

Proceedings of the  
First Workshop on  
Productivity and Performance in  
High-End Computing  
(PPHEC-04)

February 14, 2004  
Madrid, Spain

Held in conjunction with the  
Tenth International Symposium on  
High Performance Computer Architecture

**Program Chair:**

Ram Rajamony, IBM Research

**Program Committee:**

Victor Basili, University of Maryland at College Park

James Browne, University of Texas Austin

John Carter, University of Utah

John Grosh, OSD

Mary Hall, USC Information Sciences Institute

Jeremy Kepner, MIT Lincoln Labs

Karthick Rajamani, IBM Research

Ram Rajamony, IBM Research

Marc Snir, University of Illinois at Urbana-Champaign

1<sup>st</sup> Workshop on Productivity and Performance in High-End Computing  
February 14, 2004  
Madrid, Spain

Morning Keynote: The Coming Crisis in Computational Science .....	6
<i>Douglass E. Post, Los Alamos National Laboratory</i>	
Application Development Productivity Challenges for High-End Computing .....	14
<i>Vivek Sarkar, Clay Williams, and Kemal Ebcioğlu, IBM T. J. Watson Research Center</i>	
Comparing Network Processor Programming Environments: A Case Study .....	19
<i>Niraj Shah, William Plishker, and Kurt Keutzer, University of California, Berkeley</i>	
Templating Transformations for Bitstream Programs .....	27
<i>Armando Solar-Lezama, and Rastislav Bodik, University of California, Berkeley</i>	
Performance and Productivity in Parallel Programming via Processor Virtualization .....	40
<i>Laxmikant V. Kalè, University of Illinois at Urbana-Champaign</i>	
Afternoon Keynote: Building High Performance Scientific Applications for Parallel and Distributed Systems using a Software Component Architecture .....	51
<i>Dennis Gannon, University of Indiana</i>	
Introducing the “Application Kernel Matrix” .....	52
<i>Brad Chamberlain, John Feo, and David Mizell, Cray Inc.</i>	
Managing Complexity in Modern High End Scientific Computing through Component-Based Software Engineering .....	56
<i>David E. Bernholdt, Oak Ridge National Laboratory; Robert C. Armstrong, and Benjamin A. Allan, Sandia National Laboratories</i>	
The High-Level Parallel Language ZPL Improves Productivity and Performance .....	66
<i>Bradford L. Chamberlain, University of Washington and Cray Inc.; Sung-Eun Choi, Los Alamos National Laboratory; Steven J. Deitz, and Lawrence Snyder, University of Washington</i>	
Raising the Level of Programming Abstraction in Scalable Programming Models .....	76
<i>David E. Bernholdt, Oak Ridge National Laboratory; Jarek Nieplocha, Pacific Northwest National Laboratory; and P. Sadayappan, Ohio State University</i>	

Proceedings of the  
First Workshop on  
Productivity and Performance in  
High-End Computing  
(PPHEC-04)



# Morning Keynote

## The Coming Crisis in Computational Science

Douglass E. Post  
Los Alamos National Laboratory

# The Coming Crisis in Computational Science

Douglass E. Post

*Los Alamos National Laboratory, Los Alamos, NM 87544*

*post@lanl.gov*

## Abstract

The enormous increase in computing power over the last forty years has opened up new opportunities to analyze and solve important problems that face society. Computer simulations and analysis have unprecedented potential to address complex, non-linear phenomena with realistic data and geometries. Modern scientific research, originally grounded in experiment and its theoretical interpretation, is becoming a triad of experiment, theory and computation. Computer simulations and analysis are beginning to be used for engineering design and public and private policy decisions. Yet computational science is not nearly as mature as the traditional methodologies and is undergoing “growing pains”. This is not surprising as history indicates that it takes time, and quite a few major and possibly dramatic mistakes, for a methodology to mature. These major mistakes are beginning to occur. The computational science community needs to analyze these mistakes and learn from them if it is to succeed.

## 1. Introduction

Computational science—the use of large scale computers to address and solve important technical problems—is becoming an everyday tool for design and analysis of complex technical issues. Applications include scientific research, engineering design, policy analysis, training and emergency response and environmental analyses. Computational science has the potential to address complex issues with a degree of realism that has heretofore only been imagined. This exciting and very important—indeed revolutionary—potential is due to the enormous growth in computer power (speed and memory) over the last 50 years. This growth shows no sign of slowing in the near term. Yet computational science is a very new and immature discipline. It has not achieved the level of maturity of traditional methodologies such as experiment, theory, engineering design and conventional policy analysis for solving problems.

For example, when a scientist receives a computational science paper from a journal to referee, he has no definitive way to determine if the paper is correct. He cannot reproduce the results in the paper, and generally he can’t check the important results with experimental data. The most important results typically make predictions for situations for which there is no data. That’s often the purpose of the calculation, after all. Even if he had a listing of the code—the referee usually doesn’t—that is not enough to determine the validity of a very complex and large calculation. All that he can do is to subject it to a series of “plausibility” checks. Is the paper consistent with known physical laws? Is the author a reputable scientist, known for careful work? Are the results consistent with other work in the field? Is the code validated with data as close as possible the regimes of application? Do the computational methods seem sound and applicable to the problem? Are the original model and the fundamental equations correct? These criteria are not nearly as reliable as the criteria used for theoretical or experimental papers. A reviewer can re-derive the important formulae in a theoretical paper. Experimental science is a well-established methodology, and important experiments are duplicated fairly quickly. In fact, important experimental results are usually not accepted by the general scientific community until they are confirmed by independent experiments. “Discoveries” like cold fusion have their moment of fame then fade into infamy as “irreproducible” results. Reproducibility is a cornerstone of sound science.

In addition, these criteria discriminate against the reporting of new and exciting results, since such results usually cannot be thoroughly checked, and are probably, but not always, wrong. Major new contributions are thus less likely to survive the refereeing process in favor of more modest extensions of previously accepted work.

Many things could be wrong with the computational science paper that the referee could not detect. The code could have errors in the way it was written such as bugs, the wrong use of computer or mathematical algorithms, inadequate resolution in time or space, unconverged solutions, etc. Even if the code had few errors, the models

and equations in the code could be inadequate or wrong. As Robert Laughlin points out, “One generally can’t get the right answer with the wrong equations.” [1] The physical data used in the code may not have adequate resolution or may be inaccurate. The scientist or engineer running the code may not know how to set up or run the problem correctly. He may not know how to interpret the results of the code accurately. Yet referees are expected to judge the correctness of the paper. It’s a challenge the community needs to address.

Even more importantly, significant scientific, engineering design and public policy questions are beginning to be decided on the basis of computational science results. As a community, it is our responsibility to ensure that computational science achieves the same level of reliability as theoretical and experimental science and engineering design. Otherwise computational science will not be a credible methodology, and its potential for contributing to the betterment of the human condition will not be realized. If a significant number computer predictions and analysis are wrong, and there is no way to determine which ones are right and which are wrong, people will not rely on them and will not support the development of the field.

In “Design Paradigms”, Henry Petroski traces the roughly four steps needed for an engineering technology to reach maturity[2]. Among his examples is suspension bridges (Figure 1). The first step involves the design and construction of the first suspension bridges. The designers and construction crews did not know the design limits. The designs therefore were very conservative and extensively over-engineered. Although there are often initial failures, the designers generally got it right fairly quickly. An example is the Széchenyi chain bridge over the Danube joining Buda and Pest constructed in 1840. It was stood for 105 years until the Germans destroyed it in World War II. It was rebuilt in the 1990’s and stands today. The second step involves cautious design improvement and optimization. An example is the Brooklyn Bridge constructed in 1880 by John and Washington Roebling. It is still standing and carrying a modern traffic load after over 120 years. The third step involves the development of continually more ambitious designs that push the limits of the existing technologies until failures occur. The cautious approaches and the deep fear of failure of the prior generations of designers are often forgotten in the enthusiasm to go beyond the achievements of the past. The Tacoma Narrows bridge, constructed in 1940 failed catastrophically due to the excitation of harmonic oscillations driven by wind. Such bridge failures are spectacular. Almost everyone who reads this paper has seen the short movie of the galloping Tacoma Bridge as it bucked and moved in the wind until it

collapsed into the river. The civil engineering community studied and analyzed the causes for the failures, then developed solutions that became part of the design methodology for all future suspension bridges. This fourth step leads to a mature field based on the development and adoption of the “lessons learned” from the failures and successes. Now very large suspension bridges are being built such as the 1991 m span Akashi Kaikyo Bridge in 1998.



**time**

**Lessons Learned  
Case Studies**

Figure 1 History of Suspension Bridges[2].

Computational science is in the midst of the third step on the path to maturity. The first generation of computational science involved the use of the supercomputers of the 1950's, 1960's and 1970's. The authors of these applications used the codes to analyze data, design nuclear weapons, model supernovae, conduct engineering analyses, etc. Computational science was a new field and everyone was very aware that it had limitations. Due to restrictions in memory and processing speed, the problems generally did not have adequate spatial or temporal resolution and the solutions were often not converged. Many times only very approximate models were employed for the problems being addressed. Nonetheless, computational tools were a step forward, and—used with caution and carefully verification and validation—produced better answers than could be obtained with traditional analytic techniques. As computers became more powerful, the DOE and the NSF established “supercomputer” centers in the US to provide supercomputer capability to the academic and general national laboratory community. The DoD used supercomputers to address important national security issues. Industries such as Boeing and General Motors used supercomputers for engineering analyses of aircraft, engine and structural automobile components. There was still generally a strong component of skepticism about computational results and as a consequence, computational predictions were thoroughly checked and validated.

By the 1990's, computing power had reached the point where many of the prior limitations on resolution and ability to solve complex mathematical systems had been overcome. Computational techniques began to really have the potential to address difficult and important problems such as climate change and weather prediction, nuclear weapons design, astrophysics, non-linear turbulence, chemistry, biology and human event simulation. This coincided with the advent of a new generation of scientists and engineers who were specifically trained as computational scientists. They began to use computational techniques to tackle many very difficult and complex problems. While these scientists and engineers were highly skilled at using computers, many did not have the inherent skepticism about computational results that was characteristic of prior generations. Although they knew that computational models are only an incomplete model of nature, they have sometimes placed an unwarranted faith in the validity of the computational results.

There are many examples of computational analyses and results that were important elements of a policy or design decision, but were later determined to have been inadequate or seriously flawed. Among these, in my talk I will

discuss some examples in enough detail to illustrate the main elements of a “lessons learned” exercise. One candidate example involves an assessment of potential failure modes of a large engineering system. The assessment indicated that there was sufficient margin against failure. However, the system failed resulting in destruction of the system and loss of life. Further analysis after the failure indicated that the computational analysis procedure was seriously flawed, leading to a false sense of confidence in the reliability of the system.

The second assessment involved the use of computer modeling to “explain” a new and very exciting, but very surprising, experimental discovery. A computer simulation was able to “reproduce” the experimental result by changing the boundary conditions much more than was reasonable based on the physics of the problem. By treating the boundary conditions as “free parameters”, the simulation could “reproduce” the experimental results. With this confirmation, the authors published their results. The publication caused a great deal of excitement. Many groups began to repeat the experiment, but none were able to achieve the results reported in the paper. The original experimental group brought in a second group from their laboratory to repeat the measurement, and the second group found a null result. The original analysis of the experimental data was determined to have been flawed, and the result was withdrawn. All of this might have been avoided had the computer simulation group not treated important physics constraints as “free parameters”. The simulation could have served to alert the experimental team that their results were not correct, but instead it gave them false confidence in their results.

Another example involved a theoretical prediction of the performance of a proposed new facility. Based on extensive analysis of the results of smaller facilities by the international community in a particular field, it had been proposed to build a large experiment. Just as the design of the large experiment was being completed by an international team, a small group of three theorists completed a computer simulation based on a new code of the expected performance of the proposed facility. Their initial results indicated that the performance would be poor due to instabilities and that the proposed facility would not be able to achieve its goals. Their results were widely distributed in the popular media, and contributed strongly to several partners withdrawing from the project. Extensive analysis by the international community led to the realization that the three theorists had left out important effects that stabilized the instabilities, and that the expected performance would be roughly what the original design team had predicted. In this case, a computational prediction that was later proven to be

wrong had an important impact on a scientific policy issue.

There are many other examples that are also available. These examples illustrate that the computational science results are beginning to play an important role in society, but not always a positive role. If this role is to be a positive one, we, as a community, must work to achieve the level of maturity for which our results are accurate and reliable. As in the case of suspension bridges, we must start analyzing our failures and successes, and learn from them. To illustrate some of the kinds of “lessons learned” analyses we will need to conduct, I describe an exercise that Richard Kendall and I carried out for six computer simulation projects in the nuclear weapons program[3]. The analysis we carried out emphasizes the code development process more than the validity of the results of the computations, but reliable answers requires a mature methodology for development of the analysis tools.

## 2. “Lessons Learned from ASCI”

Richard Kendall and I developed a set of “lessons learned” from the US Department of Energy’s Accelerated Strategic Computing Initiative (ASCI) program[3]. Since 1996, the ASCI program has spent over \$3B to develop the predictive nuclear weapons simulation capability required for certification of the stockpile without nuclear testing. The ASCI program has elements that focus on the development of large scale, massively parallel computer platforms, the associated operating systems and code development tools, application codes and supporting algorithms and models. Some of the applications development ASCI projects have been successful in meeting their objectives and some have not. We analyzed the application projects at the Lawrence Livermore National Laboratory (LLNL) and the Los Alamos National Laboratory (LANL) utilizing metrics and case studies that focused on the history, organization and institutional support of the code projects. By identifying the common elements that led to success or failure to achieve objectives and comparing them to the experience of the information technology (IT) community (e.g. [4]), we developed a set of recommended practices for large-scale technical code projects (Table 1).

Table 1: Code Development “Lessons Learned” from the ASCI Program at LANL and LLNL

1. Build on the successful code development history and prototypes for your organization.
2. Good people in a good team are essential for successful code development projects.
3. Software Project Management: Run the code project like a project.

4. Risk identification, management and mitigation are essential for successful code development.
5. Determine the schedule and resources from the requirements (goals and objectives, quality, team building and survival, and added value), not independently.
6. A strong customer focus is essential for success.
7. Better physics in a physics code is much more important than better computer science.
8. Use modern but proven Computer Science techniques; do not let your project become a Computer Science research project unless it is one.
9. Train the teams in project management, code development techniques and the physics and numerical techniques used in the code
10. Software Quality Engineering: Use Best Practices to improve quality rather than processes.
11. Validation and Verification of codes are essential.

While the “lessons learned” list may seem obvious and certainly contains no surprises, every code project we studied violated at least a few. Almost all were violated for the least successful projects. These lessons are generally not new. Indeed, many of these lessons can be found in Fred Brooks’ 1975 classic: “The Mythical Mon-Month” [5]. Also, these principles apply to many other organized human activity[6].

1. Identify the things your organization or institution does well and build on them. Introduce change with clearly defined goals in an evolutionary fashion. Even though you may think that the ideal structure for effective code development might be radically different from the existing organization and culture, radical, instant changes will disrupt whatever is working, and likely will not lead to success. Successful change takes time and requires that the people in the institution feel “safe” and trust the management to treat them fairly.

2. Teams, not organizations or processes, develop software. Form the best team you can, support it, and help it “jell.” A good team is the strongest asset an institution can have. Developing good teams is the key to developing good software. A good team is also a crucial deliverable for a successful project because any further progress must build on the team.

3. Run the code project like the project that it is, with requirements, deliverables, a sound plan, realistic schedules and adequate resources. Align authority with responsibility. The project manager must be able to control the resources and the team, and have the active support of senior management. Otherwise, he is a project “cheerleader,” not a project leader, and the project will fail.

4. The development of large, complex technical codes is inherently very risky. Many, if not most, development efforts fail to meet their initial objectives, and quite a number fail completely. Identifying the major risks,

minimizing them and providing contingency and mitigation is essential. The major risk factors for software projects are [7]:

- uncertain goals, objectives and requirements;
- inadequate resources and support, including an overly ambitious schedule;
- institutional turmoil, including too much employee turnover;
- requirements and goals that change too rapidly or increase too fast; and
- poor team performance.

Poor team performance is the smallest risk factor for the ASCI code projects [3] and for the general software industry [7]. The other risk factors strongly dominate. Most code project failures (for ASCI as well as the general IT community—[4, 7]) are due to the failure of senior management to fulfill its responsibility to provide guidance and support for the code projects. The ASCI projects, and most technically oriented code development efforts, have additional risks because they must develop new algorithms and rely on other projects and groups to provide essential components.

5. If adequate resources and schedule are not provided, the project will fail to meet its objectives on time. This is a failure itself and may initiate a chain of events that will cause it to fail altogether. There is even less flexibility for software development than for conventional projects where one can fix two, but not three of *objectives, resources* and *schedule*. For software, one can only fix the objectives. The *objectives and goals* determine the *resources* and *schedule*. The rate limiting process for code development is the rate at which people can analyze problems and develop solutions. The ability to increase the schedule is severely limited. Similarly, the maximum size of a code team is limited by the ability of people to communicate complicated information with each other. This is reflected in the quantitative analysis that follows in the next section. The standard estimation techniques indicate that the optimal schedule and team size are a function only of the size and complexity of the code[8]. Frederick Brooks put it another way: “Adding more staff to a late project will only make it later.”[5] Ed Yourdon wrote a book entitled “Death March” about the disastrous consequences of overly ambitious code project schedules[9].

6. Codes that customers do not want to use are like experiments that do not take data or equipment that people do not use. Such codes are a waste of resources and the efforts of creative people. The code team and management must focus on providing what the customer both *needs* and *wants*. If the customer does not want or like the product, the code will fail even if it is what he really needs.

7. The value of the code to the ASCI customer is the physics capability of the code. The degree of innovative computer science in the code is of little interest to him. The most successful ASCI codes have concentrated on

improved physics and have been very conservative in their use of cutting edge computer science.

8. Similarly, computer science research within the context of an application project greatly adds to the risks and often results in code project failure. Use modern, but proven techniques. Improving the physics is risky enough. Leave computer science experiments to those who can afford to fail a few times.

9. Invest in your people through training and professional development. They will become more capable as they acquire new skills and will be more productive. It is a good way to encourage change and to get the team members to see how other groups and industries tackle their problems. In addition, their morale will increase in proportion to the support of their management. Training also provides an opportunity for code team members to share experiences with the rest of the team and with other teams.

10. Software quality is important. However, research-oriented staff are not going to take a series of processes defined in a book and follow them blindly because someone in authority tells them to. They will apply the same standard to software development methods that they apply to their science. They have to be convinced that it is right and adds value to their work. It is more successful to convince the teams that individual practices add value (configuration management, etc.) than to try to convince them that a large system of processes is something they should blindly embrace on faith.

11. Physics codes are an incomplete representation of reality. The models have shortcomings and often have mistakes in their implementation. Without a verification and validation program for the codes and their applications, there is no reason to believe that the code results have any validity at all.

The software quality issue is important. If poor quality codes are being produced, the sponsors and customers will take action. Already the DoD and other customers have developed fairly rigid processes for code development and software quality assurance in response to disastrous results due to poor quality software aircraft and satellite control software. The Air Force cannot tolerate bugs in aircraft control software that result in plane crashes. They have established a very rigorous procedure for vendors to follow to develop such software[10]. An analogous situation existed in the automobile industry in the 1970's and 1980's. The American automobile was producing poor quality cars and people were beginning to stop buying them. The Japanese were building cars that had high quality and people were buying them. A basic difference was that the US automobile industry did not sufficiently emphasize quality on the assembly line and in components. They tested the cars after they came off the assembly line and fixed the worst ones. The Japanese, on the other hand, emphasized quality at every step of the assembly process and for components. They tested the cars at many points of the assembly and tested the components before

installation. The net result was that far fewer cars with poor quality emerged from the assembly line, and got into the hands of consumers. The value of software quality assurance is likely to be maximized when it is applied at all steps of the software development process, rather than just at the end. However, just as in the assembly line, different development processes require different methods, no one size fits all. Also, just as the Japanese emphasized input from the assembly floor, the code developers themselves are often the best judges of how to implement quality. A process blindly imposed from above will likely get the same type of malicious compliance observed in the US auto industry.

### 3. Quantitative Estimation

The above lessons learned were based on a qualitative and a quantitative analysis of the history of the different code projects and comparison with the information technology industry and conventional project management and scientific research. The quantitative analysis was a key element in establishing that the ASCI code projects had not been given a consistent set of requirements, resources and schedules. Our analysis [3] was relatively simple compared to the methods often employed in the Information Technology community[11], but was nonetheless revealing. We analyzed simplified case studies for many of the DOE's Accelerated Strategic Computing Initiative (ASCI) code projects to identify the key factors that determine the success or failure of complex scientific code projects. We found that the key predictor was the age of the code project and the amount of time allocated to complete the project and meet milestones. Our analysis of the historical data indicated that it takes about 8 years to develop an ASCI weapons code. The projects that had 8 years of development often succeeded, and those that did not have 8 years of development all failed to meet their initial milestones. This result emphasized the crucial need for a consistent set of requirements, resources and schedule.

The case studies included metrics (code size, team size, age, etc.). To see if the ASCI experience is consistent with the Information Technology (IT) community, we analyzed the case studies using a generic "function point" model [11] widely used by the IT industry. We calibrated this model for scientific code projects using the ASCI case study data. Function points are a weighted total of inputs, outputs, inquiries, logical files and interfaces [11, 12]. Functions Points were not developed for technical software, but were the best measure we could find.

$$FP = \left( \frac{C++ \text{ SLOC}}{53} + \frac{C \text{ SLOC}}{128} + \frac{F77 \text{ SLOC}}{107} \right) \quad (eq. 1)$$

$$Schedule(months) = FP^x \quad 0.4 < x < 0.5; \quad use \quad x = .47 \quad (eq. 2)$$

$$Team \text{ Size} = \frac{FP}{150} \quad (eq. 3)$$

We first converted the single lines of code to Function Points (FP)(e.g. eq. 1). T. Capers Jones lists the equivalent single lines of code (SLOC) per function point (FP) for the common computer languages [11]since computer languages have different information densities.

$$Schedule = Contingency \times Function \text{ Point} \text{ schedule} + Delays \quad (eq. 4)$$

In this model, the required schedule and average team size are determined by the Function Point (FP) count (eqs. 2,3). We calibrated and modified these general scalings to account for the added complexity and viscosity associated with developing scientific codes

$$Team \text{ Size} = 3 + \frac{FP}{150} \cdot 0.6 \quad (eq. 5)$$

specifically for the nuclear weapons complex. We increased the schedule by 1.5 years to account for the additional time it takes to recruit, hire, train and get security clearances for code development staff. Using a methodology developed by the Lawrence Livermore National Laboratory Engineering Department[13], we calculated a contingency factor of 1.6 to account for the additional risks, uncertainties, complexities, etc. for the LANL and LLNL computing environments. We modified the standard FP scaling for the size of the code team (eq. 5)[11] to match the ASCI data. We included a correction for small code teams.

We analyzed seven code projects, three at LLNL and four at LANL (Table 2). For security reasons, we have identified the LLNL codes with the letters A, B and C. Table 2 lists the size of the code in function points, the time estimated by equation 4 to develop the initial capability of the code project, the actual age of the code at the point it was expected to accomplish its first milestone, whether or not the project succeeded, the optimal code team size estimated from equation 3 and the actual size of the team.

The case histories and the estimation procedures indicate that it generally takes a minimum of 8 years for a code team to develop an initial capability for a weapons code project. The requirements for a weapons code are determined by the physics necessary to simulate a nuclear weapon. LANL and LLNL have over 50 years of experience in this area, and know these requirements in detail. Weapons code projects require between 3000 and 6000 function points (Fig.2).

Table 2 Software Resource Estimates for the LLNL and LANL Code Projects

	LLNL			LANL			
	ASCI A	ASCI B	Legacy A	Antero Code Project	Shavano Code Project	Blanca Code Project	Crestone Code Project
Single Lines of Code	184000	490000	410550	300000	500000	200000	314000
Function Points (Eq.1)	4800	4000	5400	2900	4800	3800	2900
estimated schedule(Eq.4)	8.7	7.6	6.9	6.6	8.1	7.4	6.7
Project age (at initial milestone date)	3	9	N/A	4	3.5	8	8
Successful in achieving initial ASCI milestone	No	Yes	N/A	No	No	No	Yes
Estimated staff requirements (Eq.3)	22	27	24	14	22	18	14
real team size	20	22	8	17	8	35	12

Yellow shading indicates historical data; white background denotes computed numbers.

**Time in years and average code team size required to complete a project with a specified number of function points (corrected for LANL/LLNL experience)**

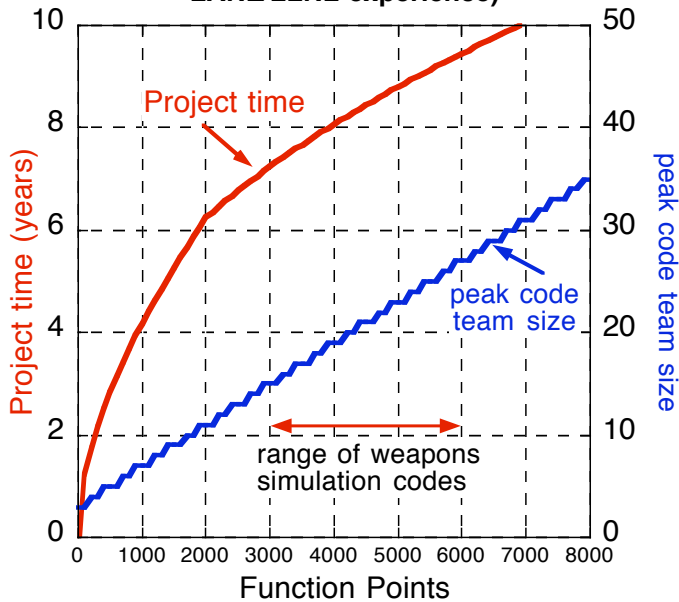


Figure 2. Time required to complete a project and average code team size as a function of code capability measured in Function Points[11, 12].

Some of the ASCI codes were started before ASCI began in 1996 (ASCI B, Legacy A for LLNL, and the LANL Crestone code project). ASCI B was started roughly in 1992 and had a working prototype in 1994. The Crestone code project was started before 1992. ASCI A and the Shavano and Antero code projects were started around early 1997. Legacy A was started over 30 years ago and was included for comparison. Since we are

able to match the history of weapons codes with scalings derived from the experience of the commercial software industry, we conclude that the constraints, computer science practices and management issues that generally apply to the IT industry apply to the development of weapons codes as well (i.e. there is no “Silver Bullet” that can radically reduce the development time [14]).

We found that the dominant factor for success is the age of the code project. The code projects that did not have sufficient time (8 years) to complete their projects failed to meet their milestones. All but one of the code projects that had 8 years succeeded in meeting their milestones. This is clear evidence that schedules and requirements must be consistent. The schedule cannot be fixed independently of the requirements, a fact long appreciated by the IT industry[9] but not adequately taken into account in the early planning for ASCI. Unfortunately, the ASCI program set the milestone for demonstrating the capability of each code project to be three and a half years (December 1999) after the beginning of ASCI (~mid 1996) and three years after the date (~January 1997) many of the code projects were launched. Having recently recognized this, partially as a result of our analysis, the ASCI program has revised the program milestones.

Adequate development time is necessary—but not sufficient—for success. Several code projects failed in spite of having adequate time. Poor practices and inadequate support—implicitly included in the contingency factor—hurt many of the projects as well.

A final point is that it is clear from the function point scaling relations (eqs. 1-5) that the code requirements determine both the schedule and resources needed for success. This estimating analysis indicates the importance of a realistic set of requirements,



schedule and resources. Without them, projects will fail and the needed applications will not be developed.

#### 4. Conclusions

Computational science has the potential to play an important role in society in many, many fields. However, if computational science is to reach the level of maturity necessary to play that role, it must increase the level of confidence in its predictions. It must develop methods to ensure that the equations and models accurately reflect the real world, that the equations and models are solved correctly, that the applications are set up and run correctly by knowledgeable and careful people, and that the results are interpreted correctly. Accurate equations and correctly implemented models requires attention to the code development process. The process must follow the general “lessons learned” discussed in the paper. Particular attention must be given to building in quality and accuracy. An intensive verification and validation program is essential. Finally, those developing the code and those using the code must have a deep appreciation of the limits of the code and a deep rooted appreciation that the results may not be correct.

#### 5. Acknowledgements:

The author is grateful for discussions with Don Remer, Rob Thomsett, Tom DeMarco, Don Burton, Richard Kendall, Dale Henderson, Ken Koch, Larry Cox, Larry Votta, and Jeremy Kepner.

#### 6. References

- [1] Laughlin, R., *The Physical Basis of Computability*. Computing in Science and Engineering, 2002. **4**(3): p. 27-30.
- [2] Petroski, H., *Design Paradigms: Case Histories of Error and Judgement in Engineering*. 1994, New York: Cambridge University Press. 221.
- [3] Post, D. and R. Kendall. *Lessons Learned From ASCI*. in *DOE Software Quality Forum 2003*. 2003. Washington, DC: Los Alamos National Laboratory.
- [4] Thomsett, R., *Radical Project Management*. 2002, Upper Saddle River, NJ: Prentice Hall.
- [5] Brooks, F., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. 1995, Menlo Park: Addison-Wesley Publishing Co. 322.
- [6] Verzuh, E., *The Fast forward MBA in Project Management*. 1999: John Wiley.
- [7] DeMarco, T. and T. Lister, *Waltzing with Bears, Managing Risk on Software Projects*. 2003, New York, New York: Dorset House Publishing. 196.
- [8] Capers-Jones, T., *Estimating Software Costs*. 1998, New York: McGraw-Hill.
- [9] Yourdon, E., *Death March*. 1997, Upper Saddle River, NJ: Prentice Hall PTR.
- [10] Paulk, M., *The Capability Maturity Model*. 1994, New York: Addison-Wesley.
- [11] Jones, T.C., *Estimating Software Costs*. 1998, New York: McGraw-Hill. 720.
- [12] Symons, C.R., *Function Point Analysis: Difficulties and Improvements*. IEEE Transactions on Software Engineering, 1988. **14**(1): p. 2-11.
- [13] Remer, D. *Managing Software Projects*. in *UCLA Technical Management Institute*. 2000. Los Angeles, CA: UCLA Extension Courses.
- [14] Brooks, F.P., *No Silver Bullet: Essence and Accidents of Software Engineering*. Computer, 1987. **20**(4): p. 10-19.

# Application Development Productivity Challenges for High-End Computing (Extended Abstract)

Vivek Sarkar\*

Clay Williams

Kemal Ebcioglu

## Abstract

*Application development productivity can be a significant bottleneck in the time to solution for deploying production applications on High-End Computing (HEC) systems. We discuss the fundamental reasons for these productivity barriers, and outline the solution being pursued for application development productivity in the IBM PERCS project (Productive Easy-to-use Reliable Computing Systems). We also introduce a simple model for defining application development productivity, and use this model to guide our choice of solutions.*

## 1 Problem Formulation

There are multiple phases in the software lifecycle of HEC applications – algorithm development, application requirements and written specification, design/code/test of a functional implementation, porting, parallelization and tuning to obtain a scaled parallel implementation, deployment, execution and debugging of parallel implementation on a production HEC system, and feedback from production runs to earlier lifecycle phases. While some of these lifecycle phases are similar to those found in commercial applications, the HEC domain poses many *unique challenges to application development productivity* especially in the phases relevant to large-scale parallel implementations of HEC applications. One reason for this the fact that HEC systems exhibit *large degrees of parallelism* at different levels — intra-chip, intra-node and inter-node — that must be exploited to obtain production-level performance. In addition, production applications are executed on *large data sets* that increasingly stress the *severe non-uniformities and discontinuities in data*

*accesses* that are present at the different levels of HEC systems due in part to the widening gap between intra-chip and inter-chip data access times in both shared and distributed memory configurations. Further, each system layer — hardware, operating system, middleware, compiler, programming language — adds a layer of opacity in controlling the non-uniform data accesses supported by the hardware.

After examining why application development productivity is so low in the HEC domain, we have concluded that the two fundamental problems that need to be addressed are the Expertise Gap and the Programming Complexity, both of which are described below. *The grand challenge for this area is to improve overall HEC application development productivity by 10× by the year 2010.*

- 1. Expertise Gap.** Programming today’s HEC systems requires a high level of expertise, but the current trend of available skills is increasingly one of few HEC expert programmers, and many domain scientists and software engineers who are not trained or experienced as expert HEC programmers. Only the HEC expert programmers can contribute to production-level HEC applications, thereby limiting the number of HEC solutions that can be developed. Some HEC development groups have indicated that the ratio of expert to non-expert programmers can be as low as 1 : 100 in their organizations. The HEC expert programmers are skilled in parallel programming, and have “top-gun” experience with all aspects of processor architecture, system hardware, system software, compilers, runtime systems, and parallelization/communication libraries. This level of expertise is typically acquired over a long period (10+ years) of hands-on experience with leading-edge HEC systems. In contrast, most qualified software engineers have little or no experience with HEC systems, and usually find low-level

---

\*IBM T.J. Watson Research Center. Email: {vsarkar, clayw, kemal}@us.ibm.com.

parallel programming constructs (such as those in OpenMP or MPI) too complex to understand or use. Similarly, while domain scientists have an in-depth knowledge of their application domain (*e.g.*, fluid dynamics, bio-informatics), they lack the necessary training and experience to be able to contribute to programming production HEC applications.

2. **Programming Complexity.** In addition to the expertise gap, tackling the non-uniformities in data access and the multiple levels of parallelism in HEC systems requires a high level of effort even from the experts who can write production-level HEC applications. One reason why these non-uniformities become a critical barrier to HEC application development productivity is because the programmers must be aware of discontinuities in each level of the system (L1 cache, L2 cache, L3 cache, and the interconnection networks) before they can write production-level HEC applications. In addition, the ability to translate experience with a prior HEC system to a new HEC system is very limited — it can take many months even for an expert HEC programmer to become productive on a new HEC machine. The net effect of all this programming complexity is a slower software lifecycle in HEC environments.

We now present a simplified *application development productivity model* to demonstrate the importance of addressing the Expertise Gap and Programming Complexity problems. Consider an average-sized organization that develops HEC applications. First, let us assume that all organization members can be grouped according into *expertise levels*, and that  $e(i)$  is the *fraction of organization members that are at expertise level  $i \geq 0$* .  $e$  defines an *expertise profile* for the entire organization, and it can also be viewed as a probability density function. There are many possible definitions for expertise levels. One simple definition is as follows: an organization member is placed in expertise level  $i$  if and only if they have successfully contributed to developing  $i$  production HEC applications in their career thus far.

Next, we define  $t_S(i)$  to be the *average completion time of an HEC application development task of size  $S$  by an organization member at expertise level  $i$* .  $t_S(i)$  essentially defines a *learning curve* or *experience curve* for HEC application developers in the organization. (Note that while the phrase "steep learning curve" is often used to represent a hard learning experience, it is a misnomer in this case because a steep descent in the learning curve actually represents a very suc-

cessful learning experience.) In this model, we will assume that  $t_S(i)$  is monotonically non-increasing with increases in expertise level  $i$ . This is a statement for a given point in time — there are certainly cases when HEC technology advances introduce more software complexity over time, thereby raising  $t_S(i)$  over time for all expertise levels  $i$ .

The two functions,  $e$  and  $t_S$  can now be combined to obtain an *application development productivity metric* as follows. For simplicity, let us assume that the organization needs to complete HEC application development tasks of a fixed size  $S$  in a given duration  $D$ . In this case, productivity can be measured as the number of such tasks completed by the organization in time  $D$ . To answer this question, we can first examine the learning curve and identify the smallest expertise level  $EL(D)$  for which  $t_S(EL(D)) \leq D$ .  $EL(D)$  represents the *expertise gap threshold* in our model. If  $EL(D) > 0$ , it must be the case that all members with a lower expertise level than  $EL(D)$  will take longer than time  $D$  to complete a task of size  $S$ . We assume that none of the members below the *expertise gap threshold* can contribute towards HEC application development productivity since they will not complete their task in the assigned duration. Therefore, the productivity  $P$  can be expressed as

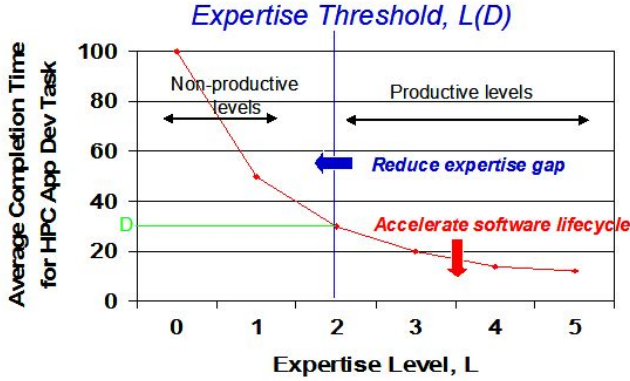
$$P = \sum_{i \geq EL(D)} \frac{e(i) \times D}{t_S(i)}$$

which is the average number of tasks of size  $S$  that can be completed (per organization member) in a given duration  $D$ . We observe that solutions to the Expertise Gap problem will lead to a smaller value of  $EL(D)$ , that solutions to the Programming Complexity problem will lead to smaller values of  $t_S(i)$ , and that both sets of increases will contribute to increasing the overall productivity.

Note that the productivity discussed above differs from the standard software engineering definitions of productivity. In those definitions, productivity is in terms of the *size  $S$*  of the application and the *effort  $E$*  it takes to develop it. The computation of size is an area of considerable debate and research in the software engineering community, with proposals ranging from lines of code to function / feature points. For an overview of productivity in software engineering, see [3]. In the traditional definition, both size and time are measured independently of the development team and their skill levels. Productivity is then defined as

$$Q = \frac{S}{E}$$

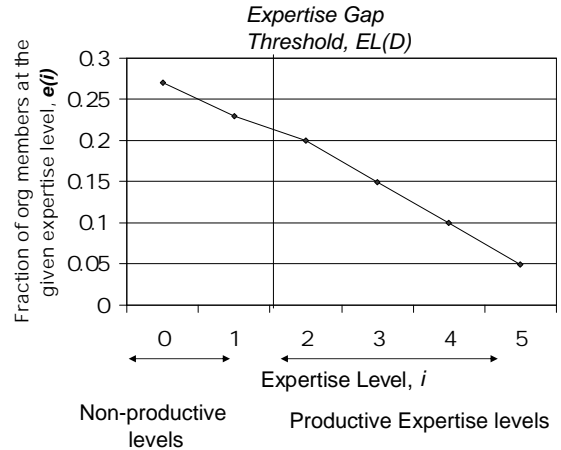
Our definition of productivity differs considerably from



**Figure 1. Learning Curve for HEC Application Developers**

this, as it has expertise and learning curve as its primary focus rather than size and effort. Size and effort are not absent from the measure, but are implicit. Our measure explores the effect of expertise and learning curve on productivity for applications of fixed size and projects of a fixed duration. In our approach, size could be measured using any of the techniques proposed to date. HEC applications are profoundly affected by the skills of the application development team and the complexity of the programming constructs. Because of this, we believe that our productivity model offers important insights into improving productivity for HEC application development. We plan to validate the model using data collected during as part of the PERCS project. In the remainder of this section, we illustrate this model through Figures 1 and 2.

Let us assume that we are interested in identifying the expertise levels that can contribute towards tasks of durations  $\leq 30$  time units (*e.g.*, 30 months). Figure 1 can be used to identify the expertise gap threshold,  $EL(D) = 2$  for  $D = 30$ . Looking at where this threshold falls in Figure 2, we see that only 50% of the members can productively contribute to these tasks. Therefore, the average organizational productivity is  $P = 30 \times (0.2/30 + 0.15/20 + 0.1/14 + 0.05/12) \approx 0.76$  tasks (per organization member) completed in 30 time units. We expect to deliver productivity boosts by both lowering the Learning Curve in Figure 1, and shifting the Expertise Profile in Figure 2 to the right so as to increase the expertise level of organization members. We expect that a judicious combination of these two productivity boosts could lead to our goal of a 10 $\times$  productivity improvement by 2010.



**Figure 2. Expertise Profile for HEC Application Development Organization**

## 2 Overview of our Solution

In this section, we give a high-level overview of our solutions to the Expertise Gap and Programming Complexity problems identified in Section 1.

We plan to address the Expertise Gap problem on two fronts:

1. *High-level tools to make parallelism understandable by non-expert HEC programmers.*

All HEC applications involve parallel programming of some kind. There has been some good progress over the last two decades on standardization of language constructs for parallel programming *e.g.*, OpenMP, MPI, threads. However, these constructs are still too complex and low-level for use by non-expert programmers. We believe that higher levels of abstraction — such as data parallelism, stream parallelism, and recursive divide-and-conquer parallelism — will be more natural in making parallelism understandable by this community of programmers. Our approach will enable non-expert programmers to a) express parallelism in new HEC applications by domain-specific tools for these higher level of abstraction, and b) transform the parallel structure of the application into useful parallelism, by using a new set of tools based on analytical, simulated, and actual performance feedback.

Our infrastructure strategy for building high-level tools for HEC programmers is to extend the Eclipse tools framework (eclipse.org) with new

tools and plug-ins for existing languages such as C/C++, Fortran, and Java, as well as new languages such as StreamIt, and the new parallel programming language constructs discussed later (see [6] for an example). In addition, we plan to extend the *refactoring* functionality in Eclipse to specifically simplify the task of parallelization. Refactoring is a well-known technique in the software engineering domain for providing tools that can guide programmers in performing semantics-preserving transformations, typically for improving the modular structure of a body of code. In our extensions, we will use the refactoring tools to guide non-expert programmers in correct and efficient parallelization, thereby reducing the expertise gap. One of the main research challenges is to identify and communicate performance bottlenecks at a high level to guide users in selection of transformations.

2. *Leveraging component-based development in the HEC domain.* It has been observed that the programming tools and practices used by the HEC community is at least a decade behind those used by developers of commercial applications. Though commercial and web applications are intrinsically concurrent and asynchronous in nature, developers of these applications are highly productive today through the use of comprehensive component and application frameworks. While the HEC community has demonstrated the ability to share numerical libraries across multiple applications, they have not as yet adapted first-class component models with strong guarantees of semantics and performance. Our implementation strategy is to leverage and extend the work under way in the CCA Forum ([cca-forum.org](http://cca-forum.org)). As components become available for reuse, they will raise the level of abstraction for building a system. When component technology matures for HEC, lower level coding will be replaced by component composition. With advanced tool support, composition is a much simpler and more uniform activity than source level coding. Thus, the level of expertise required to build HEC systems will be reduced. Furthermore, the expertise for performing certain computations can be encapsulated in components. Users of the components will only need to understand what they want to do, not the details of how to implement the computation efficiently. This also reduces the level of expertise necessary to build HEC systems. Our current plan is to select a suitable CCA framework, port it to Eclipse, and use it as the basis for research on extending CCA

to support productivity enhancements. Using Eclipse will ensure that the component work and the other Eclipse-based capabilities mentioned in this paper (refactoring, new parallel programming constructs, AOSD, etc.) will be integrated and support seamless development in the HEC tools. In the component area, a key research challenge is to deliver optimized performance akin to that of dynamically optimized libraries such as FFTW, ATLAS and SPIRAL.

In addition to the Expertise Gap, we plan to address the Programming Complexity problem on three fronts:

1. *New Programming Constructs for exploiting Parallelism and Locality in HEC systems.* As discussed earlier, the learning curve for HEC applications can be formidable due to the complexity of HEC systems. The research challenge is to simplify programming of parallelism, synchronization, consistency, and object lifetimes by providing first-class support for primitive, object, matrix and stream data types. Our proposed approach is to build the programming constructs around the following new constructs:
  - (a) *Enclaves*, which provide static and/or dynamic grouping of objects with common properties (such as object lifetime, confinement, ownership, immutability, performance hints, etc.) and enable remote (distributed) communication of objects.
  - (b) *Atomic sections*, which provide high level abstractions of locks without burdening the HEC programmer with details of lock management. Atomicity is a high-level property that is usually implemented by synchronization and other low-level mechanisms. After determining the parallel structure of the application, the programmer may realize that some sets of statements need to be performed atomically to guarantee correct parallelization. The user identifies these sets of statements as atomic sections and leaves the implementation of synchronization to the compiler and runtime system.
  - (c) *Asynchronous operations*, which provide high level abstraction of threads. By identifying an operation invocation as asynchronous, a user is logically creating a fine-grained thread for the invocation. We believe that this and similar high-level constructs will be necessary for generating the large number of threads needed for scalable performance.

- (d) *Safety checks* to guarantee the correctness of high-level assertions such as the absence of data races [2, 5] and/or the preservation of immutability properties [4]. Extensive safety checks reduce debug and test efforts by improving software observability and diagnosability. The checks can be static or dynamic. We believe that the relative overhead of runtime safety checks will shrink as compute cycles become cheaper in 2010.

## 2. Morphogenic Software.

In much of real world software, different *concerns* may often be tangled together in an undesirable way. For example, data logging, debugging, accounting, and performance measurement may all be intertwined with the main calculations of a program. Aspect-Oriented Software Development (AOSD) has gained quite a bit of attention recently as a technique for reducing programming complexity by enabling *separation of concerns* [1]. However, most of the separation and composition of concerns is done at a coarse granularity such as methods or procedures. This limits its applicability to HEC applications, where one can often find a lot of complexity at finer granularities such as individual statements. We plan to address this limitation with the morphogenic software approach, which will support fine-grained extraction and composition of concerns (extension of aspect-oriented programming) as well as support effective integration of heterogeneous code (C, C++, Fortran, Java, Matlab).

- 3. *Debugging, Validation, and Verification.* A key research challenge is to enable effective debugging of large-scale HEC applications with minimally invasive sampling, tracing, and instrumentation. These debugging systems will give the impression of reversible execution when needed, though this functionality will most likely be implemented by resuming execution from a prior checkpoint, and then using forward execution. The scope of the debugging agenda also includes blame analysis for functional and performance failures, validation rule-sets, domain constraints from physical systems.

## 3 Conclusions

In this paper, we outlined two fundamental problems facing HEC application developers — Expertise Gap and Programming Complexity. We proposed a simple

model for measuring productivity, and briefly described how we plan to address these two problems to improve overall productivity.

## Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. We would like to thank all contributors to the PERCS Programming Model and Tools technical agenda outlined in this paper — from IBM, and from our university partner groups at Massachusetts Institute of Technology, Purdue University, University of California at Berkeley, University of Delaware, University of Illinois, University of Texas at Austin, and Vanderbilt University.

## References

- [1] Special issue of Communications of the ACM on Aspect-Oriented Programming. October 2001, Volume 44, Number 10.
- [2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *SIGPLAN '02 Conference on Programming Language Design and Implementation*, June 2002.
- [3] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, second edition, 1997.
- [4] I. Pechtchanski and V. Sarkar. Immutability Specification and its Applications. *Concurrency and Computation Practice & Experience (CCPE)*, 2003.
- [5] M. Prvulovic and J. Torreallas. ReEnact: Using Thread-Level Speculation to Debug Software; An Application to Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.
- [6] A. Solar-Lezama and R. Bodik. Templating Transformations for Bitstream Programs. In *Proceedings of the HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2004.

# Comparing Network Processor Programming Environments: A Case Study

Niraj Shah, William Plishker, Kurt Keutzer  
University of California, Berkeley  
{niraj,plishker,keutzer}@eecs.berkeley.edu

## Abstract

*Network processors have emerged as prominent examples of multiprocessor application-specific programmable architectures. While there have been significant architectural developments in this field, widespread adoption will be predicated on productively programming high performance applications on these architectures. This paper presents a case study of two programming environments for a common network processor, the Intel IXP1200. We compare the development process, achievable performance, and resource usage of the final implementations using these two programming approaches and draw conclusions regarding the advantages and disadvantages of these approaches.*

## 1. Introduction

The costly unpredictable nature of ASIC design coupled with increased silicon capability is fueling the rise of multi-processor application-specific architectures. Examples of this trend have already been demonstrated in networking, multimedia, and graphics. Architectures for these application-specific programmable processors have been explored in great depth. For example, current network processors share many of the complex architectural features of traditional high-performance computing systems: multiple processing elements each with multiple hardware-supported threads, distributed memories, special-purpose hardware, and a variety of communication mechanisms.

Despite the deep architectural emphasis, there has been relatively modest investigation of their programming environments. For many system designers who chose to use these programmable solutions, their success will depend largely on their ability to implement a high-performance application in a short (or at least predictable) design time. Current options for programming network processors are: assembly language, C language variants [1] [2] [3], and block-based frameworks [4] [5]. The most common approach is to

use assembler or a subset of C with architecture specific extensions (e.g. Intel Microengine C [1], Motorola's C-Ware [2], Teja Technologies Teja C [3]). In addition, a few efforts have raised the programming abstraction to a higher level. For example, Intel has released a framework called ACE [4] that is a library-based approach built on assembly code. Another such effort, NP-Click [5], combines Click [6], a popular networking specification language, with an abstraction of the target network processor architecture.

This paper is a case study comparing two different software development approaches for a representative network processor, the Intel IXP1200 [7]: Microengine C (which we refer to as IXP-C) and NP-Click. We chose these two software development approaches to compare different levels of abstraction: a C-like language and a domain-specific language. To our knowledge, this is the first research effort to compare in a detailed manner the software development methodologies for a network processor. We chose the IXP1200 over newer architectures based on the availability of software development environments. However, we believe the results of this study are applicable to recently released architectures like the Intel IXP2xxx family as well as other network processor families.

To compare these two programming approaches, we use each of them to implement a 16 port IPv4 packet forwarder and a 4 port Differentiated Services (DiffServ) interior node. The IPv4 packet forwarder is a performance focused benchmark, while the DiffServ application contains more functionality, but supports fewer ports. We compare these two software development approaches across three categories: development process, achievable performance, and resource usage of the final implementation. We analyze these results and compare and contrast the advantages and disadvantages of the IXP-C and NP-Click programming environments.

The remainder of this paper is organized as follows: Section 2 provides a brief architectural description of the Intel IXP1200. Section 3 describes the two software development approaches we chose: IXP-C and

NP-Click. Section 4 describes the applications we implemented: IPv4 packet forwarding and a DiffServ interior node. Results are presented in Section 5. Finally, sections 6 and 7 summarize our results and comment on future work, respectively.

## 2. Intel IXP1200

For this case study we chose the Intel IXP1200 because it is representative of other network processing and other multi-processor application-specific architectures. In addition, the IXP1200 has numerous available programming environments, relative to other network processors.

The IXP1200 is one of Intel’s first network processors based on their Internet Exchange Architecture. It has six identical RISC processors, called *microengines*, plus a StrongARM processor. The StrongARM is used mostly to handle control and management plane operations. The microengines are geared for data plane processing and each has hardware support for four threads that share a fixed size program memory. The memory architecture is divided into several regions: large off-chip SDRAM, faster external SRAM, internal Scratchpad, and local register files for each microengine. Each of these areas is under the direct control of the user and there is no hardware support for caching data from slower memory into smaller faster memory (except for the small cache accessible only to the StrongARM). There is also a hash engine coprocessor that provides hardware support for hash key creation. The IX Bus (an Intel proprietary bus) is the main interface for receiving and transmitting data with external devices such as Ethernet MACs and other IXP1200s. It is 64 bits wide and runs up to 104MHz allowing for a maximum throughput of 6.6Gbps. The microengines can directly interact with the IX bus, so any microengine thread may receive or transmit data on any port without StrongARM intervention. This interaction is performed via Transmit and Receive FIFOs which are circular buffers that allow data trans-

fers directly to/from SDRAM. For the microengines to interact with peripherals (e.g. determining their state), they need to query or write to Control Status Registers (CSRs). Accessing control status registers requires using the Command Bus which doubles as the interface to the hash engine, scratchpad memory, and Transmit and Receive FIFOs. A micro-architectural diagram of the Intel IXP1200 is shown in Figure 1.

## 3. Software Development Approaches

For this case study, we focus on software development for data-plane processing, where high throughput and low latency are required. In this section we describe the two IXP1200 microengine software development approaches we chose to compare: IXP-C and NP-Click.

### 3.1. Intel IXP1200 Microengine C (IXP-C)

The initial programming model that Intel provided for the microengines was assembly language [8]. This is the lowest level of programming the architecture as it exposes all facets of the architecture under programmer control. Intel also provided a macro assembler that supports higher-level programming constructs like conditionals and loops. There is also a register allocator so symbolic variable names can be used.

Later, Intel augmented their assembly language interface to the microengines with a subset of C (which we refer to as IXP-C) [1]. IXP-C supports loops, conditionals, functions, intrinsics (function calls using C syntax that direct instruction selection), basic data types, and abstract data types such as structs and bit-fields. However, data allocation to different memory regions is user defined; for practical applications, explicit binding is necessary at declaration time. In addition, the multithreading model is explicit: the programmer must manually divide their application across microengines and threads, control all inter-processor and thread communication, and arbitrate access to shared resources. Intel also provides a library that defines additional data types, macros, and functions that provide a slightly higher abstraction of the hardware. For example, there are bit-fields that export the format of control status registers and intrinsics for assembler instructions that use the hash engine.

### 3.2. NP-Click

NP-Click [5] is a programming model implemented for the Intel IXP1200 that combines the “flow-based router” concept from Click [6] with an abstraction of

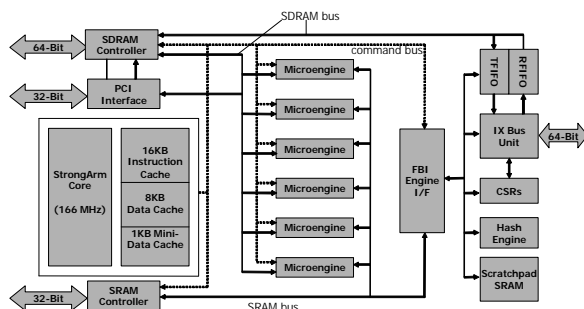


Figure 1. Intel IXP1200 microarchitecture



the target architecture. Like Click, *elements* are the base unit of computation in NP-Click. Elements correspond to common networking operations like classification, route table lookup, and header verification. Elements communicate by passing packets using well-defined semantics. There are two types of communication: push and pull. Push communication is initiated by the source element and effectively models the arrival of packets into the system. Pull communication is initiated by the sink element and often models available memory in hardware resources for egress packet flow. In NP-Click, the elements are implemented in IXP-C to leverage the existing compiler for the IXP1200.

NP-Click also provides visibility into salient architectural details that greatly affect performance. Specifically, it enables programmers to:

- control thread boundaries to effectively manage processor and thread utilization
- map data to different memories (registers, Scratchpad, SRAM, and SDRAM)
- separate design concerns of arbitration of shared resources and functionality

NP-Click separates the application description from implementation choices, such as functional partitioning across microengines or arbitration schemes for shared resources. For example, after initially implementing their application in NP-Click, the application programmer is able to change the thread boundaries to try different implementations without changing the application description.

## 4. Applications

For this case study, we use IXP-C and NP-Click to implement two applications: packet forwarding and a DiffServ interior node. The IPv4 packet forwarding application is a performance-centric benchmark with relatively narrow functionality. The second application, a DiffServ interior node, is a functionally rich application with lower performance requirements.

### 4.1. IPv4 Packet Forwarding

IP Version 4 packet forwarding [9] is a common kernel of many network processor applications. We chose to implement the data plane of a 16 port Fast Ethernet (16x100Mbps) IPv4 router. The major features of this benchmark are listed below:

- incoming packets are checked for validity, including proper version number and correct header length

- the egress port of a packet is determined by a longest prefix match route table lookup based on the IPv4 destination address field
- after the egress port has been determined, the time-to-live (TTL) and checksum fields in the packet header are updated

### 4.2. Differentiated Services Interior Node

The differentiated services architecture (DiffServ) [10] is a method of facilitating end-to-end quality of service (QoS) over an existing IP network. In contrast to other QoS methodologies, it is a provisioned model, not a signaled one. This implies network resources are provisioned for broad categories of traffic instead of employing signaling mechanisms to temporarily reserve network resources per flow. A DiffServ network relies on traffic conditioning at the boundary nodes to simplify the job of the interior nodes. The boundary nodes of a DiffServ network aggregate ingress traffic into a number of categories, called *behavior aggregates* (BAs), using the *differentiated services code-point* (DSCP) as specified in [11]. The interior nodes apply different *per hop behaviors* (PHBs) to each of the BAs. The classes of PHBs recommended by IETF include:

- Expedited Forwarding (EF): low packet loss, latency and jitter
- Assured Forwarding (AF): 4 classes of traffic, each with varying degrees of packet loss, latency and jitter
- Best Effort (BE): no guarantees of packet loss, latency, or jitter

The PHBs in a DiffServ implementation are defined by a combination of:

- classifiers: elements that select a subset of the packet stream based on packet header fields
- traffic conditioners: elements that measure, mark, shape and drop packets

For this case study, we implemented an interior DiffServ node. While there is less monitoring and shaping than in a boundary node, we believe an interior DiffServ node is a good benchmark for network processors as it is a functionally rich application that stresses different aspects of the development process.

Our DiffServ application begins with data-plane IPv4 packet forwarding functionality. After an ingress packet passes through IP verification, IP lookup, and time-to-live decrement, it is classified based on its DSCP, with each class of traffic receiving different treatment. For example, EF traffic is first metered, with traffic below the specified data rate queued for

transmit, while traffic above this data rate is discarded. On egress, we use two cascaded packet schedulers:

- deficit round-robin scheduling (DRR) [12] for EF and AF classes with a weighting toward EF and higher priority AF classes
- strict priority scheduling between the output of DRR scheduling and BE traffic

The DiffServ application we implement supports 4 Fast Ethernet ports (4x100Mbps). Though the IXP1200 may seem like overkill for this application, we attempted to implement an 8x100Mbps version of this application, but neither the IXP-C nor NP-Click implementation could support this line rate.

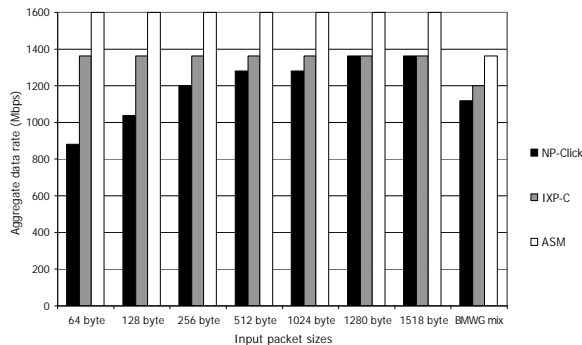
## 5. Results

To test each implementation, we used a cycle-accurate architecture simulator of the IXP1200 [13] assuming a microengine clock rate of 200MHz and an IX Bus clock rate of 100MHz. Our simulation environment also models the Ethernet MACs (Intel IXF440s) connected to the IX Bus. For both applications, the destinations of the input packet streams were randomly distributed evenly across the output ports. In addition, the routing table contained 1000 entries and measurements were not taken until steady state was reached.

### 5.1. Performance

This section describes the measurement methodology and performance results of the IXP-C and NP-Click implementations. We first present the results for IPv4 packet forwarding, then for the DiffServ interior node.

**5.1.1. IPv4 Packet Forwarding.** To measure performance for the IPv4 packet forwarding application, we consider the packet forwarder to be functional if it



**Figure 2. Performance comparison of 16 port IPv4 packet forwarding implementations**

has a steady state transmit rate that is within 1% of the receive rate without dropping any packets. We test each of the implementations with a variety of single packet size input streams (64, 128, 256, 512, 1024, 1280, and 1518 bytes) and the IETF Benchmarking Methodology Workgroup (BMWG) mix [14]. We use 64 and 1518 byte packet streams as they represent the minimum and maximum frame sizes permitted by the Ethernet standard. The packet sizes in between are included to give additional insight into the performance of the different implementations. The BMWG packet mix provides a more realistic input data set as it contains an even random distribution of seven packet sizes ranging from 64 bytes to 1518 bytes. For each input packet stream, we measure the maximum sustainable aggregate data rate.

The results of our experiments for IPv4 packet forwarding are shown in Figure 2. The IXP-C implementation is able to perform at 85% of line rate (1360Mbps aggregate) across all single packet size input streams. For the BMWG packet mix, the performance is slightly lower (1200Mbps aggregate) because of dynamic load balancing effects. We attribute the consistent data rate across all packet sizes to sub-optimal arbitration of multiple threads accessing the shared transmit FIFO as the performance limiting factor.

The aggregate bandwidth of the NP-Click implementation ranges from 880-1360Mbps. NP-Click's implementation has more processing overhead per packet than IXP-C's. As a result, for data streams composed of smaller packets, NP-Click's throughput suffers. For the BMWG packet mix, a more realistic data set, the NP-Click version also suffers from load balancing issues, but is able to achieve 93% of the IXP-C implementation (1120Mbps aggregate).

For reference, we show the performance of a hand-tuned assembler implementation based on an Intel reference design. The assembler implementation was able to meet line rate (1600Mbps aggregate) for single packet size streams, but only maintains 1360Mbps when tested with the BMWG packet mix. Both the IXP-C and NP-Click implementations fall short of this, by 11.7% and 17.6% respectively, because of the fine degree of scheduling that is available only when programming at the assembly language level.

**5.1.2. Differentiated Services Interior Node.** For the DiffServ application, measuring performance is not as simple since the specification requires non-conforming packets to be dropped. Thus, the transmit data rate will always be less than received data rate. To gauge performance we compare the egress data rates of the constituent traffic flows. For the baseline setup for all

measurements, the ingress data rates were set to the following percentages of ingress bandwidth:

- Assured Forwarding, Class 1 (AFC1): 20%
- Assured Forwarding, Class 2 (AFC2): 15%
- Assured Forwarding, Class 3 (AFC3): 10%
- Assured Forwarding, Class 4 (AFC4): 5%
- Best Effort (BE): 10%

We measured egress data rates for all traffic flows as the Expedited Forwarding (EF) traffic grew from 0% to 40% of ingress bandwidth in 5% increments. When the EF flow is set to 40% of ingress bandwidth, the aggregate ingress bandwidth is at line rate (100Mbps/port). Since this is only a 4 port design, we do not see the load balancing issues witnessed in the IPv4 packet forwarding implementations. Thus, to measure the worst case, the packet sizes of all flows were set to 64 bytes.

Both the IXP-C and NP-Click implementations were able to receive packets at line rate. The egress data rates of all flows for both implementations are shown in Figure 3 and Figure 4. Ideally, the graph should be a horizontal line for Assured Forwarding (AFCx) classes indicating their egress data rates were not affected by increase in EF traffic. For EF traffic, the egress and ingress data rates should be equal through 20%, then egress EF traffic should level off. This is due to the setting of the bandwidth meter in the EF PHB. The egress Best Effort (BE) data rate is bound to decrease as EF increases since BE packets are subject to a strict priority scheduling with respect to all other flows.

For the IXP-C implementation, as the Expedited Forwarding (EF) flow increases, we see a minor decrease on the egress data rate of Assured Forwarding (AFCx) flows. This is caused by an overall increase in the ingress data rate, which results in more packets to process. This increases the total amount of computation required, which results in a slower packet processing rate. The decline in Best Effort (BE) egress band-

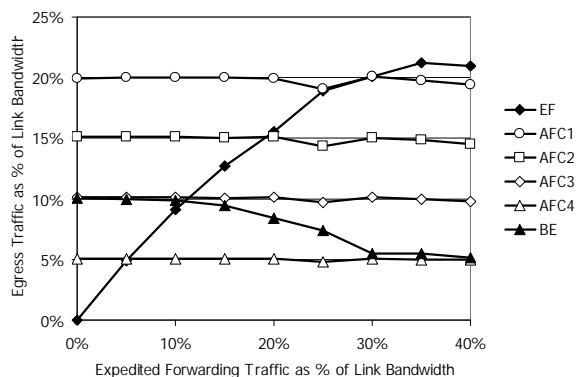


Figure 3. IXP-C DiffServ egress data rates

width is caused by the increased availability of packets from other flows with higher priority.

The NP-Click implementation performs similarly, but experiences some additional degrading effects as the EF flow increases beyond 20% of ingress bandwidth. However, the egress bandwidths of EF and AFCx flows in NP-Click's implementation remain within 10% of IXP-C's. Since the transmit function is slightly slower in NP-Click, the BE data rate takes a performance hit earlier in the ramp up of EF traffic.

## 5.2. Development Process

This section compares and contrasts the development process of the applications using IXP-C and NP-Click. Specifically, we focus on the debugging and performance improvement process, design time allocation, and total design effort. Number of lines of code is often used as a proxy for design effort. However, when comparing vastly different programming methodologies, this metric can be very misleading. Instead, we measure person-hours.

**5.2.1. IXP-C.** When using IXP-C to implement IPv4 packet forwarding and the DiffServ interior node, most of the development effort was spent arriving at a functionally correct initial implementation. The remainder of the design effort was spent improving the implementation. More than half of that was fixing bugs that arose from thread interactions. Some of these interactions were not obvious from the design or the code, which made debugging even more difficult.

Given the relatively low level abstraction of IXP-C, both profiling and optimizing the implementation proved difficult. As a result, we were able to only implement and test a few design alternatives. All the design alternatives we implemented were incremental changes on the initial implementation. Large design changes to the implementation would have required even more design effort, with no guarantee of performance improvement.

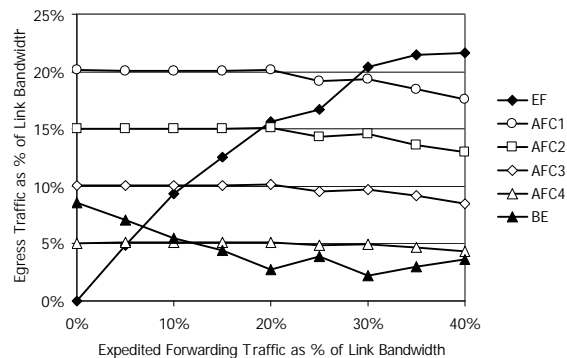


Figure 4. NP-Click DiffServ egress data rates

Our total design effort for the IPv4 packet forwarder using IXP-C was 400 person-hours. For the DiffServ implementation, we started with a hand-coded data-plane implementation of an IPv4 packet forwarder, then added the DiffServ functionality. The total design effort for this implementation was 320 person-hours.

**5.2.2. NP-Click.** Using NP-Click, we began with a Click description of the application and we were able to create an initial functionally correct design within a few days. We spent the majority of the design effort exploring the design space of implementations: pinpointing design bottlenecks, changing the mapping of elements to microengines, and simulating. Due to the modularity of NP-Click, profiling different implementations was easy. Performance improvement consisted of three major categories: changing mappings of elements to threads/microengines, better implementations of elements, and lower overhead arbitration schemes. Relatively little effort was spent debugging and when an error was spotted, it was easy to pinpoint which portion of the implementation was the cause. Common errors included incorrectly specifying element configurations and minor functional bugs within an element.

Our initial implementation of the IPv4 packet forwarding application had very low performance (480Mbps aggregate). The limiting factor of this implementation was a poor arbitration scheme of a shared resource (Transmit FIFO). With NP-Click, we were able to implement a new arbitration scheme which increased performance by 83-133%. The total design effort of this application was 100 person-hours. For the DiffServ application, we were quickly able to arrive at an initial functional implementation. This initial implementation was a naïve mapping of elements to microengines, which we then optimized. The total design effort for the NP-Click 4 port DiffServ implementation was 120 person-hours.

### 5.3. Resource Usage

Resource usage of the final implementation is a key comparison metric for any embedded software development approach. For this case study, we compare the number of microengines used, MIPS required, and code size for the IXP-C and NP-Click implementations.

For the IPv4 packet forwarding application, both the IXP-C and NP-Click implementations used 6 all six microengines on the IXP1200. For DiffServ, IXP-C required only 3 microengines, while NP-Click required 4. The additional microengine for the NP-Click implementation was needed because of instruction

store limitations. The instruction store on the Intel IXP1200 is limited to 2048 instructions per microengine. It is important to note that this memory is not a cache, but a flat memory. As a result, all instructions to be executed on the microengines must fit in this space. We attempted to fit the NP-Click DiffServ application on fewer than 4 microengines, but often created functional partitions that exceeded the instruction store on a particular microengine. Thus, some of the NP-Click implementation effort was spent optimizing for code size, not performance. The modularity of NP-Click is responsible for the higher code size. We suspect better IXP-C compilation technology could significantly reduce this. A summary of the number of required microengines and code sizes for the four implementations is given in Table 1.

The number of microengines used does not give a clear measure of the amount of computing power required for the different implementations. To measure this, we calculated the MIPS executed per implementation using a representative packet flow. On every clock cycle, a microengine on the IXP1200 is either: executing an instruction, aborting an instruction (due to a mispredicted branch), stalled (waiting for multi-cycle access to return without context switching), or idle. We calculate MIPS by aggregating the number of cycles spent executing and aborting instructions across all microengines and dividing by time. Since the microengines are running at 200Mhz, the peak MIPS rate is 200 per microengine. Table 1 includes a summary of our results. For IPv4 packet forwarding, the IXP-C and NP-Click implementations executed at similar MIPS rates. We attribute the extra MIPS in the NP-Click DiffServ implementation to overhead introduced by NP-Click’s modularity and time spent in polling loops.

## 6. Summary and Conclusions

In this case study, we have compared two different software development approaches for the Intel IXP1200, a common network processor. We compared IXP-C and NP-Click by implementing a 16 port

**Table 1. Statistics of the final implementations**

	IPv4 Packet Forwarding		DiffServ	
	IXP-C	NP-Click	IXP-C	NP-Click
<b>Number of microengines</b>	6	6	3	4
<b>MIPS</b>	1196.6	1197.8	585.3	719.9
<b>Code Size</b>	3870 instrs	5591 instrs	2363 instrs	6090 instrs

IPv4 packet forwarder and a 4 port DiffServ application. The advantages and disadvantages of each approach are given immediately below.

### 6.1. Microengine C (IXP-C)

The principal advantage of the IXP-C software development approach is resource usage of the final implementation. For the DiffServ implementation, we were able to meet line rate using only 3 microengines and used significantly fewer instructions. IXP-C also has a slight performance edge over NP-Click: it supports 75Mbps per port (BMWG packet mix) for the IPv4 packet forwarder and has a slightly higher egress data rate across all traffic flows for the DiffServ application.

Using IXP-C for software development had some drawbacks too. The overall development effort per application was much longer than NP-Click's, as was the time to reach functional correctness of the application. After meeting the functional requirements, performance tuning was also quite difficult. Hence, all performance improvements were incremental changes on the initial implementation. As a result, with IXP-C, the final implementation is largely dependent on the initial implementation. This places a large burden on the programmer's intuition of how the application should be partitioned.

### 6.2. NP-Click

NP-Click's primary advantage is ease of programming. With NP-Click, we were able to rapidly implement applications to meet the functional specification. Then, we were able to easily explore the design space of implementations to further improve performance. As a result, the total design effort was 2.5-4x shorter than IXP-C's.

For the IPv4 packet forwarder, the NP-Click implementation was able to route packets at 70Mbps per port (BMWG packet mix), 6.7% less than the IXP-C implementation. For DiffServ, the egress data rate of higher priority flows in NP-Click's implementation was within 10% of the IXP-C implementation.

The major weakness of using NP-Click is resource usage. For the DiffServ implementation, NP-Click was able to receive packets at line rate, but required an additional microengine. This was mainly due to NP-Click's code size overhead when compared to IXP-C. The instruction store limitations on the IXP1200 forced us to use one more microengine.

It is important to note that we spent significantly less effort using NP-Click for both applications. We believe further effort using NP-Click will result in bet-

ter implementations. For the 16 port packet forwarding design, we believe NP-Click can produce an implementation that can meet a higher data rate. For the DiffServ implementation, we can likely achieve similar performance with lower resource usage (but not as low as IXP-C's).

### 6.3. Conclusions

Embedded systems software development often falls into two categories:

- Performance-focused: the system is required to meet certain performance requirements
- Effort bound: a fixed number of person-hours are allocated for a design

If performance is the primary goal, IXP-C has a slight edge, at the cost of design effort. If resource usage is predicted to be tight, IXP-C is the preferred approach. For effort-bound projects, NP-Click is preferred. NP-Click gives designers a fast path to an initial implementation and the facilities to try many different implementations to improve performance and resource usage. In either case, if there is little intuition about an ideal functional partitioning *a priori*, NP-Click is more attractive.

It may be possible to combine the two software development approaches: use NP-Click to quickly try many different functional partitionings, yet write the final implementation in IXP-C. This would provide a compromise between design effort, performance and resource usage.

The Intel IXP1200 shares many of the salient features with other network processors: multiple multi-threaded processors, disparate memories with varying latencies, numerous heterogeneous shared resources, and different on-chip communication mechanisms. As result, we believe these results are applicable to many other network processors, including newer architectures.

## 7. Future Work

This case study compares two software development approaches for the Intel IXP1200. We would like to have compared other approaches, like Intel's ACE framework [4] as well. However, time limitations prevented us from doing so. We anticipate this framework enables a limited exploration of functional partitionings, but authoring functional blocks in assembler is error-prone and time consuming. Thus, we believe IXP-C should still be used for resource constrained applications while NP-Click should be used if reducing design effort is a primary concern.

Newer architectures like the Intel IXP2800 [15], which has 16 microengines and a more complex memory architecture, are exacerbating the difficulty of the programming task. Implementing applications on these newer architectures will be even more difficult as the design space is larger. In addition, the programmer will also have little intuition of which functional partitioning will meet the specification. For performance-centric applications, we still anticipate IXP-C will be able to produce higher performance implementations at the cost of significantly more design effort. For functionally rich applications, we anticipate an increased utility of NP-Click's ease of programming and facility to explore different mappings to the architecture. To confirm our hypothesis, we aim to perform a similar study for these architectures when tools for them become available to the university community.

## 8. References

- [1] Intel Corp., "Intel Microengine C Compiler Language Support Reference Manual," March 2002.
- [2] Motorola, Inc., "C-Ware Software Toolset: Product Brief," 2003.
- [3] Teja Technologies, Inc., "Teja C: A C-based Programming language for Multiprocessor Architectures," *Network Processor Design: Issues and Practices, Volume 2*, November 2003.
- [4] Intel Corp., "Intel IXA SDK ACE Programming Framework Developer's Guide," June 2001.
- [5] N. Shah, W. Plishker, and K. Keutzer, "Programming Models for Network Processors," *Network Processor Design: Issues and Practices, Volume 2*, November 2003.
- [6] E. Kohler et al. The Click Modular Router. *ACM Transactions on Computer Systems*. 18(3), pg. 263-297, August 2000.
- [7] Intel Corp., "Intel IXP1200 Network Processor," Product Datasheet, December 2001.
- [8] Intel Corp., "Intel IXP1200 Network Processor Family: Microcode Programmer's Reference Manual," March 2002.
- [9] F. Baker, "Requirements for IP Version 4 Routers," Request for Comments - 1812, Network Working Group, June 1995.
- [10] S. Blake, et al., "An Architecture for Differentiated Services," Request for Comments - 2475, Internet Engineering Task Force (IETF), December 1998.
- [11] K. Nichols, et al., "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," Request for Comments - 2474, Internet Engineering Task Force (IETF), December 1998.
- [12] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 375-85, June 1996.
- [13] Intel Corp., "IXP1200 Network Processor Family Development Tools User's Guide," March, 2002.
- [14] S. Bradner, J. McQuaid, "A Benchmarking Methodology for Network Interconnect Devices," Request for Comments - 2544, Internet Engineering Task Force (IETF), March 1999.
- [15] Intel Corp., "Intel IXP2800 Network Processor: Product Brief," 2002.

# Templating Transformations for Bitstream Programs

Armando Solar-Lezama  
UC Berkeley  
asolar@eecs.berkeley.edu

Rastislav Bodik  
UC Berkeley  
bodik@eecs.berkeley.edu

## Abstract

*High-performance code is typically the product of a tight collaboration between a domain expert, who writes the high-level model, and a system expert, who tunes the high-level code for a particular machine. Lacking tool support, this collaboration is not very productive: the performance-tuning process involves actually rewriting the original, clean code into a large, hard-to-maintain high-performance code. The result is that once the system expert is done tuning, the domain expert has a hard time modifying his model.*

*We present a tool for making the collaboration between system and domain experts more productive. Specifically, we focus on the domain of bitstream programs. In our tool, the domain expert first writes his algorithm in a high-level domain-specific language (DSL). The system expert then optimizes the original program not by manually rewriting it, or by developing a standard optimizing compiler, but instead by specifying his domain- and machine-specific transformations in a higher-order transformation-specification language (TSL).*

*Our key novelty is that the system expert needs to merely sketch the desired transformation. The system fills in the details missing in the expert's template by constraining the optimized code to behave like the original one. Such "templating" has the potential to improve productivity by (a) allowing rapid prototyping of transformations; and (b) making a transformation applicable even after its original source DSL is to some extent modified.*

## 1 Introduction

Many high-performance codes start as a relatively clean implementation written almost entirely by domain experts, e.g., mathematicians specializing in cryptographic algorithms. Starting from this high-level implementation, domain experts will work closely with

system experts to do extensive performance tuning. In the process, large sections of code are modified to achieve better parallelization or vectorization of loops, better cache performance, or elimination of redundant operations. Often, the result is low-level code that is not maintainable: it is hard to read, debug, or port to a different machine.

Domain experts working in such a software development process often face a dilemma. Whenever they want to explore different high-level algorithms, they must either (i) implement their high-level changes in the heavily optimized low-level code, where they will be expensive to implement and debug; or (ii) write the changes into the original unoptimized version and go through the extensive performance tuning all over again, throwing away the results of the previous tuning effort. The first option is error-prone; the second is laborious; both are inefficient.

This paper focuses on the domain of bit-stream programs, which appear in a wide range of settings, from data compression, to encryption, to coding. In high-performance bit manipulation, there can be large performance differences between naive and hand-tuned implementations, and the latter tend to be hard to debug. For example, in the BitTwiddle benchmark from the NSA suite, the algorithm specification takes less than half a page, whereas a single-processor high-performance implementation in FORTRAN extends over 600 lines of code filled with complicated array index computations. Bit-streaming programs are complicated because the system expert must trade-off several conflicting metrics. As our running example will illustrate, he must determine how to (i) fill computer words with as many useful bits as possible, in order to maximize bit-vector parallelism; (ii) reduce the number of bit shifts within the word, by shifting multiple bits; and (iii) reduce the word-to-word copies.

We present a tool that facilitates the interaction between domain and system experts without both sacrificing high-level clarity and throwing away the tuning effort when the source code is modified. In the

scheme we are proposing, collaboration is achieved by relying on two separate languages, one for the domain experts and one for the system experts. The first language is a DSL that specifies the high-level algorithms to be implemented; the other specifies the performance-improving transformations on the high-level program. Since the high-level code is never directly modified, our approach allows system experts to specify performance improving transformations in a way that preserves the integrity of domain experts' code.

Our approach can be viewed as empowering the system experts with a tool for rapidly developing optimizations closely tailored to the original high-level program. Our tool makes three contributions:

- We observe that a large class of bit-stream programs can be expressed as a composition of affine transformations, which can be specified as Boolean-valued matrices. This observation allows us to use the StreamIt [12] language for expressing bit-stream programs.

It is interesting to note that the process of optimizing bit-stream operations can be viewed as a sort of vectorization of bit operations (where the vectors are the computer words). The StreamIt language, however, does not directly identify or express vector operations. Instead, this vectorization is found during the system expert's tuning process.

- We develop a TSL that allows system experts to specify a large number of transformations. The TSL is a high-order language: it manipulates the program written in StreamIt by the domain expert.
- We allow the user to specify the transformation partially, using a transformation template; the tool then figures out the remaining details. To this end, we develop a simple constraint solver which uses the original reference program and the template to derive a complete transformation. Specifically, the constraint solver looks for decompositions of matrices that specify the transformations on the bit stream.

Since the template is often rather general, it can tolerate small changes in the program. The template with the solver thus act as an optimizer designed by the system expert for a narrow class of the optimizations.<sup>1</sup>

---

<sup>1</sup>The “narrowness” of the class should be viewed as an advantage, as we seek a method for writing optimizations that are powerful because they are tailored to a particular program, or a class of similar programs.

Templating has three important properties:

- The transformations do not change the semantics of the original program, because we constrain the resulting code to behave like the original, reference implementation. As a result, system experts will be free to experiment with different transformations without worrying about introducing bugs into the program.
- Our specifications are rather concise, because we allow the system expert to specify only a template of the transformation. Specifically, we want to manually specify those parts of the transformation that an automated algorithm would have trouble finding.
- The specifications are robust in the sense that they will continue to be applicable even when small changes are made to the original program.

The three properties are actually somewhat complementary. Allowing system experts to specify only a template of a transformation makes the specifications more robust, since changes to the code that affect only the details of the transformation will require no changes to the specification. At the same time, ensuring that the transformed program is semantically equivalent to the original one is what allows us to derive complete transformations from their convenient partial specifications.

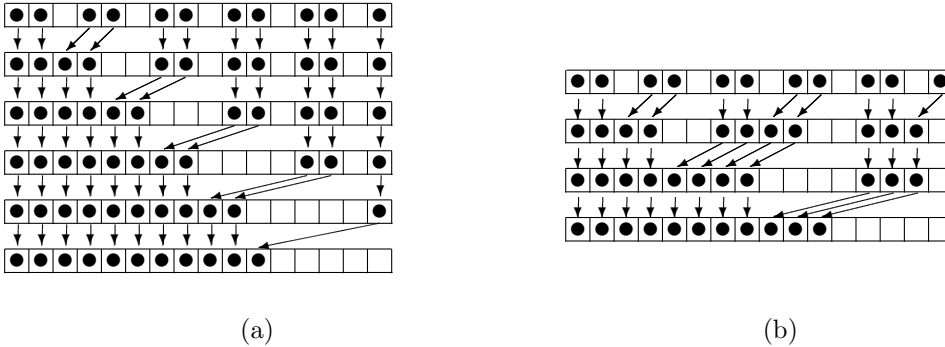
## 2 Bitstream programs and their transformations

To illustrate the bit-stream programs of interest, as well as their high-performance implementations, this section gives an example of a simple bit-stream program and sketches an efficient way of implementing it on a general-purpose processor.

Somewhat surprisingly, the seemingly simple task “drop every third bit from the input stream” can serve as a sufficient canonical example for many complex bit-stream manipulations. While bitstream manipulations can of course be more complex than this, efficiently implementing this manipulation calls for the same transformations as the more complex manipulations we have encountered (e.g., permuting bits in the input stream).

To see why efficiently implementing this manipulation in a 64-bit machine (with the usual operations of bitwise and/or and left/right shifts) is difficult, one needs to realize that efficiently implementing bit operations amounts to vectorizing with additional tweaks: Packing bits into words for parallelism is like vectorization. However, since bits needs to be shifted within





**Figure 1. Two bit compaction algorithms. (a) A naive  $O(n)$  algorithm for compacting bits within a word. (b) An  $O(\log(n))$  algorithm for the same problem. Our running example illustrates how to effectively implement bit compaction in a *stream* (not just a word) using the approach in (b).**

a word and then assembled into output words, we have to keep track of where all the bits are at different points of the execution.

More importantly, there are exponentially many ways in which we could shift the bits. Most of the schemes, including the one that a greedy search would find, are of order  $O(n)$ , but given that the bit shift instructions can move multiple bits within a word at a time, there is an algorithm that allows one to shift the bits within the word in  $O(\log(n))$  time. Figure 1 shows the difference between a naive scheme and the  $O(\log(n))$  scheme for a word-size of 16. The main difference is that the first one moves bits one set at a time, and is therefore not taking full advantage of all the parallelism available. The second algorithm on the other hand moves every other set of bits at every step, therefore reducing the number of gaps between sets of bits by half on every step. In order to exploit this trick effectively, however, it is necessary to minimize the number of times that we move bits across words, since bits that go to different words can not be moved simultaneously with bits that have to stay in the same word.

It is not difficult to see that the amount of code required for this task will be disproportionate to the one sentence description of the task. In addition, a code that really takes advantage of the wide datapath and uses the optimal algorithm would be very hard to modify given even simple changes to the description, like deciding to drop the second bit of every three instead of the third one.

### 3 The stream manipulation language

The domain expert specifies bitstream manipulations with a Domain Specific Language based on the StreamIt language that was originally developed for

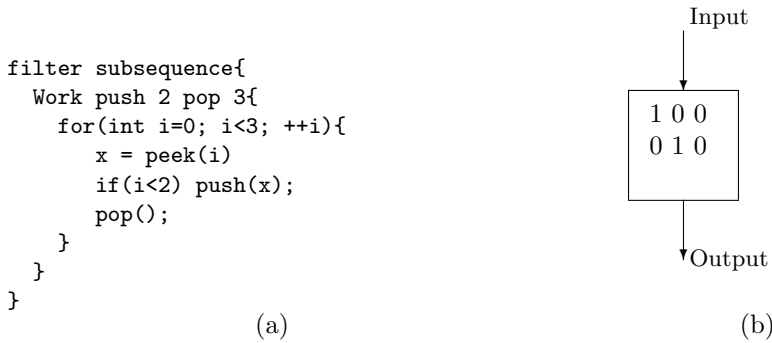
stream programs for signal processing applications [12]. The StreamIt language provides an *abstraction* for manipulating streams of word-sized entities (e.g., floating-point numbers), but this abstraction is suitable also for streams of bits, and we use it without any change.

As far as efficiently *compiling* bitstream programs, while the StreamIt compiler provides a number of optimizations aimed at optimizing word-sized stream programs with linear filters, the optimizations don't help with *bitstreams*, because in the case of bitstreams, the key translation task is how to “pack” elementary operations into bigger ones (i.e., how to pack bits into words), which is not an optimization performed by the StreamIt compiler. Therefore, we need to develop these optimizations ourselves. Developing these optimizations rapidly is the focus of this paper.

The building blocks for StreamIt are filters that can consume  $N_{in}$  elements from an input stream, and compute  $N_{out}$  elements to be written to an output stream. Filters can be of three different types:

**Basic filters** specify the mapping from input to output imperatively in a language similar in syntax to C and can therefore be any arbitrary function, although the transformations presented in this paper apply only to affine filters. In figure 2 we show the description for a filter that performs the task described above, along with a matrix representing the transformation implemented by the filter. The matrix representation for the filters is used through out the paper instead of the actual code to make the illustrations more concise, and to emphasize the fact that the filter is performing a linear transformation on the bit stream.

**Pipelines** chain several filters together. The input to the pipeline is fed to the first filter, and the output of each filter is fed as input to the next filter until you



**Figure 2. A bitstream filter that generates its output stream by dropping every third bit in its input stream. This high-level code specifies the filter’s function, but not a particular implementation of the filter on the target machine. (a) StreamIt code for the filter. (b) A graphical representation of the same filter. Because the filter is linear, it can be represented by a matrix: Each three bits popped from the input stream form a vector; this vector multiplied by the matrix produces the corresponding part of the output stream. That is, there is a column in the matrix for each input bit, and this column specifies where the input bit appears in the output.**

get to the last filter, and then its output becomes the output of the pipeline.

**Splitjoins** are composed of three parts. A splitter, receives the input stream and creates  $N$  streams. Each stream is fed to a filter, and the output of each filter is fed to a joiner which combines them into a single stream that becomes the output of the filter. For the splitter, we implement the same splitters implemented by StreamIt: round robin, and duplication. In the case of round robin, you must give as parameters a list of  $N$  numbers,  $k_1, k_2, \dots, k_N$  where  $k_i$  is the number of bits to pass to filter  $i$  on every round, and  $N_{in}$  for the entire splitjoin will be  $N_{in} = \sum_{i=1}^N k_i$ . In the case of duplication, a copy of the input stream is sent to each filter. As for joiners, we implement again round robin, and xor. In the case of round robin, we need to specify once again  $N$  parameters, but in this case the  $i^{th}$  parameter  $k_i$  is the number of bits to get from filter  $i$  on each time step. The xor joiner will xor together the streams coming out of each filter to produce an output stream.

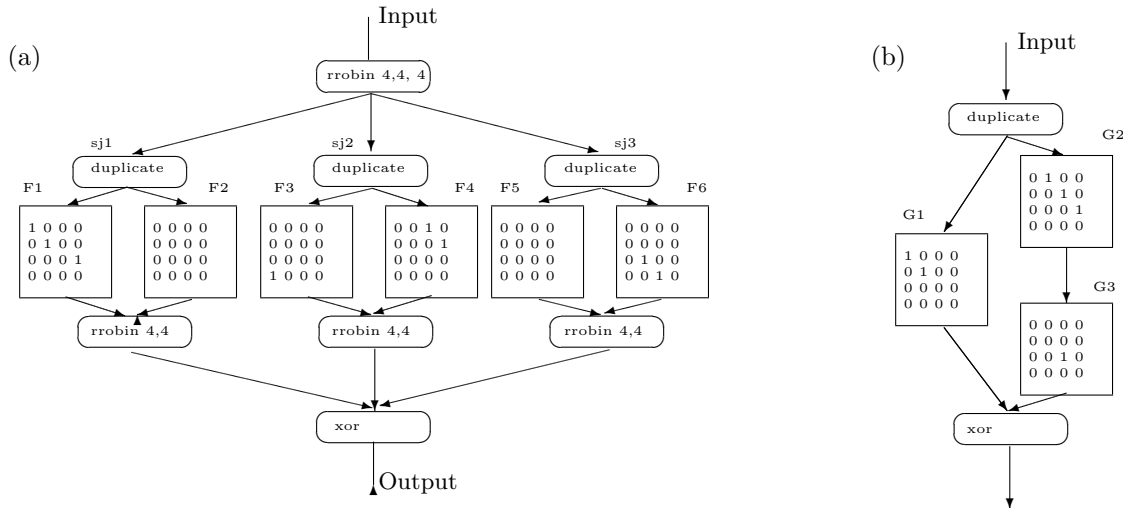
#### 4 Compiling the stream manipulation language

In order to understand how our compilation process works, one must first notice that it is easier to generate code for some programs than for others. In particular, there is a class of programs where each of the basic filters that compose them corresponds to an instruction available in our machine. In general, we refer to filters that can be easily mapped into machine instructions

as being in *low level form* for that particular machine. For a machine with standard bitwise AND, SHIFT, and XOR, table 1 summarizes all the filters that our system regards as being in low level form.

We exploit the distinction between high level and low level stream programs by dividing the compilation into two stages: a transformation stage and a code generation stage. The code generation stage receives a stream program and assumes that the program is already in low level form; therefore it has only to identify to which class each filter belongs, and generate the corresponding statement. This makes the implementation of the code generator quite simple. The only complex algorithm in the code generation phase is the scheduling algorithm, which is the same as the one used by the StreamIt compiler.

In turn, the transformation stage will be responsible for bringing a program to low level form. It is in this stage that most of the optimization can take place, and therefore it is here where the input from the system expert will become important. The transformation stage takes place in two steps. In the first step, the transformer takes in a stream program, and a specification of the transformations provided by the system expert. The transformer then applies the transformation to the program to generate a new program. During the second stage, the program is checked to see if it is already in low level form. If it is not, the system will derive a set of transformations to get the program to low level form before handing it off to the code generator. These transformations will be described in some detail in the next section when we introduce the TSL. The transformer will then generate a new transformation specification that incorporates both the specifications from



**Figure 3. Transforming the high-level algorithm into low-level form, for a 4-bit-word machine. (a) Figure 2 transformed part-way to the low-level form. In this transformed filter, the round robin splitter distributes each 12 bits of input, 4 bits to sj1, 4 to sj2 and 4 to sj3. The sj1 filter will then make two copies of the 4 bits it is given, and pass the copies to F1 and F2. F1 and F2 will each output 4 bits, and the round robin joiner will concatenate their outputs. The outputs of all three splitjoins will be XOR-ed by the outer joiner. See Figure 4 for formal description of the transformation. (b) Filter F1 from (a) after it has been converted to low-level form. This filter is in low-level form (for a 4-bit machine) because G1–3 can be each directly implemented on the target machine: G1 is an AND with the bitmask 1100, G2 is a shift left by one, and G3 is an AND with the bit mask 0010.**

the system expert and the transformations derived by the transformer and returns it to the system expert. This allows the system expert to examine the transformation specification, and to modify it to perform the transformation in a way that generates a more efficient low level program.

In figure 3(a), for example, we show a graphical representation of the filter in fig 2 after it has been partially transformed to low level form for a 4 bit machine. All the filters are already square matrices of size 4x4, but the filter is not yet in low level form because we do not have any instructions in our machine that perform the transformations F1 or F6. In figure 3(b), we show how F1 can be brought to low level form. In this case, we took each one of the two diagonals from the matrix in filter F1 that have non-zero entries, and turned them into separate filters joined by a splitjoin. For the one which had the non-zero in the first diagonal above the main one, we did a further factorization, and the result is a set of filters that correspond to shifts and ANDs with bitmasks.

## 5 Automatically Transforming the program

To better understand how the transformation from figure 2 to the one in 3(a) was performed, we must first introduce the TSL. The TSL is a high order language that allows us to specify transformations in a concise and robust way. The main component in the TSL is a set of transformations which take in a reference to a filter and return a transformed filter. The main transformations are listed in table 2.

When the system has to bring a program to low level form on its own, the system will start by making all splitjoins operate on word size chunks. In particular, we want the number of bits passed to a each filter by the splitter, and the number of bits received from each filter by the joiner to be a multiple of the word-size, since we can only pass complete words to each filter. This goal is achieved by the `SRRtoDup` and `JRRtoXOR` transformations. These transformations turn round robin splitters and joiners into duplication splitters and xor joiners respectively, adding additional filters to preserve semantics of the splitjoin. Figure 6 shows an example of the use of these two transformations.

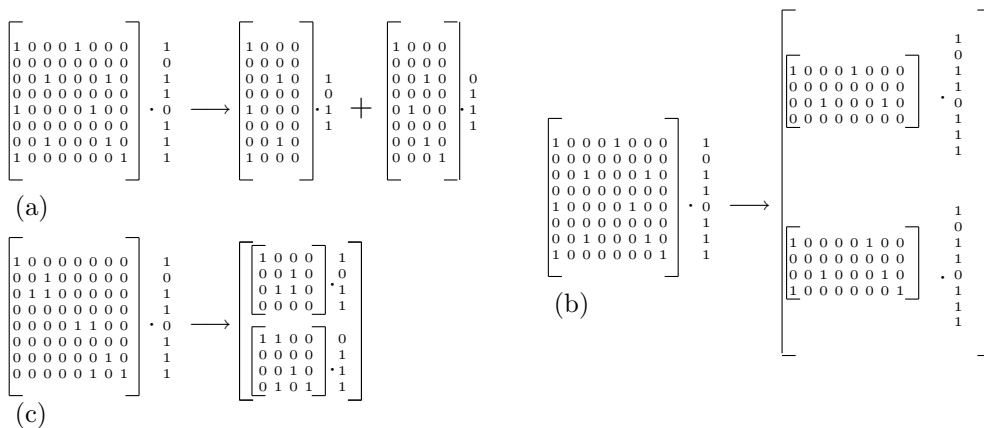
The system will then apply the `Unroll [N]` transfor-

```

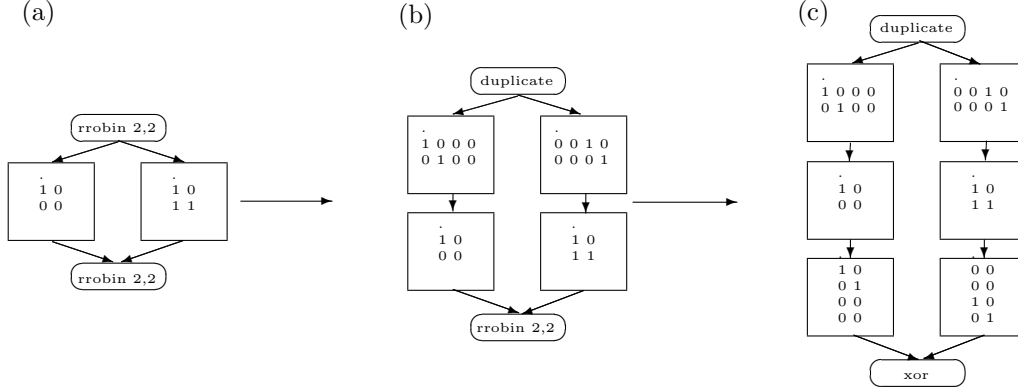
subsequence = Unroll[4](subsequence); // 1
subsequence = ColSplit[4](subsequence); // 2
subsequence.subsequence_1 = RowSplit[4](subsequence.subsequence_1); //3
subsequence.subsequence_2 = RowSplit[4](subsequence.subsequence_2); //4
subsequence.subsequence_3 = RowSplit[4](subsequence.subsequence_3); //5

```

**Figure 4.** The TSL specification for transforming the program in Figure 2 into the one in Figure 3(a), for a 4-bit machine. Statement 1 unrolls the filter (called subsequence) so that it takes 12 bits as input and returns 8 as output. Statement 2 decomposes the unrolled filter into a splitjoin of three different filters, each of which consumes 4 bits and outputs 8 bits (see also Figure 5). Each of the new filters correspond to four columns of the matrix generated by Statement 1. The new filters are named subsequence.subsequence.i. Finally, each of the three new filters is decomposed into a splitjoin of two filters, each of which consuming 4 bits and producing 4 bits.



**Figure 5.** The three splitting transformations. (a) ColSplit[4] transforms a filter into two filters, each receiving only half of the input of the original filter, since the input is now distributed round robin among the two. Their output is XOR-ed, so the semantics of the filter is unchanged. (b) The effect of RowSplit[4] on the same filter. In this case, the input is duplicated, and passed to the two new filters. Their results are concatenated by a round robin join. (c) DiagSplit[4] creates two new filters that correspond to 4x4 blocks from the diagonal of the original matrix. The input is distributed round robin, as in (a), but the output is concatenated as in (b). In order to guarantee semantics preservation, the system checks that the rest of the matrix is all zeros.



**Figure 6. The splitjoin transformations. (a) The original filter. (b) The result of applying `SRRtoDUP` on the splitjoin in (a). (c) The result of applying `JRRtoXOR` to (b). Note that the two transformations eliminated the round robin splitters and joiners, which is useful when the size of the round robin splitters and joiners is not a multiple of the word size.**

mation with

$$N = \frac{\text{word-size}}{\text{gcd}(N_{in}, N_{out}, \text{word-size})}$$

This transformation will make the number of inputs and outputs from each filter be multiples of the word size. After unrolling, the `ColSplit[N]` and `RowSplit[N]` transformations are then used to break transformations which read more than word-size bits from the input or write more than word-size bits from the output, as shown in Figure 5.  $N$  will be set to the word size, in order to obtain matrices that read and write exactly word-size bits from the input and output. Figure 4 shows the TSL specification relying on the three transformations, which transforms the program in Figure 2 into the one in Figure 3(a).

Finally, when all matrices will be square of word size, `StandardDecomp` will be applied to them. This transformation was applied to F1 in figure 3(a) to obtain the filter in low level form shown in figure 3(b). The transformation decomposes the filter into a splitjoin of filters whose corresponding matrices contain non zero elements in only one of their diagonals. There will be one filter whose matrix will contain nonzero elements only in the main diagonal, in the case of our example it is G1, and this filter can be translated directly to an AND with a bitmask. The rest of the filters will be further split into a pipeline where one filter contains all ones in one of its diagonals, like G2 in the figure, and one that contains ones and zeros in the main diagonal, like G3.

Figure 8 shows the transformation specification that

the transformer would generate for a 16 bit machine for this same problem if no specification is given by the system expert. The code resulting from this transformation will probably be much faster than anything that could be written by hand in 9 lines of code, which is what our specification takes. Nevertheless, this code will still be largely suboptimal, and would not match the performance of an optimized hand written implementation. The main problem with the generated code is that when using `StandardDecomp`, the resulting code will use the bit shifting scheme from 1(a). In order to get the system to use the  $O(\log(n))$  scheme, the system expert will have to provide a template for the transformation.

## 6 Aiding the Automatic Transformation through Templating

In order to obtain a program that uses the fast bit shifting algorithm, the system expert will have to provide a better TSL transformation than the one synthesized by the system in the previous section. Our TSL transformations are sufficient for the purpose, but specifying the transformation exhaustively may be tedious and not robust across changes in the input program. Therefore, we allow the system expert to specify merely a template of the transformation.

The two TSL factorization transformations `Factor` and `PermutFactor` are the most powerful tools for the system expert to create the template. The factorization transformations break up a basic filter into pipelines of simpler filters. Both are parameterized by either a

complete specification of the decomposition or a template for it. The main difference between them is that where as `Factor` will work on any arbitrary filter, `PermutFactor` is specialized for filters that simply perform a permutation of the bits in the input stream. Such filters can be very common in many encryption algorithms like DES, and focusing on them allows us to use more powerful algorithms for deriving the decompositions.

`Factor` will receive as its arguments a decomposition consisting of a sequence of filters into which we want to decompose the given filter. A placeholder can be used in place of a filter that we don't want to specify, and the system uses basic linear algebra to determine what this filter should be. If it is not possible to solve for a missing filter, or if the factorization is fully specified, but the specification results in a filter different from the one passed as an argument, the transformation terminates with an error message.

`PermutFactor` will decompose a transformation that performs permutations of bits into a pipeline where each of the stages moves some of the bits by a certain amount. A complete specification will say which bits to move at each step and by how much to move them. In the case of a template, we must specify into how many filters to decompose the permutation. For each filter, we can give the system a set of constraints as to what it can do with each of the bits. The constraints are given in the form of shift templates, and the system will consider itself free to move any bit not mentioned in a shift template in any way necessary to satisfy the other constraints. Table 7 describes the 5 types of shift templates allowed by the system, along with the syntax for combining them.

In order to see how `PermutFactor` derives complete specifications from the templates, consider the case where we are given shift specifications of type 1, 2 and 3 only. For this case, we can use the constraints to construct a system of linear equations as follows. First, let  $x_{i,j}$  correspond to the amount that filter  $j$  shifts the bit that was originally in position  $i$ , and let  $K_i$  be the total amount we will have to move bit  $i$  which is derived from the semantics of the original program. Then, we have that for  $i = 1 \dots \text{word-size}$ ,  $\sum_{j=0}^{j=N} x_{i,j} = K_i$ . Now, constraints of type 1 and 2 will make some of the  $x_{i,j}$  constant, while constraints of type 3 will allow us to replace all the  $x_{i,j}$  listed in the shift specification with a single variable, therefore reducing the number of unknowns. The resulting system has to be solved over the integers, since we can only shift bits by an integer amount, but this can be done efficiently even for large

systems (see [15]). If the system has no solution within the allowed range of shifts, the user is notified that it is not possible to satisfy the template while preserving the semantics of the original program.

In the presence of constraints of type 4 and 5, we have to search through many of the combinations of values allowed by the constraints of this type, and find a set that satisfies all the constraints. The search can be worst case exponential, but if the system fails to find a solution in a reasonable amount of time, it will ask the system expert to refine the specification.

In the case of our example, the system expert can use the `PermutFactor` both to specify the  $O(\log(n))$  algorithm for shifting bits within a word, and to tell the system to pack all the bits within a word before moving bits across words. Figure 9 shows the transformation specification for doing this, again for a 16 bit machine. The specification is fairly concise due to the use of templating. In addition, small changes to the algorithm will require no changes to the specification, for example, if I decide to pick the first and third bit instead of the first and second, the transformation specification will not need any modifications at all.

## 7 Related Work

The issue of productivity has been a theme to varying degrees in much of the high performance literature. Optimizing compilers, for example, allow programmers to write code in a high level but general purpose language while the compiler takes care of transforming the program to run efficiently in the target machine. Modern compilers can perform a number of transformations aimed at eliminating redundant or unnecessary operations, improving memory access locality, and for exploiting particular machine features such as vector processors (see [3] for a very complete survey).

Our approach, complements traditional compiler technology by allowing the users to specify transformations that are too specific to a particular problem to be worth including in a compiler. At the same time, by compiling into a mature language like C, we benefit from the optimizations performed by high end C compilers. Our project is part of the larger PERCS project, whose Programming Model effort investigates a spectrum of approaches to improving productivity in high-performance application development [6]. In the context of the productivity metric defined in [6], the approach in our paper seeks to allow developers with higher level of expertise finish a task sooner.

Domain specific languages have also been widely studied for their potential to improve productivity. There have been a number of recent efforts to improve

```

Syntax for PermFactor: PermFactor[shiftSpec]( filter );
shiftSpec := [ shiftList ] shiftSpec
shiftList := shift( numberList by amount ) , shiftList
numberList := number , numberList | number:number , numberList
amount := ? | optionList
optionList := number || optionlist

```

---

type 1	shift( i by j)	Shifts bit i by j
type 2	shift( set by j)	Shifts all the bits listed in set by j
type 3	shift( set by ?)	Shifts all the bits listed in set by the same amount, but gives the system freedom to select the amount.
type 4	shift( i by a    b ...)	Shifts bit in position i by either a, or b, or any other amount in the optionlist.
type 5	shift( set by a    b ...)	Shifts each of the bits listed in set by either a or b, or any other amount listed in the option list. They don't have to all move by the same amount.

**Figure 7. Syntax for PermFactor, and clasification of the shift templates.**

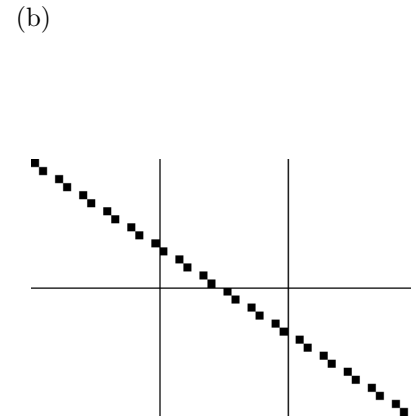
```

(a)
WSIZE=16;
subsequence = Unroll[WSIZE](subsequence); // 1
subsequence = ColSplit[WSIZE](subsequence); // 2

for(i=0; i<3; ++i){
  subsequence.filter(i)=
    RowSplit[WSIZE](subsequence.filter(i));
}

for(i=0; i<3; ++i){
  for(j=0; j<2; ++j){
    subsequence.filter(i).filter(j)=
      StandardDecomp(subsequence.filter(i).filter(j));
  }
}

```



**Figure 8. The naive transformation for compiling the filter in Figure 2 to low level form. (a) the transformation specification of the transformation. The code generated from the low-level form will compact bits within each word using the slow algorithm in Figure 1(a). (b) The filter matrix after the Unroll operation. The squares correspond to non zero entries. Each of the six submatrices corresponds to one of the basic filters obtained after performing the ColSplit and RowSplit operations.**

```

WSIZE=16;
subsequence = Unroll[WSIZE](subsequence);
subsequence = PermutFactor[ [shift(1:2 by 0), shift(17:18 by 0), shift(33:34 by 0)],
                           [shift(1:16 by ?), shift(17:32 by ?), shift(33:48 by ?)]
                           ] ( subsequence );

subsequence.subsequence_1=DiagSplit[WSIZE](subsequence);
for(i=0; i<3; ++i){
subsequence.subsequence_1.filter(i) =
    PermutFactor[ [shift(1:16 by 0 || 1)],
                  [shift(1:16 by 0 || 2)],
                  [shift(1:16 by 0 || 4)]
                ]( subsequence.subsequence_1.filter(i) );
}

```

**Figure 9. The system-expert-provided transformation for compiling the filter in Figure 2 to low level form. The code generated from the low-level form will compact bits within each word using the fast algorithm in Figure 1(b). The first `PermutFactor` instructs the system to decompose the unrolled filter into two filters. The first filter can move bits any way it desires, but it is prohibited from moving either bits 1,2 17, 18, 33, or 34. The second filter must shift all the bits within a word by the same amount. With these constraints, the system determines that in order to satisfy both constraints, in the first step it must pack all the bits within the word, and on the second step it can pack bits across words. The second `PermutFactor` provides a template for the fast bit packing algorithm by instructing the system that in the first step all the bits must either stay in place or move by 1. In the second step, they can only move by two, and in the third step, can only move by 4.**

the performance of domain specific languages. Padua et.al., for example, have made significant progress towards making the performance of MATLAB comparable to that of handwritten code through the use of aggressive type and maximum matrix size inference[1, 5].

Kennedy et. al. have worked to improve the performance of domain specific languages through the use of an approach called telescoping languages. The idea is to build libraries to provide component operations accessible from the domain specific language, and pre-compile several specialized versions of them tailored for different sets of conditions that may hold when the routine is invoked, therefore avoiding the expense of having to perform extensive interprocedural optimization at the time the program in the domain specific language is compiled, while still achieving good performance [10] [4].

The StreamIt language on which our abstraction is based, builds upon a large body of work on Synchronous Data Flow programming languages (see [8] or [14] for examples of this work). It's compiler automatically identifies linear filters, and performs many optimizations targeted towards DSP applications.

Aspect Oriented Programming aims at supporting the programmer in “cleanly separating concerns and aspects from each other, by providing mechanisms that

make it possible to abstract and compose them to produce the overall system” [11]. Our approach can be understood as a form of Aspect Oriented Programming, where the algorithm specification and the performance improving transformations are the two aspects we are dealing with. There have been other efforts at applying aspect oriented programming to restricted application domains, for example Irwin et. al. demonstrate the use of Aspect Oriented Programming in the domain of sparse matrix computations[9].

Finally, one of the most widely used methods for improving productivity in high performance computing is the use of libraries. Successful domain specific libraries in widespread use include BLAS [13] and LAPAC [2]. Code that uses the high performance libraries can be clean and fast, and composed relatively quickly, but within the library, the problems described above sometimes even become amplified, since the library must be performance tuned for lots of different platforms. More recently, a lot of work has been devoted to the development of self tuning libraries like FFTW [7], or SPIRAL [17], as well as runtime adaptive libraries like STAPL [16]. Our approach could provide an alternative for these approaches for cases when the narrow applicability of a library does not warrant the high development effort of a self tuning or runtime adaptive system.



## 8 Conclusion and Future Work

We have presented a tool for making the collaboration between system and domain experts more productive in the domain of bitstream programs. The key to this scheme is to allow the system expert to optimize the original program not by manually rewriting it but instead by specifying his domain- and machine-specific transformations in a higher-order, transformation-specification language. The TSL allows the system expert to write special purpose optimizations in a way that is safe, robust and concise.

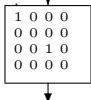
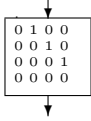
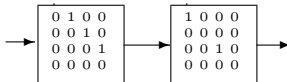
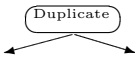
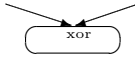
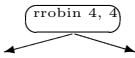
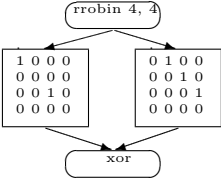
We have implemented code to take an AST of the transformation specification, apply the transformations specified onto an AST of the main program, and then apply any additional transformations needed to bring the program to LLF. We have also implemented the code generator described in section 4. Within the next few weeks, we would like to get some performance numbers for the generated code for some sample bit manipulation tasks to compare performance of code generated from the default transformation versus that with the user directed transformation.

## Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056.

## References

- [1] G. Almasi and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.
- [2] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings, Supercomputing '90*, pages 2–11. IEEE Computer Society Press, 1990.
- [3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
- [4] A. Chauhan, C. McCosh, and K. Kennedy. Automatic type-driven library generation for telescoping languages. In *Proceedings of SC: High-performance Computing and Networking Conference*, Nov. 2003.
- [5] L. DeRose and D. Padua. A matlab to fortran 90 translator and its effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96, Philadelphia, PA*, pages 309–316, May 1996.
- [6] K. Ebcioğlu, V. Sarkar, and C. Williams. Application development productivity challenges for high-end computing. In *The HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004)*, 2004.
- [7] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.
- [8] N. Halbwegs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proceedings of the IEEE*, pages 1305–1320, September 1991.
- [9] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, number 1343 in LNCS, Marina del Rey, CA, 1997. Springer-Verlag.
- [10] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.
- [11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997.
- [12] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear analysis and optimization of stream programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.
- [14] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, September 1987.
- [15] T. Mulders and A. Storjohann. Diophantine linear system solving. In *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 181–188. ACM Press, 1999.
- [16] A. Ping, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *Int. Workshop on Languages and Compilers for Parallel Computing*, August 2001.
- [17] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, accepted for publication.

Description	Example	
Simple filters of size NxN with non zero entries only in the main diagonal		bitwise AND with 1010 out = in & 1010;
Simple filters of size NxN with all ones in one of their diagonals		shift left by one out = in << 1;
Pipeline where all the filters are in low level form.		chain the operations tmp = in << 1; out = tmp & 1010;
Duplication splitter for splitjoin		give a copy to each filter in_1 = in; in_2 = in;
Xor joiner for splitjoin		aggregate outputs from all filters out = 0; out = out xor out_1; out = out xor out_2;
Round robin splitter or joiner where all the block sizes are multiples of N.		Pass one word to the first filter and one word to the second. in_1 = in[0]; in_2 = in[1];
Splitjoin where the splitter, the joiner and all the filters are in low level form.		in_1 = in[0]; in_2 = in[1]; out_1 = in_1 & 1010; out_2 = in_2 << 1; out = 0; out = out xor out_1; out = out xor out_2;

**Table 1.** Listed in the table are all the filters for which it is possible to generate code without further transformations. For the sake of clarity, the examples assume a 4 bit word length, and constants are shown in binary.

Name	Type	Description
<code>Unroll [N] (filter)</code>	<code>BasicFilter</code> → <code>BasicFilter</code>	Copies the body of the filter N times, so the filter will now take in N times as many bits as before, and take out N times as many bits as well. Useful for getting the filter size to be a multiple of the word size.
<code>ColSplit [N] (filter)</code>	<code>BasicFilter</code> → <code>SplitJoin</code>	Splits a large filter into several small filters, each taking in N bits, and putting out the same number of bits as the original basic filter. The splitjoin will do roundrobin split, and xor join. Figure 5(a) illustrates the behavior of <code>ColSplit</code> .
<code>RowSplit [N] (filter)</code>	<code>BasicFilter</code> → <code>SplitJoin</code>	Similar to <code>ColSplit</code> , but this maintains the number of inputs to take, while changing the number of outputs. Its behavior is illustrated in 5(b).
<code>DiagSplit [N] (filter)</code>	<code>BasicFilter</code> → <code>SplitJoin</code>	This function will return a splitjoin with round robin split and round robin join, where each of the filters is an NxN block from the diagonal of the original filter. Figure 5(c) shows the behavior of this filter.
<code>SRRtoDup (filter)</code>	<code>SplitJoin</code> → <code>SplitJoin</code>	Turns a splitjoin with a round robin splitter into a splitjoin with a duplication splitter. In order to preserve the semantics, it will add a filter before every filter in the splitjoin to select out of the stream only bits that the filter would have received with the round robin splitter.
<code>JRRtoXOR (filter)</code>	<code>SplitJoin</code> → <code>SplitJoin</code>	Turns a splitjoin with a round robin joiner into a splitjoin with an xor joiner. As before, filters must be added at the output of each filter in the splitjoin to preserv semantics.
<code>StandardDecomp (filter)</code>	<code>BasicFilter</code> → <code>SplitJoin</code>	Performs a standard decomposition that in the case of NxN filters is guaranteed to bring them to low level form.
<code>Factor [filterList] (filter)</code>	<code>BasicFilter</code> → <code>Pipeline</code>	Decomposes a filter into a pipeline of filters. filter list is simply a list of comma delimited filters, and can have the symbol ? to represent a wildcard. In which case, the system derives a description for the missing filter.
<code>PermutFactor [SpecLst] (filter)</code>	<code>BasicFilter</code> → <code>Pipeline</code>	The given filter must represent a permutation of bits. It can only move bits from one position to another, or take bits out of a stream. The specification is described in figure 7

**Table 2. Transformations available in the system**

# Performance and Productivity in Parallel Programming via Processor Virtualization

Laxmikant V. Kalé  
kale@cs.uiuc.edu  
Department of Computer Science  
University of Illinois at Urbana-Champaign

## Abstract

*We have been pursuing a research program aimed at enhancing productivity and performance in parallel computing at the Parallel Programming Laboratory of University of Illinois for the past decade. We summarize the basic approach, and why it has improved (and will further improve) both productivity and performance.*

*The centerpiece of our approach is a technique called processor virtualization: the program computation is divided into a large number of chunks (called virtual processors), which are mapped to processors by an adaptive, intelligent runtime system. The runtime system also controls communication between virtual processors. This approach makes possible a number of runtime optimizations.*

*We argue that the following strategies are necessary to improve productivity in parallel programming:*

- *Automated resource management via processor virtualization*
- *Modularity via concurrent composability*
- *Reusability via frameworks, libraries, and multi-paradigm interoperability*

*Of these, the first two directly benefit from processor virtualization, while the last is indirectly impacted. We describe our research on all these fronts.*

## 1. Introduction

Parallel programming is more difficult than sequential programming because of the additional issues of determinism, synchronization, communication costs, load imbalances and performance portability that must be addressed by the programmer. As a result, productivity of parallel programming efforts tends to be low.

Recognizing the importance of high productivity, in the early days of parallel computing researchers aimed at automatic parallelizing compilers. However, after decades of very stimulating research [7, 38, 17, 18, 9], it has become clear that although some of the tools produced can indeed extract almost all the parallelism from the *given* code, a from-scratch parallel reformulation is often required to attain higher performance.

We have been pursuing an approach to high productivity with scalable performance even for complex, dynamic parallel applications for the past decade [25]. One of the guiding principles for us is to seek an optimal division of labor between the programmer and the “system”. The human programmers do what they can do best, while leaving only what can be efficiently automated to the system. Specifically, we find that programmers are best at finding and expressing the natural parallelism of the application, but the runtime system can efficiently carry out resource management and many performance optimizations.

We think that parallel programming productivity can be increased by advancing the state of art on the following fronts:

- **Automatic resource management:** Writing a parallel program involves managing and allocating resources including processors, memories and networks to application data and computations. Especially for irregularly structured and/or dynamically varying applications, such resource management entails a significant programming effort. At the same time, advances in algorithms create smarter algorithms (with lower total operation counts) that tend to be irregular in structure. For example, for N-body interactions, a simple  $O(n^2)$  algorithm is easy to parallelize, where  $O(n \log n)$  algorithms (such as Barnes-Hut) or  $O(n)$  algorithms (such as Fast Multipole) are more complex. Applications themselves are tackling more dynamically evolving

scenarios, typically requiring adaptive refinements as the computation progresses. If the programmer were freed from dealing with resource management issues, their burden would be significantly reduced.

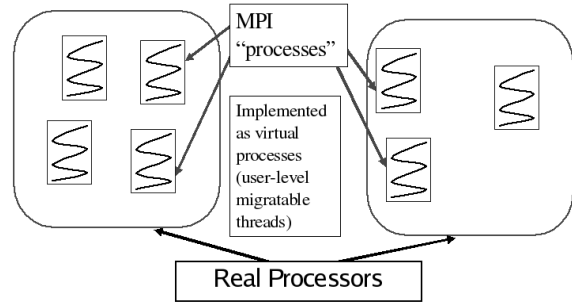
- **Concurrent compositionality:** It should be possible to compose independently developed parallel modules into an application, in such a way that the execution of composed modules may overlap in time or space (i.e. processors); moreover this “concurrent composition” must be achieved without losing efficiency. With this capability, it would be possible for application structure to be based on logical interactions of its modules, automatically overlapping the computation and communications across modules.
- **Techniques for promoting reuse of parallel software components:** Because a parallel module operates in a more complex context, it is more difficult to reuse it than a sequential component. Yet, the complexity of parallel software puts a higher premium on reuse. Thus, we must develop techniques that eliminate the barriers to reuse of parallel software.

In this paper, we illustrate the research we have been carrying out towards these objectives. One of the enabling factors in our research is the idea of processor virtualization. We begin with a brief exposition of this idea.

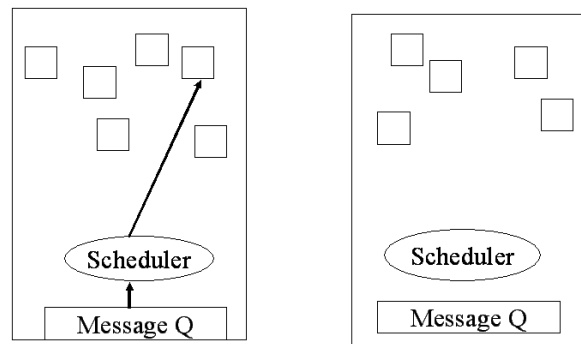
## 2. Processor Virtualization

Processor virtualization is a simple idea: the programmer decomposes the computation, without regard to the physical number of processors available, into a large number of objects, which we call **virtual processors (VPs)**. The programmer leaves the assignment of virtual processors to physical processors to the runtime system. The virtual processors themselves can be programmed using any programming paradigm: e.g. they can be MPI “processes” implemented as user-level, extremely lightweight, threads (NOT to be confused with system level threads or Pthreads), that interact with each other via messages, as in Adaptive MPI [20]. Alternatively, they can be organized as indexed collections of C++ objects that interact via asynchronous method invocations, as in Charm++ [28].

This simple idea has significant consequences. Most importantly, from the point of view of this paper, it empowers the **run-time system (RTS)** to optimize resource allocation by migrating VPs across processors. The RTS



**Figure 1. Processor Virtualization in Adaptive MPI: An MPI process is implemented as a user-level thread, several of which can be mapped to one single physical processor.**



**Figure 2. Message-Driven Execution with a processor-level scheduler**

is also involved in delivery of messages to VPs. as a result, it can optimize communication as well.

Let us first sketch the direct consequences of processor virtualization: since each physical processor may house hundreds (or even thousands) of VPs, the RTS needs to have a scheduler to decide which VP executes next. This scheduler can (and indeed must) be message-driven: it only schedules VPs that are ready to execute because they have a message pending. This message-driven scheduler turns out to be a critical component from the point of view of concurrent composition.

Second, since VPs may migrate as a program evolves, the RTS needs to maintain information about where each VP is located. This can (and must) be done efficiently, without bottlenecks. Our implementations ensure that in most cases, messages are delivered to VPs without any forwarding, with the assumption that migrations are not as common as messages [34].

Charm++ and Adaptive MPI are systems we have de-

veloped over the past 14 years that embody this idea of processor virtualization. For concreteness, the next few sections assume each VP as running an MPI “process” and interacting with others via the usual MPI sends and receives.

### 3. Automated Resource Management

Processor virtualization empowers the run-time system (RTS) to incorporate intelligent optimization strategies. We discuss two categories of such strategies below.

#### 3.1. Automatic Load Balancing

Probably the most obvious advantage of processor virtualization is that the runtime system can do automatic load balancing dynamically. Since the application program never sends messages directly to physical processors, the RTS is free to migrate the VPs across processors any time it pleases.

Of course, the RTS must be quite “intelligent” for this to work, but it is certainly possible [34]; what is more, only one RTS needs to have these smarts, whereas all the application programs can just use it.

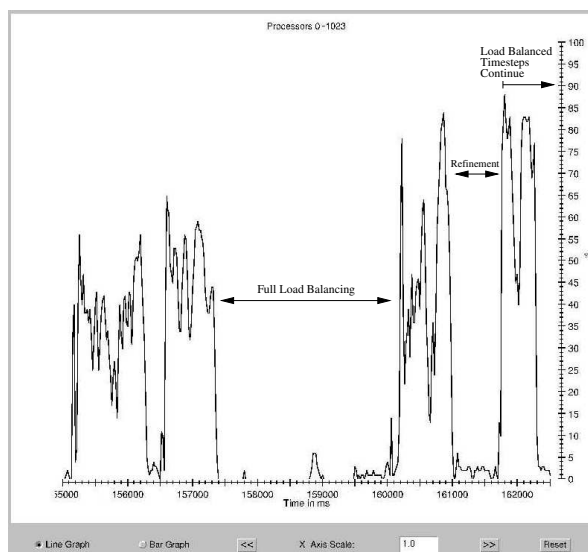
In the simplest possible setting, the RTS can monitor the load on a processor and its neighbors. If/when a (physical) processor goes idle, the RTS sends a request for additional VPs from a neighboring processor. Other variations on this idea are possible [23, 40].

A more interesting and fruitful set of strategies becomes feasible when we observe a property of many parallel computations, especially those involving physical systems. Even for dynamic applications, the computation loads and the communication patterns exhibited by the VPs tend to persist over time for most of the VPs. This is because often dynamic variations happen abruptly but infrequently (as with periodic mesh refinements) or frequently but slowly (as with migrations of particles in n-body codes including molecular dynamics).

Based on this “principle of persistence” (which is a heuristic principle, like the principle of locality), one can now build measurement based runtime load balancing strategies. The RTS can instrument the VPs to record computational load and communication patterns. It can do this automatically, without user code, since the RTS is the intermediary for both scheduling and communication. Load balancing strategies can then use this database in a centralized or distributed manner to effect remapping decisions periodically. (These periodic decisions can be augmented by “idleness-based” schemes as mentioned above when necessary.)

We and others have implemented many such strategies, and work is ongoing on strategies that observe more subtle patterns, such as dependences, critical paths, multiple-phases-within-iterations and so on. However, the main point is that the application programmer doesn’t have to worry about this important aspect of their parallel program.

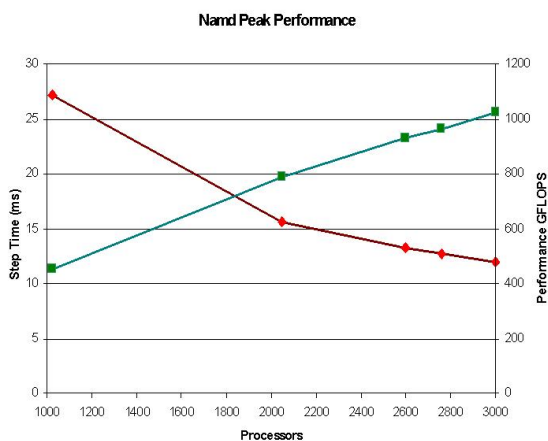
As a concrete example, we show dynamic load balancing in action in NAMD, the highly scalable molecular dynamics program used routinely by biophysicists. Figure 3 shows processor utilization against time for a NAMD run of 1024 processors [29]. The initial greedy balancer works from 157 through 160 seconds (the period in the graph with the dip in utilization), leading to some increase in average utilization. Further, after the refinement strategy finishes (within about .7 seconds) at around 161.6 seconds, we can see that utilization is significantly improved. In this figure, we may appear to spend too much time on load balancing; however, in molecular dynamics, such load balancing is needed only after several thousand timesteps [39].



**Figure 3. Processor Utilization against Time on 1024 processors**

As a result of such runtime optimizations, NAMD has attained an unprecedented high performance on several thousand processors, leading to a Gordon Bell award [39]. The performance of NAMD using Charm++ on PSC Lemieux is shown in Figure 4. Each timestep takes around 25 seconds on 1 processor. This drops to 27 milliseconds on 1000 processors and finally down to 12 milliseconds when scaled to 3000 processors with a corresponding performance level of 1000 GFLOPS. Not only are the achieved speedups impressive, the absolute time

taken per timestep (12ms) is also lowest by a considerable margin compared with other molecular dynamics programs.



**Figure 4. Processor Utilization against Time on 1024 processors**

### 3.2. Communication Optimizations

Since the RTS mediates communication, it can intercept the communication and replace communication algorithms as required by the patterns observed. This is especially true for collective operations. By keeping track of the number of processors (VPs and physical) involved, the amount of data, and the state of the rest of the computation, the RTS can decide which of the available collective communication algorithms will be better suited, and switch it at runtime. Our own results in this area have been promising [33].

Automatic runtime communication optimizations can also be performed for other non-collective operations. For example, a graph partitioning application written by a newcomer to parallel programming used an extremely fine-grained communication style: very short messages with a few bytes of data were being sent, which would have led to bad performance. However, by interposing the streaming library of the communication subsystem of the RTS (which collects short messages locally, and sends them using a virtual mesh), this was optimized without changing user code. In this case, the interposing was done manually, but as the RTS capabilities are improved it can make such decisions itself. (Even with the manual interposing, the advantage still remains that the user code didn't have to change.)

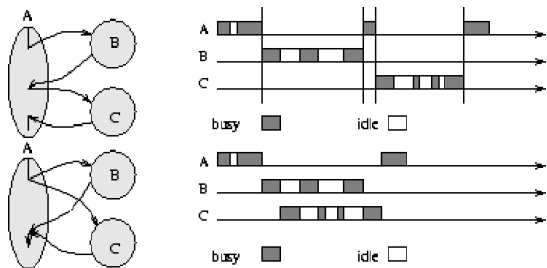
## 4. Modularity via Concurrent Composition

For high productivity in parallel programming, one should be able to modularize the program. In particular, it should be possible to compose independently developed parallel modules into a single parallel application (or into higher level modules, composed hierarchically). Further, the modules being composed should be allowed to overlap their execution in time, and over processors. Without this flexibility, one risks the danger of fragmenting the set of processors (especially when a large number of modules are being composed) and certainly loses the ability to exploit adaptive overlap of communication and computation across modules. This is illustrated with a schematic and application example below.

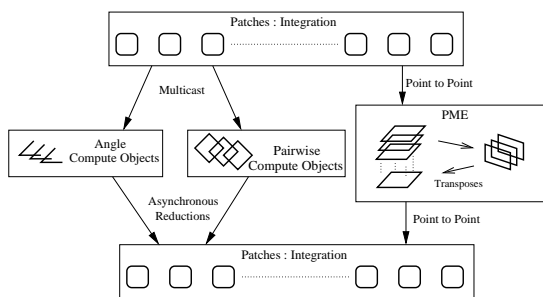
Consider the situation in Figure 5(a). A, B and C are each parallel modules spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style programming, one must choose one of the modules (say B) to call first, on all the processors. The module may contain sends, receives, and barriers. Only when B returns can A call C on each processor. Thus idle time (which arises for a variety of reasons, including load imbalance and critical paths) in each module cannot be overlapped with useful computation from the other, even though there is no dependence between the 2 modules.

In contrast, with processor virtualization (and the message-driven execution induced by it), A invokes B on each processor, which computes, sends initial messages, and returns to A. A then starts off module C in a similar manner. Now B and C interleave their execution based on availability of data (messages) they are waiting for. This automatically overlaps idle time in one module with computation in the other, as shown in the Figure. One can attempt to achieve such overlap in MPI, but at the cost of breaking the modularity between A, B and C. With processor virtualization, the code in B does not have to know about the code in A or C, and vice versa.

This phenomenon is illustrated in NAMD (Figure 5(b)). The computation partitions atoms into a set of cubic cells called “patches”. Interactions between atoms in adjacent cells are computed by separate virtual processors called the “pairwise compute objects” in the Figure. The PME (Particle-Mesh Ewald) module involves two 3D-FFTs (each with a communication intensive transpose operation) over a relatively small grid (192x144x144 in one case). By concurrently composing the PME and force-calculation modules, it becomes possible to use the considerable latency of the transposes in the PME algorithm with pairwise-force computations adaptively. Neither partitioning of processors among the two modules, nor sequencing their execution one after



(a) Modularity and Adaptive Overlapping: Schematic



(b) Concurrent Composition of PME and Force Computations in NAMD

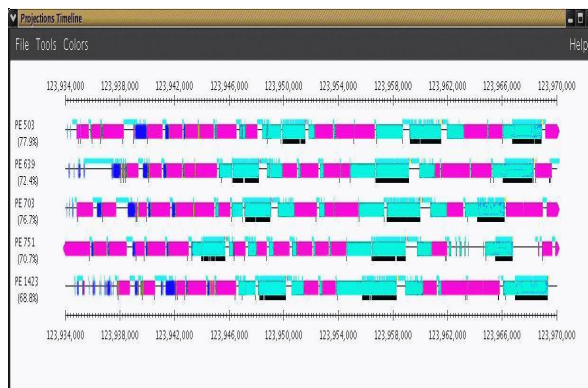
**Figure 5. Concurrent Composition**

the other will yield the same efficiency of concurrent composition employed by NAMD. Moreover, this efficiency is attained without any coding by the programmer to juggle execution between the two modules.

Figure 6 shows the timeline of a few processors in a 2112 processor NAMD run on PSC’s Lemieux alpha-cluster. The light gray rectangles (as well as the darkest gray rectangles at the beginning, around 123.938 secs) represent components of the PME computations, whereas the medium gray rectangles are pairwise (and bonded) force computations. The overlap of the two modules’ operations can be clearly seen. On PSC’s Quadrics communication network, the communication co-processors ensure that the CPU spends only a small time on communication. Therefore, all the latency of the transpose operation (between the yellow sections) is available for doing useful pair-wise force computations, which are adaptively scheduled by the system.

#### 4.1. Software Engineering Benefits

Virtual processors are logical entities, and can be made to correspond to the structure of the application. In contrast, shoe-horning the application structure into

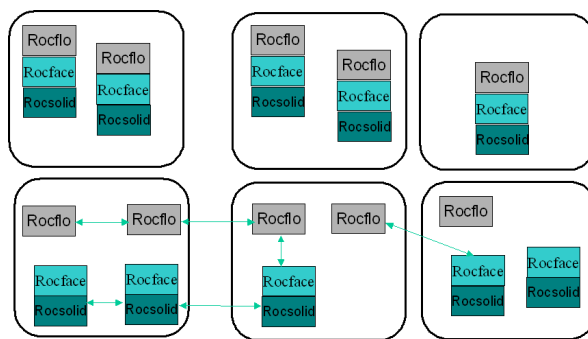


**Figure 6. PME Execution in NAMD**

physical processors leads to inelegant software.

The simplest example of this is in the number of processors used. MPI programs modeling a physical domain via structured grids often require the number of processors to be a cube (and even a power-of-two cube). With virtualization, one can decompose the data into a power-of-two cube virtual processors, yet be able to use the available number of physical processors. Attempting to do that without explicit support for virtualization leads to multiple-block codes that have to deal with messages for different blocks at various points in the code, and can spoil neat expression of the evolution of a single block when it engages in multiple phases of communication.

In software engineering terminology, such parallel software (based on physical processors) often lacks “cohesiveness”. Code and data are brought together simply because they are on the same physical processor.



**Figure 7. Rocket simulation via virtual processors**

Consider a version of the rocket simulation application consisting of two parallel modules: Rocflo (a fluid simulation of the burning gases in the Rocket interior) and Rocsolid (structural dynamics of the solid



fuel). These were derived from independently developed codes. Since the fluids and solids meshes were decomposed separately by each module, the portion of space simulated by Rocflo on processor  $i$  had no logical connection with that simulated by Rocsolid on processor  $i$ . However, an MPI implementation required them to be fused together on each processor (Figure 7 top). An AMPI implementation, on the other hand, (Figure 7 bottom) provided each module with its own set of virtual processors, and allowed for communication across them by supporting inter-communicators across multiple `MPI_COMM_WORLD`s. Among other benefits, this allows the number of pieces of Rocflo to be determined independently of that of Rocsolid, and the RTS is able to bring together (on one processor) pieces of Rocflo and Rocsolid that directly interact (because they are physically abutting, for example).

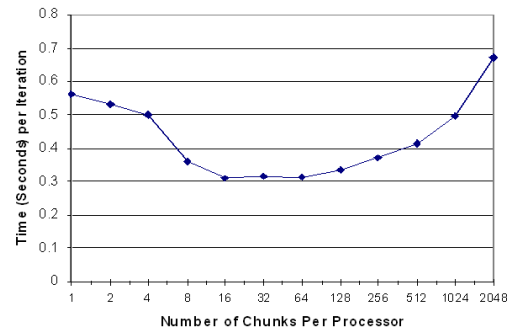
## 5. Cost of Processor Virtualization

An important question often raised is about the cost of processor virtualization. Although users may be willing to concede a certain amount of performance in return for benefit in productivity, they would like to know the extent of the performance loss. The situation is reminiscent of early days of (Fortran) compilers, when users were unwilling to switch away from assembly language programming. In fact, then as now, since programmers are highly conscious of performance issues, and they already have (by compulsion) mastered the intricacies of low level programming, they will not want to switch to a new paradigm unless assured of “as good” performance with lower effort.

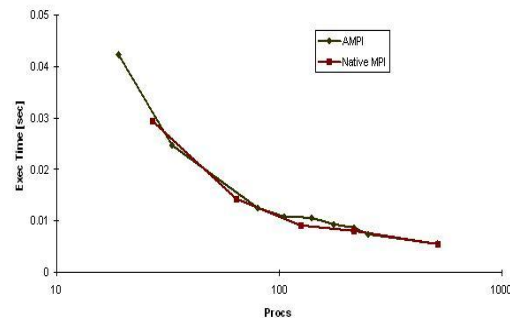
To be sure, the adaptive runtime systems enabled by processor virtualization achieve such performance enhancements as automatic dynamic load balancing. But it can be argued that expert programmers will be able to achieve such performance by programming load balancing code into their application themselves.

So, the question of overhead is still important. Luckily, in most situations, the overhead of processor virtualization is insignificant. Context switching between virtual processors requires less than a microsecond on current processors. (Recall that our virtual processors are not system level threads, or pthreads; they are user level threads). The number of messages increases with multiple VPs per processor. Messages have a software overhead of a few microseconds. So, the degree of virtualization chosen should be such that the computation per message is substantially larger than these overheads. This is clearly reasonable for most applications. For example, in a particular molecular dynamics benchmark, we used about 30,000 VPs spread over 3000 processors;

the average computation per VP per timestep was 900 microseconds, and the average computation per message was about 200 microseconds.



**Figure 8. “Overhead” of Multipartitioning in an FEM application**



**Figure 9. 7-point stencil on a 3D problem size 240<sup>3</sup> run on PSC Lemieux.**

To compare the performance of MPI and AMPI, we compared the performance of a 7-point stencil code, doing Jacobi relaxation for 3-D data, in Figure 9. AMPI achieves nearly identical performance as MPI, but it runs on any number of processors, whereas MPI requires a cubic number of processors. Additional performance data can be found in [20].

Cache performance typically improves with processor virtualization, because of its blocking effect. A study of the effect of virtualization we did with an unstructured mesh application showed (Figure 8) that performance actually improves with the degree of virtualization, and only after over 1024 VPs per physical processor does the overhead start showing its effect.

Since Charm++/AMPI are often implemented on top of native MPI, the communication costs can be expected to be higher. This is not a fundamental cost: on many

machines, our implementation uses lower level communication APIs (E.g. Elan, GM and VMI), where our performance is comparable to MPI [33]. A comparison of MPI and AMPI versions of the rocket simulation code at Illinois also showed the performance of two versions to be almost identical [11].

In some applications, virtualization leads to a large number of small messages. This can be mitigated by using a streaming library available in the Charm++ runtime that uses message combining to optimize performance.

### 5.1. Limitations and Remedies

There are situations where processor virtualization may lead to poorer performance.

When large layers of ghost cells (instead of the common 1-layer ghosts) are used, virtualization may be constrained by the memory overhead of the extra ghost cells. However, we believe it is possible to alleviate this overhead by a combination of techniques alluded to in [26].

Parallel algorithms whose costs increase with the number of processors can also limit the benefits of virtualization. Fortunately, such algorithms are rare, and they exist as components of large applications (e.g. parallel prefix, which has a complexity of  $2n + \log p$ ), and they can be allowed to run with a lower virtualization factor by concentrating data in  $p$  virtual processors.

Another source of overhead arises when processors use a large amount of remote data. If each VP uses its own memory for such data, *and if* such data overlaps significantly (i.e. multiple VPs request the same data) then both memory and communication overheads may increase. This happens, for example, in gravity computations, where each cell containing a bunch of particles is a VP, and the computation requests particles from other cells. This can be remedied by using an abstraction for requesting and caching remote data, which is implemented by a lower level library that is aware of physical processors. We are using such a library in a collaborative project in computational astronomy. NAMD also uses a similar technique in the form of proxy objects [24].

## 6. Reuse of Parallel Software Components

Reuse of parallel components can be promoted by domain-specific frameworks and allowing composition of modules written in different parallel programming paradigms.

### 6.1. Frameworks and Libraries

One method for improving productivity is to reuse a collection of techniques that are commonly needed in a particular application domain. Even though parallel applications are diverse, one can find such commonalities. For example in simulations of physical models (which constitute the dominant use of parallel computers now), one finds that only a few distinct parallel data structures are used: structured grids (arrays), unstructured meshes, spatially decomposed particles, tree structures (e.g. in multi-grid and AMR), along with a collection of linear system solvers, cover a large fraction of the application space. To improve productivity we should therefore extract the domain specific techniques into frameworks so that they don't have to be recoded for every application.

One can take two approaches to design domain specific frameworks: Vertical integration or horizontal layering. Vertical integration leads to highly specialized problem solving environments (e.g. for structural dynamics), while horizontal layering leads to a collection of capabilities that can be composed in different ways for different applications.

Our experience with horizontally layered frameworks has been quite positive. Specifically, we have developed an unstructured-mesh framework [4] that can partition a mesh and set up communication lists for a user-specified layer of ghosts. Although originally used for Finite Element computations, the framework is now used for finite volume, discontinuous Galerkin, as well as space-time meshes. Other capabilities such as collision detection or matrix-free solver interfaces are available as separate components.

However, the simplicity of problem solving environments (PSEs) suggests that horizontally layered components should be used to put together specialized PSEs, which will cut down on the development cost of PSEs themselves.

It is tempting for application developers to decide to code such capabilities themselves, since they seem relatively simple. However, once one takes the maintenance cost of software into account, and considers the fact that many focused optimizations and capabilities may have been implemented by the framework developer in the context of real applications, the advantage of frameworks becomes clear. Also, frameworks can make use of complex and tricky features of the RTS that application developers may require a significant effort to use.

### 6.2. Multiparadigm Interoperability

Although reuse of parallel components is desirable for enhancing productivity, another obstacle to such

reuse occurs because of the use of different parallel programming paradigms in different modules. One module may be written using Charm++, while another might use a DSM system, or Global Arrays [37], or the FEM framework, or MPI, or BSP, etc. By requiring that all the modules being composed into an application use the same paradigm, we give up a large opportunity for reuse.

The modules themselves may be written in different paradigms for two distinct reasons: First, a particular paradigm may be better suited for the algorithms being specified by the module. Second, it may be simply a matter of subjective choice of the programmer of that module (which might have been developed independently at an earlier time).

Concurrent composition capabilities of message-driven execution come in handy in this context, with one proviso: If all the (message-driven) paradigms share a single scheduler (See Figure 2), and possibly a common runtime support layer, then such interoperability is possible. With this in mind, we designed the Converse [27] framework, which provides (a) common capabilities such as a scheduler, user-level thread package, portable low-level communication interface, and encapsulation of other machine capabilities and (b) methods to allow the concurrent interoperation of modules written in different paradigms. In addition to providing interoperability, Converse also simplifies the task of writing runtime systems for new parallel programming paradigms.

Converse and Charm++ together now support a wide variety of programming paradigms in our infrastructure, including ARMCI, Global Arrays, Adaptive MPI, Jade (a parallel Java-like language), PVM, specific forms of DSM systems, etc. Of course, adoption of such multi-paradigm frameworks is possible only when runtime developers agree to a common standard, for which Converse is but one candidate.

## 7 Related Work

This paper focused on productivity-and-performance oriented ideas developed by the author, for pedagogical clarity. However, many of the central ideas have appeared in other research as well.

Chare Kernel, the C-based progenitor of Charm++ was developed around 1989 [22]. This system, with function calls to remote processors with objects encoded as global pointers, and a message-drive scheduler, is similar to Nexus [14]. Active Messages [42] shared message-driven execution ideas with Charm, but not processor virtualization. The Actors model [1], with its message-driven objects, is quite similar to Charm++ at its lower level, and is considered useful for specifying

and understanding the semantics of message-driven programs. However, the intellectual progenitors of our early work were the RediFlow project [31] for parallel execution of functional programs, and the Dataflow research. The basic low-level ideas in Charm++ can be considered to be macro data-flow, extended with high-level notions of automatic resource management. Other research with overlapping approaches include work on Percolation and Earth multi-threading system [21], work on HTMT and Gilgamesh projects [15], and the work on Diva [16].

Virtualization itself is not a new concept. Geoffrey Fox's 1986 textbook on parallel programming describes virtualization, for example (it was used to load balance the sharks-and-fishes application by dividing the domain into a large number of blocks, and sprinkling them across the processors randomly). The DRMS system [36] is an example of an approach based on virtualization that is closer to our work. Our approach (embodied in programming systems such as Charm++ and AMPI) can be thought of as virtualization++ : we support virtualization at the language and run-time level, and exploit it to the hilt to optimize application performance.

In the direction of interoperability, recent work on Common Component Architecture [2] is important, as it provides a method for interconnecting independently developed modules, enhancing reuse. We believe that it needs to be extended to allow processor virtualization, which is infeasible in the current form.

Several other domain specific frameworks exist that aim to raise the level of abstraction in programming. For structured grids, with possible adaptive mesh refinements, they include KeLP [13], Paramesh [35], and Chombo [8]. For computations on unstructured meshes, frameworks such as Sierra [41] exist.

Since linear system solvers arise in many current parallel applications, several libraries provide good support for them such as ESI [32] and PETSc [3]. Numerical libraries such as Ellpack [19] and Linpack [12] also help enhance reuse.

Shared memory programming models, and especially with its standardization via OpenMP, must also be considered. Via systems such as TreadMarks [30], such models are now available on distributed memory machines. However, the claim that shared memory abstraction simplifies parallel programming has not quite been substantiated. Although some programs look simpler with shared memory, others get more complex, especially if they have to deal with race conditions. It is possible that the full generality of a shared variable is unnecessary, while limited use of shared variables, in specific modes, might be productive (E.g. GA [37]).

## 8. Productivity Metrics

We have not performed any quantitative studies of improvement in productivity with Charm++/AMPI yet. We will state some anecdotal evidence instead.

Clearly, when an application requires dynamic resource management, the savings in writing code are apparent. For example, in multi-block codes, one has to write by hand how the blocks should be distributed after new refinements. All that code is in the Charm++ runtime system, and is being reused.

Virtualization also confers benefits by reusing the ability to migrate objects in different contexts. For example, once one has written a Charm++ or AMPI program with migratable objects, the runtime system can automatically carry out efficient check-pointing, support out-of-core execution, change the set of processors (shrink or expand) used by the application at runtime, vacate a machine that is about to go down or needs to be relinquished to the owner, and support fault tolerance. All of these new functionalities can be available without the user having to write new code. Of these, fault tolerance is still being worked on. Once the runtime implements this feature, it will be available for all applications without significant new application code.

Further productivity enhancements are expected when we are able to develop a “standard library for parallel programming” which will eliminate having to write code for commonly needed parallel operations.

## 9. Conclusion and Future Work

We presented a research agenda, and our progress along it, which has been explicitly aimed at improving programmer productivity and computer performance on complex parallel applications.

Processor virtualization was seen as a key to some productivity enhancements. Via this, the runtime system is empowered to carry out intelligent optimizations, including dynamic load balancing and communication optimizations, without programmer intervention. It also leads to message-driven execution, and thus to the ability to concurrently compose multiple independently developed modules effectively without losing efficiency. Separation of virtual processors from physical resources in the programmer’s mind also leads to a separation of concerns and better software engineering practices.

It can be argued that manual, application specific resource management can always do at least as well as automated techniques, in terms of performance. However, with increasing complexity of applications, and the increasing number of processors in large supercomputers,

we believe that automated techniques, culled from experience on a wide variety of applications, will be more efficient than what most actual programmers accomplish by themselves, even with a lot of effort.

Concurrent composition enables flexible reuse of parallel modules. But to make such reuse happen, reusable parallel modules must be developed. Based on the idea that a relatively small number of basic data-structures account for a large number of application components, we advocate the building of domain-specific frameworks. If each such framework provides encapsulation of a limited but useful capability, it becomes possible to compose such frameworks into vertically integrated problem solving environments.

Modules written in different parallel programming paradigms can be integrated if they share common runtime structures, and especially a message-driven scheduler (in case of virtualized or user-level thread based systems). We described our experience with Converse, an infrastructure explicitly designed to support interoperability and easy development of runtime systems for new programming paradigms.

In this paper, we kept the focus on our research in order to present a single point of view. There are many other approaches aimed at productivity. Interaction and cross-fertilization of ideas among them will lead to better systems/approaches for the future. As an example, we hope that the common component architecture effort can be extended to permit virtual-processor based formulations.

We have identified some additional future directions towards productivity. The research on intelligent adaptive runtime systems, although quite fruitful, has only picked the “low-hanging fruit”. We see potential for much more sophisticated runtime techniques, based on self-observing systems. In terms of low-level support, co-processors that can handle remote requests (beyond just puts and gets) are essential for effective deployment of composable systems. At the higher end, it seems possible to include compile-time support in a comprehensive approach aimed at productivity. The current obstacles for this include the fact the compiler-support issues that arise in this context are often considered mundane, and are not the usual issues (such as automatic parallelization) that are considered attractive by the compiler community. Telescoping languages being proposed and developed by several researchers might be able to bridge this gap.

We also see potential to increase productivity via additional language support. For instance, Jade [10], a language based on Java, provides the ability to take advantage of some of the simplifications in Java, such as the use of references instead of programmer managed stor-

age, while still having access to the features of Charm++ and the Converse runtime.

Also, we see potential in the ability to build systems of composable parallel components. The Charisma system [6] is a start in this direction, with the concept of explicit runtime support for components. A future orchestration language, which will allow the interactions among components to be defined in a scripting language, will also improve the reuse of parallel components and make the logic of parallel applications more explicit.

## 10. Acknowledgments

The research alluded to in this paper would not have been possible without generations of graduate students. For specific help in preparing this paper, I am grateful to Gengbin Zheng, Sameer Kumar, Chee Wai Lee, Chao Huang, Mark Hills and Orion Lawlor. The paper builds upon an overview of processor virtualization presented at LACSI 2002 [26]. The work which is summarized in this paper has been supported by grants from NSF, NIH and DOE over the years, including recent grants (NSF NGS 0103645, NSF ITR 0121357, NIH PHS 5 P41 RR05969-04, DOE ASCI B341494).

## References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, pages 115–124, Redondo Beach, California, August 1999.
- [3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.1, 2001.
- [4] M. Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000)*, *Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.
- [5] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.
- [6] M. A. Bhandarkar. *Charisma: A Component Architecture for Parallel Programming*. PhD thesis, Dept. of Computer Science, University of Illinois, 2002.
- [7] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwenger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.
- [8] Chombo – infrastructure for adaptive mesh refinement. <http://seesar.lbl.gov/anag/chombo/>.
- [9] R. Cytron, D. J. Kuck, and A. V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. *Computer Physics Communications*, 37(1–3):39–48, 1985.
- [10] J. DeSouza and L. V. Kalé. Jade: A parallel message-driven java. In *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.
- [11] E. deStruler, J. Hoeflinger, L. V. Kale, and M. Bhandarkar. A New Approach to Software Integration Frameworks for Multi-physics Simulation Codes. In *Proceedings of IFIP TC2/WG2.5 Working Conference on Architecture of Scientific Software, Ottawa, Canada*, pages 87–104, October 2000.
- [12] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, 1979.
- [13] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible communication mechanisms for dynamic structured applications. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 203–215, 1996.
- [14] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, May 1994.
- [15] H. Gao, K. Theobald, A. Marquez, and T. Sterling. The gmt program execution model, 1997.
- [16] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping irregular applications to DIVA, A PIM-based data-intensive architecture. pages ??–??, 1999.
- [17] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine independent parallel programming in fortran-d. In J. Salz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier Science Publishers B.V., 1992.
- [18] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for fortran-d on mimd distributed memory machines. In *Proceedings of Supercomputing 1991*, Nov. 1991.
- [19] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, K. Y. Wang, and S. Weerawarana. //ellpack: a numerical simulation programming environment for parallel mimd machines. In *Proceedings of the 4th international conference on Supercomputing*, pages 96–107. ACM Press, 1990.
- [20] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

- [21] H. Hum. A design study of the earth multiprocessor, 1995.
- [22] L. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.
- [23] L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.
- [24] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.
- [25] L. V. Kale. Application oriented and computer science centered HPCC research. In *Proceedings of Developing a CS Agenda for High-Performance Computing*, pages 98–105, March 1994. Position Paper.
- [26] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.
- [27] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.
- [28] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
- [29] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop (ICCS'03)*, Melbourne, Australia, June 2003.
- [30] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.
- [31] R. Keller, F. Lin, and J. Tanaka. Rediflow Multiprocessing. *Digest of Papers COMPCON, Spring'84*, pages 410–417, February 1984.
- [32] M. G. Knepley and et al. Solvers as operators - proposal for the esi solver interface.
- [33] L. V. Kale and Sameer Kumar and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS 2003*, 2003.
- [34] O. S. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.
- [35] P. MacNeice, K. M. Olson, C. Mobarrey, R. de Fainchtein, and C. Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.
- [36] J. E. Moreira and V.K.Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303, 1997.
- [37] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. In *Journal of Supercomputing*, volume 10, pages 169–189, 1996.
- [38] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):829–842, dec 1986.
- [39] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.
- [40] V. A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference (5th DMCC'90)*, volume II, Architecture Software Tools, and Other General Issues, pages 994–999, Charleston, SC, Apr. 1990. IEEE.
- [41] L. M. Taylor. Sierra - a software framework for developing massively parallel, adaptive, multi-physics, finite element codes. In *Presentation at the International conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, USA, June 1999.
- [42] T. von Eicken, D. Culler, S. Goldstein, and K. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

# Afternoon Keynote

Building High Performance Scientific  
Applications for Parallel and  
Distributed Systems using a Software  
Component Architecture.

Dennis Gannon  
University of Indiana

# Introducing the “Application Kernel Matrix”

Brad Chamberlain

John Feo

David Mizell

Cray, Inc.

{bradc, feo, dmizell}@cray.com

## Abstract

*The goal of this project is to provide a publicly-available web site at which programming languages designed for high performance computing can be compared and discussed. A small number of important application kernels will be specified, and implementations of any of them in programming languages relevant to HPC can be submitted. Information will be collected and displayed regarding the performance time on various computer systems, as well as anecdotal information about how much time it took to implement a given kernel in a given language.*

## 1. Introduction

The context of this work is Cray’s “Cascade” project. This is an effort to design and prototype a multi-petaflops supercomputer by 2010. Cascade, along with efforts by IBM and Sun with the same general objective, is funded by DARPA’s High Productivity Computer Systems program. Note that DARPA has emphasized productivity, not just performance, in the program’s title. The government is calling for the next generation of supercomputers to be substantially easier to program, debug and tune for performance.

Cray’s intended contributions to supercomputer programming productivity are threefold: first, we are designing a multithreaded, single-address-space system, which we expect to be easier to program (and to compile for) than the current generation of distributed-memory architectures. Second, we are designing Chapel [2], a programming language aimed at spanning the gap between programming expressiveness and runtime performance. Third, we are offering

the “application kernel matrix,” the subject of this paper, as a repository of comparative information relevant to the HPC programming language research community.

## 2. Purpose of the Application Kernel Matrix

The intent of the application kernel matrix (AKM) is to be a publicly-accessible repository of application kernel examples, each coded in a variety of HPC-oriented languages. We conceptualize it as a three-dimensional matrix, as described below:

- The rows are application kernels. We have selected an initial set of ten (see Section 3). They will be derived from important HPC applications, and selected to represent a variety of interesting parallel programming problems. We will provide a sequential, probably Fortran 95, implementation of each. We expect the F95 implementations to be at most a few hundred lines long – long enough to be nontrivial, but not so long as to discourage participation from the always-busy research community. Standard data sets will be provided with each kernel, so that performance results (which will exclude file I/O) will be comparable on the basis of the same inputs.
- The columns are programming languages. We will initially populate the matrix with columns for the widely-used HPC languages, such as Fortran/MPI [10, 11], Co-Array Fortran [12], OpenMP [5], UPC [3], etc., and we look for programming language researchers to add more columns for the new languages they are developing and experimenting with: Titanium [15], A-ZPL [4, 14], Chapel, etc.
- The third-dimension planes represent high-performance computer systems. We envision each  $(i, j, 0)$  entry to be a



“portable,” or “generic” implementation of kernel  $i$  in language  $j$ . Each  $(i,j,k)$  entry for  $k > 0$  would be an implementation of kernel  $i$  in language  $j$ , performance-tuned for computer system  $k$ . System vendors would be encouraged to add planes to the matrix, containing kernel implementations in the language or languages of interest to them, tuned for high performance on their hardware.

There will be a monitored submission process to be used by researchers and vendors wishing to add entries to the AKM. We will require each person or team submitting an entry to the AKM to provide information about their submittal, that we anticipate would be of interest to the HPC programming language research community. First, the source code of the implemented kernel will be posted on the web site. Second, each submitter will provide performance data with their submission, specifying the system on which the program was executed and which standard data set served as input. In addition, each submitting person or team will be asked to keep an activity log and fill out a questionnaire. They will use the activity log to record the time the programmer spent implementing kernel  $i$  in language  $j$ , broken down into these categories: studying the problem, designing the code, editing the code, debugging, and tuning for performance. The questionnaire will elicit information about each programmer: overall years of experience, amount of experience with this language, amount of experience performance-tuning for this hardware, etc. We make no claim that this information we collect in the AKM will have any statistical validity in terms of the inherent productivity advantages of one programming language or computer system over another. This data will necessarily be anecdotal. A separate group of researchers funded by DARPA, the HPCS Productivity Team, is expected to carry out carefully-designed, multiple-subject experiments in programmer productivity. The most we could hope to provide, by collecting information about time taken and programmer expertise in the AKM, is an early look at some interesting anomaly which the Productivity Team might choose to explore with a controlled experiment. We believe that even the data in the AKM that is less subject to issues of statistical validity, such as how many source lines of code of language  $j$  it took to optimize kernel  $i$  for system  $k$ , will still be debatable. However, we also believe that it will be useful for the

community to have these comparative examples available in one place. We see the AKM as an extension of the kind of comparative study done in [6], which can now be sited on the Internet and expanded as participants submit more entries.

### 3. Initial Kernel Selections

We wanted to choose application kernels which are (1) relevant to important HPC applications, (2) interesting and somewhat challenging parallel programming problems, and (3) different from each other in terms of the parallel programming challenges they pose. Our initial selections are the following ten kernels:

1. FT, the multidimensional FFT from the NAS parallel benchmarks [1]
2. MG, the multigrid kernel from the NAS parallel benchmarks
3. CG, the conjugate gradient kernel from the NAS parallel benchmarks
4. the Smith-Waterman bioinformatics code, a dynamic programming algorithm [13]
5. SWEEP3D, the ASCI benchmark [8, 9]
6. a triangular backsolve for a dense matrix
7. a triangular backsolve for a sparse matrix
8. finding the connected components of a graph
9. a first-order linear recurrence: given a vector of floating point values, find the maximum value, index of the first occurrence of the maximum, and prefix sum of the vector
10. a branch-and-bound problem: the chip “floorplan” optimization problem from [7]

### 4. Implementation Plans

Our plan is to have the initial version of the AKM available on the Internet by the end of March 2004. Figure 1 is a sketch of how the matrix might be represented so as to enable site visitors to browse the submitted solutions.

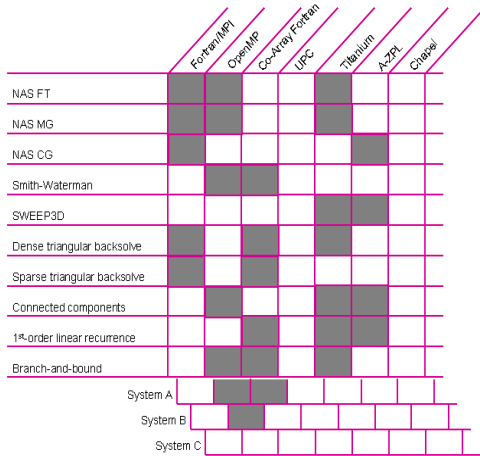


Figure 1: anticipated web site representation of the Application Kernel Matrix

The darkened boxes would represent kernel implementations that had been successfully submitted. Clicking on one of them would take the visitor to a page that provided links to the source code, performance data and information about the programmer's prior experience and the programming time required. Elsewhere on the web site, the visitor would be able to download the programmer's questionnaire and programmer's log forms, or participate in the monitored e-mail discussion forum.

## 5. Envisioned Usage Modes

We anticipate that designers of HPC programming languages and vendors of HPC systems will be motivated to submit entries to the AKM. Language designers will want to show that their language covers a broad spectrum of systems (portability) and kernels (generality), and that it performs well compared to standard languages. Vendors will want to show that they support a wide range of languages and that users can obtain good performance without requiring many changes from the reference implementation of a given kernel.

People browsing the AKM may prefer to do so across any of its dimensions:

- An MPI programmer who needs to port a code to a new system might study the examples in order to learn how MPI codes are usually modified for performance on that system, and what kind of performance to expect.

- A lab director whose users program in MPI, OpenMP, CAF and UPC might look for a system that supports all of these languages reasonably well without requiring substantial modifications from the reference implementations.
- A programmer who is dissatisfied with the current generation of languages might look at each of the implementations of the kernel most similar to her own work, to see which of them seems easiest to learn and use without sacrificing much performance.

Note that because, as in the second scenario, the AKM may be used to compare computer systems, it would be appropriate for a disinterested party, such as a government or academic lab, to assume responsibility for managing the AKM once Cray has completed the initial implementation.

## 6. Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003.

## 7. References:

- [1] Bailey, David H. *et al.*, "The NAS Parallel Benchmarks," in *The International Journal of Supercomputer Applications*, Vol. 5 No. 3, Fall 1991, pp. 63-73.
- [2] Callahan, David, *et al.*, "The Cascade High Productivity Language" to appear in *the 9<sup>th</sup> International Workshop on High-Level Programming Models and Supportive Environments*, April 2004.
- [3] Carlson, William W. *et al.*, "Introduction to UPC and language specification," Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.
- [4] Chamberlain, Bradford L., *The Design and Implementation of a Region-Based Parallel Language*, PhD thesis, University of Washington, November 2001.
- [5] Dagum, Leonardo and Menon, Ramesh, "OpenMP: An Industry-Standard API for Shared-Memory Programming" in *IEEE Computational Science & Engineering*, Vol. 5, No. 1, January/March 1998.

- [6] Feo, John, ed., *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, North-Holland, New York, 1992.
- [7] Foster, Ian, *Designing and Building Parallel Programs*, Addison-Wesley, Boston, 1995.
- [8][http://www.llnl.gov/asci\\_benchmarks/asci/limited/sweep3d/asci\\_sweep3d.html](http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html)
- [9] Koch, K. R. *et al.*, "Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine" in *Transactions of the American Nuclear Society*, Vol. 65, pp. 198–9, 1992.
- [10] Message Passing Interface Forum, "MPI: A message passing interface standard," in *International Journal of Supercomputing Applications*, Vol. 8 Nos. 3–4, pp. 169–416, 1994.
- [11] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997.
- [12] Numrich, R. W and Reid, J. K., "Co-Array Fortran for parallel programming," Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.
- [13] Smith, T. F. and Waterman, M. S., "Identification of Common Molecular Subsequences," *Journal of Molecular Biology* (1981) **147**:195-197.
- [14] Snyder, Lawrence, *Programming Guide to ZPL*, MIT Press, Cambridge, MA, USA, 1999.
- [15] Yelick, Kathy *et al.*, "Titanium: A High-Performance Java Dialect" in *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

# Managing Complexity in Modern High End Scientific Computing through Component-Based Software Engineering

David E. Bernholdt  
Oak Ridge National Laboratory  
P. O. Box 2008, MS 6016, Oak Ridge, TN 37831-6016  
bernholdtde@ornl.gov

Robert C. Armstrong, and Benjamin A. Allan  
Sandia National Laboratories  
7011 East Avenue, MS 9915, Livermore CA, 94550-0969  
{rob,baallan}@sandia.gov

## Abstract

*The ever-increasing complexity of modern high-performance scientific simulation software presents a tremendous challenge to the development and use of this type of software, with significant impacts on productivity. Component-based software engineering is a means of addressing complexity that has been developed primarily in response to the needs of business and related software environments, but which has not yet had a significant impact on high-end computing. In this paper, we present the Common Component Architecture (CCA) as a component model designed to meet the special needs of high-performance scientific computing, focusing on how the CCA addresses issues of complexity. Unique among component architectures is the technique presented here by which a CCA component can act as a container to encapsulate and control other components without itself having to implement the functionality of a framework.*

## 1. Introduction

Complexity of software is one of the greatest single challenges facing modern high performance scientific computing. It comes from several sources. As hardware manufacturers strive to provide ever faster systems, they become more complex, with deep non-uniform memory access hierarchies, CPU hierarchies (clusters of SMPs and similar models), widely varying architectures and capabilities for both interconnects and I/O systems. These “features” are almost always exposed to the programmer, and in order to achieve maximum performance, the programmer must take

responsibility for tuning their code to each platform of interest. The second source of software complexity is the scientific problem being addressed. As computers have become more capable, and together with advances in software been able to deliver interesting and useful results through simulation, researchers demand more. So scientific simulation software expands to encompass larger problems, higher fidelity simulations, and the coupling of simulations across multiple time and length scales. In each case the complexity of the software must increase to answer the new challenges.

Studies have shown that the human mind is able to handle a limited amount of complexity [7, 13, 18], so that at some point the complexity of HPC software will outstrip the ability of programmers to deal with it and the pace of software development will slow. Assembling teams of programmers to create large-scale codes is a response to the size and complexity of the software and the breadth of knowledge required to successfully create it. However adding more workers, while necessary to deal with complexity, is not sufficient. The coordination required between workers imposes significant overheads that can limit the software scalability for the same reasons that Fred Brooks famously observed that “adding programmers to a late project only makes it later” [11].

Facing similar problems of software complexity, other communities, most notably the business and internet software communities, have invested heavily in *component-based software engineering* (CBSE) as a means to help address these issues. CBSE is based around the idea of software components, or units of programmatic functionality, that can be composed together to build an application. Components effectively break the complexity into people-sized chunks. Except to their developers, components are

treated as black boxes which interact with other components and the rest of the external environment only through well-defined interfaces. In this way, components encapsulate complexity with which users of the component need not concern themselves. Users create applications by composing components together in a “plug and play” fashion (which is very amenable to visual programming techniques) based on their interfaces. This provides a new level of abstraction for most software development, and thus a means of managing the complexity at a higher level.

The CBSE approach also provides a natural means to help control the complexity that arises due to multi-person interactions in team-developed software. Since interfaces are the key to component interoperability, the initial design of a component-based application can focus on the overall architecture and “componentization” of the problem and on defining the interfaces through which the components interact. With this task completed, individuals or small groups can then split off and focus on developing components conforming to the specifications without the need to interact with the creators of other components.

Component-based software engineering may seem like a natural approach to the creation of complex scientific software, and can be thought of as an extension of widely used approaches, such as the creation of software libraries, and object-oriented programming. But CBSE has not yet made significant inroads into HPC software, primarily because the “commodity” component models currently available, such as CORBA [14, 15, 23], COM/DCOM [19, 22], and Enterprise JavaBeans [20] were developed primarily for the business/internet software communities and do not address the needs of HPC scientific software very well [6]. Most commodity component environments have been designed primarily for distributed computing, and do not recognize or support the need for local performance and the use of tightly-coupled parallel computing as being more important than distributed computing. In scientific computing, it is common to have large codes which evolve over the course of many years, or even decades. Therefore, the ease with which “legacy” codebases can be incorporated into a component-based environment, and the cost of doing so, are also important considerations. Additional considerations include support for languages, data types, and computing platforms important to high-performance scientific computing.

The Common Component Architecture (CCA) [6, 12] was conceived in 1998 as a grass-roots effort to address the need of the scientific community for approaches to address the complexity of scientific software development and to facilitate and promote the creation of reusable, interoperable software for scientific high performance computing [3]. In this paper we describe features of the Common Component Architecture which simplify the management of soft-

ware complexity.

## 2. The Common Component Architecture

The Common Component Architecture is the nucleus of an extensive research and development program in the Dept. of Energy and academia. On the research side, the effort is focused on understanding how best to utilize and implement component-based software engineering practices in the high-performance scientific computing area. In addition to the definition of the CCA specification itself, the development effort is aimed at creating practical reference implementations conforming to the specification, helping scientific software developers use them to create CCA-compliant software, and, ultimately, at creating a rich “marketplace” of scientific components from which new component-based applications will be built. Space constraints require that we limit our presentation here to those aspects of the CCA which bear directly on dealing with complexity: a description of the basic elements of the CCA’s component model, and the mechanism by which components are created, formed into applications, and executed. However, a comprehensive overview will be published soon [10] and tutorials are already available [1].

The specification of the Common Component Architecture defines the rights, responsibilities and the relationships between the various elements of the model. Briefly, these are as follows:

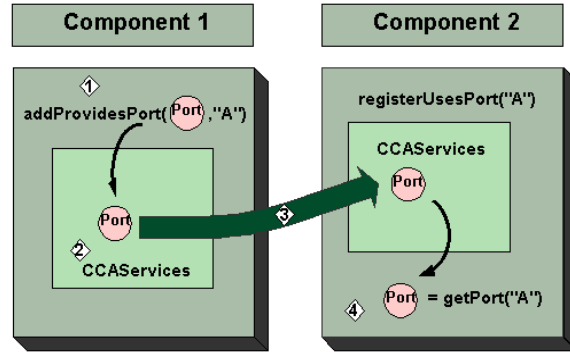
- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces to other components.
- *Ports* are interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of sub-routines, or a module in a language such as Fortran 90. Components may provide ports, meaning they implement the functionality expressed in the port (called *provides ports*), or they may use ports, meaning they make calls on that port provided by another component (called *uses ports*).
- The *framework* holds CCA components as they are assembled into applications and executed. The framework is responsible for connecting uses and provides ports without exposing the components’ implementation details. It also provides a small set of standard services, defined by the CCA specification, which are available to all components. The `BuilderService` and `AbstractFramework` ports are two of these

standard services which are both central and novel with respect to the way the CCA deals with complexity.

The CCA employs a minimalist design philosophy to simplify the task of incorporating pre-existing HPC software into the CCA environment. CCA components interact with the CCA framework via the *Services* interface, which provides the means for components to register the ports they provide and use (`addProvidesPort()`, `registerUsesPort()`), and to obtain “handles” to ports so that they can be used (`getPort()`). This makes it possible for the framework to effectively and efficiently mediate component connections. To be “CCA compliant”, components are required to implement the `gov.cca.Component` class, which includes just one method: `setServices()`. This method is invoked by the framework immediately after the component is instantiated, passing in a CCA *Services* object (later referred to as `svc`). The primary purpose of `setServices()` is for the component to tell the framework what ports it provides and uses.

The uses/provides design pattern for ports and the framework’s role in mediating the connection of ports is also important in the CCA’s ability to transparently support both local high-performance and distributed computing models. Prior to actually invoking a method on another port, the component uses the `svc.getPort()` method to obtain a handle to the port. In the distributed computing case, the handle would be a pointer to a local proxy for the provides port created by the framework, and the framework is responsible for conveying the remote method invocations to the actual provider, including marshaling and unmarshaling arguments. In the local high-performance (also referred to as “direct connect” or “in-process”) case, the framework typically loads components into separate *namespaces* within the address space of a single process, so in this case the handle can be a pointer to the virtual function dispatch table for the providing port. In this case the method invocations take place directly without intervention by the framework and without CCA-imposed overheads beyond the virtualization of the function call (common in object-oriented languages anyway). Since the caller and callee share the same address space, all arguments are commonly passed by reference without the loss of performance indirection entails. Measurements show that the CCA-imposed overhead on calls between components in the direct connect case is minimal, and does not impact performance relative to traditional (non-component) programs [9, 21]. In some cases, the Bable language interoperability tool [8] may need to translate datatypes between languages, but for most scientific computing these overheads can be avoided.

In the high-performance *parallel* context, the CCA’s model is that of many of the local high-performance “in-process” component assemblies running in parallel across



**Figure 1. A schematic representation of the sequence of interactions between the component and framework via the CCA Services object that allow ports to be connected and used.**

many processors. Components in each process operate via the usual CCA mechanisms, while the parallel instances of a given component can utilize whatever parallel communications model they prefer, without any CCA-imposed overheads. Both single-component/multiple-data and multiple-component/multiple data paradigms are supported, analogous to SPMD and MPMD programs without any CCA-imposed performance overheads [21].

Figure 1 illustrates more specifically the sequence of interactions between the component and framework via the CCA *Services* object that allow ports to be connected and used. In step 1, Component 1 calls `svc.addProvidesPort()` (and Component 2 calls `svc.registerUsesPort()`) to express their intent. The CCA *Services* object caches the information about the port it got from `addProvidesPort()` (step 2). In the third step, the framework connects the uses port to the provides port, and the framework copies information about the provides port over to the user’s (component 2’s) CCA *Services* object. Finally, when Component 2 wants to invoke a method on the port provided by Component 1, it issues a `svc.getPort()` call to obtain a handle for the port. Not shown in the diagram is the `svc.releasePort()` call, informs the framework that the caller is (temporarily) done using the port. A port may be used only *after* a `getPort()` call is made for it, and before its companion `releasePort()` call; `getPort()` and `releasePort()` can be used repeatedly throughout the body of the component. This is considered better CCA programming practice than acquiring handles to all relevant ports once at the beginning of the component execution and releasing them only at the end, because it allows the use of a more dynamic component programming model, through the `BuilderService` port.

In “normal” use of the CCA model, steps 1–3 would take place during the “assembly” phase of the applications. Specifically, steps 1 and 2 would take place with the component’s `setServices()`, invoked by the framework when the component is instantiated, and step 3 would take place as the component instructs the framework how to connect the uses and provides ports for the application. Step 4 would take place during execution of the component’s code. Finally, when not within a `getPort()/releasePort()` block, connections between uses and provides ports may be broken, and components may be destroyed.

In general, components cannot use ports on other components during the assembly phase (i.e. within the component’s `setServices()` routine) because there is no guarantee that the components providing those ports have been instantiated and connected to this component’s “uses port”. There is one exception, however. As a reuse of concepts, the CCA also casts framework services as ports, and such services *are* available to components as soon as they have been instantiated.

While this explanation has portrayed the phases of the lifecycle as “collective”, with the entire application being assembled, executed, and then disassembled, this is not necessarily the case. Through the `BuilderService` framework service port, applications can have extremely dynamic behavior. The motivating example for the development of `BuilderService` was the desire to be able to swap out one numerical solver for another during a simulation because, for example, the solution might be moving into a region where another solver would provide better performance or numerical quality [16]. `BuilderService`, together with the `AbstractFramework` service also allow hierarchies of components to be created, encapsulating many components and treating them as one.

### 3. Application Complexity in the CCA

While the CCA does a very good job of encapsulating the complexity of thousands of lines of source code into black-box components, the model, as described in Section 2, has only one level. Modern HPC scientific applications often grow extremely large and involve the coupling of simulations at different time or length-scales. Eventually even componentized versions of such applications become too complex for software developers and users to deal with all at once.

As an example, consider the study of a reaction-diffusion simulation under varying numerical and geometric parameters, where the user may be interested in performance, convergence, and efficiency. Figure 2 shows the CCA “wiring diagram” for a modestly complex “production quality” application of this type, developed by Jaideep Ray, Sophia Lefantzi, and their co-workers in the Center for React-

ing Flow Simulation lead by Sandia National Laboratory [2, 17].

This figure, derived from a screen capture of the visual programming interface currently available with the CCA framework [4, 5], shows the numerous components as dark boxes decorated with smaller boxes representing the provides ports (left side of each component) and uses ports (right side). Lines show connections between uses and provides ports. The component layout is very cluttered and quickly fills most of the screen. As is typical in component-based applications, multiple components are used to implement the various high-level elements of the application. In this case, three components together provide the reaction kinetics functionality (heavy oval) and a second group of four components that handle the diffusion equation (heavy rectangle). These components dominate the work area, but are of little interest to the planned study, because they will not be changed in any way.

This example makes the visual case for the need to be able to group components to further hide complexity. In this case, our purposes would be well served if we could group the three chemistry components together into one a single `ChemSolver`, and the four diffusion components into a single `DiffusionIntegrator` component. This would simplify the visual programming picture, making it easier to work with the remaining components, which are of interest in the planned study. It may also be of interest to export these groupings as components in their own right, available for use in other applications. Flexibility of the mechanism is important too: the target of the next study could be a comparison of the numerical and performance characteristics of the `CvodeSolver` against other equivalent solvers, looking for a possible replacement. In this case, we would want to see all of the structure for the chemistry part of the application, but could black-box other component groupings.

### 4. Reusing Component Concepts for Aggregation and Scalability

For a peer object model like CCA, there is really only one option to deal with the need to provide multiple levels of encapsulation: a peer container object for networks of components. Although there are notable exceptions (e.g. Visual Basic™), it was considered a best practice to make each container itself a component and therefore achieve a self-similar answer to component aggregation.

Because the `BuilderService` port exports framework functionality to the user, it serves two vital functions that normally are under framework control: containment and composition. Containment allows an entire component composition (network of connected components) to be black-boxed as a single component. Dynamic composition allows changes in the way a component network is

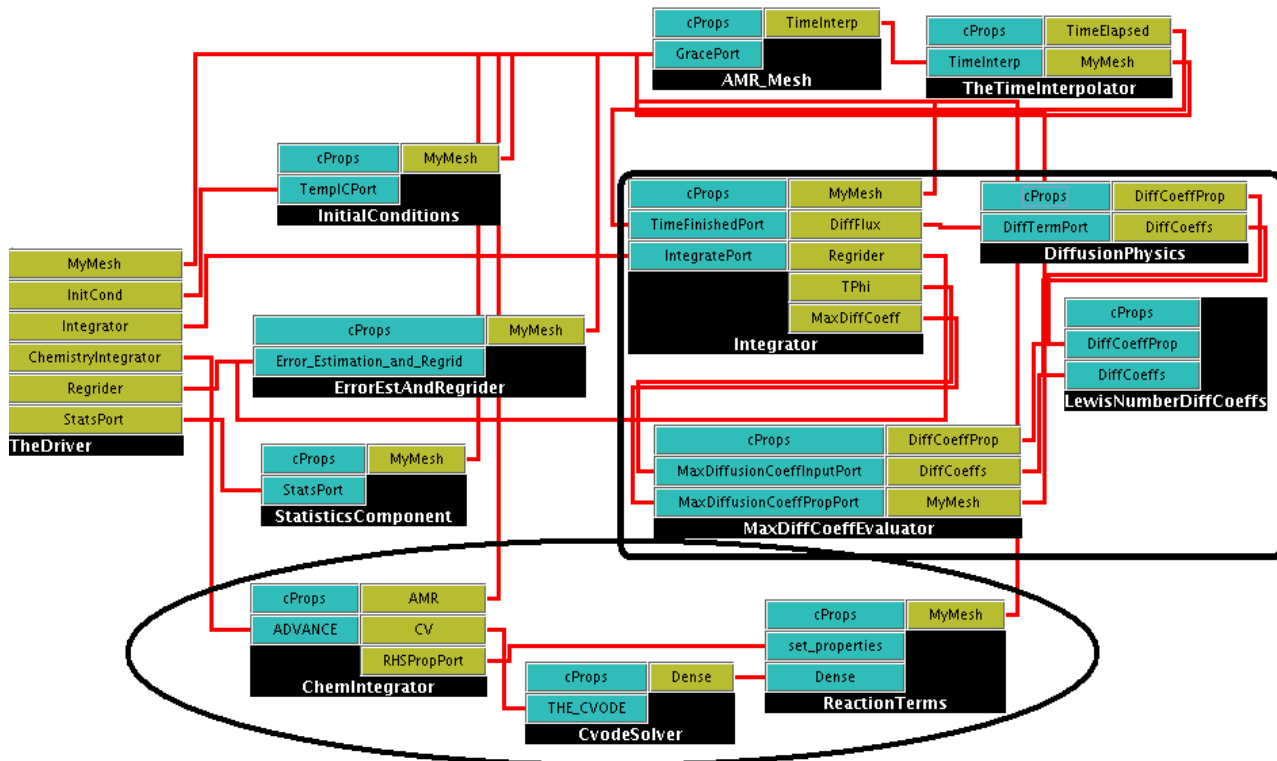


Figure 2. Assembly diagram of a reaction-diffusion simulation using an implicit/explicit integration scheme on an adaptively refined grid. The heavy lines highlight the components related to the reaction kinetics (oval) and the diffusion equation (rectangle).

connected at any time during the execution of the program.

#### 4.1. BuilderService: Component Containers in a High Performance Setting

The BuilderService interface allows the high-performance computational scientist the ability to take on the role of the framework programmer. BuilderService is a standard framework service port and the user requests this interface through the usual CCA mechanism of `svc.registerUsesPort()` and `svc.getPort()`. The entire interface for BuilderService can be found on the web at <http://www.cca-forum.org/specification/>. Figures 3–6 show an entire scenario for containing a more complicated component network within a controlling component that uses BuilderService.

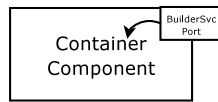
The idealized scenario of these figures is to encapsulate a two-component network, but proxy an unconnected provides port and an unconnected uses port on the outside of the container making them available for connections by a user. In the figures, a component called Container Component is located and instantiated in the usual way of Section

2. Container Component requests a BuilderService port (Fig. 3). Because BuilderService is a CCA service port provided by the framework, it can be retrieved immediately, during its initial `setServices()` call. During the same call, two components are instantiated, and then connected together through use of the BuilderService interface (Fig. 4). Next the Container Component connects the component network to itself by exporting the same type of ports on itself (Fig. 5) and connecting them to the contained network. Finally, the single Container Component presents the two proxied ports encapsulating, in this case a two component network (Fig. 6). The Container Component here does not have any functionality other than as a program to create an interior encapsulated network of components, re-exporting provides ports and proxying uses ports that are left unconnected.

It is worth noting that once all of the connections to the containing component are made, there is no further involvement by the container in the execution of the program. All of the encapsulated components are dealt with directly. This means that taking advantage of the CCA containment mechanism inflicts no performance penalty on the user's application.



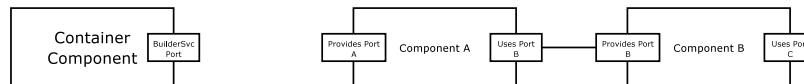
**Container Component advertises and gets a BuilderService port.**



```
// Request a BuilderService port and retrieve it (svc is the Services object):
svc.registerUsesPort("gov.cca.BuilderService", "myBuilderService",properties)
gov::cca::BuilderService bSvc = getPort("myBuilderService");
```

**Figure 3. CCA Containment Mechanism Using BuilderService: Step 1: Create the container component.**

**Container creates two components and connects them:**



```
// Create two components and connect them with BuilderService:
gov::cca::ComponentID cmptA = bSvc.createInstance("AType", "ComponentA",properties);
gov::cca::ComponentID cmptB = bSvc.createInstance("BType", "ComponentB",properties);
bSvc.connect("componentA","UsesPortB","ComponentB","ProvidesPortB");
```

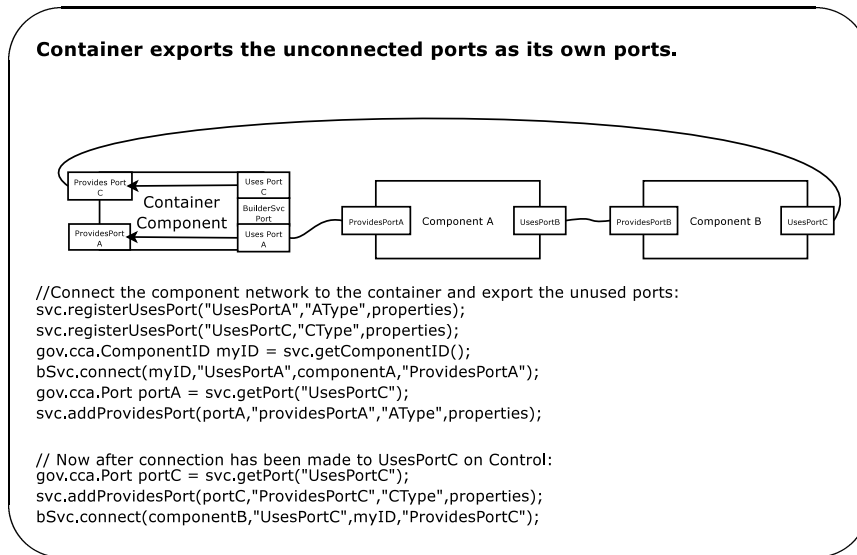
**Figure 4. CCA Containment Mechanism Using BuilderService: Step 2: Container component composes a network of components.**

## 4.2. Using CCA from Main: AbstractFramework

Component-oriented programming implies that there is an overarching framework that instantiates, manipulates, and destroys components and otherwise manages components on the behalf of the user. The downside is that the user is unable to write or control the `main()` program. In most cases a well-written CCA-compliant framework will cover what 90% of the users would like to do. However since high performance computing involves ever more sophisticated hardware, and hence runtime environments, some setup may be needed ahead of the framework to prepare it for queuing systems, message passing layers, or other nonstandard facilities that could not be anticipated by the framework developers. The CCA's answer to this requirement is another interface called `AbstractFramework`. This interface is not a `gov.cca.Port` but one that allows an instantiation of a CCA framework from a library. Beyond creation and destruction there is only

one method on the interface: `getServices()` which returns a `Services` object identical to the one received in the `setServices` call by a normal component. The `getServices()` call effectively creates an image of the main program inside the framework allowing `addProvidesPort()` and `registerUsesPort()` calls from the main program the same as any other component. From then on the `BuilderService` interface can be requested and the process can proceed as before. Listing 1 shows an example of this in Python.

`BuilderService` and `AbstractFramework` interfaces are considered by the CCA working group to be "advanced" behavior, and would probably only be undertaken by a user that is already well versed in CCA component semantics and behavior. In essence the user is taking over the role usually occupied by a CCA compliant framework. It is important that the user of the `BuilderService` containment be cognizant of what CCA components are entitled to expect and to respect the component life cycle



**Figure 5. CCA Containment Mechanism Using BuilderService: Step 3: Container component connects itself to unconnected ports.**

that the component writers depend on.

Mentioned previously, another important function of `BuilderService` is the automation of the componentized programs at the component level. For example, an equation solver component used in the solution of a PDE might work fastest with an **LU** preconditioner for some number of time steps, but later in the calculation might require a multigrid method. A component that monitors the convergence behavior (possibly the condition number) could disconnect the **LU** component and plug in the multigrid method as the need arises. `BuilderService` allows high-performance components to be programmed dynamically, as any other object in the calculation [16].

### 4.3. The Simplified Reaction-Diffusion Application

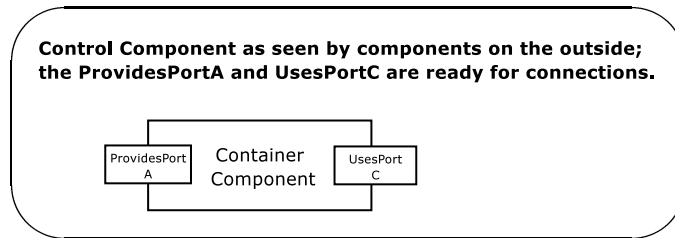
The mechanisms described in this section can be applied to our reaction-diffusion simulation example, producing the result shown in Figure 7. The core chemistry and physics of the problem are now encapsulated within the black-box `ChemSolver` and `DiffusionIntegrator` components. The `ErrEstAndRegrid` and `TimeInterpolator` components are readily accessible, and there is enough screen real estate available to easily manipulate the components of interest as needed.

## 5. Conclusions

A certain amount of complexity is unavoidable in high-performance scientific computing due to the complexity of the problems being solved. The Common Component Architecture is design specifically to meet the needs of this community, including the need to better manage complexity.

Because CCA's target developers are computational scientists who wish to focus not on software development, but on their scientific simulations, the tools and concepts of the CCA must be both simple to grasp and scalable to the problems of interest to the computational scientists. Reuse of concepts is an important means for the CCA to achieve this necessary simplicity – in other words, to reduce the complexity inherent in the CCA itself. An example is the use of the uses/provides concept for ports to transparently enable both high-performance local component assembly and distributed computing. In the parallel computing case, the CCA's approach allows the programmer to reuse the tools and techniques with which they are most comfortable for parallel programming, rather than imposing a new model or tools on them.

The CCA also deals with software complexity directly. At the first level, components provide black-box encapsulation of complex pieces of source code so that the user of the component (as opposed to its developer) need not be concerned about its internals. Through the `BuilderService`



**Figure 6. CCA Containment Mechanism Using BuilderService: Step 4: Finally, only proxied ports are available for further connections.**

**Listing 1. Abstract Framework example in Python**

```

#!/usr/bin/python
import ccaffeine.AbstractFramework # load the framework
# Framework-specific portion, in this case Ccaffeine:
a = ccaffeine.AbstractFramework.AbstractFramework() # create it
# initialize telling the framework what components we will use and
# where they are located.
args = "--path /home/rob/cca/lib/components"
a.initialize(args)
# From here on, this main program is using only standard CCA,
# nothing implementation specific.
# We create this main python program as a component in the framework by
# getting gov.cca.Services:
svc = a.getServices("main", "MainComponent", properties);
myid = svc.getComponentID(); # this is our ComponentID
svc.registerUsesPort("bs", "gov.cca.BuilderService", properties)
port = svc.getPort("bs")
import gov.cca.ports.BuilderService
bs = gov.cca.ports.BuilderService.BuilderService(port)
# From here on everything is the same as if it were
# a "normal" CCA component.
  
```

vice and AbstractFramework interfaces, the CCA also provides an approach to hierarchically encapsulate a network of components as a single component, a design pattern which is unique (as far as we know) in the world of components.

In this way, application developers can manage the complexity presented by their CCA-based applications in a flexible and general fashion. It is hoped that by introducing fewer new concepts, it will be easier to employ BuilderService in applications, making even large-scale multi-physics applications more manageable. An important side effect of this approach is that CCA-compliant frameworks need to add little to support this style of containment because most of the existing infrastructure can be reused. Because there are numerous CCA-

compliant frameworks specialized in various areas of high end computing, a beneficial artifact of the BuilderService/AbstractFramework approach is that disparate frameworks can be linked together using only CCA ports.

**6. Acknowledgments**

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom are gratefully acknowledged. We further acknowledge our collaborators outside the CCA Forum and the early adopters of the CCA for the important contributions they have made both to our understanding of CBSE in the high-performance scientific computing context, and to making the CCA a practical and usable environment. We

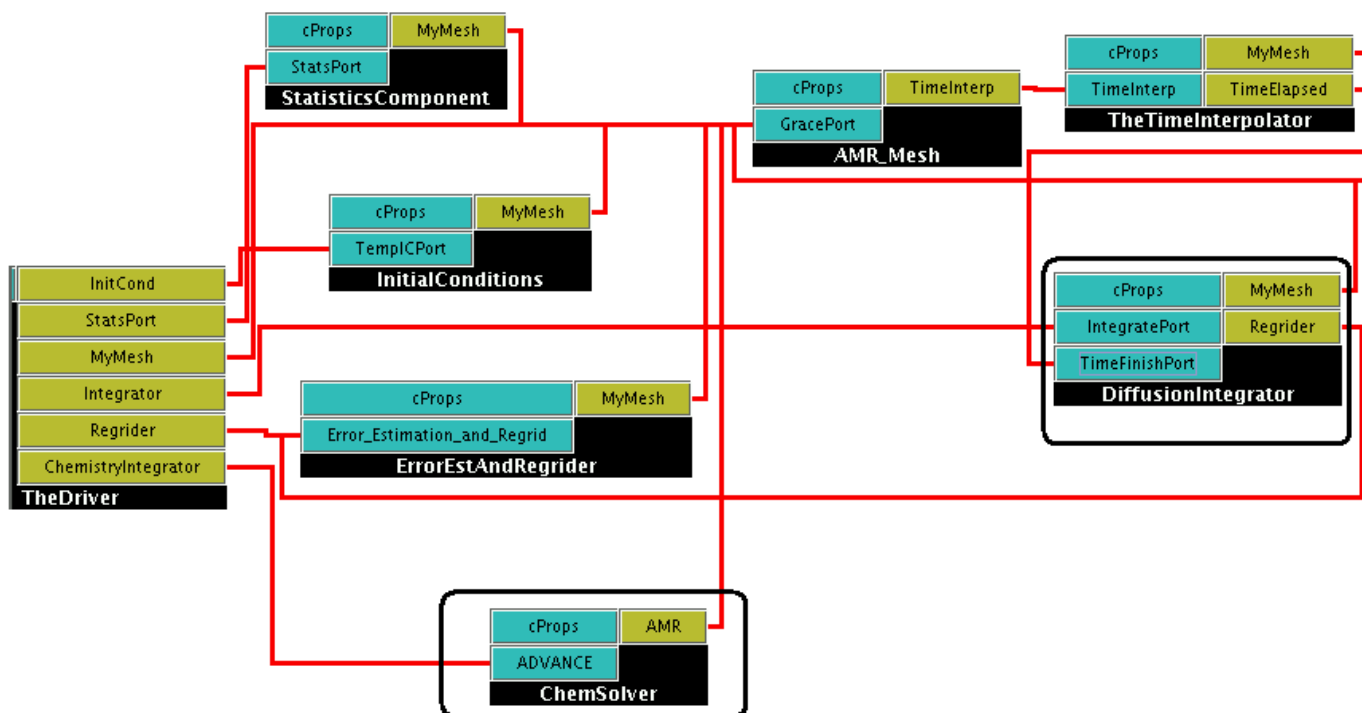


Figure 7. Assembly diagram of the simulation with reaction and diffusion black boxes.

particularly thank Jaideep Ray and Sophia Lefantzi for providing the original components used in our chemical sciences example.

This work has been supported in part by the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing initiative, through the Center for Component Technology for Terascale Simulation Software, of which ORNL and SNL are members.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.

## References

- [1] CCA tutorials. <http://www.cca-forum.org/tutorials/>.
- [2] CFRFS webpage. <http://cfrfs.ca.sandia.gov>.
- [3] Requirements of component architectures for high-performance computing. <http://www.cca-forum.org/documents/requirements.shtml>.
- [4] B. Allan, R. Armstrong, S. Lefantzi, J. Ray, E. Walsh, and P. Wolfe. Ccaffeine - a CCA component framework for parallel computing. <http://www.cca-forum.org/ccafe/>.
- [5] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The cca core specification in a distributed memory spmd framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.
- [6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.
- [7] R. Armstrong and R. B. McCoy. The common component architecture: Fostering an open source community in high performance computing. In *Proc. of the Advanced School for Computing and Imaging*. Center Parcs Het Heijderbos, Heijen, Netherlands, 4–6 June 2003.
- [8] Babel homepage. <http://www.llnl.gov/CASC/components/babel.html>.
- [9] D. E. Bernholdt, W. R. Elwasif, and J. A. Kohl. Communication infrastructure in high-performance component-based scientific computing. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI User's Group Meeting Linz, Austria, September/October 2002. Proceedings*, volume 2474 of *Lecture Notes in Computer Science*, pages 260–270. Springer, September 2002.
- [10] D. E. Bernholdt et al. A component architecture for high-performance scientific computing. *Intl. J. High Perf. Comp. Appl.*, in preparation for ACTS Collection special issue.
- [11] F. P. Brooks, Jr. *The Mythical Man-Monday: Essays on Software Engineering*. Addison Wesley Professional, second edition, 1995.

- [12] Common Component Architecture Forum. <http://www.cca-forum.org>.
- [13] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000.
- [14] O. M. Group. *The Common Object Request Broker: Architecture and Specification*. OMG Document, 1998. <http://www.omg.org/corba>.
- [15] O. M. Group. *CORBA Components*. OMG TC Document orbos/99-02-05, March 1999.
- [16] P. Hovland, K. Keahey, L. C. McInnes, B. Norris, L. F. Diachin, and P. Raghavan. A quality of service approach for high-performance numerical components. In *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*, Toulouse, France, June 20 2003.
- [17] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France*. IEEE Computer Society, 2003. Distributed via CD-ROM.
- [18] M. M. Lehman. A brief introduction to the FEAST hypothesis and projects (feedback, evolution and software technology). <http://www.doc.ic.ac.uk/~mml/feast>.
- [19] Microsoft COM Web page. <http://www.microsoft.com/com/about.asp>.
- [20] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, June 1999.
- [21] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28 (12):1811–1831, 2002.
- [22] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.
- [23] J. Siegel. OMG overview: CORBA and the OMG in enterprise computing. *Communications of the ACM*, 41(10):37–43, 1998.

# The High-Level Parallel Language ZPL Improves Productivity and Performance

Bradford L. Chamberlain<sup>\*†</sup>      Sung-Eun Choi<sup>‡</sup>      Steven J. Deitz<sup>\*</sup>      Lawrence Snyder<sup>\*</sup>

<sup>\*</sup>University of Washington  
Seattle, WA 98195  
{brad,deitz,snyder}@cs.washington.edu

<sup>†</sup>Cray Inc.  
Seattle, WA 98104  
bradc@cray.com

<sup>‡</sup>Los Alamos National Laboratory  
Los Alamos, NM 87545  
sungeun@lanl.gov

## Abstract

*In this paper, we qualitatively address how high-level parallel languages improve productivity and performance. Using ZPL as a case study, we discuss advantages that stem from a language having both a global (rather than a per-processor) view of the computation and an underlying performance model that statically identifies communication in code. We also candidly discuss several disadvantages to ZPL.*

## 1. Introduction

*In the spring of 2003, we encountered a curious bug in one of the NAS parallel benchmarks. To evaluate the scalability of ZPL, we were comparing our ZPL implementation of the NAS CG benchmark against the provided Fortran+MPI implementation on an increasing power-of-two number of processors of a new 1024-node cluster at Los Alamos National Laboratory (LANL). Both implementations ran flawlessly on up to 512 processors but, on our first 1024 processor run, the Fortran+MPI failed to verify correctly even as the ZPL worked. A day after we reported the failed verification to NAS, they were able to produce identical erroneous results on an IBM SP.<sup>1</sup> It wasn't a strange interaction between LANL's experimental cluster and ZPL, but rather a bug in the long-standing Fortran+MPI benchmark...*

\* \* \*

ZPL is a high-level parallel programming language developed at the University of Washington. Our implementation is based on a compiler that translates ZPL programs to C code with calls to MPI, PVM, or SHMEM, as the user chooses. Since the first release of this compiler in 1997, there have been significant improvements as we have evolved the language. This paper discusses some of the lessons we have learned over this time.

<sup>1</sup>Personal Communication. Rob F. Van der Wijngaart. April 9, 2003.

Like Co-array Fortran, High Performance Fortran, Titanium, Unified Parallel C and other parallel languages, ZPL offers scientists who are frustrated by MPI a much improved parallel programming experience. The anecdote above, which we will come back to later in this paper, illustrates this point and is the sort of issue we will discuss in this paper. The point of this anecdote is not that the provided Fortran+MPI benchmark was poorly written. Indeed, the NAS benchmarks are well-known for being well-written and highly-optimized. The point, as we will see later, is that the high-level nature of ZPL virtually eliminates a wide class of parallel programming bugs, thus making parallel programming easier.

Focusing on ZPL, this paper addresses how high-level parallel languages improve both productivity and performance. Throughout this paper, we will present anecdotes, code segments, and qualitative arguments as evidence of this improvement. The purpose of this paper is not to advertise ZPL but rather to encourage researchers to explore the space of language abstractions which ZPL champions.

This paper is organized as follows. In the next section, we characterize the design space of ZPL. No introduction to the language is offered; the interested reader is instead referred to the literature [4, 21]. In Section 3, we examine aspects of ZPL that increase productivity and performance. In Section 4, we discuss limitations of ZPL and, in Section 5, we conclude.

## 2. Characterizing ZPL

Figure 1 shows C+MPI and ZPL implementations of a trivial benchmark. The idea behind the benchmark is to iteratively replace each element in a 1D array with the average of its two neighboring elements until the change between the values in the array on successive iterations is small. Though admittedly contrived, the codes effectively illustrate two important characteristics of ZPL.

First, ZPL is a global-view parallel language. The programmer writes code that largely disregards the processors that will execute it. Thus array A is declared based on the

```

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int n;
double *A, *Tmp;
const double epsilon = 0.000001;

int main(int argc, char* argv[]) {
    int i, iters;
    double delta;
    int numprocs, rank, mysize;
    double sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (argc != 2) {
        printf("usage: %line %n\n");
        exit(1);
    }
    n = atoi(argv[1]);
    mysize = n * (rank + 1) / numprocs -
            n * rank / numprocs;
    A = malloc((mysize+2)*sizeof(double));
    for (i = 0; i <= mysize; i++)
        A[i] = 0.0;
    if (rank == numprocs - 1)
        A[mysize+1] = n + 1.0;
    Tmp = malloc((mysize+2)*sizeof(double));
    iters = 0;
    do {
        iters++;
        if (rank < numprocs-1)
            MPI_Send(&(A[mysize]), 1, MPI_DOUBLE, rank + 1,
                    1, MPI_COMM_WORLD);
        if (rank > 0)
            MPI_Recv(&(A[0]), 1, MPI_DOUBLE, rank - 1,
                    1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (rank > 0)
            MPI_Send(&(A[1]), 1, MPI_DOUBLE, rank - 1,
                    1, MPI_COMM_WORLD);
        if (rank < numprocs-1)
            MPI_Recv(&(A[mysize+1]), 1, MPI_DOUBLE, rank + 1,
                    1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (i = 1; i <= mysize; i++)
            Tmp[i] = (A[i-1] + A[i+1]) / 2.0;
        delta = 0.0;
        for (i = 1; i <= mysize; i++)
            delta += fabs(A[i] - Tmp[i]);
        MPI_Allreduce(&delta, &sum, 1, MPI_DOUBLE,
                    MPI_SUM, MPI_COMM_WORLD);
        delta = sum;
        for (i = 1; i <= mysize; i++)
            A[i] = Tmp[i];
    } while (delta > epsilon);
    if (rank == 0)
        printf("Iterations: %d\n", iters);
    MPI_Finalize();
}

```

(a)

```

program line;
config var
    n : integer = 6;
region
    R = [1..n];
    BigR = [0..n+1];
direction
    east = [1];
    west = [-1];
var
    A, Tmp : [BigR] double;
constant
    epsilon : double = 0.000001;
procedure line();
var
    iters : integer;
    delta : double;
begin
    [BigR] A := 0;
    [n+1] A := n + 1;
    iters := 0;
    [R] repeat
        iters += 1;
        Tmp := (A@east + A@west) / 2.0;
        delta := +<< abs(A - Tmp);
        A := Tmp;
    until delta <= epsilon;
    writeln("Iterations: %d": iters);
end;

```

(b)

Figure 1. A trivial benchmark written to compare (a) C+MPI and (b) ZPL. This benchmark measures how many iterations are needed for an array to reach a fixed point. The user sets  $n$ , the size of the problem, at the command line. The program starts by initializing the array to zero with left and right borders set to 0 and  $n+1$  respectively. On each iteration, the elements in the array are replaced by the average of their two neighbors. The program terminates when the sum of the changes is less than the fixed constant  $\epsilon$ . The number of iterations is reported. The use of the italics in the C+MPI code indicates the changes that are necessary to make when parallelizing the sequential language C using MPI. The vertical bars on the left indicate new lines of code; in addition,  $mysize$  replaces occurrences of  $n$ .

global bound of  $n + 1$ . In contrast, C+MPI is a local-view parallel language, and array  $A$  is declared based on per-processor bounds of `mysize + 2`. The highlighted parts of the C+MPI code show the changes that needed to be made from a sequential C code and the burden that is placed on the local-view programmer. In addition to using local bounds to size arrays on a per-processor basis, inter-processor communication must be explicitly managed in tedious detail.

It is important to note the difference between global-view languages such as HPF [14] and ZPL and local-view languages with global address spaces such as Co-array Fortran [19], Titanium [24], and UPC [3]. The latter are sometimes referred to as fragmented languages because they require programmers to divide the expression of their computation between the processors in an SPMD style of programming. They are significantly easier to use than C+MPI because of their global address space but, unlike global-view languages, they require the user to manage low-level synchronization.

Second, despite its global view, communication is explicit in ZPL. The details of communication are managed by the compiler, but the ZPL programmer is readily aware of where communication is induced. This provides a simple, but powerful performance model called the what-you-see-is-what-you-get (WYSIWYG) performance model [5]. The only communication in the simple ZPL program in Figure 1 is induced by the *at operator* (`@`), which shifts data across processor boundaries, and the *reduce operator* (`op<<`), which determines the sum of values distributed across all the processors.

ZPL is the only language to offer both properties. In the most well-known global-view language, High Performance Fortran, the programmer achieves parallelism by supplying directives of distribution and parallel computation. As a parallel extension to Fortran 90, its easy to reason about what is computed. However, communication requirements for a given statement are invisible in the syntax, thus making it a challenge for both programmers and compilers to optimize communication. On the other hand, local-view languages tend to make communication explicit but at the expense of the global view of computation.

### 3. Advantages of ZPL

This section is composed of seven parts, each of which addresses some aspect of the advantage of high-level parallel programming languages. The first two parts look at advantages of having a global view of computation; it makes parallel programming easier and provides for more general parallel programs. The next two parts focus on language abstractions; structural abstractions improve programmability while orthogonal abstractions make it easy to tune paral-

lel codes. The fifth part discusses how a high-level performance model makes it easy to maintain fast code, and the sixth part discusses how high-level languages stop programmers from over-specifying code and keeping the compiler from making effective optimizations. In the final part, we show some performance results which suggest that the bottom line of high-performance is still achievable.

#### 3.1. Global-view languages make parallel programming easier

As a case in point, we will elaborate on our introductory story. The NAS Parallel Benchmarks (NPB) [1] have long served as a way for us to evaluate the performance of ZPL. These benchmarks were designed to assist in evaluating the performance of parallel supercomputers. Derived from Computation Fluid Dynamics (CFD) applications and implemented in Fortran or C and MPI, they are “intended to be run with little or no tuning, [and] approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer.” [1] This statement is actually too modest. These benchmarks are highly-tuned and represent the upper end of achievable performance with a message-passing library. The benchmarks are well-written, stable programs that garner a substantial degree of respect in the high-end computing community.

The NAS CG benchmark estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The bug we encountered on 1024 processors was all the more curious because of the relative age of CG. Not only did the CG benchmark run flawlessly on up to 512 processors, but it had also been used for years in evaluating parallel systems. The 1024-processor bug was found to be in the initialization and was fixed by the NAS team within a week.<sup>2</sup> What happened was that an array used later in the computation and treated as scratch in the initialization was sized based on the problem size divided by the number of processors. As the number of processors grew, it became too small to fit the initialization data.

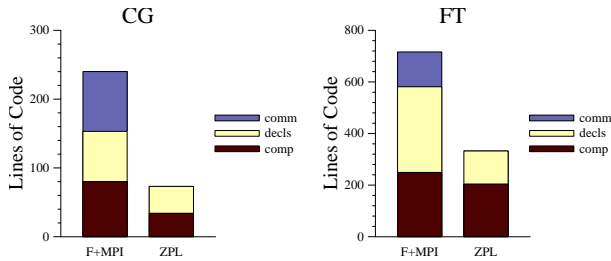
Because of the local view of computation, the array was sized based on local per-processor bounds. Given a global-view language like ZPL, that same array would be sized based on the global bounds. Had the programmer made a similar array sizing mistake in ZPL, the benchmark would have failed on any number of processors, not just when the number became large. Thus global-view languages make the development of working parallel programs easier.

Global-view languages make parallel programming easier for many other reasons too. Here is another story of using ZPL.<sup>3</sup> A professional programmer at HP with over “5 years of experience ... doing regular product development

<sup>2</sup>Personal Communication. Haoqiang Jin. April 10, 2003.

<sup>3</sup>Personal Communication. George Forman. December 5, 2003.





**Figure 2. Charts showing line counts of the Fortran+MPI and ZPL implementations of the NAS CG and FT benchmarks. The counts are subdivided into lines used for communication, declarations, and computation. These counts are being reprinted from a previous paper which can be consulted for more detailed information on the ZPL implementations of these benchmarks [11].**

work” wrote a code to test clustering algorithms. “It was a code he cared about and had cultivated for many research experiments over the course of a year+ for testing different kinds of clustering. It was tuned for performance, because he had to do many runs for research significance.” The core of the computation was 355 lines but, by describing it in Matlab, the programmer was able to explain it to a second HP programmer easily.

This second programmer, having worked on ZPL in the past, was eager to try writing the code in ZPL. Sequential runs took copious amounts of time, and both researchers expected they could achieve near linear speedup. In roughly 6 hours, the second programmer had the ZPL code working. It was 73 lines. Not only did it show nearly linear speedup [25] but, to the C programmer’s surprise, its sequential performance was better than that of the optimized C code. This application helped the HP researchers demonstrate that clustering code across continents, even with bad network latency, is better than shipping data to local clusters [13].

Though lines of code is not an ideal metric for evaluating a parallel programming language, it does provide some quantitative measure of programmability. Figure 2 counts the number of lines of code in the timed portions of the NAS CG and FT benchmarks. The ZPL codes require less than half the number of lines used to write the equivalent Fortran+MPI. Inspecting the codes reveals similar complexities and simplifications as in this paper’s examples, yet on a larger scale. This is a testament to how much easier it is to use ZPL.

### 3.2. Global-view languages provide for more general parallel programs

To keep MPI programs from requiring even more lines of code, they are often written with assumptions about the problem size or the number of implementing processors. For example, these may both be required to be powers of two. For the provided Fortran+MPI implementations of the NAS CG, FT, and MG benchmarks, the number of processors is required to be a power of two. In contrast to this restriction to the Fortran+MPI versions, the ZPL program can run on a non-power-of-two number of processors.

These assumptions in MPI are not surprising. They greatly simplify the implementation and permit optimizations that would not be possible in the more general case. However, there are times when one wants to run on a non-power of two number of processors. For example, given a 64 processor machine, scientists might not want to wait for a 16 processor job to finish if their programs could run on the 48 available processors sooner. Also, due to budget constraints, machines are often not composed of a power-of-two number of processors.

Modifying an MPI code to introduce such flexibility can often impact all aspects of the code. Moreover, as illustrated in Figure 6 and seen in the literature [7], running the more general ZPL version of the CG, FT, and MG benchmarks on a non-power-of-two number of processors results in improvements over the next smallest power-of-two number of processors for the Fortran+MPI benchmark.

### 3.3. Structural abstractions improve programmability

Sparse problems comprise a challenging and crucial class of computation in high-end computing. Yet it is important to remember that the sparsity of an array or matrix relates to its potential for optimized implementation rather than the fundamental operations it supports. As an example, matrix-vector multiplication is a mathematical operation whose definition is independent of whether the matrix operand is sparse or dense; its sparsity merely provides an opportunity for reducing the computational and storage overheads of the operation.

Most languages fail to support abstractions for sparse data structures, placing the effort of exploiting sparsity on the programmer rather than the tools. Programmers must build their own data structures to represent sparse arrays and this change in representation forces a corresponding change to the computation itself. As an example, consider the Fortran codes in Figure 3(a) which implement matrix-vector multiplication for a dense array and for a sparse array using compressed row storage. Note that the change from sparse to dense is pervasive in the code. The 2D array  $a$

DENSE	SPARSE
<pre> <b>real</b> p(n), w(n) <b>real</b> a(n,n)  (a) <b>do</b> j = 1, numrows       sum = 0.d0       <b>do</b> k = 1, numcols             sum = sum + a(j,k)*p(k)       <b>enddo</b>       w(j) = sum <b>enddo</b>  <b>region</b> R = [1..n, 1..n];       RowVect = [*, 1..n];       ColVect = [1..m, *];  (b) <b>var</b> M: [R] <b>double</b>;       V: [RowVect] <b>double</b>;       S: [ColVect] <b>double</b>;        [ColVect] S := +&lt;&lt;[R] (M * V); </pre>	<pre> <b>real</b> p(n), w(n) <b>real</b> a(nnz) <b>integer</b> colidx(nnz) <b>integer</b> rowstr(n)  <b>do</b> j = 1, numrows       sum = 0.d0       <b>do</b> k = rowstr(j), rowstr(j+1)-1             sum = sum + a(k)*p(colidx(k))       <b>enddo</b>       w(j) = sum <b>enddo</b>  <b>region</b> R = [1..n, 1..n] <b>where</b> /* pattern */;       RowVect = [*, 1..n];       ColVect = [1..m, *];  <b>var</b> M: [R] <b>double</b>;       V: [RowVect] <b>double</b>;       S: [ColVect] <b>double</b>;        [ColVect] S := +&lt;&lt;[R] (M * V); </pre>

**Figure 3. Dense and sparse implementations of matrix-vector transpose in (a) Fortran+MPI and (b) ZPL**

becomes a 1D array of values with two integer vectors to provide directory information. This forces the inner loop to be restructured to iterate properly over the directory, index  $a$ , and index into  $p$  using an indirect index. This represents a substantial modification to the code considering that the mathematical operation being expressed has not changed. The problem is exacerbated in parallel codes where communication code must also be rewritten to deal with sparse structures.

In contrast, ZPL supports sparse arrays and matrices as a fundamental concept, allowing programmers to specify an array's sparsity as part of the declaration of its size and shape [8]. This results in minimal impact on the computation itself. Consider the ZPL implementations of sparse and dense matrix-vector multiplication in Figure 3(b). By isolating the impact that such a simple conceptual change has on the code, the programmer can easily switch between sparse and dense representations with little penalty. For example, in the NAS MG benchmark, the input array  $V$  is truly sparse, containing only 20 non-zeroes in its  $512^3$  elements for class C. Using a sparse representation can improve the space and computational costs associated with  $V$ , yet making this change requires significant effort in most languages and as a result, most implementations do not bother. In ZPL the change is trivial, reducing the overall memory footprint of the program by 1/3. As a second example, the NAS FT benchmark checks its results by taking a sparse walk through a dense array. Representing this subset of values directly using a sparse region is a simple change in ZPL and improves performance by making the parallelism

more explicit.

By separating the specification of sparsity from its use in computation, the compiler is also given increased flexibility in its choice of sparse data structure implementations. The ZPL compiler automatically tunes its sparse representation based on the requirements dictated by its usage in the code [4]. One could furthermore imagine allowing the user to specify a preferred sparse data structure as part of the array's declaration. In conventional languages where the user must manage sparsity explicitly, such changes tend to affect every line of code that refers to the array, violating the general principle of separating data structure from algorithm.

### 3.4. Orthogonal abstractions make it easy to tune parallel codes

Significant changes in performance can be realized by fine-tuning a parallel code after it is written. For example, a programmer could want to change the ratio between the number of processors in the column and row dimensions of a 2D processor grid. In the NAS CG benchmark, this results in improved performance when the data-to-processor ratio is large. In the Fortran+MPI implementation, this is a difficult change but, in the ZPL implementation, it is trivial.

Figure 4 shows the code involved in transposing a row to a column in Fortran+MPI and ZPL. Because the row and column arrays are replicated across their dimension of the transpose array, it makes sense, for performance of the transpose, to use a 1:1 or 2:1 row-to-column processor layout in the 2D processor grid. The 1:1 ratio is always ideal,

```

if ( l2npcols .ne. 0 ) then
  call mpi_irecv(q, exch_recv_length,
                dp_type, exch_proc, 1,
                mpi_comm_world, request, ierr)
  call mpi_send(w(send_start), send_len,
                dp_type, exch_proc, 1,
                mpi_comm_world, ierr)
  call mpi_wait( request, status, ierr )
else
  do j=1,exch_recv_length
    q(j) = w(j)
  enddo
endif

```

(a)

```

[Row] W := P#[Index2, srcindex];

```

(b)

Figure 4. NAS CG transpose code in (a) Fortran+MPI and (b) ZPL.

but if we want to run on an odd power-of-two number of processors, e.g. 8, then sometimes we need to use a 2:1 ratio, e.g. a  $4 \times 2$  processor grid. In these cases, the communication pattern is one-to-one. Each processor needs to send data to only one processor and needs to receive data from only one processor.

The Fortran+MPI code, because of its low-level of abstraction, cannot keep the processor grid orthogonal to the computation. Thus the one-to-one communication pattern is unyielding. In ZPL, on the other hand, if the processor grid has a 2:1 or 1:1 ratio, the one-to-one communication pattern is achieved, but the processor grid is not restricted to having this ratio. This is useful for the part of the code that implements the sparse computation. It turns out, for the sparse computation, that a 1:2 or even a 1:4 ratio, improves the performance of this part of the code. If the data-to-processor ratio is high then the overall performance of the code improves since the sparse computation, rather than the transpose, is the bottleneck.

### 3.5. A high-level performance model makes it easy to maintain fast code

*Wavefront computations* are common in scientific applications. A wavefront computation is one in which the value of each data element is dependent on one or more values computed in previous iterations of the loop nest. Though inherently serial, *pipelining* is a well known, but tedious, technique for efficient parallelization of wavefront computations [9, 23, 15].

The Accelerated Strategic Computing Initiative’s (ASCI) SWEEP3D benchmark solves a three dimensional neutron transport problem. Figure 5 compares the ASCI Fortran+MPI and ZPL implementations of the core computation. The Fortran version is simplified for improved clarity. As always, the reduction in code size is dramatic (over three times). Here we will focus on the pipelining itself.

The Fortran version assumes that the problem is only distributed over two ( $i$  and  $j$ ) of the three dimensions. Consequently, the  $k$  dimension is treated differently than the other

two when the computation is actually the same. For example, the first twelve lines in the main loop deal with initializing the *inflows*. Notice a subtlety in the initialization of the  $i$ - and  $j$ -inflows; they are actually performed within the pipeline loop ( $kk$ ). In other words, there is communication in the *inner loop* of the computation. This has a profound performance implication, yet the code looks nearly the same as a very simple data parallel array operation. To fix this problem, the  $kk$  loop should just be moved down below the inflow initialization.

The wavefront computation in the ZPL version begins with the `interleave` keyword. This forces the statements within its scope to execute in an interleaved manner by fusing the statements into the same scalarized loops. The encompassed “prime at” references ‘@ indicate to the programmer *and the compiler* that the operations within the statement block may require serialization of the computation. The compiler can and does implement pipelining as described above, thus relieving the programmer from worrying about the details of the implementation including the *tile size* to use for pipelining. These special prime at references explicitly indicate that those referenced values are dependent on values computed in previous iterations, resulting in serialization. Notice that in the Fortran version, indexing withstanding, it is not at all clear which references cause the serialization. Moreover, if these references were to change such that no serialization were necessary, the programmer should explicitly move the  $i$ - and  $j$ -inflow initialization outside the  $kk$  loop for fully parallel execution. Not doing so would not necessarily result in an incorrect program, just an inefficient one.

### 3.6. High-level languages stop programmers from over-specifying code

Though the key advantage of a high-level language for productivity is that it frees the programmer from the heavy burden of writing low-level implementing code, there is a further advantage: The compiler is freed from having to use that implementation. That is, low-level code over-specifies

```

do mo = 1, mmo !outer angles loop (batches of mi angles)
<initialize K-inflows -- triply nested loop>
do kk = 1, kb ! outer planes loop (batches of mk-planes)
  if (ew\rcv.ne.0) then ! I-inflows for block
    <receive boundary values>
  else
    <initialize I-inflows -- triply nested loop>
  endif
  if (ns\rcv.ne.0) then ! J-inflows for block
    <receive boundary values>
  else
    <initialize J-inflows -- triply nested loop>
  endif
  ! JK-diagonals with MMI pipelined angles
do idiag = 1, jt+nk-l+mi-l
  do jkm = 1, ndiag
    do i = i0, il, i2 ! I-line recursion
      ci = mu(m)*hi(i)
      dl = ( sigt(i,j,k) + ci + cj + ck )
      dl = 1.0 / dl
      ql = phi(i) + ci*phiir + \
          cj*phibj(i,lk,mi) + \
          ck*phikb(i,j,mi)
      phi(i) = ql * dl
      phiir = 2.0d+0*phi(i) - phiir
      phii(i) = phiir
      phibj(i,lk,mi) = 2.0d+0*phi(i) - phibj(i,lk,mi)
      phikb(i,j,mi) = 2.0d+0*phi(i) - phikb(i,j,mi)
    end do ! i
    phiib(j,lk,mi) = phiir
  end do ! jkm
end do ! idiag
<compute and send outflows>
end do ! kk
end do ! mo

```

(a)

```

[R] begin
[lasti of R] phiib := 0.0; -- boundary i inflow
[lastj of R] phibj := 0.0; -- boundary j inflow
[lastk of R] phikb := 0.0; -- boundary k inflow
ci := mu * hi;
cj := eta * hj;
ck := tsi * hk;
dl := 1.0 / (Sigt + ci + cj + ck);
interleave
  ql := phi + ci*phiib'@lasti +
        cj*phib'@lastj +
        ck*phikb'@lastk;
  phi := ql * dl;
  phiib := 2.0*phi - phiib'@lasti;
  phibj := 2.0*phi - phibj'@lastj;
  phikb := 2.0*phi - phikb'@lastk;
end;
-- final i, j, and k outflows
[lasti in R] leakage[1+i3] += wmu * phiib * dj * dk;
[lastj in R] leakage[3+j3] += weta * phibj * di * dk;
[lastk in R] leakage[5+k3] += wtsi * phikb * di * dj;
end;

```

(b)

Figure 5. Core computation of the ASCI SWEEP3D benchmark in (a) Fortran+MPI and (b) ZPL.

an implementation, possibly limiting the compiler’s ability to optimize.

Many researchers have shown that message passing is often the wrong choice for efficient communications (e.g., [22, 16, 18]). Regardless, most parallel programs written in low-level languages such as C or Fortran use message passing, partly due to the fact that a standard interface exists (MPI) but also to the fact that it is easier to use than other proposed interfaces. In fact, these other interfaces were primarily designed as targets for libraries and compilers for high-level languages rather than programmers.

The reason these alternative communication libraries perform better than message passing libraries is that the exposed interface more closely matches the implementing hardware. For example, Striker et al. [22] showed that the synchronization required of message passing limits performance compared to one-sided communication on the T3D, a machine that provided hardware support for one-sided communication via the SHMEM library [2]. ARMCI [17] generalizes the low-level libraries of modern PC network interface cards (such as Myrinet, Quadrics Elan, and Infiniband) to provide efficient one-sided communication. ARMCI has been shown to perform well on high-performance clusters [18], a platform generally accepted as one for message passing.

Like other high-level parallel languages, programmers do not write interprocessor communication commands in ZPL. Rather, the compiler determines where communication may be required and it inserts into the object code the appropriate calls to the ZPL runtime library. The compiler actually generates calls that describe the non-local data dependences, *not explicit communication calls*. The calls in this interface, called *Ironman* [6], describe four important locations in the object code. Two are for the *destination* (DR/DN), and the other two for the *source* side (SR/SV).

*Destination Ready (DR)*. The locally cached copy of the non-local data will not be read again (until DN). These memory locations on the destination processor are now *ready* to be overwritten with new values.

*Source Ready (SR)*. The values needed on the destination processor have just been written. The source processor is *ready* to transmit the values.

*Destination Needed (DN)*. The locally cached copy of the non-local data will be read. The non-local data is *needed* at the destination.

*Source Volatile (SV)*. The values needed on the destination processor will be overwritten. The values are now *volatile* and must be transmitted by this point.

These Ironman calls are bound to a specific communication library (MPI, SHMEM, etc.) at link-time. This late binding enables the use of the most appropriate communication mechanisms for a given platform *without changing the user's program itself*. For example, for MPI, DR and DN bind directly to `MPI_Irecv` and `MPI_Wait`, and SR and SV bind to `MPI_Isend` and `MPI_Wait`. For a put-based implementation of a one-sided communication library such as SHMEM or ARMCI, SR puts the data from source to destination, DR and DN perform loosely-coupled synchronization with SR, and SV is not needed.

When programmers write MPI message passing code directly, it is the semantics of message passing *not the individual implementations of MPI* that ultimately limit performance. For example, data that is irregularly laid out in memory, must be marshaled and brought together into a contiguous message buffer before it can be sent. However, some libraries (such as SHMEM and ARMCI) and PC networks interfaces (such as Dolphin SCI and Quadrics Elan) expose via their native communication library remote direct memory access (RDMA) to remote addresses. These do not require the extra copy and memory overhead. An implementation of MPI using these facilities has no control over this data marshaling because the code to perform this operation is embedded in the program itself. If a ZPL program is to be run using MPI, then the ZPL libraries for MPI would perform the marshaling; on a machine with efficient RDMA, no marshaling would be performed.

By removing the burden of writing low-level implementation code such as communication calls, the compiler is able to better optimize communication using data dependence information. Moreover, the late binding to the native communication library allows for the most efficient communication library to be used without penalties incurred when using a particular library.

### 3.7. High-level languages achieve high-performance

Performance is the bottom line of parallel programming. The whole reason to parallelize a code is to make it run faster. If the high-level language hurts either the sequential performance or the program's ability to scale to higher numbers of processors, then it loses its value.

Figure 6 shows the performance of the NAS CG and FT benchmarks in ZPL and Fortran+MPI across three platforms. These platforms are representative of the diversity of machines. The T3E provides a top-of-the-line network for low-latency interprocessor communication whereas the cluster has much higher latency, but faster processors.

Note that the generality of the ZPL implementation is apparent in the 176-processor run on the IBM SP. Here the ZPL code can improve its performance further even though the Fortran+MPI code was not written to run on a non-

power-of-two number of processors.

## 4. Limitations and Evolution

A very reasonable observation to make about ZPL is that for all of its convenient features and abstractions, it does not support arbitrary models of parallel programming. While ZPL's support for parallel computation using sparse and dense multidimensional index sets supports a wide variety of high-end applications, it lacks similar abstractions for other paradigms such as distributed hash tables, graph-based data structures, and nested parallelism. Other desirable features such as user-defined data distributions and task parallelism are only now being added to the language [10, 12].

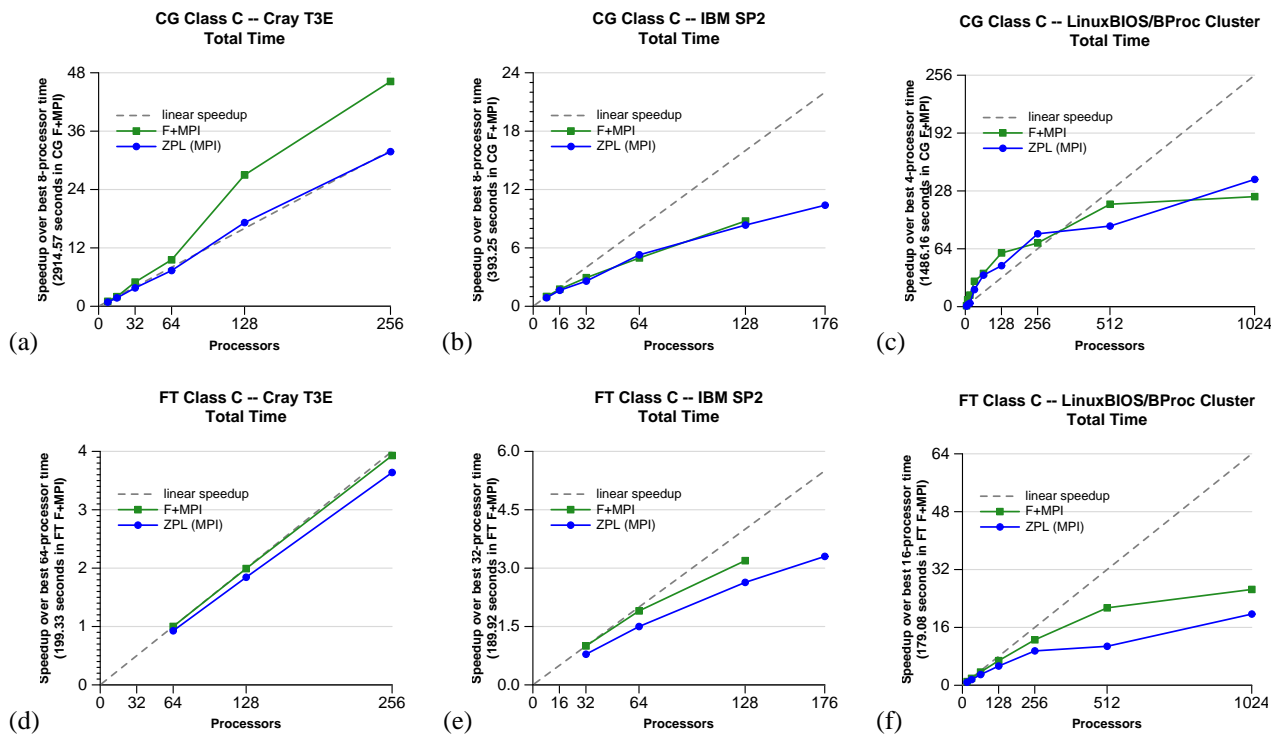
Our explanation for this lack of generality is one of philosophy. While many languages strive for complete generality from their inception, these languages tend to either provide a very low level of abstraction, to never get all of their features implemented, or to never achieve good performance for more than a narrow range of features. In contrast, our approach has been to start with those facilities we know how to compile well and then add generality to the language as our understanding and experience grow. As a result, we have managed to achieve good performance throughout ZPL's lifetime while keeping the language's concepts elegant and interoperable.

The downside to this approach is that it has taken a long time to acquire the knowledge. At times, ZPL's evolution has been slow and incremental. To reduce the effects of this problem, we are currently in the process of producing an open source release of ZPL in hopes of engaging a broader community in its support and development (previous releases have contained the compiler and runtime binaries without their sources). A private release of the source to colleagues at U. Mass-Lowell has already allowed ZPL to be ported to the unusual Mercury-Race architecture [20].

We also anticipate that the open source release should allow us to support a broader community of parallel programmers, since many potential users in the past have expressed their unwillingness to base their research on a language whose implementation they could not access directly (in part for fear that we would cease to support it in the future). Meanwhile, our evolution of the language progresses as does our enthusiasm for it, particularly as we look beyond the NAS parallel benchmarks to consider more challenging applications that push the limits of what we are currently able to express cleanly and efficiently in ZPL.

## 5. Conclusions

There is a growing consensus that the bottleneck for productivity in parallel computing lies with the low-level



**Figure 6. Graphs showing the total speedups of class C of the NAS CG and FT benchmarks across three platforms. The first column shows results on Yukon, a 272 processor Cray T3E with 260 user processors. Each processor is a 450 MHz Alpha processor with 256 MB of memory. The second column shows results on Icehawk, a 200 processor IBM SP with 176 user processors. The SP2 is composed of 44 nodes with 2 GB of memory per node. Each node contains four 375 MHz power3 processors. The third column shows results on up to 1024 processors of Pink, a 2048 processor cluster built with the LinuxBIOS/BProc technology. Pink is composed of 1024 nodes with 2 GB of memory per node. Each node contains two 2.4 GHz Intel Xeon processors. These results are being reprinted from a previous paper which can be consulted for more detailed information on the ZPL implementations of these benchmarks [11].**

programming models that users must rely on to express their programs. In this paper, we explored the benefits of languages that provide the programmer with a global view of their computation rather than a local per-processor view. In addition, we discussed why it is beneficial to allow programmers to reason about the implementation of their codes. In ZPL, this is achieved by making all communication requirements visible in the source code, allowing both the programmer and the compiler to reason effortlessly about this bottleneck of parallel computing.

By supporting a global view of computation with communication cues, ZPL provides programmers with a simpler programming model which allows for rapid development, evolution, and tuning. ZPL also makes the programmer's intentions clear to the compiler so that it can implement the code efficiently using a variety of data structures and communication protocols on any modern architecture.

## 6. Acknowledgments

During the time of this work, the second author was employed by Los Alamos National Laboratory and funded by the Mathematical Information and Computer Sciences (MICS) program of the DOE Office of Science, and the third author was supported by a DOE High Performance Computer Science Graduate Fellowship (HPCSGF). Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. This paper has been recorded under LANL LAUR number LA-UR-03-9521.

The authors would like to thank the many who have contributed to ZPL in the past, and thus made this work possible. This work was supported in part by grants of HPC resources from the Arctic Region Supercomputing Center and

Los Alamos National Laboratory for which we are grateful. Thanks are also due to the anonymous reviewers for their helpful suggestions.

## References

- [1] D. Bailey, T. Harris, W. Saphir, R. F. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NASA Ames Research Center (NAS-95-020), December 1995.
- [2] R. Barriuso and A. Knies. SHMEM user's guide. Technical report, Cray Research Inc., May 1994.
- [3] 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, Center for Computing Sciences, Bowie, MD, May 1999.
- [4] B. L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.
- [5] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.
- [6] B. L. Chamberlain, S.-E. Choi, and L. Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.
- [7] 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 the ACM Conference on Supercomputing*, 2000.
- [8] B. L. Chamberlain and L. Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.
- [9] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–844, 1986.
- [10] S. J. Deitz. Renewed hope for data parallelism: Unintegrated support for task parallelism in ZPL. Technical report, University of Washington (2003-12-04), December 2003.
- [11] S. J. Deitz, B. L. Chamberlain, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.
- [12] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Proceedings of the IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.
- [13] G. Forman and B. Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explorations Newsletter*, 2(2):34–38, 2000.
- [14] High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1*, November 1994.
- [15] E. C. Lewis and L. Snyder. Pipelining wavefront computations: Experiences and performance. In *Fifth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2000.
- [16] S. Lumetta, A. Mainwaring, and D. Culler. Multi-protocol active messages on a cluster of SMP's. In *Supercomputing*, 1997.
- [17] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Workshop on Runtime Systems for Parallel Programming (RTSPP)*, April 1999.
- [18] J. Nieplocha, J. Ju, and E. Apra. One-sided communication on the Myrinet-based SMP clusters using the GM message-passing library. In *CAC*, 2001.
- [19] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.
- [20] D. Rey, J. Stubblefield, and J. Canning. Porting the parallel array programming language ZPL to an embedded multi-computing system. In *Proceedings of the 2002 conference on APL*, pages 168–175. ACM Press, 2002.
- [21] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.
- [22] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the ACM International Conference on Supercomputing*, 1995.
- [23] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
- [24] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.
- [25] B. Zhang, M. Hsu, and G. Forman. Accurate recasting of parameter estimation algorithms using sufficient statistics for efficient parallel speed-up demonstrated for center-based data clustering algorithms. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2000.

# Raising the Level of Programming Abstraction in Scalable Programming Models

David E. Bernholdt  
*Computer Science & Mathematics*  
*Oak Ridge National Laboratory*  
*bernholdtde@ornl.gov*

Jarek Nieplocha  
*Computational Sciences & Mathematics*  
*Pacific Northwest National Laboratory*  
*j\_nieplocha@pnl.gov*

P. Sadayappan  
*Dept. of Computer and Information Science*  
*Ohio State University*  
*saday@cis.ohio-state.edu*

## Abstract

*The complexity of modern scientific simulations combined with the complexity of the high-performance computer hardware on which they run place an ever-increasing burden on scientific software developers, with clear impacts on both productivity and performance. We argue that raising the level of abstraction of the programming model/environment is a key element of addressing this situation. We present examples of two distinctly different approaches to raising the level of abstraction of the programming model while maintaining or increasing performance: the Tensor Contraction engine, a narrowly-focused domain specific language together with an optimizing compiler; and Extended Global Arrays, a programming framework that integrates programming models dealing with different layers of the memory/storage hierarchy using compiler analysis and code transformation techniques.*

## 1. Introduction

The role of computational simulation in science and engineering has blossomed in recent years to the point where it is now recognized as a peer to experimental and theoretical approaches and has become an indispensable tool to the progress of modern science and technology. Moreover, the pace of change and improvement in scientific high-end computing has been tremendous: more powerful computers allow researchers to perform larger and higher fidelity simulations, which in turn inspire the

need for yet larger and faster computers. However this progress has not been without cost. Software developers have had to face increases in the complexity of algorithms and methods concomitant with the increases in problem size and fidelity compounded by increases in software complexity required to tease the maximum performance out of hardware with deeper memory hierarchies, higher degrees of parallelism, and other “features”. The result of this burgeoning complexity is that more and more of the software developer’s effort goes into dealing with the details, with obvious impacts on overall productivity.

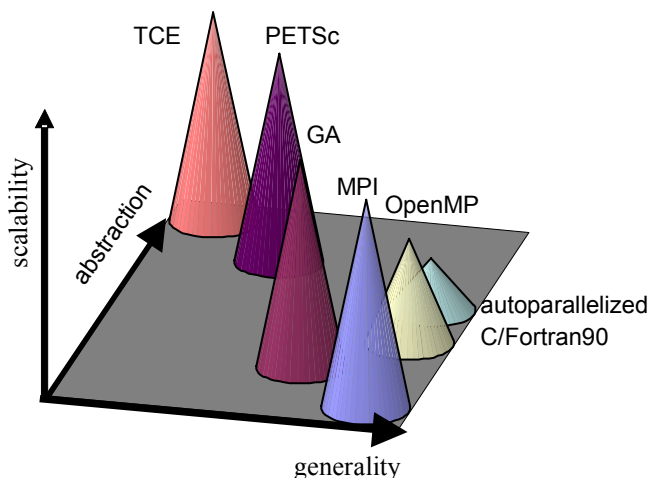
Any measure of productivity for a developer and user of software must take into account both the time required to develop the software and the time it takes to run, or the performance. A “productive” programming environment, therefore, is one that allows the programmer to easily express computational problem (i.e. a programming model which provides a high level of abstraction) while providing the highest possible performance. Based on our collective experience in high-performance scientific computing and our assessment of progress in the field over the last 10-15 years, we argue that raising the level of abstraction available to the developer has become a crucial factor in the effort to increase software productivity in scientific computing. After discussing the idea of abstraction in high-end computing, this paper presents the Tensor Contraction Engine (TCE) and Extended Global Arrays (XGA) as examples of efforts that take different approaches toward the goal of raising the level of abstraction while maintaining high performance.



## 2. Abstraction, Scalability, and Generality in High Performance Programming

For the programming model on which the software development effort is based, a key factor is the level of abstraction offered to the user. This term covers a number of factors, including the ease of expressing the (essentially mathematical) problem to be solved, and the ease of expressing the (parallel) algorithms necessary to solve it. We use NWChem [14, 16] as an example to highlight the importance of the model's level of abstraction in enhancing productivity in developing complex high-performance software. NWChem is a large (over a million lines of code) computational chemistry package that provides high-performance, scalable implementations of a broad spectrum of methods in computational chemistry. Development of NWChem began in 1993, at a time when the chemistry community had experimented with parallel computing, but had produced few general, scalable, high-performance parallel algorithms. Experience had shown that many quantum chemical methods could not be implemented easily in the traditional message-passing programming model. In addition, effective abstractions and parallel I/O techniques were needed for out-of-core chemistry algorithms. These challenges led to emergence of novel parallel programming tools that enabled rapid development and implementation of scalable algorithms in this science domain, namely the Global Array (GA) toolkit [1, 26], Disk Resident Arrays (DRA) [21], and Shared Files [22, 13]. When coupled with algorithms that appropriately consider the non-uniform memory access (NUMA) nature of modern high-performance computers, the GA model augmented with DRA has proven both very high performance and very expressive for algorithms of the type that appear in quantum chemistry. Indeed, at present, essentially all scalable parallel quantum chemistry packages utilize Global Arrays or an equivalent programming model rather than the two-sided message-passing programming model that dominates most other scientific domains. Though no quantitative data is available, the qualitative experience of the NWChem effort (now ten years old, and embodying far in excess of 100 person-years of effort) is that the high-level abstractions provided by GA, DRA, and Shared Files were found invaluable in rapid development of scalable algorithms for this scientific domain, and quickly enabled scientists without prior experience in parallel programming to become productive contributors in this large software development effort. As previously noted, despite the continued popularity of the message-passing model in other fields, all scalable quantum chemistry codes use GA-like programming models.

In addition to level of abstraction and scalability of a programming model, a third, related, dimension is the generality of the programming model – whether it is appropriate to a narrow or broad range of computational problems and scientific domains.



**Figure 1.** Relative classification of programming methodologies with respect to level of abstraction, generality, and parallel scalability

While it is hard or impossible to precisely quantify abstraction level, generality, and scalability of various programming models without reference to a particular class of problems and other factors, it is possible to estimate rough relative positions of various programming models within this three dimensional space. By way of example, Figure 1 presents such an assessment:

- MPI [3] provides a very general, but rather low-level programming model and generally supports a high degree of optimization and tuning, making it possible to obtain performance close to the raw capabilities of the underlying hardware. It thus scores high with respect to model-generality and scalability, but ranks low regarding the abstraction-level offered to the software developer.
- The Global Arrays [1, 26] library-based approach offers a global shared view of multi-dimensional array objects that can be accessed by processes via block get/put/update operations. It inter-operates with MPI and provides comparable performance and scalability. Through its shared global view of array objects, it offers a higher level of abstraction to the programmer. However, since it only applies to array objects, the GA model is less general than MPI.
- OpenMP [4] is a completely general parallel programming model that offers a shared-space view of arbitrary data structures. Thus it ranks very highly along the dimension of generality, on par with MPI. We rank it

slightly higher than MPI and GA with respect to abstraction-level (referring in this case just to the ease of expressing the parallelism of the problem) However, scalable implementations of OpenMP for large-scale systems are yet to be realized.

- Automatic parallelization of standard sequential C/Fortran programs: Standard sequential programming languages like C, C++ and Fortran have been heavily used for developing scientific and engineering applications. There has been a long history of efforts to automatically parallelize sequential programs. Although there was great optimism in the early days of parallel computing that compiler techniques could be developed to automatically parallelize sequential programs, today the prospects of achieving such a goal seem very dim. Several vendors have marketed commercial auto-parallelizing compilers, but their effectiveness has been limited, especially in the context of highly parallel systems.

- PETSc (Portable Extensible Toolkit for Scientific computation [5]) is an example of a class of tools that facilitates the parallel (as well as serial), numerical solution of PDEs that require solving large-scale, sparse nonlinear systems of equation. The user creates and manipulates matrix objects, whose underlying representation and distribution among nodes of a parallel machine are transparent to the user. A variety of linear and non-linear solvers are implemented. The level of abstraction is very high, since both the data distribution as well as the parallel nature of the underlying solvers can be completely transparent to the user. It ranks rather low with respect to generality, since the high level of abstraction is only available for the set of numerical methods implemented.

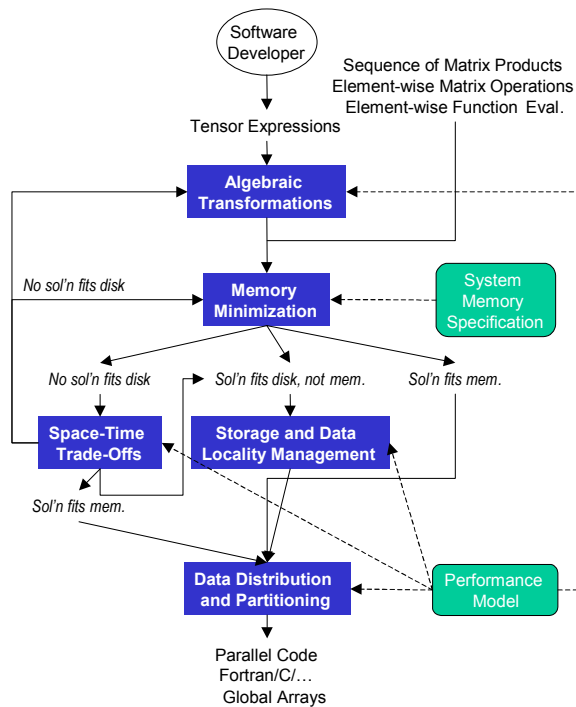
From the viewpoint of the scientist/software developer, one might describe the “holy grail” of productive scientific computing as being able to write the equations for the problem to be solved in a form that is close or identical to the way they would be expressed in a scientific paper and have tools turn this input into efficient, high-performance code. From the computer science viewpoint, the “holy grail” would be a programming model that maximizes all three axes (high abstraction/high generality/high performance), this is a daunting challenge (even for sequential computing!). However in order to address the looming crisis of software complexity, it is imperative to make progress toward solving this problem.

Such efforts typically try to move along one or more of the three dimensions, while maintaining the level of the remainder. In the remainder of this paper, we present examples of two efforts, taking different paths in the effort to raise the level of abstraction while preserving scalability. One involves the development of a high-level

language and optimizing compiler called the Tensor Contraction Engine (TCE) for a class of problems in computational chemistry, and the other an effort to generalize the Global Array programming model to transparently manage multiple layers of memory hierarchy.

### 3. The Tensor Contraction Engine

The Tensor Contraction Engine (TCE) is a domain-specific program synthesis system [6] being developed by a team of computer scientists and computational chemists. It is a system to automatically transform a high-level description of a quantum chemical model expressed in terms of complex tensor contraction expressions (essentially generalized matrix products on multidimensional arrays) into optimized parallel programs. A primary reason for the development of the system was to significantly decrease the amount of time needed to develop high-performance codes implementing accurate models for correlated electronic structure methods in computational chemistry packages. In this case, the level of abstraction (particularly the ease of expressing the problem itself) is so high that writing a program in the TCE environment is little more than writing out the tensor contraction expressions that define the method to be implemented and the parallelism is implicit in that input, but of course the TCE is limited to a



**Figure 2.** A schematic representation of the Tensor Contraction Engine’s architecture

narrow class of problems, as shown in Figure 1. Figure 2 provides a high-level picture of the transformation system. A brief description of the components of the system follows.

### 3.1. High-level language

The input to the synthesis system is a sequence of tensor contraction expressions (essentially sum-of-products array expressions) together with declarations of index ranges and symmetry and sparsity of matrices. The high-level notation offers two significant advantages:

1. For the user, the high-level representation makes it extremely convenient to express complex tensor contraction expressions.
2. For the compiler, the high-level representation provides essential information that facilitates domain-specific optimizations; such information would be difficult or impossible to extract out of code implementing such expressions in a language such as Fortran or C.

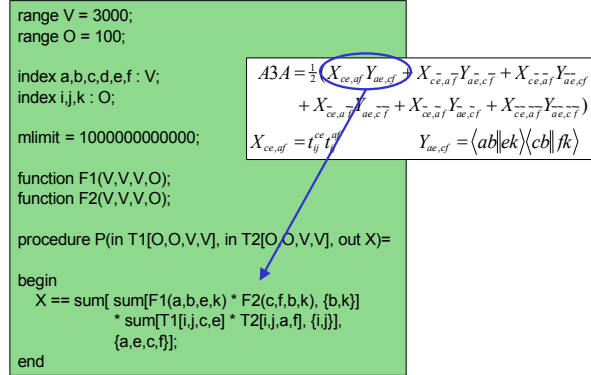
Figure 3 shows an example of a TCE program representing a term in a tensor contraction expression. It is shown along with a larger expression from a coupled-cluster [19, 20] model, shown in a notation used by quantum chemists in describing the computation. The tensor contraction expressions for accurate electronic structure models can have hundreds of such terms, and the Fortran codes implementing them often have tens of thousands of lines of code.

### 3.2. Algebraic transformations

Input from the user in the form of tensor expressions is transformed into a computation sequence. The properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition are used to search for various possible ways of applying these properties to an input sum-of-products expression. A combination that results in an equivalent form of the computation with minimal operation cost is generated. The problem of determining an equivalent operation-minimal form of the expression is NP-complete, but efficient pruning-search procedures have been developed that are very effective in practice [18].

### 3.3. Memory minimization

The operation-minimal computation sequence synthesized by applying algebraic transformation might require an excessive amount of memory due to the need to use large temporary intermediate arrays. The Memory Minimization step seeks to perform loop fusion transformations to reduce the memory requirements.



**Figure 3.** An example of the typical representation of tensor contraction expressions used in the scientific literature (inset) together with sample of the input language for the Tensor Contraction Engine for one term in the inset expression

Optimal loop fusion is also an NP-complete problem [12]. An abstraction called the fusion-graph has been developed and has served as the basis for a search process used to evaluate alternate the loop fusion choices in the context of the TCE [17]. The loop fusion transformations along with array contractions to minimize memory are done without incurring any increase on the number of arithmetic operations.

### 3.4. Space-time transformation

If the memory minimization step is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system a space-time trade-off is performed. This is done by exploring different ways of adding redundant loops that enable additional fusion and array contraction. The redundant loops increase the amount of computation, but additional array contractions so enabled might reduce space requirements of intermediate temporaries. Loop tiling can be used with the redundant loops to allow additional space-time trade-off. The fusion graph framework has been used to develop a search procedure to seek the best choice of redundant loops and tile sizes that can fit the computation within the available storage while incurring a minimal computational overhead due to the redundant loops introduced [9].

### 3.5. Storage and data locality optimization

If the space requirement exceeds physical memory capacity, portions of the arrays must be moved between disk and main memory as needed, in a way that maximizes reuse of elements in memory. The same

considerations are involved in minimizing cache misses — blocks of data are moved between physical memory and the space available in the cache. Loop blocking is used to minimize disk-to-memory transfer overhead. The issue of tile-size optimization is discussed in [11].

### 3.6. Data distribution and partitioning

This component determines how best to partition the arrays among the processors of a parallel system. We assume a data-parallel model, where each tensor contraction is distributed across the parallel machine. The arrays are to be disjointly partitioned between the physical memories of the processors. The data distribution pattern that minimizes the total inter-processor communication in executing a sequence of tensor contractions is determined. The data partitioning issue is discussed in [10].

### 3.7. Code generation

The back end of the synthesis system provides the output as pseudo-code, Fortran or C code. The generated code can be either serial or parallel, using Global Arrays (GA). Though targeting a traditional message-passing or other programming model is also quite feasible, we have found that the abstraction of globally-addressable shared data provided by the GA programming model greatly simplifies conceptual and code generation issues involved in the interface between the TCE-generated code and the supporting infrastructure provided by existing quantum chemistry packages, such as NWChem. Depending on the circumstances, the synthesized code could also call highly-tuned, machine-specific Basic Linear Algebra Subprograms (BLAS) libraries, or optimized low-level functions from the existing quantum chemistry packages.

The TCE approach has already demonstrated tremendous productivity gains. Using the prototype version of the TCE, which does yet incorporate several optimizations, more than 20 different quantum chemical methods have been implemented in just a few weeks, many receiving their first-ever parallel implementation in this way [15]. At a very conservative estimate of three months of effort each it would have required more than five years of effort to implement all these methods by hand, representing a productivity increase on the order of 50-100 fold, not including the improvements in time to solution due to the availability of parallel implementations. The ratio size of the synthesized Fortran code to the input tensor contraction expressions (measured as number of characters of source code, excluding comments) is also typically around two orders of magnitude. Work is underway on the fully optimizing version of the TCE to implement optimizations targeted at enhancing the

performance and scalability of the synthesized parallel Fortran code.

## 4. Extended Global Arrays

A logical step to raise the level of abstraction of the GA+DRA model is to integrate the management of three layers of memory hierarchy -- distributed main memory, shared memory on the SMP node of a cluster, and secondary storage -- under a single programming interface in an environment which automatically manages the hierarchy through extensions to the compilers for traditional programming languages. This effort, which we call “Extended Global Arrays” (XGA), is currently in the design stage.

### 4.1. Global Arrays

The Global Arrays toolkit presents to the application developer a distributed data structure as a single object and allows access as if it resided in shared memory. These features help the developer raise the level of composition and increase code reuse. A higher level of composition reduces the amount of code that must be written and enables scientists to program in terms of physically meaningful concepts rather than low-level manipulation of distributed data and explicit communication. Thus, it makes scientists more productive and permits more time to be spent optimizing performance-critical algorithms and application kernels. GA programming model includes as a subset message passing; in particular, the programmer can use full MPI functionality on both GA and non-GA data. The library can be used in C, C++, Fortran 77, Fortran 90 and Python programs.

GA implements a shared-memory programming model in which data locality is managed by the programmer through explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory (DSM) models that provide an explicit acquire/release protocol. However, GA acknowledges that remote data is slower to access than is local data and therefore allows data locality to be explicitly specified and hence managed. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing or redundant data transfers present in some DSM solutions. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference.

The GA toolkit provides extensive support for controlling array distribution and accessing locality information. Both task-parallel and data-parallel programming styles are possible. Task parallelism is supported through the one-sided (non-collective) copy operations that transfer data between global memory (distributed/shared array) and local memory. In addition, each process is able to access directly data held in a section of a global array that is logically assigned to that process. Atomic operations are provided that can be used to implement synchronization and ensure correctness of updates of overlapping array sections. The data parallel computing model is supported through the set of collectively called functions that operate on either entire arrays or sections of global arrays. The set includes BLAS-like operations interfaces to the parallel linear algebra libraries such as Scalapack as well as the TAO optimization toolkit [7].

## 4.2. Disk Resident Arrays

The disk resident arrays (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage [NF1996]. It introduces the concept of a disk resident array - a disk-based representation of an array. It provides functions for transferring blocks of data between global arrays and disk arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized array communication) can be extended to programs that operate on arrays that are too large to fit into memory. By providing distinct interfaces for accessing objects located in main memory (local and remote) and on the disk, GA and DRA render visible the different levels of the memory hierarchy in which objects are stored. Hence, programs can take advantage of the performance characteristics associated with access to these levels.

## 4.3. SMP Arrays

So-called SMP Arrays (SA) can be used as a shared memory cache for latency sensitive distributed arrays in cluster environments based on collection of Symmetric Multiprocessor (SMP) nodes. Due to its cost effectiveness, SMP systems are used as building blocks for both commodity clusters as well as custom architectures (e.g., IBM SP, SGI Altix, NEC SX, Cray X1). SA arrays resemble global arrays except their scope is limited to an SMP node rather than entire parallel job running on a cluster. SA are related to the mirrored arrays, that were initially introduced as an extension to Global Array model in context of wide-area-network grid computing environments [23, 24, 25] and recently proposed for reducing communication overhead on

clusters [27]. In the latter context, shared memory mirroring is used to cache entire global arrays on every SMP node. The arrays are replicated across cluster nodes and distributed within each node. The goal is to take performance advantage of the shared memory, which constitutes the fastest interprocessor communication protocol, and use it as replacement for more expensive network communication. In the mirrored approach, the user is responsible for managing consistency of the cached data and collective operations on arrays are globally synchronized. The SA arrays do not involve global synchronization in collective operations and are created and managed independently on each SMP node.

## 4.4. Integrated Programming Framework

The evolution of programming models is driven by the fundamental tradeoffs between high productivity and performance requirements in context of evolving scalable architectures. On one hand, high productivity demands high-level of abstractions that insulate the programmer from specificity of the underlying hardware details and allow describe the underlying mathematical model in terms of collection of algorithms and appropriate data structures. However, achieving high performance and scalability is difficult if the essential characteristics of the hardware, in particular the memory hierarchy, are ignored.

Intelligent and automated management of data movement is a fundamental and unifying theme for the Extended Global Array interface we are developing. The goal is to have a single interface for managing data and high level representation of the mathematical algorithms operating on multidimensional arrays while the details on the underlying data movement between secondary storage, distributed memory, shared memory, and local memory are handled by the XGA implementation. XGA attempts to address this problem while relying on three elements:

- Compiler analysis and code transformation
- Performance model for GA, SA, DRA operations
- Information on resource availability and configuration (disk space, memory, processor affinity).

The basic idea is to translate XGA programs into SA/GA/DRA code while orchestrating data movement, caching, and redistribution so that the performance is maximized while satisfying the constraints on the available resources. XGA would allow from a single source to generate in-core and out-of-core codes while reducing the programmer effort and maintenance costs.

## 5. Discussion

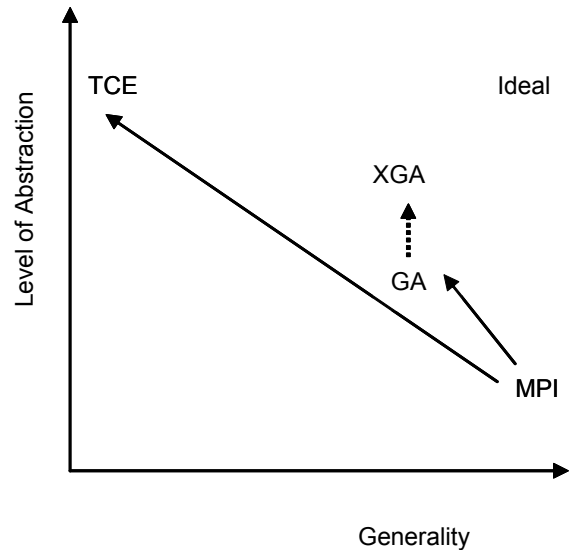
In this paper, we have discussed two efforts that we are currently engaged in, that seek to raise the level of abstraction offered to the developers of high-end software. Although message-passing with MPI can be used to develop and tune parallel programs in any application domain, we believe that the effort required to develop, validate and maintain very complex high-performance software is a deterrent and an impediment.

Historically, in the quantum chemistry domain, the need for higher-level abstractions to aid in coping with the complexity led to the development of the GA and DRA libraries; these libraries provide a programming model that has found many uses outside of the quantum chemistry domain as well. In the newer efforts described above, we are investigating other approaches to raising the level of abstraction while maintaining scalability and performance. The TCE is, once again, motivated by the needs of the quantum chemistry community, though in this case, the result is applicable to a relatively narrow domain because of the extremely high level of abstraction provided by the high-level language used. However, we believe that many of the approaches developed for the TCE can also be applied in the context of other more broadly applicable efforts at raising the level of abstraction in programming models for high-end computing. An example is the automatic memory hierarchy management aspect of XGA.

Figure 4 shows the relationships between these models in the two-dimensional abstraction/generality space. The third dimension of scalability can be made implicit when only considering models that achieve satisfactory levels of performance/scalability. The TCE uses GA in its implementation, but is not an *extension* of the GA model. The XGA effort, on the other hand is specifically an effort to extend the GA model to higher levels of abstraction. The two very different approaches to raising the level of abstraction of the programming model we have presented here have clear benefits to software productivity (some already realized, in the case of the TCE, and more expected following further development, in the case of both TCE and XGA). They also demonstrate the transferability of ideas in this space (automatic memory hierarchy management from TCE moving into XGA). Although the “holy grail” of a programming model with high level of abstraction, high generality and high scalability may be a distant goal, broad exploration of this space is likely to yield many new ideas with broad applicability and lead to the development of programming models that raise programmer productivity while delivering high performance.

In the future, we plan to explore ways of moving along the dimension of generality, while again maintaining scalability. Other approaches might seek to proceed along different paths in the three dimensional space of generality, abstraction-level and scalability, with the ultimate goal of developing very general-purpose programming language frameworks that offer high levels of abstraction and high scalability. However, there is a potential problem with approaches where the initial starting point has inadequate scalability, as exemplified by the experience with High-Performance Fortran [2]. A significant problem with HPF was that users were unable to achieve high performance for many applications with the initial releases of the compilers from vendors. This was because challenging compiler optimization problems had to be solved before performance could be delivered for a range of applications and this resulted in a vicious cycle. Vendors did not see it worthwhile investing in compiler optimization technology unless they perceived user demand; there was insufficient user demand without scalable performance.

The end goal of programming language models that rank high in all three dimensions is an extremely challenging one. Significant advances in compiler technology will be essential in achieving high scalability with general-purpose programming models offering high levels of abstraction. The sustained vision and support of governmental funding agencies towards this goal will be crucial. It will be very important for funding agencies to



**Figure 4:** GA and TCE are programming models at different levels of abstraction and generality, developed to make high-end software development easier than using MPI. XGA is a proposed model to further raise the level of abstraction above GA



engage in a long-term plan to support a variety of efforts that seek to advance the state-of-the-art in programming models offering high levels of abstraction, generality and performance. Progress will be greatly facilitated by sustained and strong interaction between application developers and systems software developers in vertically integrated teams, with expertise cutting across multiple layers: from the applications layer, programming models/frameworks layer, run-time layer, communications layer and hardware architecture.

## Acknowledgements

We wish to thank all of the members of the Tensor Contraction Engine collaboration and the Global Arrays team, without whose efforts this paper would not have been possible. This research is sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, by the Office of Mathematical, Information, and Computational Sciences (MICS) of the U. S. Department of Energy Office of Science, Environmental Molecular Sciences Laboratory at PNNL, and by the National Science Foundation through the ITR program. ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. PNNL is managed by the Battelle Memorial Institute for the U. S. Department of Energy under Contract No. DE-AC06-76RLO 1830.

## References

[1] Global Array Toolkit Home Page, <http://www.emsl.pnl.gov/docs/global>.

[2] High Performance Fortran Forum. High Performance Fortran Language Specification, Ver. 2.0, 1997. <http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/>.

[3] Message Passing Interface Forum, "MPI: a message-passing interface standard," Intl. J. Supercomputer Appl. and High Perf. Comp. 8, 159-416 (1994).

[4] OpenMP: Simple, Portable, Scalable SMP Programming, <http://www.openmp.org>.

[5] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepply, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual, Technical Report ANL-95/11 Revision 2.1.5, Argonne National Laboratory, 2002.

[6] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. Proc Supercomputing 2002, 2002.

[7] Steve Benson, Lois Curfman McInnes, Jorge J. More, and Jason Sarich. TAO Users Manual, Technical Report

ANL/MCS-TM-242-Revision 1.5, Argonne National Laboratory, 2003.

[8] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. Parallel Programming in OpenMP. Morgan Kaufman, 2000, ISBN 1-55860-671-8.

[9] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), June 2002, pp. 177-186.

[10] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS), Apr. 2003.

[11] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. Proc. of the Intl. Conf. on High Performance Computing, Dec. 2001, Lecture Notes in Computer Science, Vol. 2228, pp. 237-248, Springer-Verlag, 2001.

[12] Alain Darte. On the complexity of loop fusion. In Parallel Computing, Vol. 26, No. 9, 1175-1193.

[13] Holger Dachselt, Jarek Nieplocha, and Robert Harrison. An Out-of-Core Implementation of the COLUMBUS Massively-Parallel Multireference Configuration Interaction Program. Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, San Jose, CA, pp. 1-10, 1998.

[14] High Performance Computational Chemistry Group, NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.5 (2003), Pacific Northwest National Laboratory, Richland, WA 99352, <http://www.emsl.pnl.gov/docs/nwchem>

[15] S. Hirata, Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories., The Journal of Physical Chemistry A, ASAP Article 10.1021/jp034596z S1089-5639(03)04596-1.2.

[16] Ricky A. Kendall, Edo Apra, David E. Bernholdt, Eric J. Bylaska, Michel Dupuis, George I. Fann, Robert J. Harrison, Jialin Ju, Jeffrey A. Nichols, Jarek Nieplocha, T. P. Straatsma, Theresa L. Windus, and Adrian T. Wong, "High Performance Computational Chemistry: Overview of NWChem, a Distributed Parallel Application," Comp. Phys. Comm. 128, 260-283 (2000).

[17] C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. Proc. 12th LCPC Workshop San Diego, CA, Aug. 1999.

- [18] C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Par. Proc. Lett.*, (7) 2, pp. 157-168, 1997.
- [19] T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pp. 47-109, Kluwer Academic, 1997.
- [20] J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.), *Encyclopedia of Computational Chemistry*, Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115-128, 1998.
- [21] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations. *Proc. 6th Symposium on the Frontiers of Massively Parallel Computing*, Anapolis, MD, March 1996.
- [22] Jarek Nieplocha, Ian Foster, and Rick A. Kendall. ChemIO: High-Performance Parallel I/O for Computational Chemistry Applications, *Intl. J. Supercomp. Apps. High Perf. Comp.* 12, 345-363 (1998).
- [23] J. Nieplocha and R. J. Harrison. Shared memory NUMA programming on I-WAY. *5th International Symposium on High Performance Distributed Computer (HPDC-5)*, pp. 432-441, 1996.
- [24] J. Nieplocha and R. J. Harrison. Shared-Memory Programming in Metacomputing Environments: The Global Array Approach. *J. Supercomputing*, 11, 119-136 (1997).
- [25] J. Nieplocha, R. J. Harrison, and I. Foster. Explicit Management of Memory Hierarchy. In L. Grandinetti, J. Kowalik, and M. Vajtersic (Eds.), *Advances in High Performance Computing*, Kluwer Academic, NATO ASI Series #30, pp. 185-198, 1996.
- [26] J. Nieplocha, R.J. Harrison, and R.J. Littlefield, "Global Arrays: A Non-Uniform-Memory-Access Programming Model for High-Performance Computers," *J. Supercomp.* 10, 169 (1996).
- [27] B. Palmer, J. Nieplocha, and E. Apra. Shared Memory Mirroring for Reducing Communication Overhead on Commodity Networks. *5th International Conference on Cluster Computing (CLUSTER 2003)*, Kowloon, Hong Kong, December 2003.