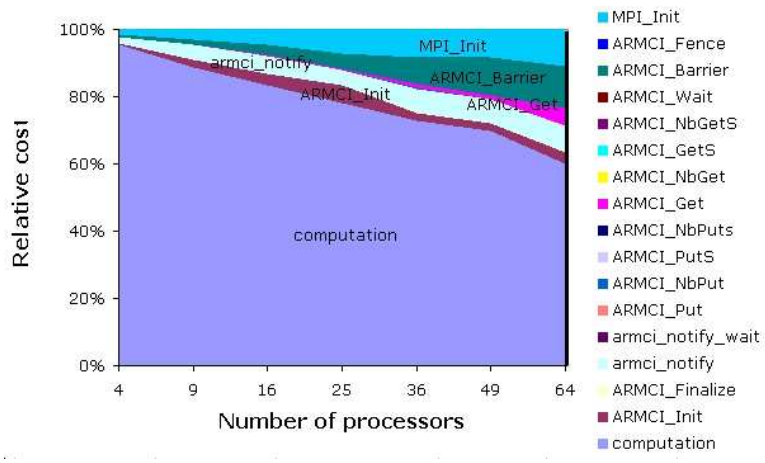


Figure 11.17: Scalability of relative costs for communication primitives and computation for the CAF version of the LBMHD kernel, size 1024^2.

The relative costs scalability graph show that the cost of barriers for the CAF version increases with the number of CPUs. Figure 11.19 shows that as the number of CPUs increases, the volume of PUTs per process image decreases, but the number and volume of GETs and the number of barriers stay constant. Both GETs and barriers were used to implement reductions at the source level in the original LBMHD source code that we received from LBNL. The CAF implementation was performing three consecutive reductions on scalars. We first replaced the three scalar reductions with a vector reduction defined at language level as described in Section 11.4.2; that solution was suboptimal since the vector reduction used multiple barriers. By replacing the three scalar reductions with a three-element MPI vector reduction, performance improved by 25% on 64 processors, as shown in Figure 11.18, that presents parallel efficiency for the timed phases of MPI and CAF versions of the LBMHD benchmark. As we mentioned in Section 11.3, the excess work indicated by the layered charts and computed by the automated scaling analysis applies to the entire application, not just the timed phase of it. This results points that it is important to use the appropriate collective primitives, but also to the need for efficient reduction support at the language level.

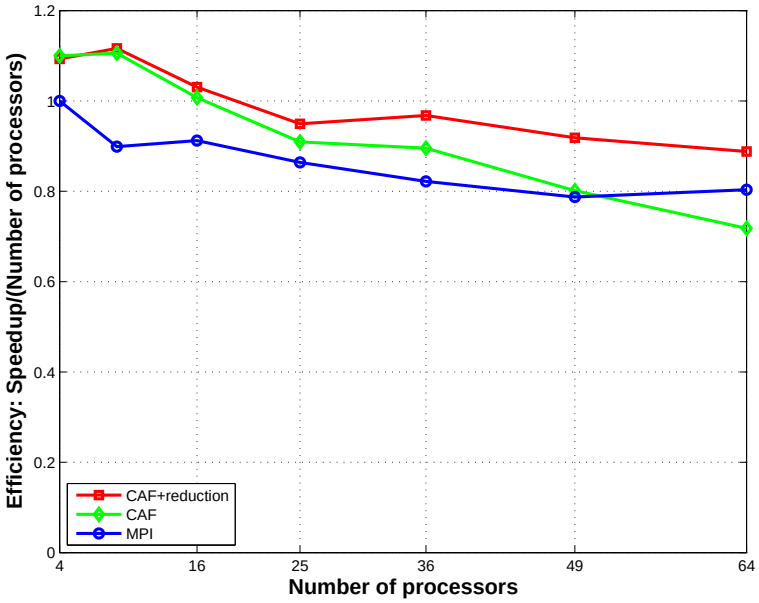


Figure 11.18: Parallel efficiency for the timed phases of MPI and CAF variants of the LBMHD kernel on an Itanium2+Myrinet 2000 cluster.

In Figures 11.20 and 11.21, we present screenshots with results of strong scaling analysis for CAF LBMHD, using relative excess work, on 4 and 64 CPUs. The results in Figure 11.20 show that the $IREW$ score for the main routine `mhd` is 53%. The routine `decomp`, which performs the initial problem decomposition, has both $IREW$ and $EREW$ scores of 14%, which means that the lack of scalability is due to local computation. `cafinit` has a $IREW$ score of 10%, caused by `MPI_Init` and `ARMCI_Init`. The routine `stream` has an $IREW$ score of 9%. The routine `caf_allsum_dp` has $IREW$ scores of 6%, 6%, and 5%. This points to the inefficiency of user handcoded reductions, similar to the lesson learned from the bottom-up semiautomatic analysis. In Figure 11.21, we present the analysis results for the routine `stream`. The main routine contributing to the non-scalability score of `stream` is `neighbors`, which updates the ghost cells of the neighbors; `neighbors` has a value of 7% for $IREW$ and 1% for $EREW$. Within `neighbors`, one calls to `notify` has an $IREW$ score of 1% and three calls to `notify` have a score of 1% each. Note that the overall excess work of 53% is significantly higher

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barriers
4	200	22195200	33	264	200	200	107
9	200	14808000	33	264	404	404	107
16	200	11136000	33	264	404	404	107
25	200	8889600	33	264	404	404	107
36	200	7420800	33	264	404	404	107
49	200	6384000	33	264	404	404	107
64	200	5606400	33	264	404	404	107

Figure 11.19: Communication and synchronization volume for the CAF version of LBMHD (size 1024^2).

than the excess work of 39% indicated by the layered chart in Figure 11.17; this is due to poor scaling of local computation, such as the one performed by the routine `decomp`. In Appendix B we present a proposed extension of CAF with collective operations primitives at language level and an MPI-based implementation strategy. By using the CAF extensions, we were able to achieve an improvement of LBMHD of 25% on 64 processors, and the translation to MPI collectives didn't introduce significant overhead.

11.4.6 Analysis of a MILC Application

MILC [25] represents a set of parallel codes developed for the study of lattice quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics. These codes were designed to run on MIMD parallel machines. They are written in C, and they are based on MPI. MILC is part of a set of codes used by NSF as procurement benchmarks for petascale systems [152, 153]. The latest version of MILC, 7 as of the time of this writing, uses the SciDAC libraries [4] to optimize the communication in the MILC application. We present an analysis of the version 7.2.1 of MILC using MPI as communication substrate. Our goal is to demonstrate the applicability of our method to MPI-based codes that are used with weak scaling.

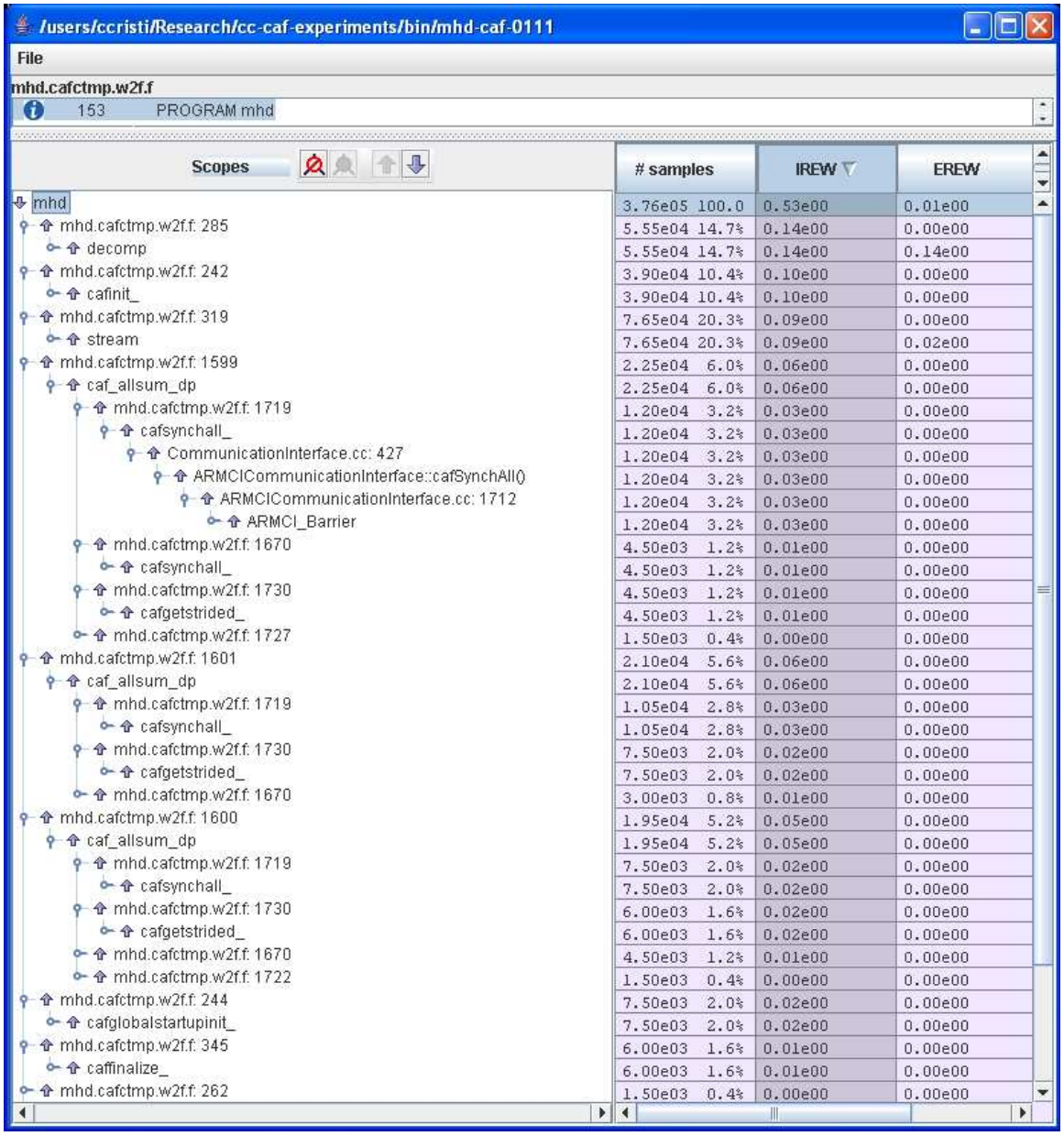


Figure 11.20: Screenshot of strong scaling analysis results for CAF LBMHD (size 1024^2), using relative excess work, on 4 and 64 CPUs.

From the MILC codes we analyzed the su3_rmd application, which is a Kogut-Susskind molecular dynamics code using the R algorithm. We chose our input sizes so that as we increased the number of processors, the work on each processor remains constant. The

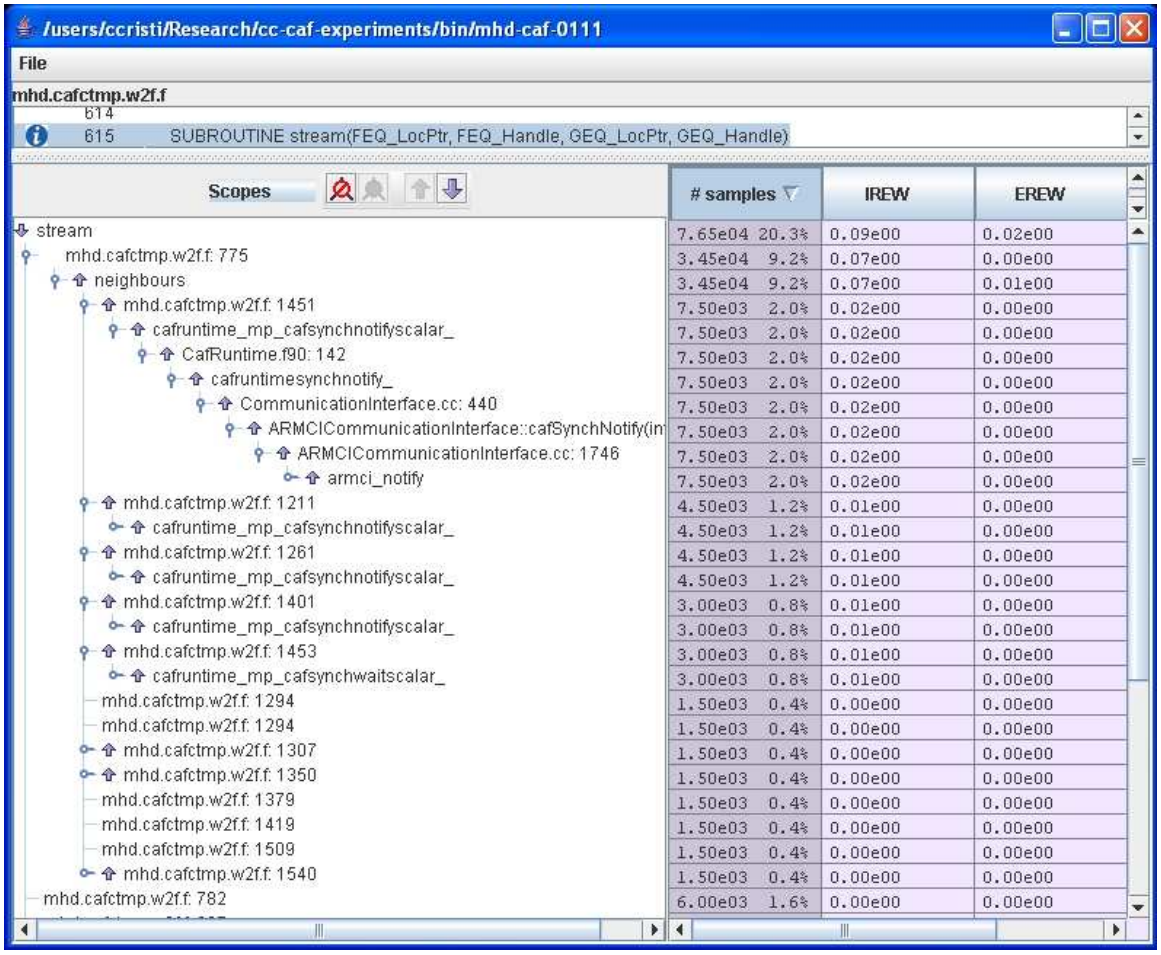


Figure 11.21: Screenshot of strong scaling analysis results for CAF LBMHD (size 1024^2), using relative excess work, on 4 and 64 CPUs, for the routine `stream`.

expectation is that the overall running time is the same on any number of processors. In Figure 11.22, we present a screenshot of the weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 CPUs. Overall, su3_rmd loses 32% efficiency. A call to `ks_congrad_two_src` is responsible for 10% $IREW$, a call to `update_h` leads to 7% loss of efficiency, and two calls to `grsource_imp` cause 7% $IREW$ each.

Next, we focus on the loss of scalability within `ks_congrad_two_src` in Figure 11.23. A call to `ks_congrad` accounts for 8% $IREW$, while a second call to `ks_congrad` leads to 2% $IREW$. Within the first call to `ks_congrad`, the routine `load_fatlinks`

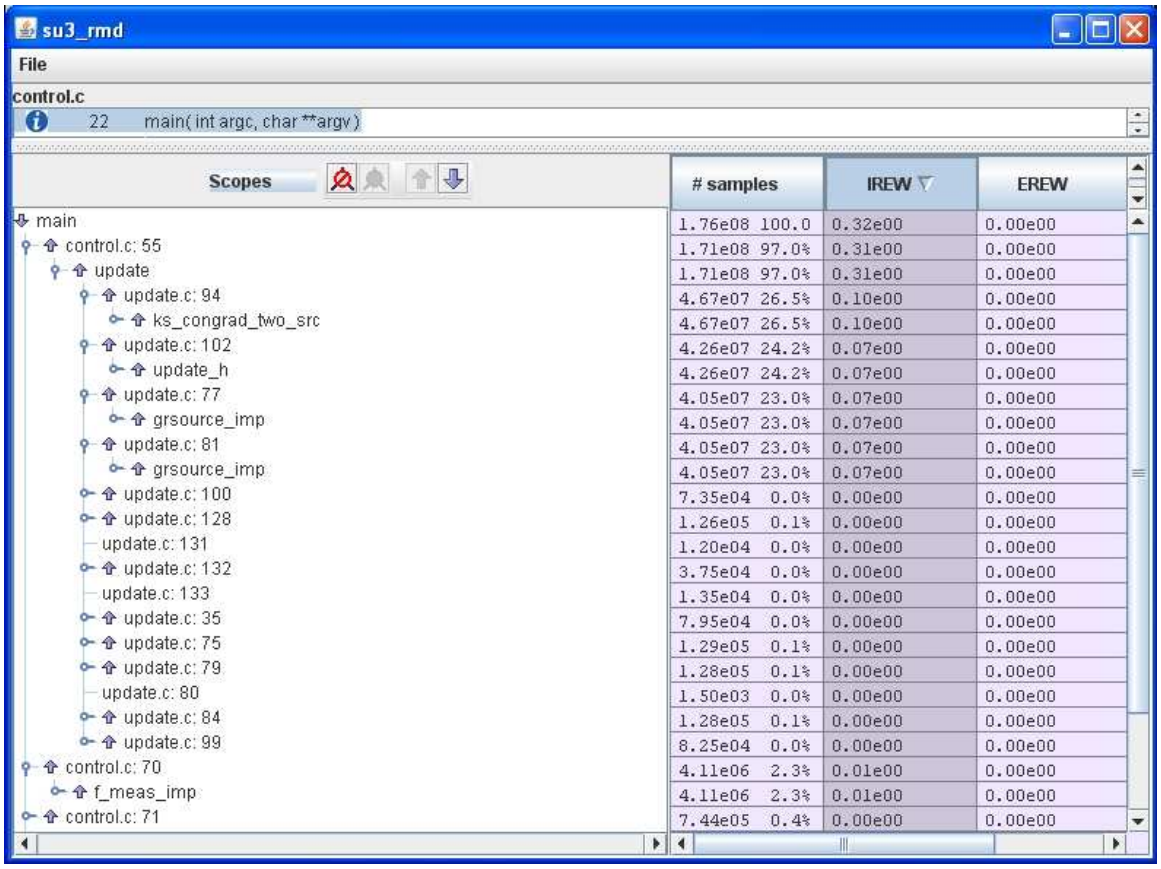


Figure 11.22: Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors.

has an $IREW$ of 7%. Within both calls to `ks_congrad`, several calls to the routine `dslash_fn_field_special` have a cumulated $IREW$ of 4%.

In Figure 11.24, we present a screenshot of weak scaling results for `grsource_imp`. The results show that the routine `load_fatlinks` loses again 7% $IREW$. Overall, `load_fatlinks` is responsible for 21% of the loss of scaling. We present a screenshot of scaling analysis results for `load_fatlinks` in Figure 11.25. The `path_product` routine accounts for 4% $IREW$ out of 7% $IREW$ for `load_fatlinks`. Inside the routine `path_product`, several calls to `wait_gather` account for 3% $IREW$. By correlating the CCT node with the source code, we determined that `wait_gather` waits

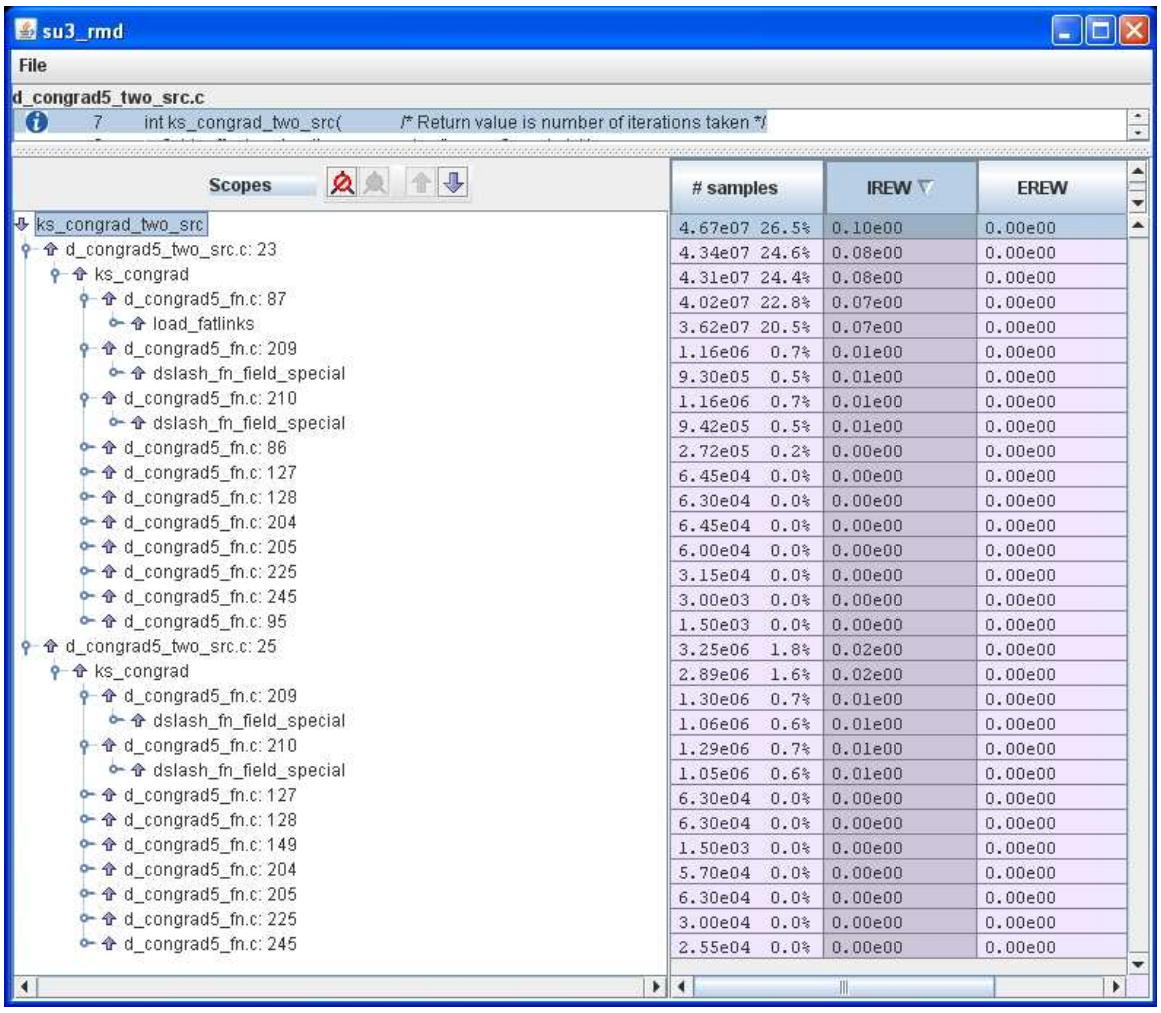


Figure 11.23: Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `ks_congrad_two_src`.

for a series of MPI sends and receives to complete. `path_product` also exhibits a 1% $EREW$, showing that the time spent inside `path_product` increases as the number of processors increases.

In Figure 11.26, we focus on the cost of the routines `dslash_fn_field_special` called in `ks_congrad`. The results showed that again `wait_gather` is a culprit.

Overall, we demonstrated that our scaling analysis technique can be applied as well to the analysis of weak scaling parallel codes, and it pointed to a communication routine,

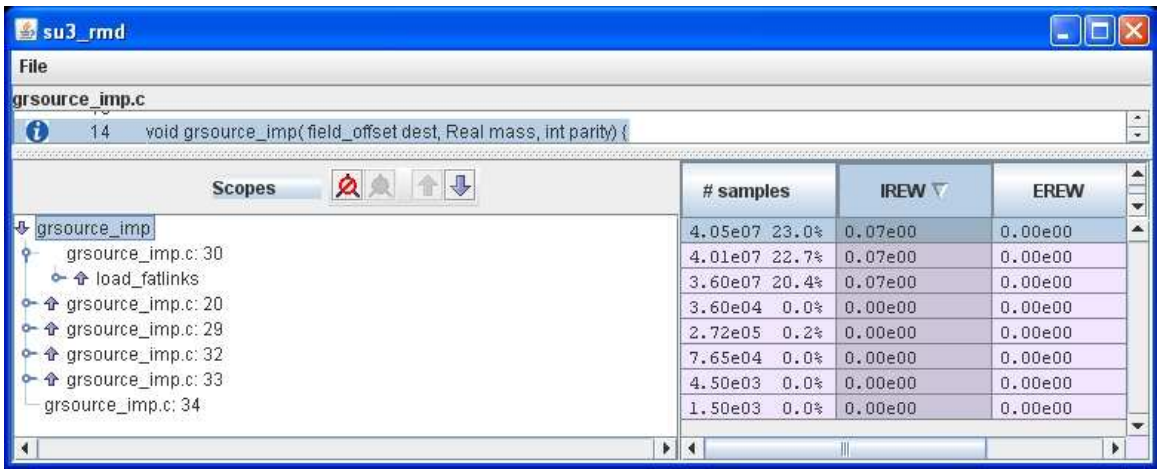


Figure 11.24: Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `grsource_imp`.

`wait_gather`, as a significant source of inefficiency.

11.5 Discussion

Performance analysis based on expectations is a powerful technique. It is applicable to a broad range of applications because it is not limited to any particular programming model. By using a metric based on the fraction of excess work present in an execution, we focus attention on what matters; absolute scalability is less relevant than the overall cost incurred in an execution due to lack of scalability.

In this chapter we presented novel parallel scalability analysis method based on call path profiles, which automatically computes scalability scores for each node in a program's calling context tree. We focused on using the expectation of linear scaling to analyze parallel executions that represent studies of strong scaling, and used expectations of constant time for a weak scaling study. We also described a semiautomatic performance analysis of scalability for computation and synchronization primitives for MPI and CAF benchmarks. We presented the insight gained with our analysis methods into the scalability problems of the NAS benchmarks (MG, SP, and CG), the LBMHD benchmark, LANL's POP application,

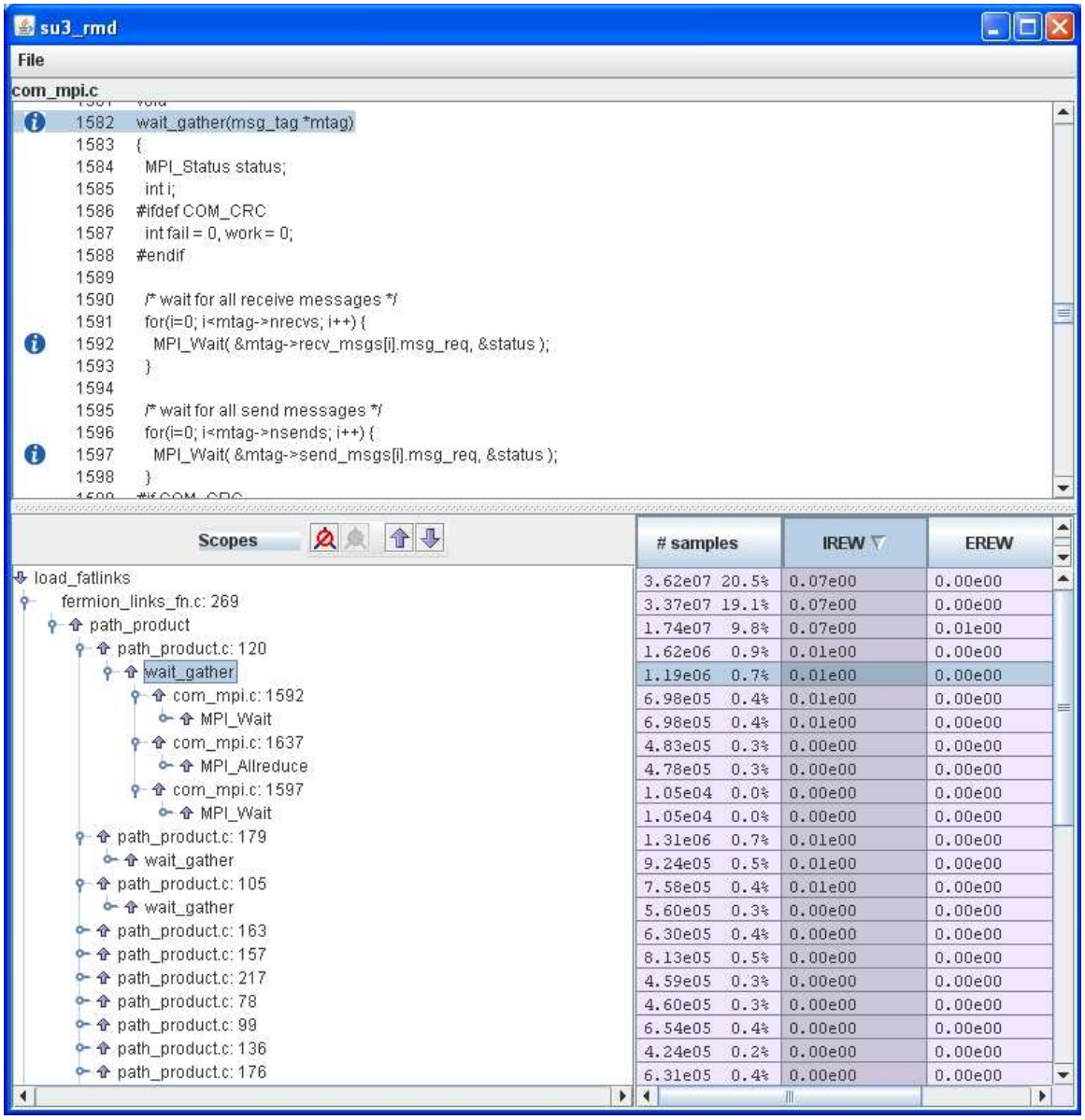


Figure 11.25: Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `load_fatlinks`.

and of an MPI-based MILC benchmark. We determined that the lack of reductions support in the CAF language led to suboptimal and non-performance portable implementations of reductions as CAF source-level libraries; replacing naive reductions with MPI reductions yielded time improvements as high as 25% on 64 processors for the LBMHD benchmark.

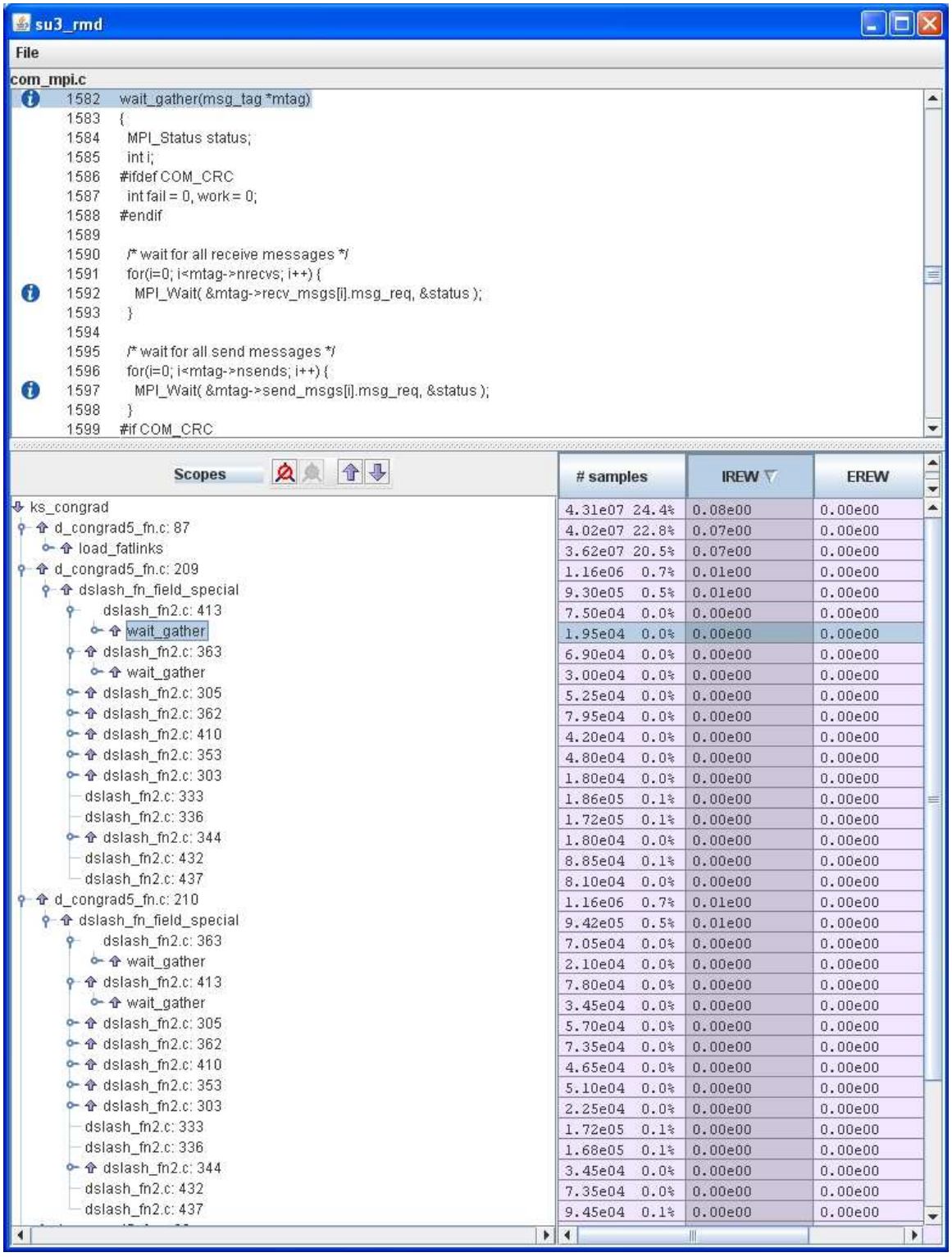


Figure 11.26: Screenshot of weak scaling analysis results for su3_rmd using relative excess work on 1 and 16 processors, for the routine `ks_congrad`.

We also determined that the the lack of a non-blocking implementation of `armci_notify` in the ARMCI communication library caused a scalability bottleneck in NAS SP.

This study showed that the results obtained by the automatic scaling analysis method are consistent with those obtained by the semi-automatic method using the communication primitives scalability plots and the bottom-up view. This means that even though one may use many metrics to quantify scalability, the ones we utilized sufficed for both strong and weak scaling analyses.

We explored an extension of the CAF model with collective operations, and evaluated their impact; using the language-level collective led to a reduction of 60% on 64 CPUs of the initialization time for the NAS MG benchmark and to gains of 25% in execution time on 64 CPUs for the LBMHD kernel. The language extensions are described in Appendix B.

We demonstrated the utility of our technique for pinpointing scalability bottlenecks no matter what their underlying cause. Our scaling analysis method works regardless of the SPMD parallel programming model, of the underlying communication fabric and processor type, of the application characteristics, and of the scaling characteristics (e.g. weak scaling vs strong scaling). When used in conjunction with performance analysis based on expectations, our performance tools are able to attribute scalability bottlenecks to calling contexts, which enables them to be precisely diagnosed.

In the future, we intend to explore using performance analysis based on expectations for analyzing codes written using other parallel programming models, *e.g.* OpenMP and MPI-2. We plan to use our method to perform a thorough scaling study of petascale NSF procurement benchmarks. Finally, we plan to incorporate support for performance analysis using expectations into the distributed version of Rice University's HPCToolkit performance analysis tools.

Chapter 12

Conclusions

We are fast approaching the point when petascale machines will be available to scientists and engineers. Exploiting these machines effectively will be a challenge. To rise to this challenge, we need programming models and tools that improve development time productivity and enable us to harness the power of massively parallel systems. Because programmers rarely achieve the expected performance or scaling from their codes, they need tools that can automatically pinpoint scaling impediments to direct and prioritize their optimization efforts, and thereby improve development time productivity.

In the quest for easy to use, performance portable, and expressive parallel programming models, Co-array Fortran represents a pragmatic alternative to established models such as MPI, OpenMP and HPF. While MPI, a library-based message passing programming model, is the *de facto* technology used for writing parallel codes, it is difficult to use. HPF and OpenMP are language-based programming models; they rely exclusively on compilers to achieve high-performance, and are not able to deliver performance on a broad range of codes and architectures. CAF offers a one-sided programming model, where only one process needs to specify `PUT` or `GET` communication, without interrupting the other process; CAF is easier to use than MPI, especially for irregular applications. In contrast to HPF and OpenMP, a CAF programmer has more control over the final performance and only modest compiler technology is needed to achieve high-performance and scalability.

The thesis of our work is that *Co-array Fortran codes can deliver high performance and scalability comparable to that of hand-tuned MPI codes across a broad range of architectures. When CAF codes or other SPMD programs do not achieve the desired performance and scalability, we can automatically diagnose impediments to their scalability*.

To demonstrate this thesis, we implemented `cafc`, a prototype multi-platform source-to-source CAF compiler. We demonstrated through experiments on several platforms that CAF versions of such regular codes as the NAS benchmarks SP, BT, and LU, of irregular codes such as NAS CG, and of the magnetohydrodynamics code LBMHD can yield performance comparable to or better than that of their MPI counterparts on both cluster-based and hardware shared memory platforms.

This dissertation presents key implementation decisions regarding the implementation of a multiplatform CAF compiler, and describes automatic and source level optimizations for achieving local and communication performance on clusters and distributed shared memory systems.

To achieve *efficient node performance*, the `cafc`-generated code must be amenable to backend compiler analysis and optimization. To avoid the penalty of overly conservative assumptions about aliasing, `cafc` implements an automatic transformation that we call *procedure splitting*, that conveys to a backend compiler the lack of aliasing, co-array shape and bounds, and the contiguity of co-array data. This enables a backend compiler to perform more accurate dependence analysis and apply important optimizations such as software pipelining, software prefetching and tiling. Our experiments showed that procedure splitting yielded benefits as high as 60% on Itanium2 and Alpha architectures.

To achieve scalable *communication performance*, we used source-level transformations such as *communication vectorization*. An advantage of CAF is that it can express vectorization at source level without calls to bulk library primitives. Communication vectorization yielded benefits as high as 30% on Myrinet cluster architectures. When writing CAF communication, the Fortran 95 array sections enable a programmer to express communication of strided data that is noncontiguous. We showed that even when using communication libraries that support efficient non-contiguous strided communication, it is beneficial to perform *communication packing* of strided data at source level, sending it as contiguous message, and unpacking it at its destination. We also showed that one-sided communication aggregation using active messages is less efficient than library optimized strided communi-

cation transfers, because libraries such as ARMCI can overlap packing of communication chunks at the source, communication of strided chunks and unpacking of chunks on the destination. Communication packing at source level boosted performance about 30% for both CAF and UPC on clusters, but yielded minuscule benefits on shared memory platforms. To give a CAF programmer the ability to overlap computation and communication, we extended CAF with *non-blocking communication regions*. Skilled CAF programmers can use pragmas to specify the beginning and the end of regions in which all communication events are issued by using non-blocking communication primitives, assuming the underlying communication library provides them. Using these regions enabled us to improve the performance of NAS BT by up to 7% on an Itanium+Myrinet2000 cluster.

To further improve parallel performance of CAF or other SPMD codes, we need to determine the impediments to scalability. To understand how scaling bottlenecks arise, we need to analyze them within the calling context in which they occur. This enables program analysis at multiple levels of abstraction: we could choose to analyze the cost of user-level routines, user-level communication abstractions, compiler runtime primitives, or underlying communication library.

Users have certain *performance expectations* of their codes. For strong scaling parallel applications users expect that their execution time decreases linearly with the number of processors. For weak scaling applications, they expect that the execution time stays constant while the number of processors increases and the problem size per processor remains constant. Our goal was to develop an efficient technology that *quantifies* how much a certain code deviates from the performance expectations of the users, and then quickly *guides* them to the scaling bottlenecks. We developed an intuitive metric for analyzing the scalability of application performance based on excess work. We used this scaling analysis methodology to analyze the parallel performance of MPI, CAF, and UPC codes. A major advantage of our scalability analysis method is that it is effective regardless of the SPMD programming model, underlying communication library, processor type, application characteristics, or partitioning model. We plan to incorporate our scaling analysis into

HPCToolkit, so it would be available on a wide range of platforms.

Our scaling study pointed to several types of problems. One performance issue we identified using our scalability analysis was the inefficiency of user-level implementation of reductions in both CAF and UPC codes. A drawback of source-level user-implemented reductions is that they introduce performance portability problems. The appropriate solution is to use language-level or library implementations of reductions, that can be tuned offline to use the most efficient algorithms for a particular platform. An obstacle to scalability for CAF codes was a blocking implementation of the `sync_notify` synchronization primitive. Finally, for both CAF and MPI applications we found that some codes performed successive reductions on scalars; the natural remedy for that is to perform aggregation of reductions by using the appropriate vector operations. An important result was that the relative excess work metric readily identified these scalability bottlenecks.

The scaling analysis of CAF codes indicated the urgency of language-level support for collective operations. Consequently, we explored and evaluated collective operations extensions to the CAF model and presented an implementation strategy based on the MPI collectives. For the NAS MG benchmark, using the language-level collectives led to a reduction of the initialization time by 60% on 64 processors, and led to a reduction of the measured running time for LBMHD of 25% on 64 processors.

Unsurprisingly, our scaling analysis identified exposed communication latency as a major scalability impediment. In companion work, Dotsenko [72] proposed several strategies for latency hiding: CAF language extensions for computation shipping and multiversion variables for producer-consumer patterns. However, further compiler analysis and runtime improvements are necessary to tune the granularity of communication to target architectures. We need to improve the usability of our calling context tree viewer by making it easier for users to identify trouble spots when analyzing large applications, for example by computing and displaying summary information for the scalability metrics. Our scalability analysis methodology supports SPMD programs; we need to extend it to analyze parallel programs that utilize dynamic activities.

Bibliography

[1] High Productivity Computing Systems. `http://www.highproductivity.org`, 2006.

[2] NESL: A parallel programming language. `http://www.cs.cmu.edu/~scandal/nesl.html`, 2006.

[3] Parallel numerical algorithms in NESL. `http://www.cs.cmu.edu/~scandal/nesl/alg-numerical.html`, 2006.

[4] *US Lattice Quantum Chromodynamics Software Releases*, 2006.

[5] D. Abramson and A. McKay. Evaluating the performance of a SISAL implementation of the Abingdon Cross Image Processing Benchmark. *International Journal of Parallel Programming*, 23(2):105–134, 1995.

[6] Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`, 1995.

[7] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference*. McGraw Hill, NY, April 1992.

[8] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[9] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[10] V. Adve, C. Koelbel, and J. Mellor-Crummey. Performance Analysis of Data-Parallel Programs. Technical Report CRPC-TR94405, Center for Research on Parallel Computation, Rice University, May 1994.

[11] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[12] V. Adve and J. Mellor-Crummey. *Advanced Code Generation for High Performance Fortran*, chapter 16, pages 553–596. Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques and Run Time Systems (D. P. Agrawal and S. Pande, editors), Lecture Notes in Computer Science 1808. Springer-Verlag, Berlin, 2000.

[13] V. Adve, J. Mellor-Crummey, and A. Sethi. An Integer Set Framework for HPF Analysis and Code Generation. Technical Report CS-TR97-275, Dept. of Computer Science, Rice University, Apr. 1997.

[14] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the Fifth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 251–260, New York, NY, USA, 1993. ACM Press.

[15] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[16] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification v1.0α. `http://research.sun.com/projects/plrg/fortress.pdf`, Sept. 2006.

[17] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[18] AMD. Multi-core processors — the next evolution in computing. `http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf`, 2006.

[19] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[20] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, New York, NY, USA, 1990. ACM Press.

[21] ANSI. *Myrinet-on-VME Protocol Specification (ANSI/VITA 26-1998)*. American National Standard Institute, Washington, DC, 1998.

[22] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.

[23] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Aug. 1991.

[24] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[25] J. Bailey, C. Bernard, C. DeTar, S. Gottlieb, U. M. Heller, J. Hetrick, L. Levkova, J. Osborn, D. B. Renner, R. Sugar, and D. Toussaint. MIMD lattice computation (milc) collaboration. `http://www.physics.indiana.edu/~sg/milc.html`, 2006.

[26] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[27] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the Cray X1. In *Proceedings of the 18th ACM International Conference on Supercomputing*, Saint Malo, France, June 2004.

[28] G. E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie-Mellon University, Sept. 1995.

[29] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.

[30] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[31] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[32] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. NESL user's manual (3.1). Technical Report CMU-CS-95-169, Carnegie-Mellon University, Sept. 1995.

[33] D. Bonachea. GASNet specification v 1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, 2002.

[34] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National, October 2004.

[35] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[36] S. J. D. Bradford L. Chamberlain, Sung-Eun Choi and L. Snyder.

[37] T. Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Working Conference on Massively Parallel Programming Models*, Berlin, 1993.

[38] T. Brandes. Adaptor: A compilation system for data parallel Fortran programs. In C. W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*. Vieweg, Wiesbaden, 1994.

[39] P. G. Bridges and A. B. Maccabe. Mpulse: Integrated monitoring and profiling for large-scale environments. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston,TX, October 2004.

[40] D. Cann and J. Feo. SISAL versus FORTRAN: A comparison using the Livermore loops. In *Proceedings of Supercomputing 1990*, pages 626–636, NY, November 1990.

[41] D. C. Cann. The optimizing SISAL compiler. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[42] F. Cantonnet and T. El-Ghazawi. UPC performance and potential: A NPB experimental study. In *Proceedings of Supercomputing 2002*, Baltimore, MD, 2002.

[43] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber. Fast address translation techniques for distributed shared memory compilers. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, 2005.

[44] F. Cantonnet, Y. Yao, S. Annareddy, A. S. Mohamed, T. El-Ghazawi, P. Lorenz, and J. Gaber. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2003.

[45] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[46] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.

[47] B. L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, Department of Computer Science, University of Washington, November 2001.

[48] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of Supercomputing 2000*, Dallas, November 2000.

[49] D. Chavarría-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. In *Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, Sept. 2002.

[50] D. Chavarría-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. *Journal of Instruction Level Parallelism*, 5, feb 2003.

[51] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`, nov 2005.

[52] W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grain UPC applications. In *Proceedings of the 14th International Conference of Parallel Architectures and Compilation Techniques*, Saint-Louis,MO, 2005.

[53] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[54] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th ACM International Conference on Supercomputing*, San Francisco, California, June 2003.

[55] I.-H. Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of Supercomputing 2004*, Pittsburgh, PA, 2004.

[56] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran Performance and Potential: An NPB Experimental Study. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[57] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. In *Proceedings of the Los Alamos Computer*

Science Institute Fifth Annual Symposium, Santa Fe, NM, Oct. 2004. Distributed on CD-ROM.

[58] Cray, Inc. Cray X1 Server. `http://www.cray.com`, 2004.

[59] Cray, Inc. Chapel specification v0.4. `http://chapel.cs.washington.edu/specification.pdf`, Feb. 2005.

[60] Cray, Inc. Cray C/C++ reference manual. `http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003`, 2006.

[61] Cray Research, Inc. Application programmer's library reference manual. Technical Report SR-2165, Cray Research, Inc., 1994.

[62] A. Darte, D. Chavarría-Miranda, R. Fowler, and J. Mellor-Crummey. Generalized multipartitioning for multi-dimensional arrays. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[63] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[64] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *Proceedings of Supercomputing 1995*, San Diego, CA, December 1995.

[65] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems — data copy reuse and runtime partitioning. In *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*.

[66] R. Das, J. Saltz, and R. v. Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.

[67] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[68] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder. The design and implementation of a parallel array operator for arbitrary remapping of data. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, San Diego, CA, June 2003.

[69] L. DeRose, T. Hoover, and J. K. Hollingsworth. The dynamic probe class library-an infrastructure for developing instrumentation for performance too. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001.

[70] S. J. Dietz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, Department of Computer Science, University of Washington, February 2005.

[71] S. J. Dietz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Proceedings of the 9th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, Santa Fe, NM, April 2004.

[72] Y. Dotsenko. *Expressiveness, Programmability and Portable High Performance of Global Address Space Languages*. PhD thesis, Department of Computer Science, Rice University, june 2006.

[73] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-Array Fortran Compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September 29 - October 3 2004.

[74] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on Hardware Shared Memory Platforms. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[75] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[76] T. v. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[77] T. El-Ghazawi, F. Cantonnet, P. Saha, R. Thakur, R. Ross, and D. Bonachea. *UPC-IO: A Parallel I/O API for UPC v1.0*, July 2004. Available at `http://upc.gwu.edu/docs/UPC-IOv1.0.pdf`.

[78] T. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC specifications. `http://upc.gwu.edu/documentation.html`, 2003.

[79] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[80] H. P. F. Forum. High Performance Fortran language specification, version 2.0. `http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20`, Jan. 1997.

[81] V. Freeh and G. Andrews. fsc: A SISAL compiler for both distributed- and shared-memory machines. Technical Report tr 95-01, University of Arizona, February 1995.

[82] N. Froyd, J. Mellor-Crummey, and R. Fowler. Efficient call-stack profiling of unmodified, optimized code. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, 2005.

[83] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *Proceedings of GCC Summit*, Ottawa, Canada, June 2006.

[84] M. Frumkin, H. Jin, and J. Yan. Implementation of the NAS parallel benchmarks in High Performance Fortran. Technical Report NAS-98-009, NAS Parallel Tools Groups, NASA Ames Research Center, Moffet Field, CA 94035, September 1998.

[85] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[86] D. A. Garza-Salazar and W. Bohm. D-OSC: a SISAL compiler for distributed-memory machines. In *Proceedings of the Second Parallel Computation and Scheduling Workshop*, Ensenada, Mexico, Aug. 1997.

[87] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report WRL-95-9, 1995.

[88] K. Gharachorloo, D. Lenoski, J. Ludon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, 1990.

[89] J. R. Goodman. Cache consistency and. sequential consistency. Technical Report 61, SCI Committee, March 1989. Available at `http://citeseer.ist.psu.edu/619902.html`.

[90] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall PTR, Eaglewood Cliffs, NJ, 2005.

[91] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing, Second Edition*. Addison-Wesley, Boston, MA, 2003.

[92] C. Grelck. Implementing the NAS benchmark MG in SAC. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2002.

[93] C. Grelck and S.-B. Scholz. Towards an efficient functional implementation of the NAS benchmark FT. In *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03)*, Nizhni Novgorod,Russia, 2003.

[94] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[95] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, Jan. 1997.

[96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[97] W. Gropp, M. Snir, B. Nitzberg, and E. Lusk. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, second edition, 1998.

[98] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[99] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[100] D. Han and T. Jones. Survey of MPI call usage. `http://www.spscicomp.org/ScicomP10/Presentations/2004.08.12_Scicomp10.ppt`, August 2004.

[101] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems*. PhD thesis, Dept. of Computer Science, Rice University, Dec. 1994. Available as CRPC-TR94494-S from the Center for Research on Parallel Computation, Rice University.

[102] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[103] M. T. Heath. Performance visualization with ParaGraph. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, Townsend, TN, May 1994.

[104] M. T. Heath, A. D. Malony, and D. T. Rover. Visualization for parallel performance evaluation and optimization. *Software Visualization*, 1998.

[105] Hewlett-Packard. *Compaq UPC Compiler*, 2003. `http://h30097.www3.hp.com/upc`.

[106] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[107] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, Jan. 1993.

[108] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, Aug. 1991. Springer-Verlag.

[109] S. Hiranandani, K. Kennedy, and J. Mellor-Crummey. Advanced compilation techniques for Fortran D. Technical Report CRPC-TR93338, Center for Research on Parallel Computation, Rice University, Oct. 1993.

[110] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, July 1994.

[111] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*.

[112] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[113] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[114] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[115] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.

[116] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, Apr. 1994.

[117] Y. C. Hu, G. Jin, S. L. Johnsson, D. Kehagias, and N. Shalaby. HPFBench: A High Performance Fortran benchmark suite. *ACM Transactions on Mathematical Software*, 26(1):99–149, Mar. 2000.

[118] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[119] C. Iancu, P. Husbands, and P. Hargrove. Communication optimizations for fine-grain UPC applications. In *Proceedings of the 14th International Conference of Parallel Architectures and Compilation Techniques*, Saint-Louis,MO, 2005.

[120] IBM Corporation. Report on the experimental language X10, draft v0.41. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html/FILE/ATTH4YZ5.pdf`, Feb. 2006.

[121] Intel Inc. Evolution of parallel computing. `http://www.intel.com/platforms/parallel.htm`, 2006.

[122] Intrepid Technology Inc. GCC Unified Parallel C. `http://www.intrepid.com/upc`, 2006.

[123] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York, NY, 1991.

[124] P. W. Jones. The Los Alamos Parallel Ocean Program (POP) and coupled model on MPP and clustered SMP architectures. In *Seventh ECMWF Workshop on the Use of Parallel Processors in Meterorology*, Reading, England, November 1996.

[125] P. W. Jones, P. H. Worley, Y. Yoshida, I. James B. White, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP): Research Articles. *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[126] K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[127] K. Kennedy, J. Saltz, and R. von Hanxleden. Value-based distributions in Fortran D - A preliminary report. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, 1993.

[128] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[129] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[130] J. M. Levesque. Applied parallel research's xHPF system. *IEEE Parallel and Distributed Technologies*, 2(3), 1994.

[131] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[132] K. Marsal. IBM unveils dual-core PowerPC chips up to 2.5GHz. `http://www.appleinsider.com/article.php?id=1166`, 2005.

[133] T. G. Mattson. An introduction to OpenMP 2.0. *Lecture Notes in Computer Science*, 1940:384–390, 2000.

[134] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers. `http://home.austin.rr.com/mccalpin/papers/bandwidth/`, 1995.

[135] L. Meadows, D. Miles, C. Walinsky, M. Young, and R. Touzeau. The Intel Paragon HPF compiler. In *Proceedings of the 1995 Intel Supercomputer Users Group*, Albuquerque, NM, June 1995.

[136] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002. *Special Issue with Selected Papers from the Los Alamos Computer Science Institute Symposium.*

[137] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface Standard*, 1997.

[138] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1999.

[139] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[140] B. Mohr. OPARI OpenMP pragma and region instrumentor. `http://www.fz-juelich.de/zam/kojak/opari`, 2006.

[141] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, 2002.

[142] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, 2001. Distributed on CD-ROM.

[143] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. *A Scalable Approach to MPI Application Performance Analysis*, volume 3666 of *Lecture Notes in Computer Science*, pages 309–316. Springer-Verlag, Berlin, 2005.

[144] D. Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993.

[145] Myricom. GM: A message-passing system for Myrinet networks, 1.6.5. `http://www.myri.com/scs/GM/doc/html/`, October 2005.

[146] Myricom. GM: A message-passing system for Myrinet networks, 2.0.x and 2.1.x. `http://www.myri.com/scs/GM-2/doc/html/`, October 2005.

[147] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[148] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2), 1995.

[149] S. Neuner. Scaling Linux to new heights: the SGI Altix 3000 system. *Linux J.*, (106), 2003.

[150] J. Nieplocha and B. Carpenter. *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer-Verlag, NY, 1999.

[151] J. Nieplocha, V. Tipparaju, A. Saify, and D. K. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Proc. Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02*, Ft. Lauderdale, Florida, April 2002.

[152] Benchmarking information referenced in the NSF 05-625 High Performance Computing System Acquisition: Towards a Petascale Computing Environment for Science and Engineering.. Technical Report NSF0605, November 2005.

[153] High Performance Computing System Acquisition: Towards a Petascale Computing Environment for Science and Engineering. Technical Report NSF05625, National Science Foundation, September 2005.

[154] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *ACM Fortran Forum*, 24(2):4–17, Aug. 2005.

[155] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, August 1998.

[156] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, August 1998.

[157] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proceedings of Supercomputing 2004*, Pittsburgh, PA, November 2004.

[158] Open64 Developers. Open64 compiler and tools. `http://sourceforge.net/projects/open64`, Sept. 2001.

[159] Open64/SL Developers. Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64`, July 2002.

[160] S. S. Pande, D. P. Agrawal, and J. Mauney. Compiling functional parallelism on distributed-memory systems. *IEEE Parallel and Distributed Technology*, 02(1):64–76, 1994.

[161] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: high performance clustering technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[162] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code, 1995.

[163] Quadrics. RMS user guide v 13 (eagle). `http://web1.quadrics.com/downloads/documentation/RMSUserMan_13.pdf`, July 2004.

[164] Quadrics. QsNetII installation and diagnostics manual. `http://web1.quadrics.com/downloads/documentation/QsNet2Install.pdf`, April 2005.

[165] C. Rasmussen, M. Sottile, and T. Bulatewicz. CHASM language interoperability tools. `http://sourceforge.net/projects/chasm-interop`, July 2003.

[166] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[167] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, 1979.

[168] Rice University. *HPCToolkit Performance Analysis Tools*, 2006.

[169] Single Assignment C. `http://www.sac-home.org/`, 2006.

[170] S.-B. Scholz. Single Assignment C — functional programming using imperative style. In *In Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94)*, pages 21.1—21.13, Norwich, UK, 1994.

[171] S.-B. Scholz. Single Assignment C – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, Nov. 2003.

[172] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

[173] Silicon Graphics. CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer, 1994.

[174] Silicon Graphics, Inc. MPT Programmer's Guide, MPI man pages, intro_shmem man pages. `http://techpubs.sgi.com`, 2002.

[175] Silicon Graphics, Inc. The SGI Altix 3000 Global Shared-Memory Architecture. `http://www.sgi.com/servers/altix/whitepapers/tech_papers.html`, 2004.

[176] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge,MA, 1995.

[177] L. Snyder. A programmers guide to ZPL. `http://www.cs.washington.edu/research/zpl/zpl_guide.pdf`, 1999.

[178] J. Su and K. Yelick. Array prefetching for irregular array accesses in Titanium. In *Proceedings of the Sixth Annual Workshop on Java for Parallel and Distributed Computing*, Santa Fe, New Mexico, 2004.

[179] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, 2005.

[180] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *ICS '96: Proceedings of the 10th International Conference on Supercomputing*, pages 18–25, New York, NY, USA, 1996. ACM Press.

[181] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[182] A. S. Tanenbaum. *Distributed operating systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[183] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of Supercomputing 2002*, Baltimore, MD, 2002.

[184] R. v Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[185] J. K. H. Vahid Tabatabaee, Ananta Tiwari. Parallel parameter tuning for applications with performance variability. In *Proceedings of Supercomputing 2005*, Seattle, WA, 2005.

[186] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, 2000.

[187] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 240–250, New York, NY, USA, 2002. ACM Press.

[188] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2001)*, Snowbird, Utah, 2001.

[189] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[190] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 256–266, New York, NY, USA, 1992. ACM Press.

[191] A. Wallcraft. Co-Array Fortran vs MPI. `http://www.co-array.org/cafvsmpi.htm`, 2006.

[192] A. Wallcraft. Subset Co-Array Fortran into OpenMP Fortran. `http://www.co-array.org/caf2omp.htm`, 2006.

[193] E. Wiebel, D. Greenberg, and S. Seidel. *UPC Collective Operations Specifications v1.0*, December 2003. Available at `http://upc.gwu.edu/docs/UPC_Coll_Spec_V1.0.pdf`.

[194] F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Julich, May 2004.

[195] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proceedings of the European Conference on Parallel Computing (Europar)*, Pisa, Italy, August-September 2004.

[196] P. H. Worley. MPICL: a port of the PICL tracing logic to MPI. `http://www.epm.ornl.gov/picl`, 1999.

[197] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: a performance framework for distributed parallel systems. In *Proceedings of Supercomputing*, page 50, Washington, DC, USA, 2000. IEEE Computer Society.

[198] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13), September–November 1998.

[199] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.

Appendices

Appendix A

Scaling Analysis of Parallel Program Performance

In Chapter 11 we described an automatic scaling analysis method, the software infrastructure used to implement it, and presented scaling analysis results which gave us insight into scaling problems for several applications. In this chapter we present applications of our scaling method for other MPI and CAF codes, spanning several of NAS benchmarks. For all the benchmarks analyzed we focused on small problem sizes, which tends to expose lack of scalability due to communication and synchronization inefficiencies on a small number of processors. For historical reasons, we used the average excess work scaling metric.

A.1 Analysis of the NAS MG Benchmark

The MG multigrid kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [24]. The MPI version of MG is described in Section 3.2.1. In Figure A.1 we present the scalability of the MPI version of NAS MG. The MPI primitives that display increased cost with increasing number of processors are `MPI_Send`, `MPI_Wait` and `MPI_Init`; the overall loss of efficiency is 75%. The profiling overhead was of 7-16% for the MPI NAS MG version.

In Figures A.2 we present a screenshot of the scaling analysis results for the MPI version of NAS MG. Overall, the average loss of scaling for the main routine is 34%. The MPI initialization routine accounts for 9%; the routine performing the multigrid computation, `mg3p`, accounts for 7%; the MPI finalization routine leads to a scaling loss of 3%. The routine `resid` costs 5%, and the routine `zran3` incurs a 4% loss of scalability. We display the results for `mg3p` in Figure A.3; the call to `resid` accounts for 3%, out of which

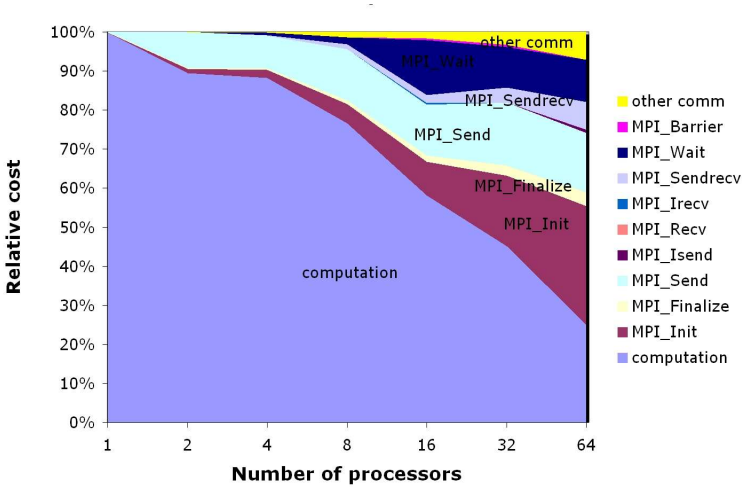


Figure A.1: Scalability of relative costs for communication primitives and computation for the MPI version of the NAS MG benchmark class A (size 256^3).

the routine `comm3` costs 2%; this cost is due to two calls to `give3`, each costing 1%. The main constributor to the scaling loss of `give3` is the MPI routine `mpi_send`. A call to `psinv` accounts for another 2% loss of scaling, mostly due to the `comm3` routine as well.

A.2 Analysis of the NAS SP Benchmark

NAS SP is a simulated CFD application that solves systems of equations resulting from an approximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [24]. SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [24]. The MPI version of SP was described in Section 3.2.1. In Figure A.4 we present the scalability of relative costs for communication primitives and computation for the MPI version of NAS SP. The graph shows that the loss of efficiency is of 46% on 64 CPUs, and that the costs of `MPI_Waitall` increase significantly with an increasing number of processors. The profiling overhead was of 2-8%.

In Figure A.5 we present the scaling analysis results for the MPI version of NAS SP,

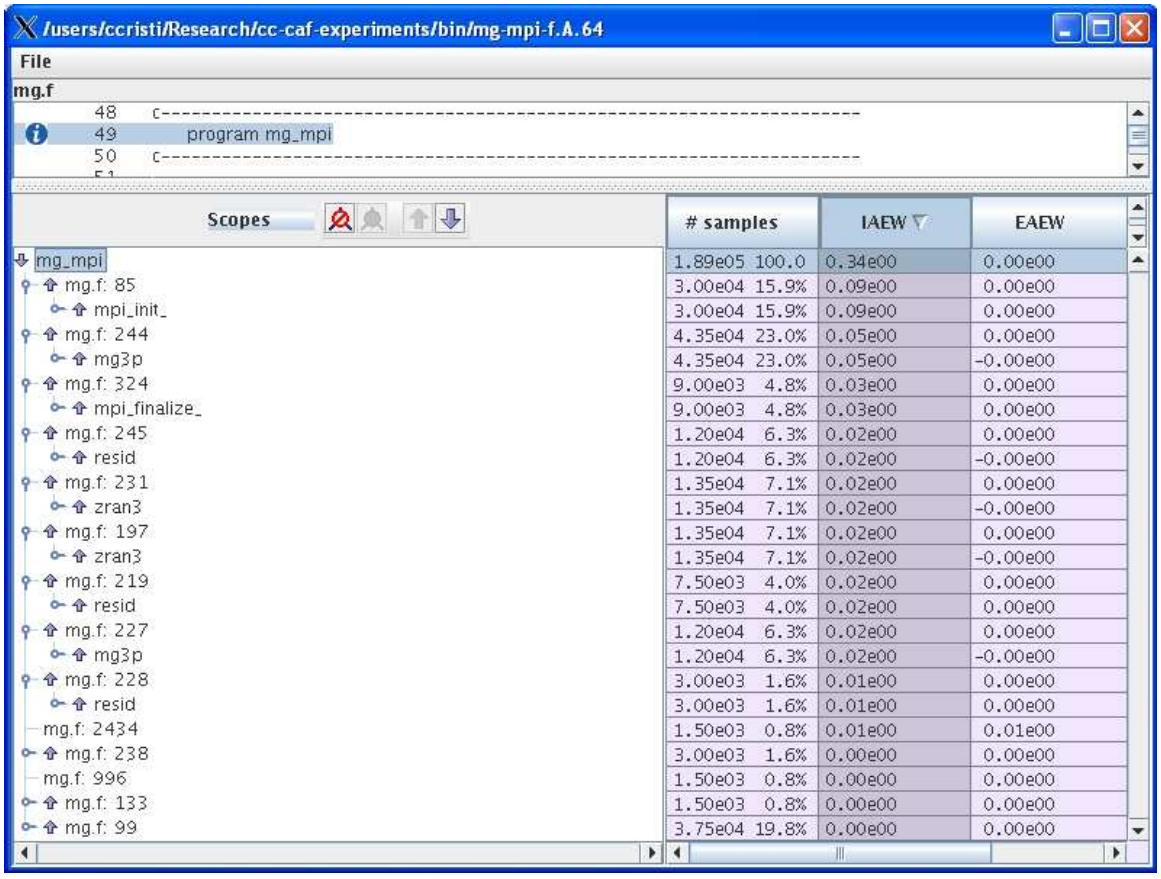


Figure A.2: Screenshot of strong scaling analysis results for MPI NAS MG class A (size 256^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 processors.

class A. The total scaling loss is 21%, out of which the alternate direction integration routine, `adi`, accounts for 19%; an initialization routine, `setup_mpi` accounts for the remaining 2%. Within `adi`, a call to `copy_faces` leads to a 12% loss of scaling, the sweeps along the x, y, and z-dimensions account each for 2% loss, and the `add` routine incurs a 1% scaling loss. We display a screenshot of the scaling analysis results for the routine `copy_faces` in Figure A.6. We notice that the culprit is a call to `mpi_wait`, with an $IAEW$ of 11%; by inspecting the source code we determine that `mpi_wait` is called to complete the communication with six neighbors performed in `copy_faces`.

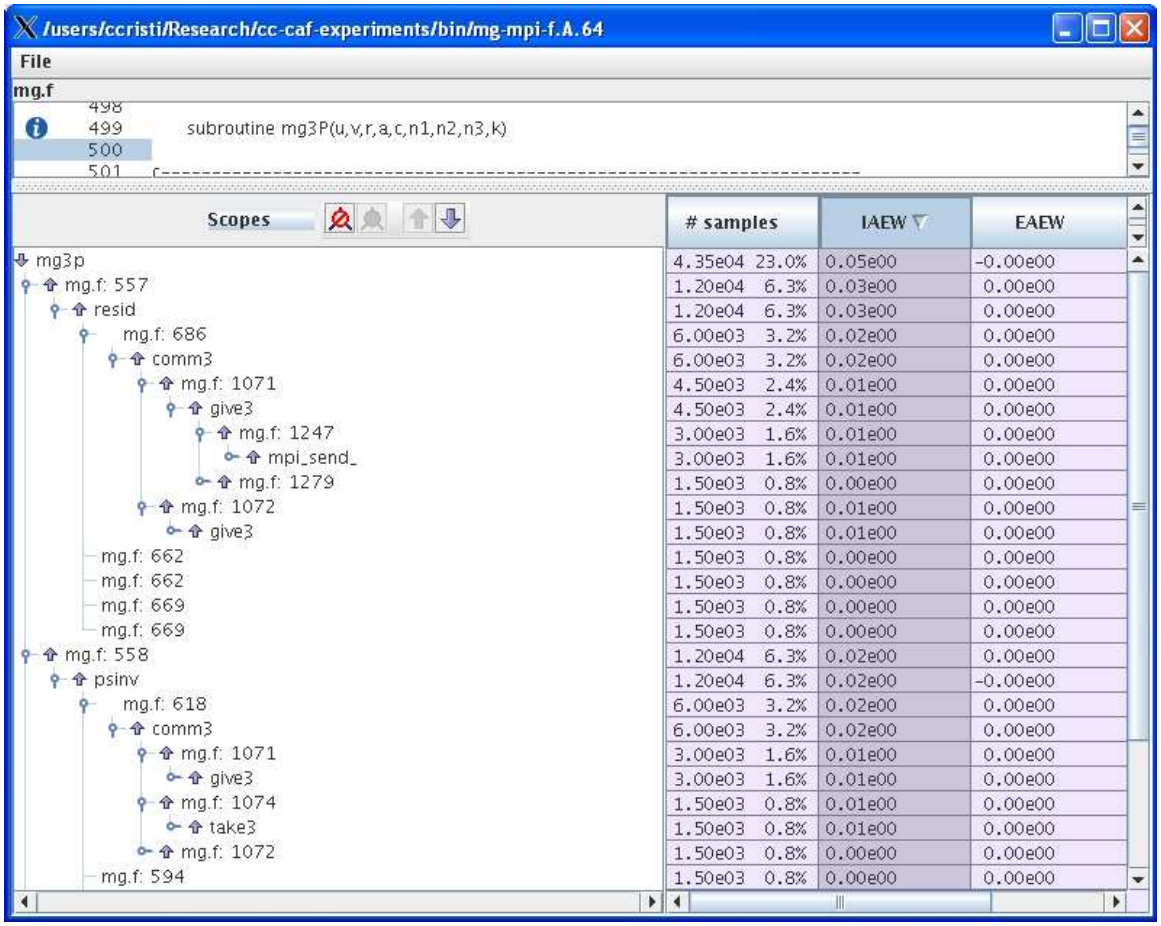


Figure A.3: Screenshot of strong scaling analysis for MPI MG class A (size 256^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 processors, for the routine `mg3p`.

A.3 Analysis of the NAS CG Benchmark

The CG benchmark uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [24]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The MPI version of NAS CG was described in Section 3.2.1, and the CAF version in Section 6.2. In Figure A.7 we present the scalability of relative costs for communication primitives and communication for the MPI version of NAS CG; the results show that the loss of efficiency on 64 processors

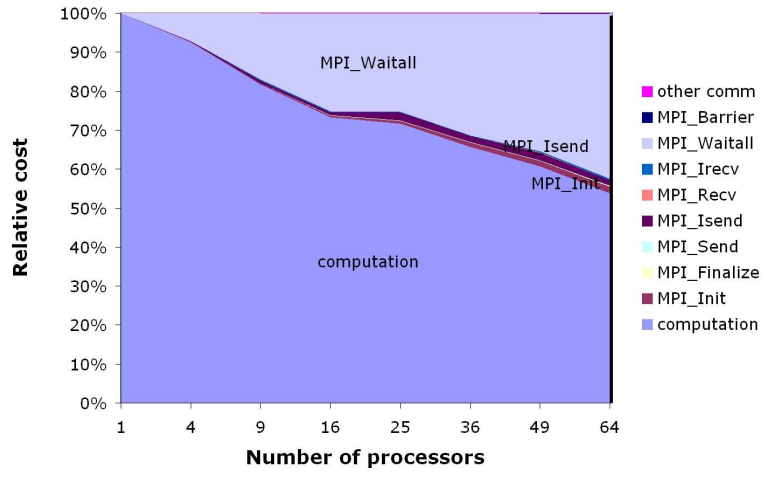


Figure A.4: Scalability of relative costs for communication primitives and computation for the MPI version of the NAS SP benchmark class A (size 64^3).

is of 73%, and `MPI_Wait` is the most nonscalable communication primitive. In Figure A.8 we present the same results for the CAF version of NAS CG; the overall loss of efficiency is of 76% and the routines which exhibit nonscalable costs are `armci_notify_wait` and `ARMCI_Put`. In Figure A.9 we present a summary of the user-defined metrics for the volume of communication and synchronization. The profiling overhead was of 2-8% for the MPI version and of 4-13% for the CAF versions.

The relative cost of communication primitives graphs show that the CAF version spends more time in `sync_wait` as the number of CPUs increases. However, by comparing the CAF and MPI communication primitives and computation scalability graphs, we determined that in this case, it is a characteristic of the algorithm rather then an inefficiency in the translation to CAF or `cafc` run-time library implementation. Notice that both CAF versions display a similar anomaly when going from 8 to 16 CPUs. The relative cost of computation is higher for 16 processors than for 8 processors; analysis of the compiler optimization report showed that the backend compiler performs the same optimizations. The relative cost difference is due to increased number of cache misses for the 16 CPUs version, due to increased conflict misses.

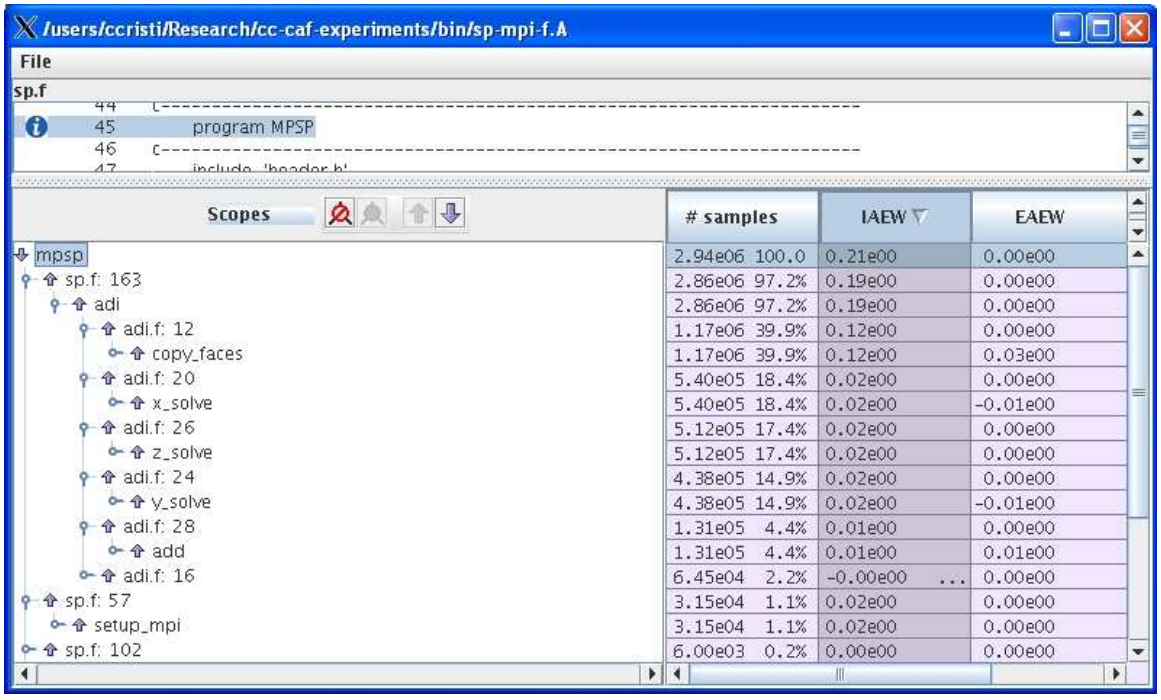


Figure A.5: Screenshot of strong scaling analysis results for MPI NAS SP class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs.

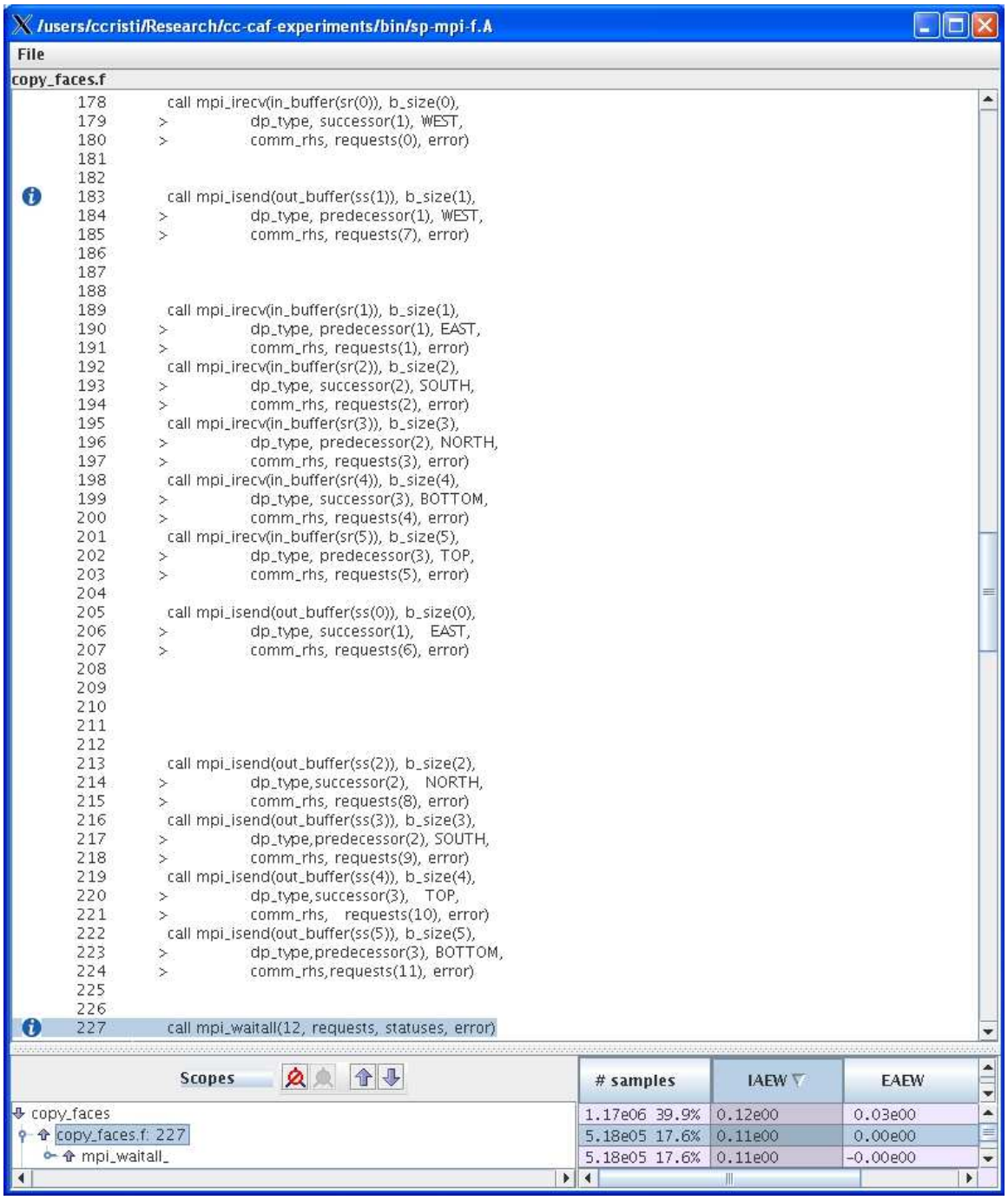


Figure A.6: Screenshot of strong scaling analysis results for MPI NAS SP class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs, for the routine `copy_faces`.

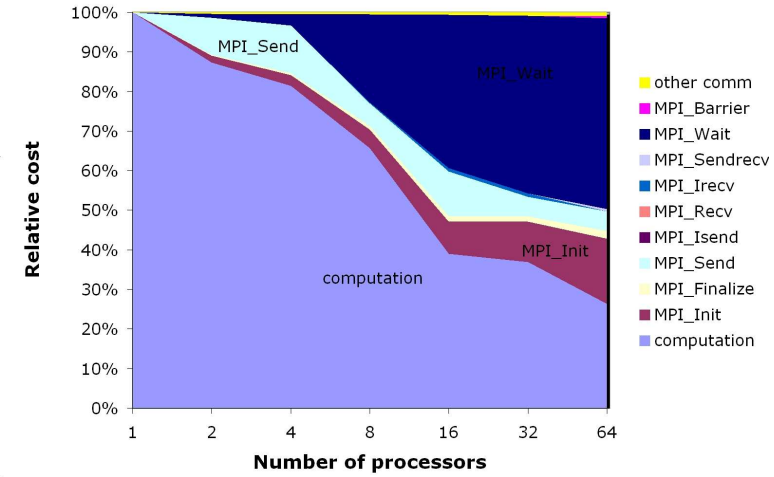


Figure A.7: Scalability of relative costs for communication primitives and computation for the MPI version of the NAS CG benchmark class A (size 14000).

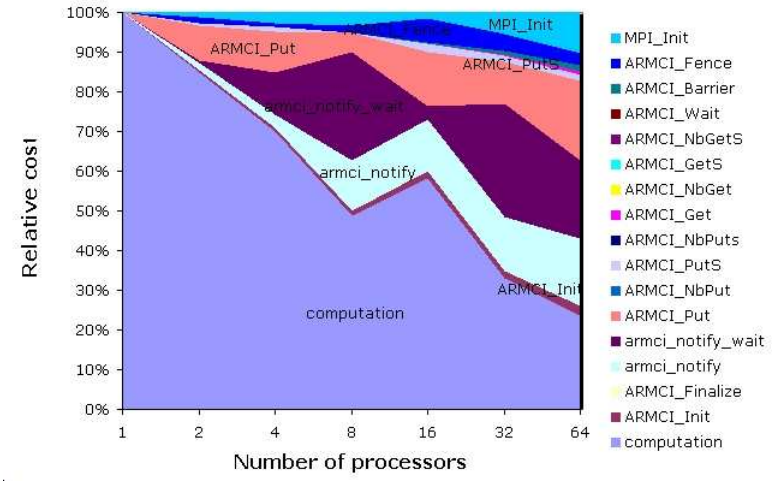


Figure A.8: Scalability of relative costs for communication primitives and computation for the CAF version of the NAS CG benchmark class A (size 14000).

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barriers
1	0	0	3	16	0	0	15
2	1680	46598912	5	40	3360	3360	15
4	1680	46598912	9	72	3360	3360	15
8	2944	34957824	17	264	5888	5888	15
16	2944	34957824	33	520	5888	5888	15
32	4208	23316736	65	2056	8416	8416	15
64	4208	233167367	129	4104	8416	8416	15

Figure A.9: Communication and synchronization volume for the CAF version of NAS CG, class A (size 14000).

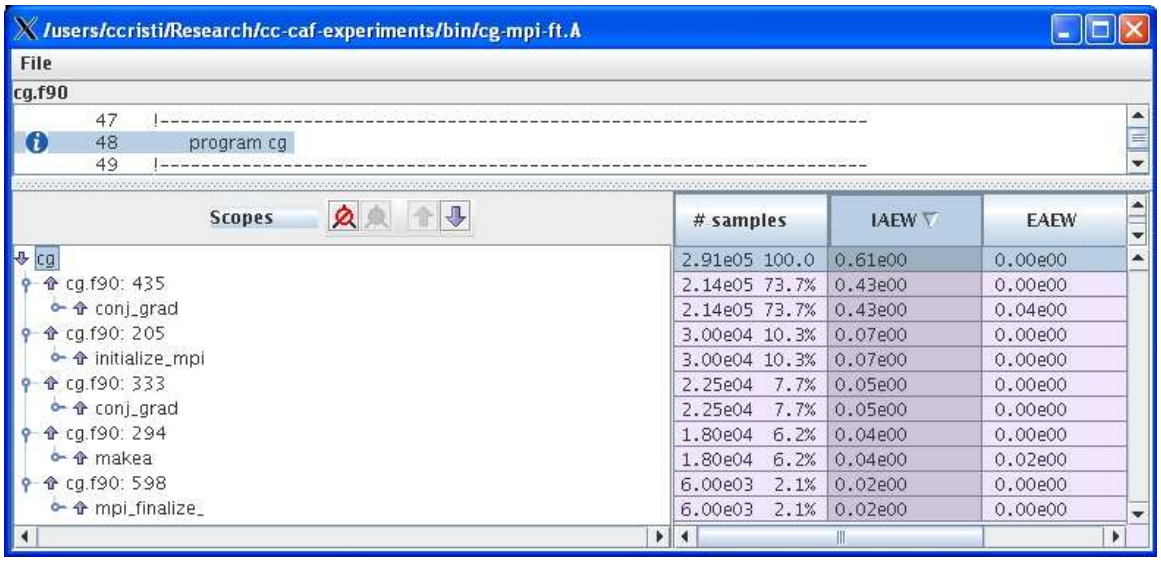


Figure A.10: Screenshot of strong scaling analysis results for MPI NAS CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs.

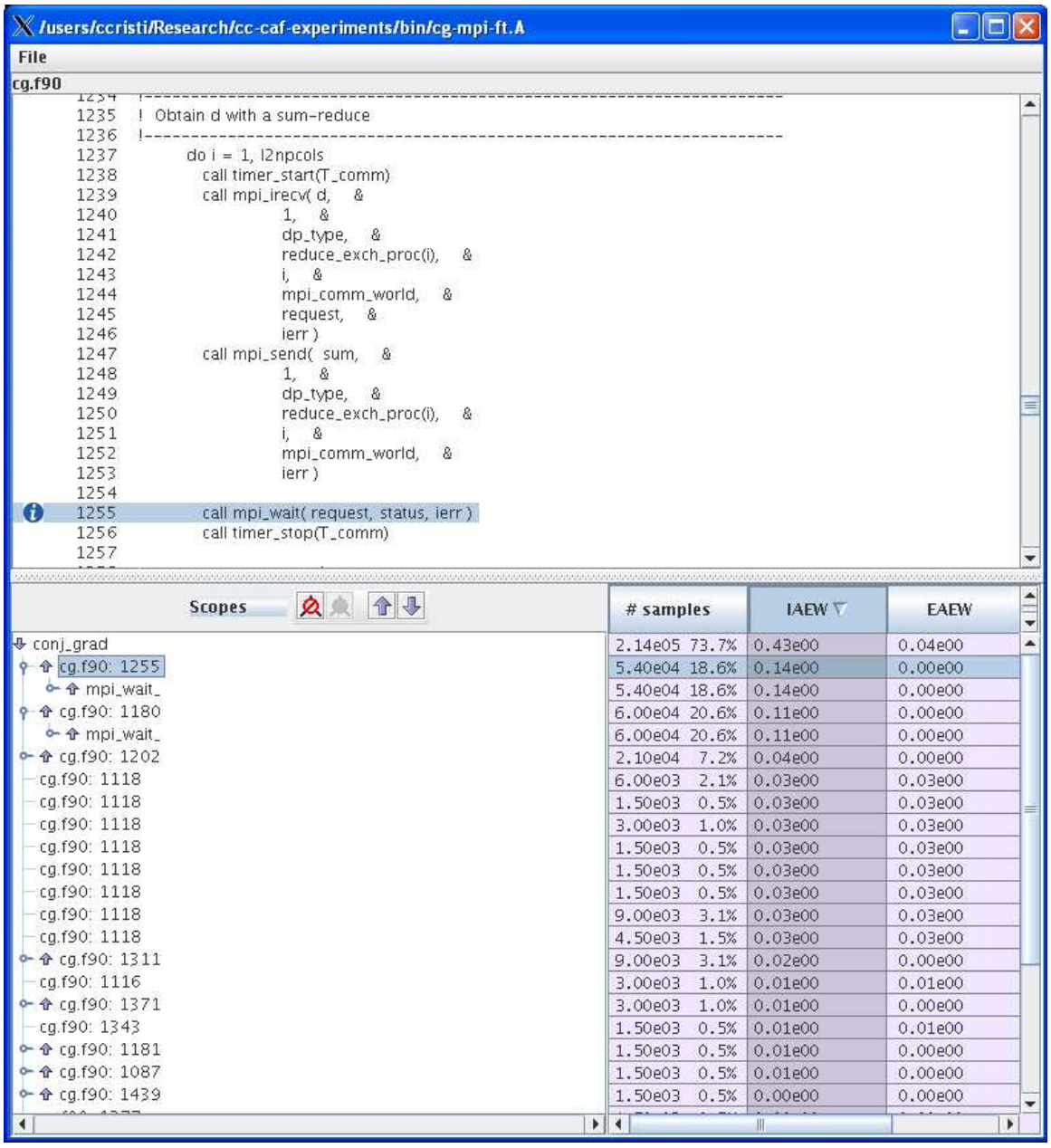


Figure A.11: Screenshot of strong scaling analysis results for MPI NAS CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the routine `conj_grad`.

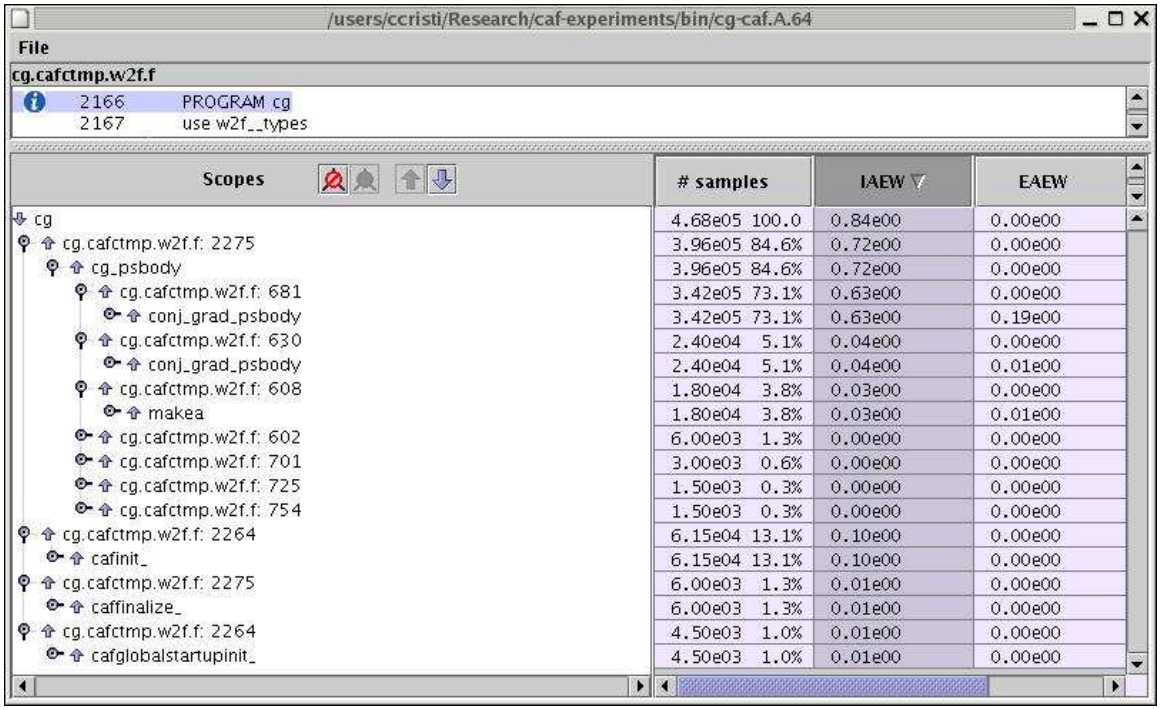


Figure A.12: Screenshot of strong scaling analysis results for CAF CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs.

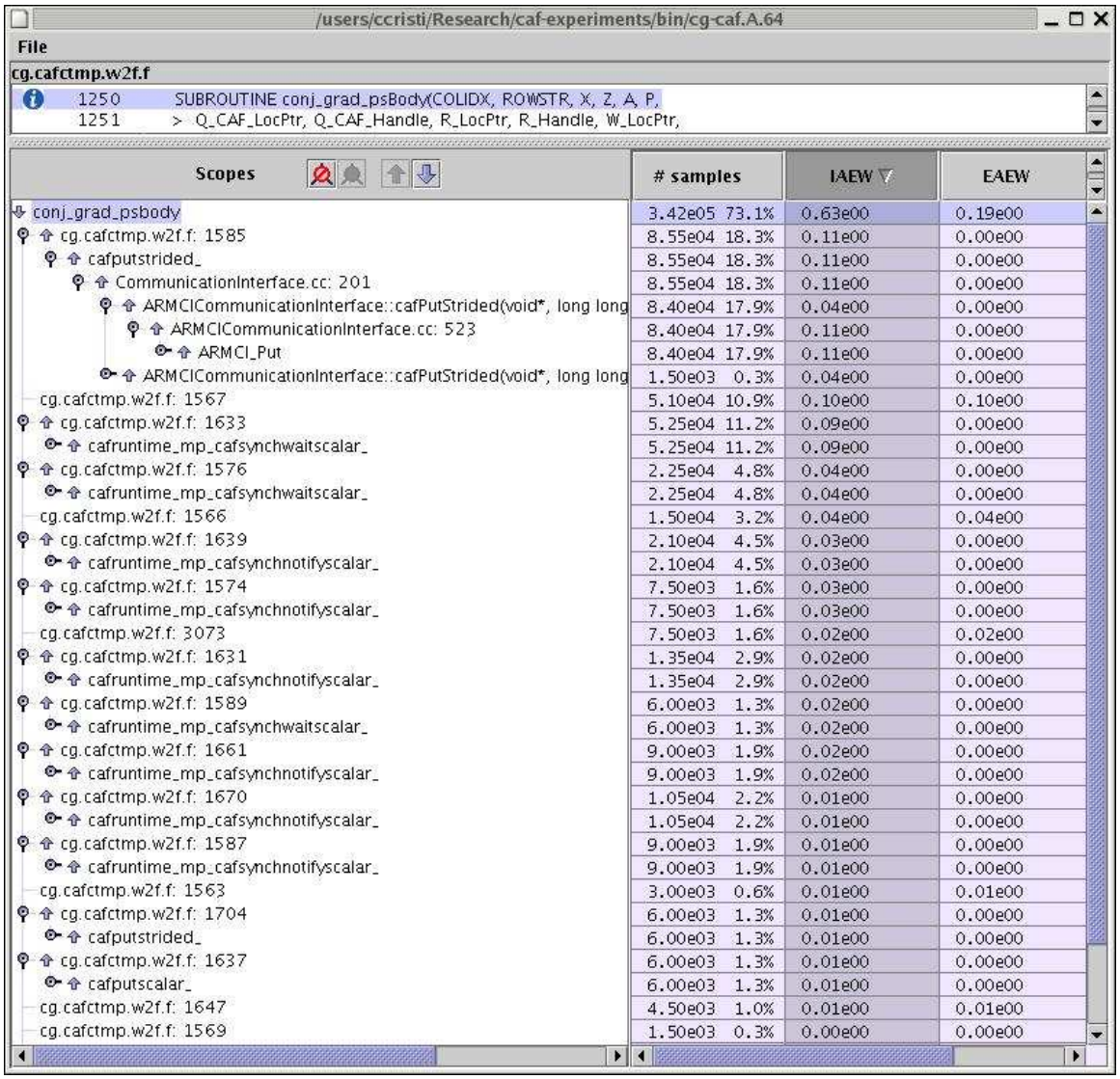


Figure A.13: Screenshot of strong scaling analysis results for CAF CG class A (size 14000), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the routine `conj_grad_psbody`.

We present strong scaling analysis results for the MPI version of NAS CG class A in Figure A.10. The total scaling loss is 61%; `conj_grad` accounts for 50%, the MPI initialization routine incurs a 7% loss, `makea` leads to a 4% loss, and the MPI finalization routine has an $IAEW$ of 2%. By analyzing further `conj_grad`, as displayed in Figure A.11, we notice that two large factors in scaling loss are two calls to `mpi_wait`, with $IAEW$ scores of 14% and 11%, respectively. By using the source correlation feature, we determine that `mpi_wait` is used to implement several sum reductions for a sparse matrix-vector product. This result is consistent with the results of relative cost scaling for selected communication primitives presented in Figure A.7. Also, the $EAEW$ for `conj_grad` is 4%, which shows that the local computation does not exhibit linear scaling either.

In Figures A.12 and A.13 we present screenshots of strong scaling analysis for the CAF version of NAS CG class A using the ARMCI communication library. The results in Figure A.12 show that $IAEW$ for the main routine `cg` is 83.6%, out of which 62.6% is due to `conj_grad_psbody`, the main timed conjugate gradient routine, 10% is due to `cafinit`, 4.5% is due the call of `conj_grad_psbody` in the initialization phase, 3.4% is due to `makea`, that generates the sparse matrix input data, 1% is due to `caffinalize`. Figure A.13 shows that for `conj_grad_psbody` 19.1% of average excess work is actually due to exclusive costs, which means that the local computation is not scalable either. `ARMCI_Put` is responsible for 11.1% excess work, calls to `armci_notify_wait` are responsible for 15%, and calls to blocking `armci_notify` are responsible for 12% IAEW. The calls to `armci_notify_wait` correspond to waiting for permission to write on the remote co-arrays, and are indicative of load imbalance between images.

A.4 Analysis of the NAS LU Benchmark

LU solves the 3D Navier-Stokes equation as do SP and BT. LU implements the solution by using a Successive Over-Relaxation (SSOR) algorithm which splits the operator of the Navier-Stokes equation into a product of lower-triangular and upper-triangular matrices (see [24] and [84])). The algorithm solves five coupled nonlinear partial differential equa-

tions, on a 3D logically structured grid, using an implicit pseudo-time marching scheme. The MPI and CAF versions of NAS LU are described in sections 3.2.1 and 6.4. In Figure A.14 we present the scalability of relative costs of communication primitives and computation for the MPI version of NAS LU. The overall loss of efficiency on 64 CPUs is 46%, and the most inefficient communication primitives is `MPI_Recv`. In Figure A.15 we present the scalability of the CAF version of NAS LU. The overall loss of efficiency on 64 CPUs is 68%, with `armci_notify_wait` most responsible for the loss of scaling. In Figure A.16 we present a summary of the user-defined metrics for the volume of communication and synchronization. The profiling overhead was of 3-10% for the MPI version and of 4-11% for the CAF versions.

For CAF NAS LU, as the number of CPUs increases the time spent in `sync_wait` increases. Even though the communication and synchronization volume point to an increases number of `PUT`s, the number of synchronization events is not double the number of `PUT`s. The bottom up view shows that the large time spent in `sync_wait` is due to load imbalance, waiting for the data to arrive, rather than to the inefficiency of the handshake. However, having non-blocking notifies might reduce the wait time as well, because the extra network latency exposed for the source processor of a `PUT` before the notification is sent is observed as well by the destination processor. The CAF versions using ARMCI and GASNet as communication libraries display the same scalability characteristics.

In Figure A.17 we show a screenshot of strong scaling analysis results using average excess work for the MPI version of NAS LU, class A. The overall nonscalability score is 19%, with the routine `ssor` responsible for 18%, and the communication initialization function, `init_comm`, accounting for 1%. Within `ssor`, `rhs` accounts for 6%, `blts` for 5%, and `jacld` for 2%. By focusing more closely on `ssor`, as shown in Figure A.18, we determined that the main culprit for the loss of scaling of `ssor` were calls to the communication routine `exchange_3`.

In Figures A.19 and A.20 we present screenshots with results of strong scaling analysis for the CAF version of NAS LU using average excess work on 1, 2, 4, 8, 16, 32, and 64

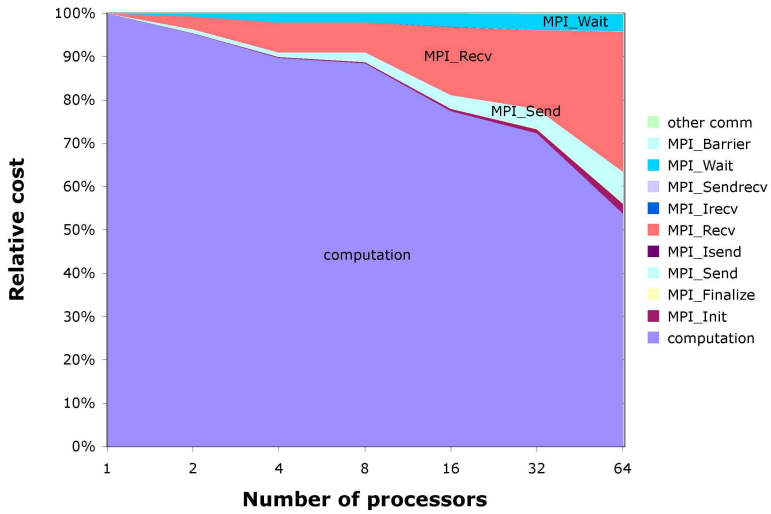


Figure A.14: Scalability of relative costs for communication primitives and computation for the MPI version of the NAS LU benchmark class A (size 64^3).

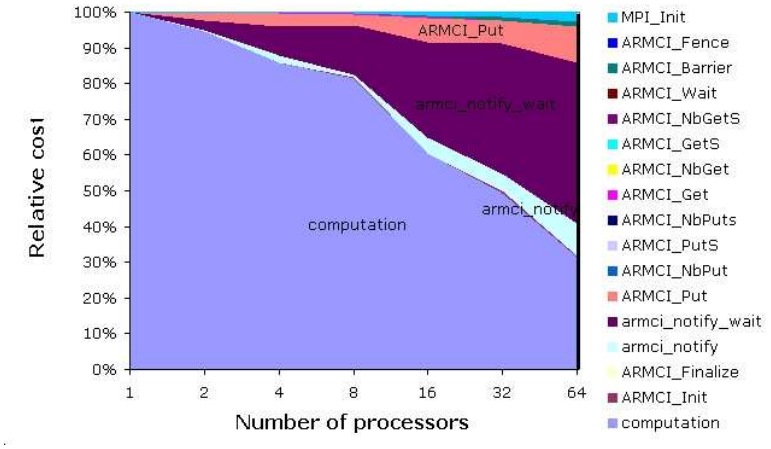


Figure A.15: Scalability of relative costs for communication primitives and computation for the CAF version of the NAS LU benchmark class A (size 64^3).

CPUs. The results in Figure A.19 show that the the nonscalability score for the main routine `applu` is 34%; the score for the routine `ssor`, which performs successive overrelaxation, is 33%, and for `cafinit` is 1%. Within `ssor`, the routine `buts`, which computes the regular-sparse block-upper triangular solution, has an $IAEW$ of 20%, the routine `blts`

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barrier
1	0	0	16	416	0	0	299
2	15756	121141440	9	232	16007	16004	299
4	31510	121141440	9	232	32012	32008	299
8	31510	95773440	9	232	32012	32008	299
16	31510	60132480	9	232	32012	32008	299
32	31510	44731200	9	232	32012	32008	299
64	63012	61375968	9	232	64018	64018	299

Figure A.16: Communication and synchronization volume for the CAF version of NAS LU, class A (size 64^3).

is responsible for 7%, `rhs` for 3%, and `jacld` for 3%. In Figure A.20 we analyze the scalability of the routine `buts`; the results show that the major reason for nonscalability is the `armci_notify_wait` primitive, used to determine if a data transfer to a local image completed. This result is consistent with the one determined using the first type of analysis.

A.5 Analysis of the NAS BT Benchmark

The NAS BT benchmark is a simulated CFD application that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. BT solves block-tridiagonal systems of 5x5 blocks [24] and uses skewed block distribution called multipartitioning [24, 148]. We discussed the MPI version of NAS BT in Section 3.2.1 and described the CAF version in Section 6.3.

In Figure A.21 we present the scalability of relative costs of communication primitives and computation for the MPI version of NAS BT. The overall loss of efficiency on 64 CPUs is 14%; the `MPI_Wait` routine shows worst scaling In Figure A.22 we present the relative costs of communication primitives and computation for the CAF version of NAS BT. On

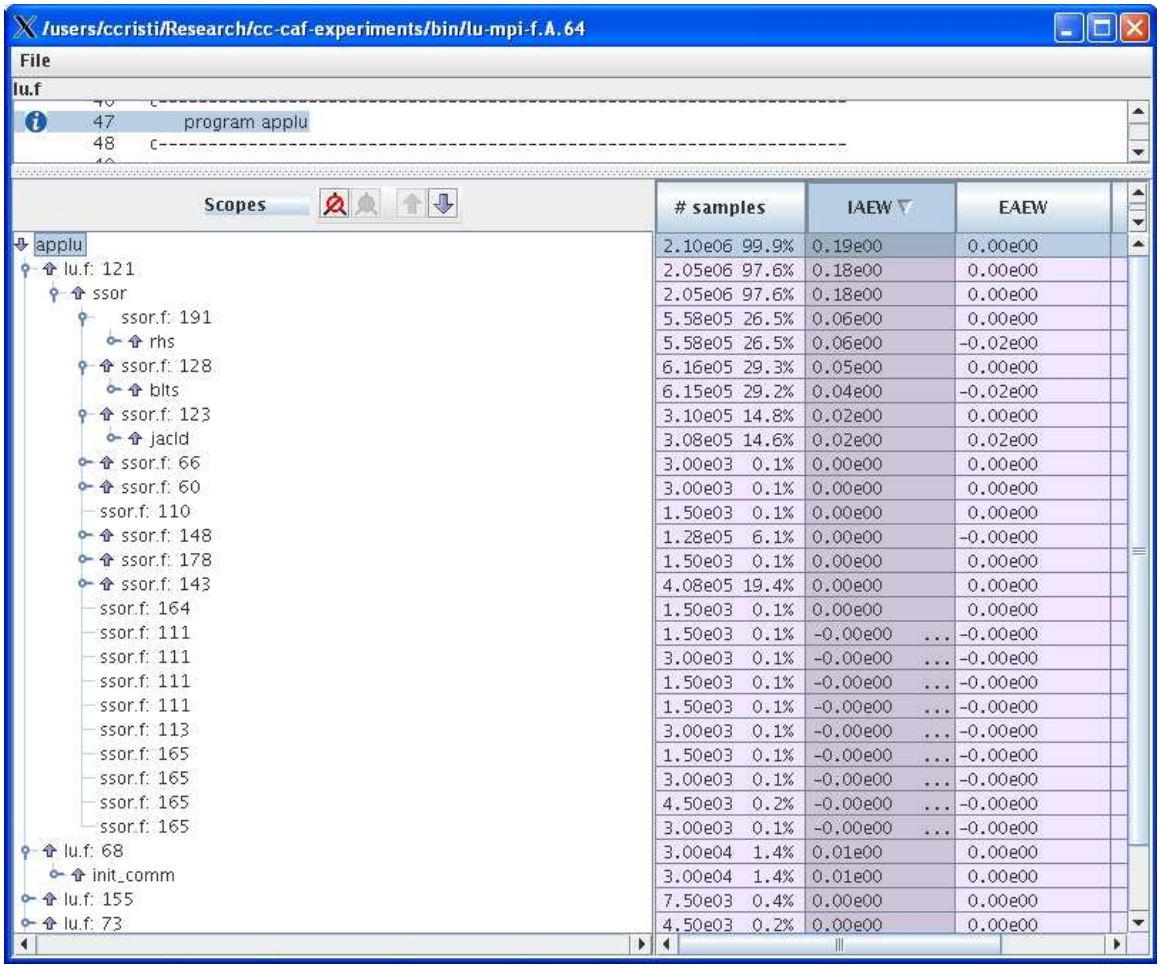


Figure A.17: Screenshot of strong scaling analysis results for MPI NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs.

64 CPUs, CAF BT loses 28%, with `ARMCI_Put` and `armci_notify_wait` being the least scalable communication primitives. In Figure A.23 we present a summary of the user-defined metrics for the volume of communication and synchronization. The profiling overhead was of 5-6% for the MPI and the CAF versions.

By inspecting the scalability graphs, we notice that computation amounts for 75-80% of the relative cost. Even though the number of `PUT`s increases also quadratically with the number of processors, the CAF implementation trades extra buffer for synchronization, reducing the cost of a handshake. The high relative cost of computation on cluster platform

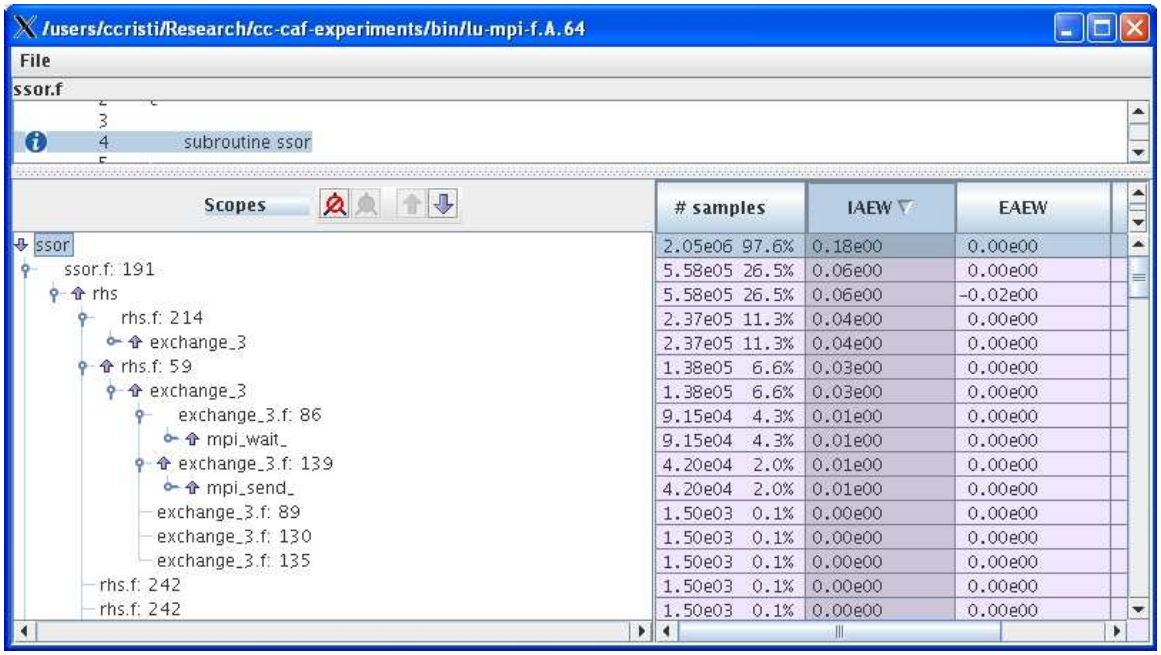


Figure A.18: Screenshot of strong scaling analysis results for the MPI version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the subroutine `ssor`.

explains why communication aggregation for BT didn't yield to significant improvement on a shared memory platform such as SGI Altix 3000.

In Figures A.24 and A.25 we present screenshots of strong scaling analysis results using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. The $IAEW$ score for the main routine is 5%, out of which the `adi` routine accounts for 4%. Inside `adi`, `x_solve` causes a 4% scaling loss, `y_solve` leads to a 2% loss, and `z_solve` causes a 1% loss. By further analyzing `x_solve`, as shown in Figure A.25, we determine that a call to `lhsx` has an $EAEW$ cost of 4%, which indicated that nonscaling node computation is a cause of nonscalability for MPI NAS BT.

In Figures A.26 and A.27 we present screenshots with results of strong scaling analysis for the CAF version of NAS BT using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs. Figure A.26 shows that the scalability score for the main routine `mpbt` is

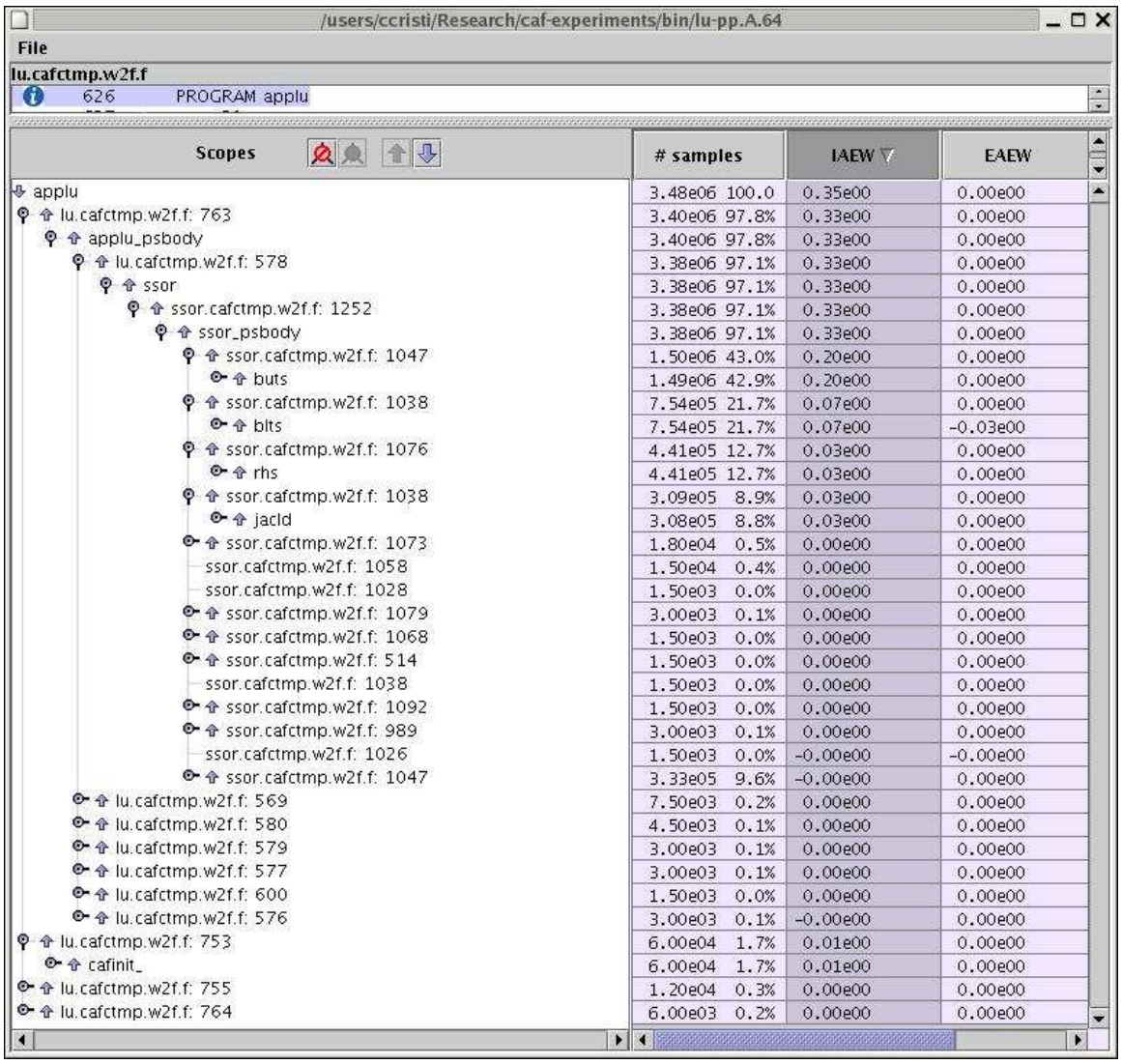


Figure A.19: Screenshot of strong scaling analysis results for the CAF version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs.

21%, with the main routine responsible for that being `adi`. Within `adi`, which performs alternate direction integration, `y_solve` has an $IAEW$ score of 12%, `x_solve` of 5% and `z_solve` of 4%. In Figure A.27 we present the analysis results for the `y_solve` routine, which performs alternate direction integration along the y dimension. The routine `lhsy_psbody`, which performs , is has an $IAEW$ and $EAEW$ values of 10%, exposing

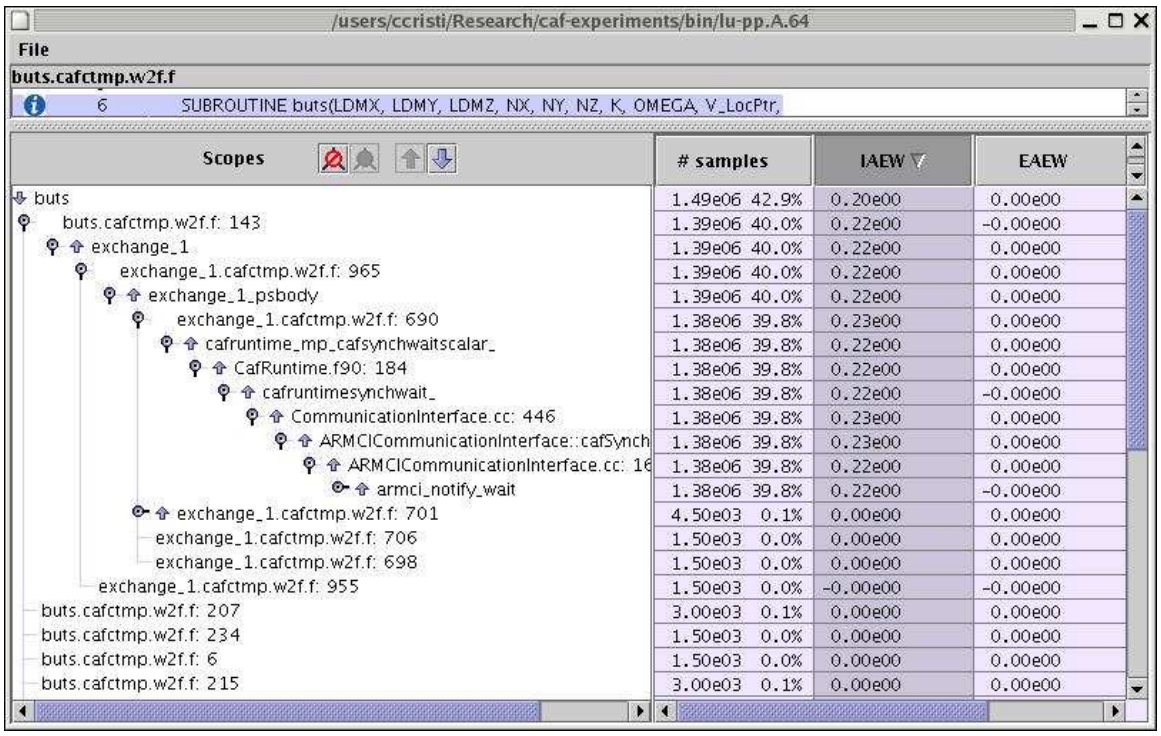


Figure A.20: Screenshot of strong scaling analysis results for the CAF version of NAS LU class A (size 64^3), using average excess work on 1, 2, 4, 8, 16, 32, and 64 CPUs, for the function `ssor`.

the fact that the computation performed by `lhsy_psbody` doesn't scale linearly with an increasing number of processors. In the routine `y_send_solve_info_psbody`, $IAEW$ for `ARMCI_Put` is 1%. Calls to `syncwait` and `y_solve_cell` have values of 1% and 1% for $IAEW$, respectively. This shows that for BT the main factor for non-scalability now is the lack of scalability of the computation.

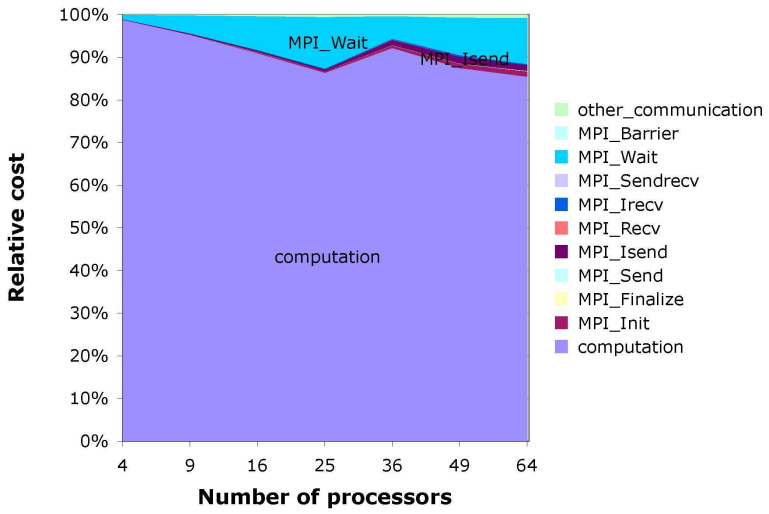


Figure A.21: Scalability of relative costs for communication primitives and computation for the MPI version of the NAS BT benchmark class A (size 64^3).

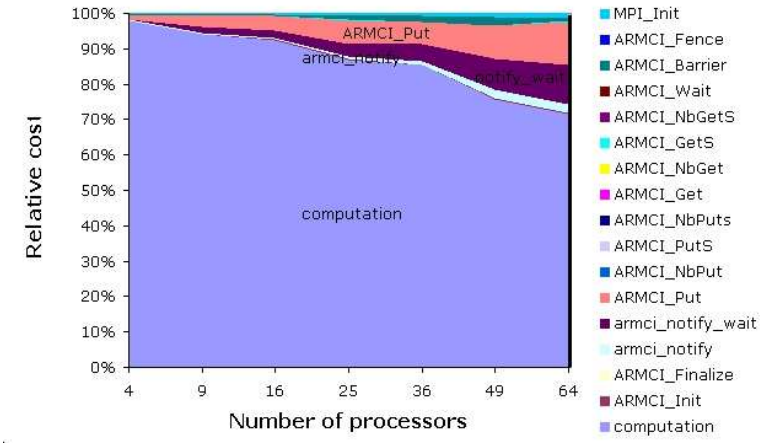


Figure A.22: Scalability of relative costs for communication primitives and computation for the CAF version of the NAS BT benchmark class A (size 64^3), using the ARMCI communication library.

CPUs	PUTs	PUT vol	GETs	GET vol	notifies	waits	barriers
4	3021	283153800	8	196	2820	2820	237
9	4830	252010080	13	424	4026	4026	237
16	6639	220849560	20	868	5232	5232	237
25	8448	178226400	29	1600	6438	6438	237
36	10257	157997160	40	2692	7644	7644	237
49	12066	149735520	53	4216	8850	8850	237
64	13875	139170360	68	6244	10056	10056	237

Figure A.23: Communication and synchronization volume for the CAF version of NAS BT, class A (size 64^3).

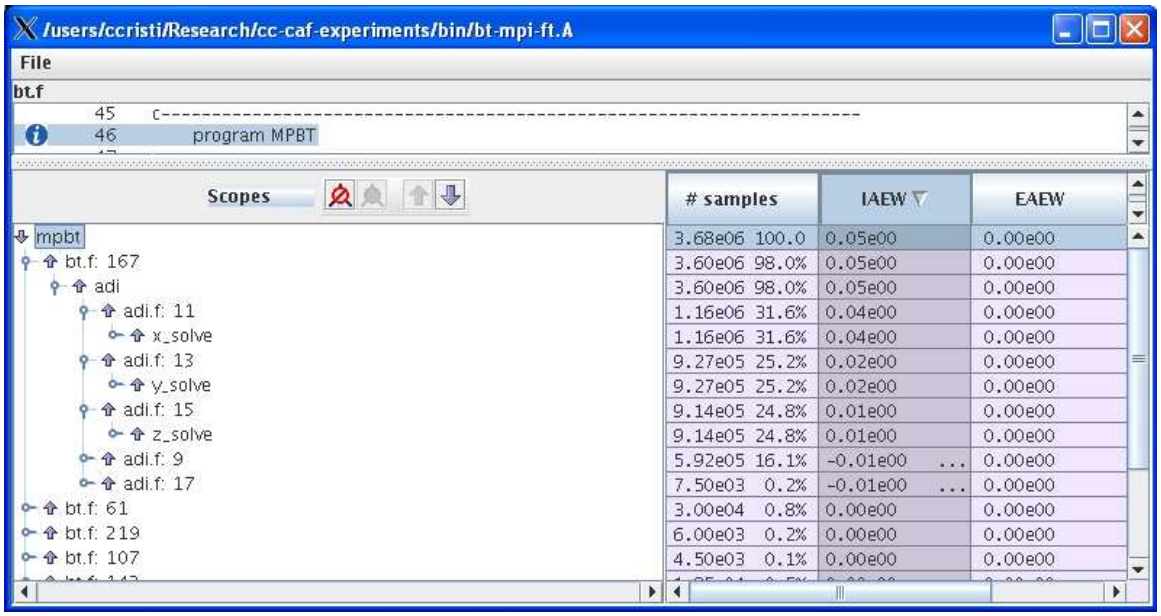


Figure A.24: Screenshot of strong scaling analysis results for MPI NAS BT class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs.

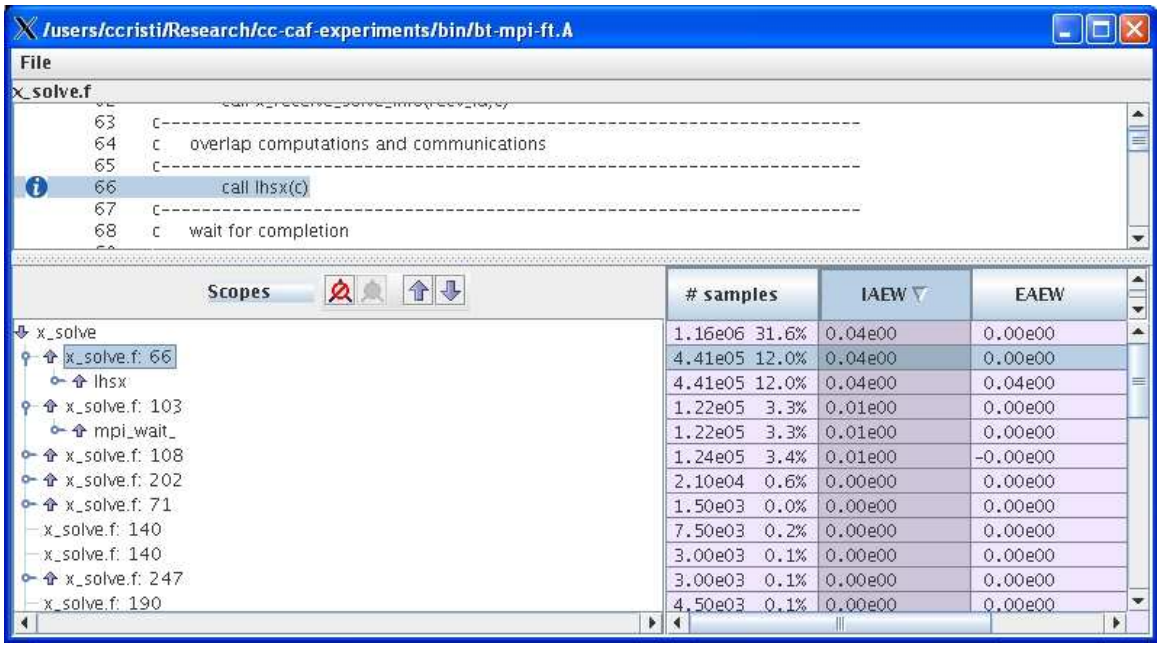


Figure A.25: Scalability of relative costs for communication primitives and computation for the CAF version of NAS BT class A (size 64^3), for the routine `x_solve`, using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs.

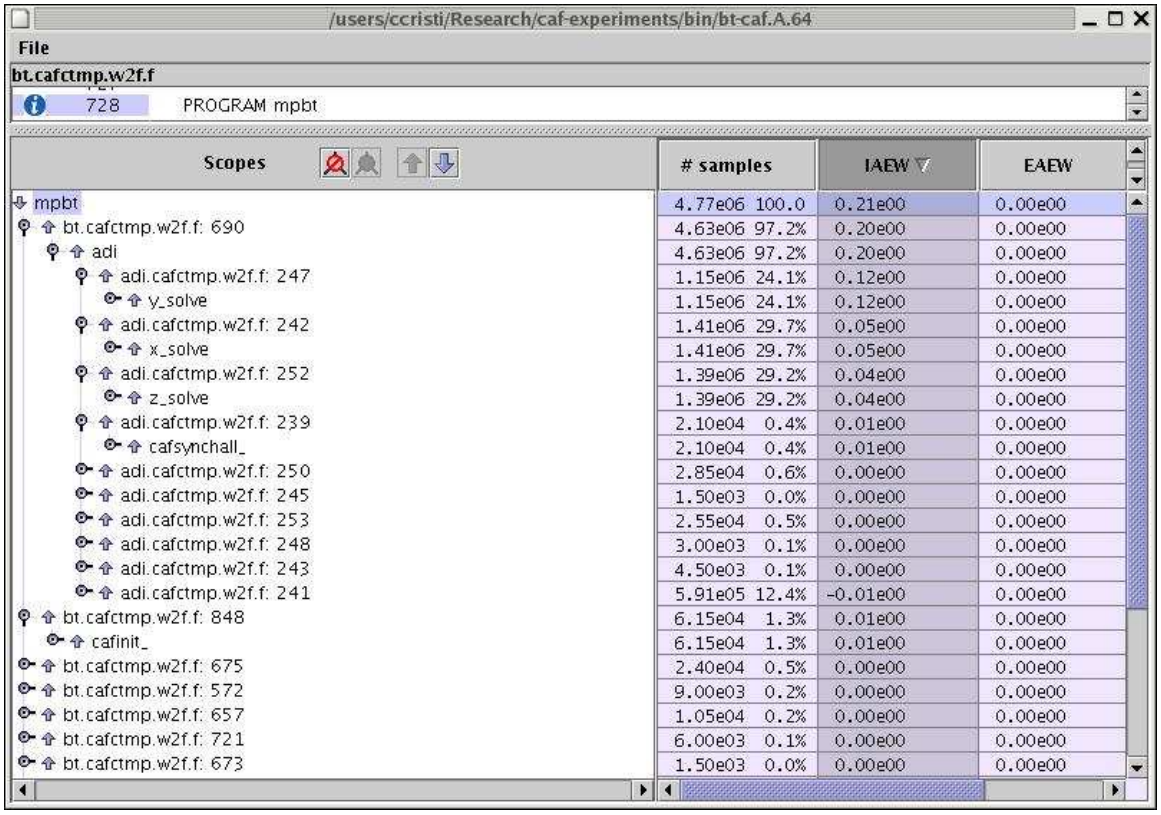


Figure A.26: Screenshot of strong scaling analysis results for the CAF version of NAS BT class A (size 64^3), using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs.

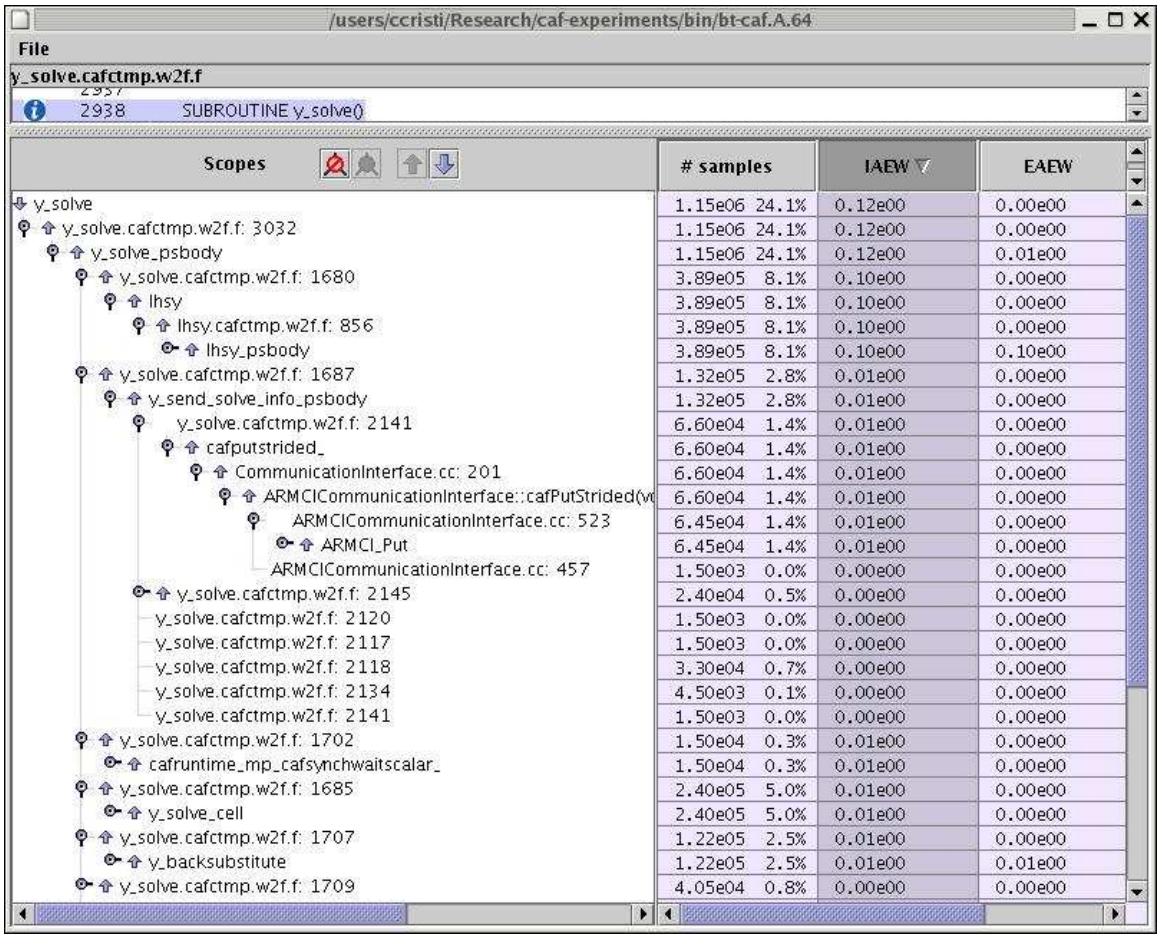


Figure A.27: Screenshot of strong scaling analysis results for the CAF version of NAS BT class A (size 64^3), for the routine `y_solve`, using average excess work on 4, 9, 16, 25, 36, 49, and 64 CPUs.

Appendix B

Extending CAF with collective operations

Our experiments showed that the lack of language support for collective operations leads to suboptimal, non-performance portable user implementation. In many scientific operations collective primitives such as reductions occur on the critical path, for example when checking a convergence criteria; having a performance portable way to provide collective operations for CAF programmers is then critical.

In this section we present an extension of the CAF model with collective operations. Many parallel algorithms [91] are designed using operations such as reductions, broadcast, scatter, gather, and all-to-all communication. We present CAF extensions that support these operations. We chose not to support the full set of reductions present in HPF, but rather support a minimal set of operations that suffice to express a wide range of commonly used parallel algorithms. We did not include for example dimensional reductions, because we did not see them utilized in the codes and algorithms that we analyzed. In particular, the proposed collective primitives were sufficient to express the collective communication encountered in our CAF benchmarks.

We describe an implementation strategy for CAF collective operations that realizes them using MPI calls. A motivation for our design choice is that, native implementations of MPI optimize the collective operations, and MPI is a performance portable translation target. However, for platforms where there are more efficient alternatives to MPI, `cafc` would choose the more efficient implementation for the collective operations.

For expressiveness and ease of use, the CAF collective operations should operate on scalar and multi-dimensional co-arrays, on private and shared variables; a CAF compiler runtime might optimize the implementation of the collective operation based on the type of

the arguments.

A CAF programmer should be able to use collective routines on the complete set of process images, but also on groups of processors. The design of process image groups is an orthogonal issue, and has been tackled by Dotsenko [72].

B.1 Reductions

CAF_REDUCE(SOURCE, DEST, SIZE, OPERATOR, root, [,UDFUNC][,group])

- OPERATOR
  - CAF_SUM
  - CAF_PROD
  - CAF_MAX
  - CAF_MIN,
  - CAF_AND
  - CAF_OR
  - CAF_XOR
  - UDFUNCCOMM: user defined reduction operator, commutative
  - UDFUNCNONCOMM: user defined reduction operator, non-commutative
- root: image that will contain the reduction result
- UDFUNC: user defined associative reduction operator
- group: group of processors

CAF_ALLREDUCE(SOURCE, DEST, SIZE, OPERATOR [,UDFUNC][,group])

- OPERATOR
  - CAF_SUM

  - CAF_PROD
  - CAF_MAX
  - CAF_MIN,
  - CAF_AND
  - CAF_OR
  - CAF_XOR
  - UDFUNCCOMM: user defined reduction operator, commutative
  - UDFUNCNONCOMM: user defined reduction operator, non-commutative
- UDFUNC: user defined associative reduction operator
- group: group of processors

CAF_PREFIXREDUCE(SOURCE, DEST, SIZE, OPERATOR [,UDFUNC][,group])

- OPERATOR
  - CAF_SUM
  - CAF_PROD
  - CAF_MAX
  - CAF_MIN,
  - CAF_AND
  - CAF_OR
  - CAF_XOR
  - UDFUNCCOMM: user defined reduction operator, commutative
  - UDFUNCNONCOMM: user defined reduction operator, non-commutative
- UDFUNC: user defined associative reduction operator

- group: group of processors

The user defined reductions operators have the following structure:

```
procedure UserDefinedOperatorInPlace(a,b)
   b = a op b
end procedure
```

where a and b have the same type and correspond to scalar types (primitive or user-defined types)

Comments and restrictions

- for CAF_REDUCE, only the image root receives a copy of the result after the reduction
- CAF_ALLREDUCE has the semantics of an all-to-all reduction: all images have a copy of the results at the end
- SOURCE, DEST have the same type and size SIZE is expressed in number of elements
- if group is not present, the reduction applies to all images
- there is an increasing, consecutive numbering of all images in group
- root is a valid image number
- arithmetic, relational and logical operators apply only for SOURCE and DESTINATION of the appropriate type
- if the type for SOURCE and DEST contains pointer fields, their values are undefined after the reductions; pointer fields cannot be used in the user-defined operator

B.2 Broadcast

CAF_BCAST(SOURCE, SIZE, root [,group])

- SIZE is expressed in number of elements
- root is a valid image number
- if the type for SOURCE contains pointer fields, their values are undefined after the broadcast; broadcast acts as if a bitwise copy is performed for the SOURCE data

B.3 Scatter/AllScatter

CAF_SCATTER(SOURCE, DEST, SIZE, root [,group])

- SOURCE and DEST have the same type
- SOURCE and DEST will be treated as one-dimensional, one-based arrays for the scatter operation
- SIZE is expressed in number of elements
- root is a valid image number
- if the group argument is not present, the scatter operation applies to all images
- there is an increasing, consecutive numbering of all images in group, from p_{lb} to p_{ub}
- Considering SOURCE as a unidimensional array, after scatter every image p (including root) contains in DEST the array section SOURCE$((p - p_{lb}) * SIZE + 1 : (p - p_{lb}) * SIZE))$ on the root image
- the argument SOURCE is optional on any non-root image

B.4 Gather/AllGather

CAF_GATHER(SOURCE, DEST, SIZE, root [,group])

- SOURCE and DEST have the same type
- SOURCE and DEST will be treated as one-dimensional, one-based arrays for the gather operation
- SIZE is expressed in number of elements
- root is a valid image number
- if the group argument is not present, the gather operation applies to all images
- there is an increasing, consecutive numbering of all images in group, from p_{lb} to p_{ub}
- after gather, the root image contains in DEST$((p-p_{lb})*SIZE+1:(p-p_{lb})*SIZE))$ the contents of SOURCE$(1:SIZE)$ on image p.
- the argument DEST is optional on any non-root image

CAF_ALLGATHER(SOURCE, DEST, SIZE [,group])

- SOURCE and DEST have the same type
- SOURCE and DEST will be treated as one-dimensional, one-based arrays for the allgather operation
- SIZE is expressed in number of elements
- root is a valid image number
- if the group argument is not present, the gather operation applies to all images
- there is an increasing, consecutive numbering of all images in group, from p_{lb} to p_{ub}
- after gather, every image contains in DEST$((p-p_{lb})*SIZE+1:(p-p_{lb})*SIZE))$ the contents of SOURCE$(1:SIZE)$ on image p.

B.5 All-to-all Communication

CAF_ALLTOALL(SOURCE, DEST, SIZE [,group])

- SOURCE and DEST have the same type
- SOURCE and DEST will be treated as one-dimensional, one-based arrays for the allgather operation
- SIZE is expressed in number of elements
- if the group argument is not present, the gather operation applies to all images
- there is an increasing, consecutive numbering of all images in group, from p_{lb} to p_{ub}
- after gather, every image q contains in DEST$((p-p_{lb})*SIZE+1:(p-p_{lb})*SIZE))$ the contents of SOURCE$((q-p_{lb})*SIZE+1:(q-p_{lb})*SIZE))$ on image p.

B.6 Implementation Strategy

A portable implementation strategy is to translate these collective operations into their corresponding MPI counterparts; both ARMCI and GASNet support interoperability with MPI. If both source and destination are co-arrays and the underlying communication library has a more efficient implementation of a collective operation than the one provided by MPI, then `cafc` would choose at runtime the native implementation of the collective over the one provided by MPI. For primitive types, the translation is straightforward. MPI provides a rich set of primitive types that matches the set of primitive types of Fortran 95; `cafc` would pass as an argument to the MPI collective operation the MPI datatype corresponding to the CAF type. For user defined types, we determine at program launch the size (including padding) of a user defined type, and declare an opaque MPI datatype of the same size as the user defined type. This approach is sufficient to support broadcast, scatter, gather and all-to-all operations. To support user defined reductions, we need to generate functions corresponding to the user defined operators in the format specified by MPI.

```
void MPIUserDefinedFunction(invec, outvec, len, mpi_datatype)
  type invec(*)
  type outvec(*)
  integer len
  integer mpi_datatype
```

A simple solution is to generate a wrapper with the proper set of arguments, iterate through `invec` and `outvec` and call the user specified reduction operator with the corresponding elements from `invec` and `outvec`. However, this version would be very inefficient, because it would incur a function call cost per array element. A more efficient approach is to declare an attribute for the user defined operators, acting as a flag to the compiler. `cafc` could then synthesize at compile time a user-defined reduction operator which follows the MPI requirements, but *inlines* rather then calls the user defined operator. Another argument for annotating user-defined operators is that the reduction operator needs to be registered with the MPI library. If we do not flag to `cafc` the user defined operators, then we have to generate the functions required by MPI per each callsite of such reduction, which might inquire a large space penalty for programs that perform many reductions with user-defined operators.

B.7 Experimental Evaluation of Reductions

We have implemented support for broadcast and reductions of primitive types in `cafc` using the MPI collectives as translation target. For MG, after replacing the suboptimal user-written collective calls (broadcast and allreduce operations) with CAF intrinsics based on MPI, the initialization time decreased by to 40% on 64 processors. In Figure B.1 we present the the parallel efficiency plot for LBMHD using the prototype implementation of CAF collective intrinsics; one observation is that our translation scheme does not introduce a high overhead over direct calls of MPI primitives.

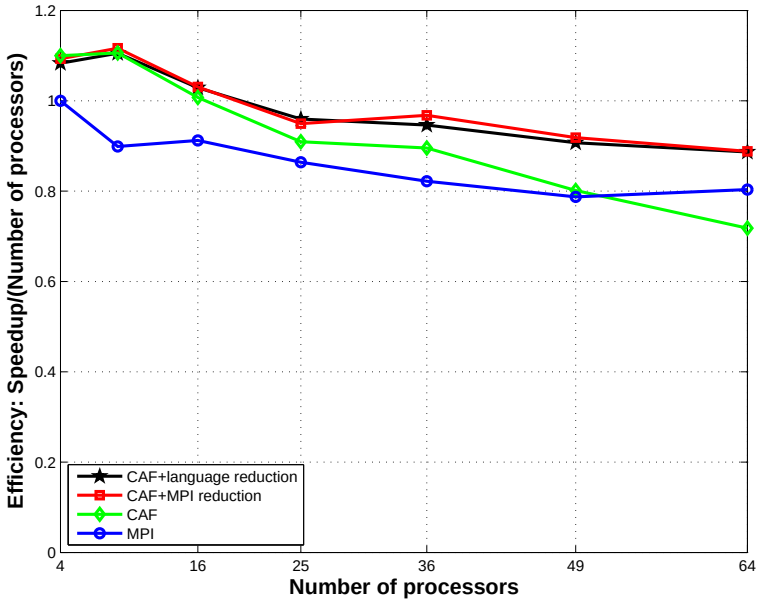


Figure B.1: Scalability of MPI and CAF variants of the LBMHD kernel on an Itanium2+Myrinet 2000 cluster.