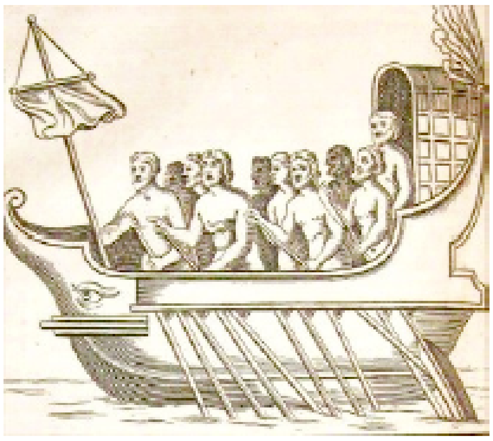
MareNostrum, the name of "our" supercomputer, means literally "our sea". It was a Roman name for the Mediterranean Sea on which Barcelona is a port.


Ancient Roman Ship

Optimizing programming models for massively parallel computers

By Montse Farreras

Advisor:
Toni Cortés

A dissertation submitted in partial fulfillment of the requirements for the degree of:

Doctor per la Universitat Politècnica de Catalunya

Barcelona (Spain)
2008

Technical University of Catalunya

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: Optimizing programming models for massively parallel computers ....
Autor de la tesi: Montse Farreras Esclusa..

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, ... de/d'.. de ....

El President	El Secretari
....	....
(nom i cognoms)	(nom i cognoms)

El vocal	El vocal	El vocal
....	....	....
(nom i cognoms)	(nom i cognoms)	(nom i cognoms)

Abstract

Since the invention of the transistor, clock frequency increase was the primary method of improving computing performance. As the reach of Moore's law came to an end, however, technology driven performance gains became increasingly harder to achieve, and the research community was forced to come up with innovative system architectures. Today increasing parallelism is the primary method of improving performance: single processors are being replaced by multiprocessor systems and multicore architectures.

The challenge faced by computer architects is to increase performance while limited by cost and power consumption. The appearance of cheap and fast interconnection networks has promoted designs based on distributed memory computing. Most modern massively parallel computers, as reflected by the Top 500 list, are clusters of workstations using commodity processors connected by high speed interconnects.

Today's massively parallel systems consist of hundreds of thousands of processors. Software technology to program these large systems is still in its infancy. Optimizing communication has become a key to overall system performance. To cope with the increasing burden of communication, the following methods have been explored:

(i) Scalability in the messaging system: The messaging system itself needs to scale up to the 100K processor range.

(ii) Scalable algorithms reducing communication: As the machine grows in size the amount of communication also increases, and the resulting overhead negatively impacts performance. New programming models and algorithms allow programmers to better exploit locality and reduce communication.

(iii) Speed up communication: reducing and hiding communication latency, and improving bandwidth.

Following the three items described above, this thesis contributes to the improvement of the communication system (i) by proposing a scalable memory management of the communication system, that guarantees the correct reception of data and control-data, (ii) by proposing a language extension that allows programmers to better exploit data locality to reduce inter-node communication, and (iii) by presenting and evaluating a cache of remote addresses that aims to reduce control-data and exploit the RDMA native network capabilities, resulting in latency reduction and better overlap of communication and computation.

Our contributions are analyzed in two different parallel programming models: Message Passing Interface (MPI) and Unified Parallel C (UPC). Many different programing models exist today, and the programmer usually needs to choose one or another depending on the problem and the machine architecture. MPI has been chosen because it is the de facto standard for parallel programming in distributed memory machines. UPC was considered because it constitutes a promising easy-to-use approach to parallelism. Since parallelism is everywhere, programmability is becoming important and languages such as UPC are gaining attention as a potential future of high performance computing.

Concerning the communication system, the languages chosen are relevant because, while MPI offers two-sided communication, UPC relays on a one-sided communication model. This difference potentially influences the communication system requirements of the language. These requirements as well as our contributions are analyzed and discussed for both programming models and we state whether they apply to both programming models. [1]

[1]This work has been partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, TIN2004-07739-C02-01 and TIN2007-60625, and by the BSC-IBM collaboration agreement SoW Scalability for BGL. I acknowledge the European Center for Parallelism of Barcelona (CEPBA), the Barcelona Supercomputing Center (BSC), the Lawrence Livermore National Laboratory, the IBM Poughkeepsie Benchmarking Center and the IBM Research T.J. Watson for supplying the computing resources for the experiments presented in this thesis.

Agraïments

Cap al Febrer del 2002, vaig iniciar una experiència que poc em pensava que tingués un impacte tan gran en la meva vida. Aquests anys de formació com investigadora han sigut una experiència molt enriquidora, tan des del punt de vista professional com personal.

Durant aquests gairebé 7 anys, he tingut el plaer de conèixer moltíssimes persones i cadascuna d'elles m'ha aportat alguna cosa, la llista de persones a qui m'agradaria donar les gràcies és tan llarga que em caldria tot un annex a la tesis, i com sempre escric això amb presses així que si em descuido algú no m'ho tingueu en compte.

En primer lloc el meu *advisor*, el Toni Cortés, que ha resistit en tot moment al meu costat a les verdes i a les madures. Li vull agrair el suport que m'ha donat, el guiar-me quan m'he encallat i la confiança que m'ha demostrat. I sobretot el que més m'ha ajudat: el seu optimisme i bon humor.

També vull mencionar el Jesus Labarta i l'Eduard Ayguadé, per l'experiència que m'han transmès, i pels seus comentaris en diferents etapes de la tesis que m'han ajudat a millorar-la. I al Xavier Martorell per la seva disposició a ajudar en tot moment i pel seu suport tècnic i logístic.

En segon lloc el George Almasi i el Calin Cascaval amb qui he treballat els darrers quatre estius durant les meves estades a IBM T.J. Watson. Són els investigadors més brillants que he conegut fins al moment i també unes grans persones. D'ells he après molt i són un gran estímul professional per continuar fent recerca.

Tot l'equip de IBM T.J. Watson mereix una menció, per omplir de bones vivències els meus estius a Yorktown Heights i pels seus bons consells: Jose B., Jose C i sobretot a Gabor, a qui també vull agrair les correccions. També l'equip amb qui he colŢlaborat de IBM Toronto, Kit, Philip i Ettore.

En tercer lloc tots els companys i amics que he fet al departament, la llista és molt llarga. Els companys de doctorat des dels inicis de la sala CEPBA: Juanjo, Alex, també Rogeli, Raul i Vicenç. Els companys d'aventures: Carles, Jesus. El meu company d'assignatura: Pau Bofill un exemple a seguir. Les meves companyes: Ana, Yolanda i

Marta. Al Pau per les converses i cafès. Els companys de Parallel Models: Jordi, Javi, Xavi i Roger. Els administradors de sistemes tant del DAC com del BSC. I totes aquelles persones amb qui he compartit el menjador, festes, esmorzars o bons moments en general.

Especialmet vull remarcar el David, el meu *roomie* amb qui he compartit més hores que ningú, sempre disposat a donar un cop de mà a les 7h del matí si cal, *a on no arriba un roomie* ..., i que m'ha marcat clarament el camí cap al final de la tesis.

En quart lloc, la meva família, per què malgrat tot és la millor família del món. I a tots els meus amics que sort n'he tingut.

I per últim vull dedicar la tesis a la persona més especial de la meva vida: al Sergi, a ell vull agrair tots els bons moments que hem compartit però també la paciència que ha tingut per haver viscut molt d'aprop tot l'esforç que he dedicat a aquesta tesis.

Acknowledgements

In February 2002, I took a challenge that has had an impact in my life: undertake a PhD in Computer Science. These years of research training have provided me with an intellectually enriching experience, from both professional and personal point of view.

Along the last seven years, I have had the pleasure to meet many people, each of them has brought some novelty and good to my life. The list of people whom I would like to thank is such a long one that I would need to add a new annex to the thesis, and as usual I rush so please do not get upset with me if I am forgetting someone.

First of all, my *advisor*, professor Toni Cortés. He has held out at my side for the good and the bad times. I am grateful to him for his support, for his guidance when I was blocked, and for his faith in me. And specially, his optimism and sense of humor have been really helpful.

I would also like to mention Jesus Labarta and Eduard Ayguadé. They have imparted their experience, and given me remarks at different stages of the development of this thesis that have improved it. Also thanks to Xavi Martorell, who has always been ready to help me with any kind of technical or logistic support.

Secondly, I have been working for the last four years during my summer internships with George Almasi and Calin Cascaval. They are the most brilliant researchers I have met so far and also very kind people. I have learned a lot from them and they inspire and encourage me to keep doing research. Thanks for your time and dedication.

The whole team at IBM T.J. Watson deserves a place here, to fill up my summers in Yorktown Heights with very nice memories and to give me good advices: Jose B., Jose C and specially Gabor, of whom I appreciate his reviews. I have also been collaborating with the compiler team in IBM Toronto: Kit, Philip and Ettore.

Thirdly, I want to mention all my department colleagues and the list is long. My PhD mates from the very beginning at the CEPBA room: Juanjo, Alex, also Rogeli, Raul and Vicenç. My adventure buddies: Carles and Jesus. My workmate in the classroom: Pau Bofill, who is a role model. My girlfriends: Ana, Yolanda and Marta. My classmate Pau for the nice coffee breaks. My teammates: Jordi, Javi, Xavi i Roger. The system administrators, from DAC and from BSC. And all the people whom I have shared beautiful moments, breakfast, lunch, dinner or parties.

I would like to emphasize David, my official *roomie*, with whom I have shared more office hours than with anybody else. He is always ready to lend a hand, even at 7am in the morning: *a on no arriba un roomie...*. He has show me the way to graduation.

Fourthly, my family, despite of the difficulties, they are the best possible family in the world. And all my friends, I am very happy to count on them.

And finally, I would like to dedicate this thesis to the most special person in my life: Sergi. I thank him for all the great time we have spent together and I also appreciate his patience because he has followed my efforts on this work closely.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time [196].

In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory. His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV. The key to its design was a fairly high parallelism, with up to 256 processors, which allowed the machine to work on large data sets in what would later be known as vector processing. However, ILLIAC IV was called "the most infamous of Supercomputers", because only one fourth of the project goals were completed, it took 11 years and cost almost four times the original estimate [154]. When it was finally ready to run its first real application in 1976, it was outperformed by existing commercial supercomputers such as the Cray-1.

In the 1980s it was believed computer performance was best improved by creating faster and more efficient processors. This idea was challenged by parallel processing in the 90s, and attention turned from vector processors to massive parallel processing systems with thousands of ordinary CPUs. Since then, there has been an increasing trend to move away from shared memory parallel supercomputers toward networks of computers (clusters). This transition has been mostly driven by the fast improvement in the availability of commodity high-performance networks and highly influenced by power and cost limitations.

As a result, scalable computing clusters, ranging from cluster of PCs or workstations to clusters of Shared Memory Processors (SMPs) have become a low-cost standard platform for high-performance and large-scale computing. Emphasis has moved away somehow from fast sequential processing toward fast communications and distributed computing.

1.1.1 Future trends in High Performance Computing

Nowadays in the highlights of top500 list, recently updated in June 2008, we read that 400 (from 500) systems are labeled as clusters, making this the most common architecture in the TOP500 with a stable share of 80 percent. Moreover, looking at the TOP50, it is stated that the dominant architectures are custom build massively parallel systems (MPPs) with 56 percent ahead of commodity clusters with 40 percent.

The headline of TOP500 introduces: *The new number one system,* **Roadrunner**, *located at Los Alamos National Laboratory (LANL) built by IBM broke (as first system ever) the petaflop/s Linpack barrier with 1.026 petaflop/s. Roadrunner is based on 6,562 dual-core AMD Opteron chips plus 12,240 Cell chips connected with a commodity*

InfiniBand network.

Some other staggering statistics from the last released top500: *The entry level to the list moved up to the 9.0 Tflop/s mark on the Linpack benchmark, compared to 5.9 Tflop/s six months ago.* This is the largest turnover rate in the 16 years of the history of the TOP500 project. And *The average concurrency level in the TOP500 is 4,850 cores per system up from 3,290 six month ago*

If we look at the news, along the years we find more evidence that innovative system architectures go for distributed designs where communication becomes a key issue: *I.B.M. Supercomputer Sets World Record for Speed By John Markoff Published.* September 2004: *An I.B.M. machine has reclaimed the title of world's fastest supercomputer, overtaking a Japanese computer that had caused shock waves at United States government agencies when it set a computing speed record in 2002.* The machine was BlueGene/L with 65,536 dual-core nodes (later upgraded to 106,496 nodes) connected through a High speed 3-dimensional torus network. It took the place of Earth Simulator, a highly parallel vector supercomputer system consisting of 640 nodes with 8 vector processors connected by 640x640 single-stage crossbar switches.

June 2005: *The 5 spot was captured by the upgraded MareNostrum system at the Barcelona Supercomputer Center. It is an IBM BladeCenter JS20-based system with a Myrinet connection network and achieved 27.91 TFlop/s - just ahead of a third Blue Gene system owned by ASTRON and installed at the University of Groningen with 27.45 TFlop/s.* The four machines, MareNostrum, Earth Simulator, Blue Gene and Roadrunner among many others are based on distributed memory and High Performance Networks.

Apparently, the focus has moved to distributed computing and MPPs lead the HPC market. Performance is then, delivered by exploiting the application parallelism in these massively parallel systems connected through High-performance Networks and two issues become critical in order to improve efficiency and gain scalability: (i) Optimize the communication management (ii) and provide a suitable parallel programming model.

The aim of this thesis is to improve scalability of programming models for current massively parallel computers, as a key issue, we will focus on the optimization of communications, and two different parallel programming models, both suitable for Large Scale Machines, will be studied. Next section examines the future trends on programming models for High Performance Computing.

1.1.2 Programming trends in HPC

Since many real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software, and a supporting architecture that are naturally parallel.

Supercomputers are used for highly calculation-intensive tasks such as problems involving quantum mechanical physics, weather forecasting, climate research, molecular modeling, physical simulations, cryptanalysis, and the like.

Although the technology of clusters is currently being deployed, the development of parallel applications is really a complex task. Adequate software and tools need to be provided. Over the years, many research on High Performance Computing has been focused on analyzing, improving or creating parallel programming models adapting to the new computer architectures with the ultimate goal of improving productivity of programming. As a result, nowadays, there is a a wide spectrum of different *parallel programming environments or models* to exploit parallel processing.

We understand parallel programming model as a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. Parallel models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or as complete new execution models. They are also roughly categorized for two kinds of systems: shared-memory system and distributed-memory system, though the lines between them are largely blurred nowadays.

Also, from the language point of view, we could broadly identify two main approaches to parallelism: message passing (explicit communication) and data-parallel (implicit communication). The first approach is treated as an assembler for parallel programming, since the programmer is responsible for most of the parallelization effort such as task decomposition, mapping tasks to processors, and the communication structure. This approach is based on the assumption that the user is often the best judge of how parallelism can be exploited for a particular application. While in the second approach the user has the illusion of a shared address space and it offers ease of programming since the user does not specify, and thus cannot control, the scheduling of calculations and/or the placement of data. A programming model is usually judged by its expressibility and simplicity, which are by all means conflicting factors.

Due to the difficulties in automatic parallelization, people have to choose a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform. Moreover, parallelism is everywhere and people writing parallel programs may not be computer specialists any longer, as a result programmability is winning attention, becoming a required characteristic of new programming models.

In this thesis we take two different parallel programming models as our case studies: the well known communication library *Message Passing Interface* (MPI), as an explicit communication programming model. And a parallel extension to the C language: *Unified*

Parallel C (UPC), where communication is implicit.

Whereas MPI is a communication library, UPC is a parallel language, it relies on a compiler that translates code to a Run Time System (RTS) library in charge of the communication.

Explicit communication: MPI

Message passing libraries allow efficient parallel programs to be written for distributed memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving packets of data. Various message passing environments were developed in early 80s. Some were developed for special purpose machines such as the Caltech's N-cube, some were developed for networks of UNIX workstations, e.g., the Argonne's P4 library and PVM (Parallel Virtual Machine) [179], which was one of the most popular, from the Oak Ridge National Laboratories. The Ohio Supercomputer Center developed a message passing library called LAM [118]. By early 1992 it was clear that the authors of these numerous libraries were duplicating their efforts.

In late 1992 a meeting was set up during the Supercomputing 92 conference and the attendants agreed to develop and then implement a common standard for message passing that would incorporate all the interesting existing ideas and build on them. And this is how MPI , the Message Passing Interface, was born. Nowadays MPI has become the **de facto standard** of parallel models for distributed memory systems and it is by far the most popular due to its portability, flexibility and good performance. MPI defines an API that allows portability while the library implementation permits multiple platform dependent optimizations. As a result, parallel applications can have good performance without having to port them to different architectures.

The first MPI specification draft [130] was introduced at Supercomputing on 1994. At this time the top MPP on the TOP500 list [188], coming from Sandia National Labs, had 3800 processors. Nowadays, the top two on the list are: Blue Gene Light, recently updated, it is installed at Lawrence Livermore National Labs, it has 212992 cores, and Roadrunner, recently created, with 122400 cores. And during the last twelve years the peak performance has increased from 180Gflops to 1,026Pflops (more than one quadrillion floating point operations per second, achieved this June 2008 by Roadrunner). The trend is to increase the number of processors, connected through different topologies of high-speed networks to gain performance, and parallel models need to catch up.

Despite of the fact that MPI specification was designed to be performant and scalable, the truth is that current implementations still need to be adjusted to this evolution on the new architectures and MPI scalability proves to be a weakness of current MPI implementations.

Implicit communication: UPC

Concerning the data-parallel models, this approach has been followed by parallel languages such as SISAL and PCN, but they found little favor with application programmers. This is because users are reluctant to learn a completely new language for parallel programming. They really would prefer to use their traditional high-level languages (like C and Fortran) and try to recycle their already available sequential software. For these programmers, the extensions to existing languages or runtime libraries are a viable alternative.

Taking this into account the partitioned global address space (PGAS) parallel programming models appeared. A PGAS programming model is a Single Program Multiple Data (SPMD) paradigm that presents the user with both a private memory space and a shared memory space. By default, all variables are private and every instruction is executed by each thread. However, the user may declare that some data be shared and that loop iterations be divided among threads. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular thread, thereby exploiting locality of reference. The PGAS model is the basis of Unified Parallel C, Co-array Fortran, and Titanium (Java).

Unified Parallel C (UPC) is an extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (clusters). The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC fixes the amount of parallelism at program startup time.

In order to express parallelism, UPC extends ISO C 99 with the following constructs: an explicitly parallel execution model, a shared address space, synchronization primitives and a memory consistency model and memory management primitives.

The UPC language, as opposed to MPI, is quite new (the first language specifications appeared on 2001) and it evolved from experiences with three other earlier languages that proposed parallel extensions to ISO C 99: AC, Split-C, and Parallel C Preprocessor (PCP). UPC is not a superset of these three languages, but rather an attempt to distill the best characteristics of each. UPC combines the programmability advantages of the shared memory programming paradigm and the control over data layout and performance of the

Table 1.1 MPI UPC Programming Models (PM) comparison

MPI	**UPC**
SPMD	SPMD
Static (MPI-1) Dynamic (MPI-2)	Static threads
Shared and distributed memory	Shared and distributed memory
Specified Data distribution	Specified Data distribution
Message Passing PM	Shared Memory PM
Explicit communication	Implicit communication
Two-sided	One-sided

message passing programming paradigm.

Regarding UPC performance, UPC community emphasized performance and scalability when the language was designed, and there are promising preliminary results [75]. However UPC scalability and efficiency still need to be proved.

MPI versus UPC Comparison

Table 1.1 summarizes the most relevant characteristics of both languages. Both languages are built on the Single Program Multiple Data (SPMD) paradigm. In MPI-1 and UPC the number of threads is static, defined at startup time, while dynamic creation of processes has been enabled in MPI-2. In both models a process may access its private memory and share data with the other processes, and the data distribution has to be specified by the programmer. The difference is that in UPC the user is presented with a global shared address space while MPI provides the user with a set of primitives allowing explicit communication among processes. In UPC the communication is hidden by the compiler and runtime while in MPI it is made explicitly by the programmer.

From the communication point of view, which is the main concern of this thesis, one relevant difference must be emphasized: MPI offers a two-sided communication model, a process sends a message to another process that has to explicitly receive it (`send, receive`), and both processes are involved in the communication (and synchronized). Whereas UPC is based on one-sided communication and only the process that initiates the communication needs to be involved (`get, put`).

1.2 Global picture

We have stated that optimized communication is crucial for scalability of large scale architectures. The aim of this thesis is to improve scalability of programming models for current massively parallel computers. As a key issue, we will focus on the optimization of communication, and two different parallel programming models will be considered: MPI following the explicit communication approach and UPC following the implicit approach. As long as they run on distributed memory machines, both languages need an underlying messaging system that is performant and scalable.

We state that in spite of the fact that UPC and MPI take rather different approaches, as far as performance and scalability concerns they may have a lot in common. In this study we will find out whether both models suffer from the same problems, and the same optimizations could be applied or on the contrary we need different solutions to end up with a scalable model. Notice that if all the experiences in one model could be applied to the other one, the contribution is doubled.

Therefore the goal of this work is to analyze the weaknesses of parallel models for massively parallel computers and propose optimizations related to the communication system in terms of scalability and efficiency.

Figure 1.1 shows the global picture of our work. Two main directions have been taken:

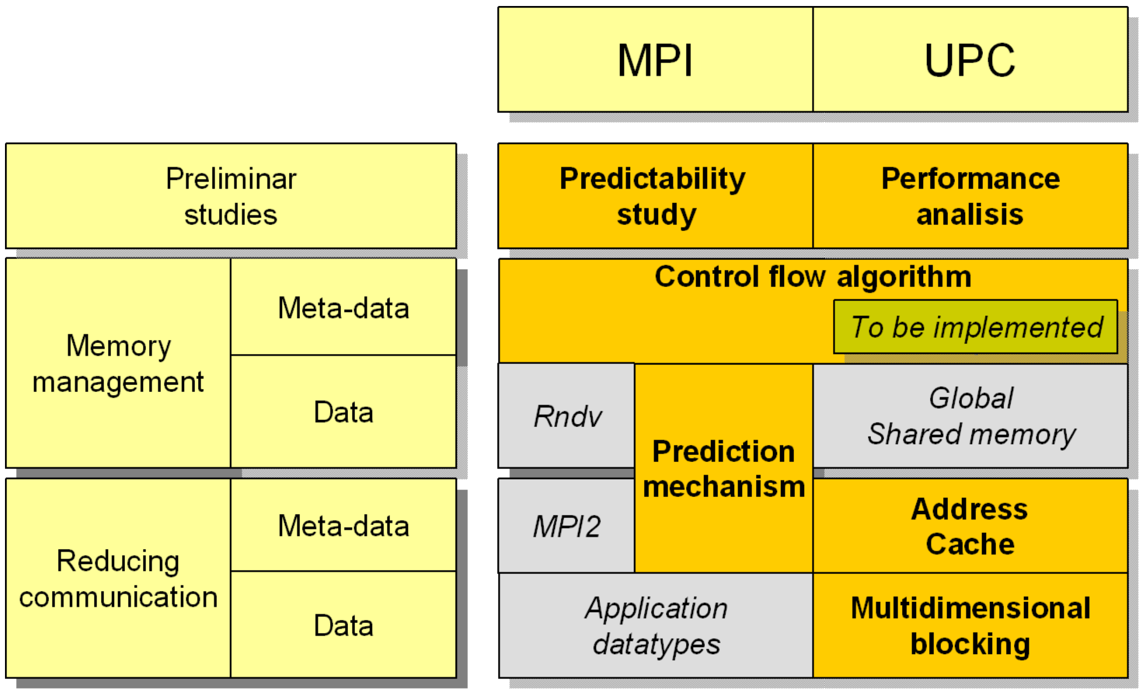


Figure 1.1 Global Picture of our thesis.

- **Optimize the communication system memory management** to make it scalable to hundreds of thousands of nodes. And we distinguish between: (i) The memory management required to receive and process the application *data*. And (ii) The memory management for *meta-data* (aka. *control data*), we call meta-data the information required to manage the data transfer but not containing application data.

- **Reduce the communication** between nodes, will obviously result in a better efficiency and scalability. Here we also distinguish between: (i) Communication of *Data* and (ii) *Meta-data* messages.

Each aspect is being considered for both the MPI and UPC parallel models. Our contributions are shown in bold face, italics denotes either the problem was already being considered in the bibliography or the language did not show this problem and in the case of the control flow algorithm it means that although the same solution applies it has not been yet implemented.

As shown in the figure, a preliminary study of MPI predictability was done to come up with a **prediction mechanism** that accomplishes two goals, it improves the communication system memory management, guaranteeing message reception for large scale machines and reduces the control-data communication. To solve the problem of control-data memory management a **memory management control flow algorithm** was proposed. This constitutes the first contribution of this thesis.

In UPC a preliminary study was also done to assure that UPC was able to scale to hundreds of thousands of nodes. Then an extension to the language was proposed, **Multidimensional Blocking in UPC** to better exploit locality and reduce communication of application data. This is our second contribution.

And finally, a mechanism to reduce control-data communication was proposed in UPC, by means of a **Remote address Cache**, accomplishing two goals: First, it reduces control-data communication, similar to the prediction mechanism in MPI. And second, it exploits the RDMA network capabilities whenever possible. It results in an improvement in terms of data communication latency. This is our third contribution.

1.3 Contributions

The main contributions of this thesis can be outlined as follows. We:

- Scale up the communication memory management: (i) *Prediction to scale up the communication system memory management* (ii) *Control flow for control data*

- Reduce communication: we introduce the *Multidimensional blocking: a language extension to reduce communication*

- Speed up communication: we present the *Remote Address Cache: Scalable RDMA performance for PGAS languages*.

1.3.1 Scale up the communication memory management

As we have seen the current trend on parallel computing is moving from big shared memory machines to large scale distributed memory machines connected through high performance networks. An increase in the number of nodes directly implies an increase in the communication requirements in terms of robustness, scalability and efficiency.

Moreover applications become more complex and resource demanding which also affects communication by increasing the amount of communication required and the communication system is stressed.

The communication system needs to be designed with scalability in mind. It needs to handle the correct reception of messages, as well as to take care of unexpected message reception despite of current resource availability.

Any message arriving before being required by the application needs to be stored somewhere into the library and solid and efficient memory management becomes essential for scalability. The scalability problem arises due to fact that available messaging systems take assumptions about resource availability (memory) which are not always true, specially considering the system architecture of current massively parallel computers.

Our *first contribution* focuses on **improving the memory management in the communication system**, dealing with memory shortage in two different scenarios: control-data and application data. This contribution has been split in two chapters:

- In the first chapter we focus on user's data message reception. In most current implementations, preference is given to the performance and control messages are avoided, so a message may be sent without receiver's permission and it is not guaranteed that the receiver will have enough memory to keep this message. Two solutions are provided: first, **prediction** was used to **improve communication system memory management on user's data reception**. The communication pattern predictability was studied showing high predictability rates. And a mechanism to guarantee the correct message reception by predicting the communication pattern and allocating the required resources in-advance was introduced, this mechanism was also reducing the amount of control-data exchanged between processes. And second, a simpler solution based on a three-way handshake protocol is also considered and analyzed.

- The second part of this contribution, *Control Flow for control data*, focuses on **short-memory MPPs memory management for meta-data reception**. For every message arrival, control-data needs to be saved in order to manage the message. When memory is short, thousands of control-data messages can flood up the receiver's memory, specially in the new era of supercomputers having very hard memory restrictions (BG/L, ASCI-Red, ASCI-White). In this chapter we present a new memory management protocol that guarantees scalability of messaging systems for large scale supercomputers.

MPI has been chosen to carry on this study since it is the most popular parallel language widely used among the HPC community and portable. MPICH has been chosen because it is an open source and broadly used implementation of the MPI standard.

The work performed in this area has resulted in the following publications:

[82] Felix Freitag, Jordi Caubet, Montse Farreras, Toni Cortes, Jesus Labarta. Exploring the Predictability of MPI Messages. IPDPS '03: Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)

[84] Felix Freitag, Montse Farreras, Toni Cortes, Jesus Labarta. Predicting MPI Buffer addresses. ICCS2004: The International Conference on Computer Sciences 2004 (ICCS'04)

[80] Montse Farreras, Toni Cortes, Jesus Labarta, George Almasi. Scaling MPI to short-memory MPPs such as BG/L. ICS06: Proceedings of the 2006 International Conference on Supercomputing (ICS'06)

1.3.2 Multidimensional blocking: a language extension to reduce communication

In the era of large scale machines, system-optimization is winning attention and performance is delivered by exploiting application parallelism. Despite the more and more efficient modern High-performance Networks being deployed, time spend in communication reduces application performance and it is crucial to provide appropriate parallel programming models that exploit data locality as much as possible.

Partitioned Global Address Space (PGAS) languages offer an attractive, high productivity programming model for programming large-scale parallel machines. PGAS languages, such as Unified Parallel C (UPC), combine the simplicity of shared-memory programming with the efficiency of the message-passing paradigm by allowing users' control over the data layout. PGAS languages distinguish between private, shared-local, and shared-remote memory, with shared-remote accesses typically much more expensive than shared-local and private accesses, especially on distributed memory machines where shared-remote access implies communication over a network.

The *second contribution* of our thesis aims to **reduce communication, by a simple extension to the UPC language that allows for better control of locality**, and therefore performance, in the language.

This work has been carried on in the UPC language because MPI, as an explicit programming model, distributes the “shared" data explicitly among the processes, all the responsibility is left to the programmer and the complexity of the program increases substantially.

A preliminary study was performed to evaluate the use of a shared memory programming language, Unified Parallel C (UPC) on large scale distributed memory machines. We demonstrate not only that shared memory programming for hundreds of thousands of processors is possible, but also that with the right support from the compiler and runtime system, the performance of the resulting codes is comparable to MPI implementations. A good balance in productivity and performance is demonstrated.

The work performed in this area has been done in collaboration with IBM. The proposed language extension, although at this point it is not yet part of the standard UPC language specifications, is supported by the IBM XLUPC compiler and this contribution will be part of the final product to be released in December 2009.

This contribution results from a collaborative effort involving work in two areas: the UPC compiler and the UPC Run Time System (RTS). For the purpose of this thesis we present the proposed language extension and the necessary support in the UPC RTS.

This work has resulted in the following publications:

[17] Christopher Barton, Calin Cascaval, George Almasi, Yili Zheng, Montse Farreras and Jose Nelson Amaral. Shared Memory Programming for Large Scale Machines. In PLDI 2006: ACM SIGPLAN Conference on Programming Language Design and Implementation

[16] Christopher Barton, Calin Cascaval, George Almasi, Rahul Garg, Jose Nelson Amaral and Montse Farreras. Multidimensional Blocking in UPC. LCPC 2007: International Workshop on Languages and Compilers for Parallel Computing

And two High Performance Computing (HPC) Challenge Competition Awards:

[34] C.Cascaval, C.Barton, G.Almási, Y.Zheng, M.Farreras, P.Luk, R.Mak. The UPC/BlueGene Class II Submission to the HPC Challenge Award Competition. HPCC

2005.

[35] C. Cascaval, G. Almási, C. Barton, E. Tiotto, G. Dózsa, M. Farreras, P. Luk, T. Spelce. HPC Challenge 2006 Awards Competition: xlUPC on BlueGene/L. HPCC 2006.

1.3.3 Remote Address Cache: Scalable RDMA performance for PGAS languages

Parallel programming for both multicore and large scale parallel machines is becoming evermore challenging; adequate programming tools offering both ease of programming and productivity are essential.

PGAS languages provide a unique programming model that can span shared-memory multiprocessor (SMP) architectures, distributed memory machines and cluster of SMPs. User can program large scale machines with easy-to-use, shared memory paradigms.

Moreover, PGAS languages are based on one-sided communication, which offers some advantages in terms of performance over the traditional two-sided communication model mostly used in MPI. In a two-sided model, the processes involved in the communication need to synchronize. In contrast, one-sided model splits communication from synchronization offering primitives for communication (`get`, `put`) and for synchronization (`barrier`, `fence`). Which allows to better overlap of communication and computation.

However, in order to exploit efficiently large scale machines, PGAS language implementations and their runtime system must be designed for scalability and performance. The IBM XLUPC compiler and runtime system provide a scalable design through the use of the Shared Variable Directory (SVD). The SVD stores meta-information needed to access shared data. The information includes the virtual memory address of data for each shared variable on the local node, the type and shape of the shared data, etc. The SVD is managed by the compiler, which creates and deletes entries in the directory whenever shared variables are allocated and de-allocated. The SVD is dereferenced, in the worst case, for every shared memory access, thus exposing a potential performance problem. In addition, the runtime system implements a protocol for obtaining the remote address of a shared variable, and thus a handshake is required to exchange this control-information which limits the exploitation of native (remote) direct memory accesses that most modern High Performance Networks offer (Infiniband and Myrinet among others).

In the *third contribution* of our thesis we present a **cache of remote addresses** as an optimization that will reduce the SVD access overhead, the handshake will not be necessary thus synchronization would be avoided and the native direct memory access will be exploited. It results in a significant performance improvement while maintaining the portability and scalability of the runtime system.

This work has been carried on in the UPC language and it applies to any other PGAS language but not to MPI. MPI is an explicit programming model, although MPI-2 offers one-sided communication, communication is explicit, the remote shared address of any one-sided communication is exchanged through a collective before the one-sided takes place. As the remote address is known the cache is not required, and all the benefits of one-sided communication apply.

The work performed in this area has resulted in the following publication:

[79] Montse Farreras, George Almási, Calin Cascaval and Toni Cortes. Scalable RDMA performance in PGAS languages. In submission process

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 presents the state of the art related to our work. In the direction of improving the memory management in the communication system of massively parallel computers, Chapter 3 explores the use of predictability to scale up the communication system memory management and Chapter 4 presents and evaluates a memory control flow algorithm that deals with control data reception in short-memory scenarios. Chapter 5 introduces a language extension that allows for better exploitation of data locality thus reducing inter-node communication. Chapter 6 presents and evaluates a cache of remote addresses that aims to reduce communication latency by reducing control-data communication and exploiting the RDMA network capabilities. Finally, Chapter 7 draws up the conclusions of our research work and discusses the future directions.

Chapter 2

Related Work

2.1 Parallel programming models

Over the years, many research on High Performance Computing has been focused on analyzing, improving or creating parallel programming models adapting to the new computer architectures with the ultimate goal of improving productivity of programming.

Among the most popular programming languages: PThreads [33] (POSIX Threads) and OpenMP [148] are two of most widely used for shared memory machines. Whereas Message Passing Interface (MPI) stands out in High Performance Computing among other message-passing systems PM [184], CORBA [147] or SOAP [178] for distributed memory machines. In MPI, communication between processes is made explicit which allows better performance to the detriment of programmability.

Due to its complexity, there have been many projects over the years that have aimed to improve productivity of parallel programmers. One of the major directions has been to provide important computational tasks as sets of libraries that run over MPI. Popular examples include SCALAPACK [117, 48] and PBLAS [49]. Large software engineering efforts such as PETSc [15] provide a common framework that encompass many popular tools. In practice these distributions have been successful.

In order to provide a richer abstraction and expression of the higher level semantics there have been many language efforts to improve productivity. Popular examples of these language include UPC, Co-Array Fortran [146, 51], Titanium [200], ZPL [43], HPF [81], Chapel [42, 36], Fortress [5], X10 [198], Cilk [180], Charm++ [151], and many others.

While each of the languages has its own corresponding performance and scalability studies, we highlight those related to the once we choose to study on our thesis: UPC and MPI.

2.1.1 MPI

Concerning MPI, there exist many research groups that have provided their own MPI implementation: MVPICH [138, 134], MP-LITE [126], LAM [118], CHIMP [47] or OpenMPI [149]. The most relevant are:

- Argone National Laboratory develop MPICH [132]. It is high-performance widely portable implementation supporting different computation and communication platforms such as commodity clusters, high-speed networks and propietary high-end computing systems (BlueGene, Cray). It provides a very modular design framework that allows an easy extension to other derived implementations. For this reason it is by far the most popular and widely used implementation.

- Ohio State University develops and does research over MVAPICH, extending MPICH over InfiniBand [186].
- Sandia National Laboratory implements MPICH over Portals: Portals MPI [133].
- Myricom provides two Myrinet [140] drivers: GM and MX, with its correspondent MPICH extensions: MPICH-GM and MPICH-MX [140].
- IBM provides its own MPI implementation based also on MPICH for the BlueGene family [6].

2.1.2 UPC

Concerning UPC, the family of UPC implementations include:

- Berkeley UPC: The goal of the UPC effort at **UC Berkely** is to build portable, high performance implementations of UPC for large-scale multiprocessors, PC clusters, and clusters of shared memory multiprocessors. There are three major components to this effort: lightweight communication, compilation techniques and application benchmarks.
- MTU-UPC: Michigan Technological University (MTU) [168] projects include the recent release of the MuPC run time system for UPC as well as collective specification development, memory model research, programmability studies, and test suite development.
- GCC-based Intrepid UPC[87] toolset provides a compilation and execution environment for programs written in UPC. The GCC UPC compiler extends the capabilities of the GNU GCC compiler. The GCC UPC compiler is implemented as a C Language dialect translator, in a fashion similar to the implementation of the GNU Objective C compiler.
- The High Performance Computing Laboratory of George Washington University (GWU) is involved in a number of efforts: UPC specification, UPC testing strategies, UPC documentation, testing suites, UPC benchmarking, and UPC collective and Parallel I/O specification.
- Other implementations are provided by HP UPC [95] and Cray UPC [58].

Our thesis is focused on optimizing the communication system for Massively Parallel Computers. The proposed optimization have been studied in these two parallel models:

MPI and UPC. Next sections present the research work found in the bibliography for each contribution of this thesis.

2.2 Improving memory management in the communication system

The first contribution of this thesis is focused on memory management in the communication system, dealing with memory shortage in two different scenarios: control-data and application data.

For every message arrival, control-data needs to be saved in order to manage the message. When memory is short thousands of control-data messages can flood up the receiver's memory.

Another memory management issue is to be aware of is the application data. In most current implementations, preference is given to the performance and control messages are avoided, so a message may be sent without receiver's permission and it is not guaranteed that the receiver will have enough memory to keep this message. Section 2.2.2 analyze both aspects of some messaging systems. Since one of our proposed solutions uses prediction to guarantee data message reception, a preliminary study about message predictability was done, research work related with predictability is presented in next section.

2.2.1 Communication Patterns Prediction

Regarding the characterization of the temporal communication patterns of MPI applications and the proposal of a prediction scheme based on periodicity detection, there has been some research on the topic.

Kim and Lilja [113] describe the communication patterns of MPI applications in the spatial domain (communication locality). In their experiments it was found that the processes communicate only with a small number of the partners (message-destination locality). Also, it is observed that the MPI applications usually have only 2-3 distinct message sizes (message size locality). Another result is that while the problem size and the number of processes vary, the size and number of messages vary, but the observed communication localities are quite unaffected. The characteristics of the temporal patterns in MPI messages, however, were not reported.

In our work we considered the temporal domain of the message stream. We identified the size of the iterative patterns appearing in the temporal domain. Although the spatial locality shown in [113] suggests the possibility of an iterative behavior in the temporal domain, it is not guaranteed by the spatial locality. In fact, we observed in the data streams of the physical level that communication latencies can be affected by randomness, which as a consequence affects the temporal pattern. When the communication latencies become significant compared to the computation time, randomness will not influence in the spatial locality of the messages, but the temporal message behavior may become unknown and unpredictable.

In his work, Afsahi et al. [3, 1, 2] propose the prediction of MPI messages. In this work a number of heuristics for the prediction of MPI messages are presented. The prediction heuristics are used to predict only the next value of a given data stream. The proposed application is to cache the incoming message to be nearby when the consuming thread accesses it. It was shown that the prediction heuristics provide very high hit rates for the studied streams.

Alsahi et al. [3] also propose the use of prediction to handle dynamic memory allocation for user's data, to guarantee correct message arrival. Our work is different in that the predictor we propose is based on the computation of distances between patterns in the stream. This has the benefit that the iterative pattern in the data stream is identified. In contrast to these prediction heuristics, knowing the iterative pattern allows not only the next value to be predicted, but several future values of the data stream.

Regarding to the applicability of MPI predictions, Iwamoto et al. [105] use next message prediction to speculatively execute the message reception beforehand. Our work differs in the purpose. While Iwamoto et al. seek for performance improvement, correct message reception is not guaranteed. In our work prediction is not used to receive the message but to prepare the receiver's environment to guarantee enough resources for message reception.

Our approach analyzes and predicts MPI communication patterns at run time. This analysis is indeed possible at compiler time. Shao et al. [172] present a compiler framework which can identify communication patterns for MPI-based parallel applications. The analysis has been used to minimize power consumption [173], and to guide source code transformations [163]. This approach has the drawback that it is limited for those applications that show static communication patterns.

2.2.2 Memory management in the communication system

To guarantee message reception, some contributions have taken the approach of statically allocate memory space to guarantee message arrival, this is the case for AIX [103], it allocates space for user's data but not for control-data. Also Fast messages in its credit-based control flow algorithm [150] preallocates space for both data and control-data for every other process in the system. PortalsMPI[133] pre-allocates space for control-data, while data is dropped if no memory is available. We should note though, that memory pre-allocation has the drawback that it does not scale well, just imagine if we decide to allocate 10Kb buffer per process in a 132K node machine like BGL, each process would need to allocate more than 1Gb of memory, while BGL has 512Mb of available memory per node. If we reduce the memory allocated, and use it only for control-data, it is always a limitation that turns to be too restrictive in practice, especially as applications scale beyond a thousand processes.

The next logical step is to allocate the memory dynamically. Portals 3.1 [29, 28] softens the problem with control-data reception by allocating memory for control-data dynamically and the number of outstanding unexpected messages becomes dependent on memory space rather than count, not having enough memory to allocate control-data is not considered, (probably because to our knowledge it has scaled up to 1792 nodes [29]).

Regarding dynamic memory allocation for user's data other directions have been taken that do not use prediction. An on-demand credit management was explored for use in Fast Messages making the previous credit-based control flow mechanism more scalable[37, 116]. The idea is to distribute available credits on an on-demand basis so it does not depend on the number of processors and each process receives credit according to its communication needs.

Finally, most MPI implementations running on supercomputers use a rendezvous-based protocol to guarantee correct message reception [135, 11, 10, 65, 74, 104]. In this mechanism, aka. 3-handshake protocol, the sender asks permission to send the message and waits for confirmation, which guarantees data message arrival.

Although all directions are very interesting, they are only partially valid to our scenario since they all assume that control-data is able to be received at the other end, hence many changes would have to be done to cover control messages resulting in a completely new solution.

Facing both, control-data and data memory issues, we should mention the work from Myricom [140] in both MPICH-GM and MPICH-MX, and a contribution from Ohio State University [110].

MPICH-GM holds up to 256 eager messages, and both data and control-data messages are dropped afterward, to be resend after a timeout. Basically, the problem moves from a memory issue to network congestion. MPICH-MX is newer and it solves the scalability problems found in MPICH-GM by means of a control flow algorithm and pre-allocated space in the NIC memory for outstanding messages [26]. If memory is short, sender is stalled, messages are copied into a buffer and a different thread will send them. Memory shortage has to be detected in advance to be able to apply this solution, so a threshold is defined.

About the contribution from Ohio, a memory management protocol ("control flow") with dynamic pre-allocation of buffers is implemented. However, the solution has some weaknesses: it is not scalable because buffers are never freed and keep increasing resulting in a waste of memory (the authors are aware of it). Moreover, as far as we understand the paper explanations, it seems that the solution may have problems concerning the reception order of messages, it may violate MPI semantics (see section 4.3.3). And finally, the approach is adding overhead even when there are no memory problems.

To sum up, memory management for unexpected user's data has been updated to the current trend in some implementations running in supercomputers. However in all proposed solutions, either some space is allocated a priori, which may waste critical memory resources or may be too restrictive, or memory is allocated on demand during program execution, which may scarify the robustness and reliability of the system. Only MPICH-MX deals with both problems (data and control-data), the weakness of MPICH-MX is that a threshold is defined from where all senders are stalled. As we will show in chapter 4 this behavior may cause a deadlock situation in extremely adverse situations. The advantages of our solution are that it is deadlock free and our threshold is dynamic, depending on the available memory and the application needs.

2.3 Extending the language to reduce communication

In our second contribution, we first demonstrate how the memory programming model found in UPC is a good fit for large distributed memory machines, and then we propose a language extension to better exploit locality and so reduce communication while maintaining good programmability.

2.3.1 Scalability

There is a considerable amount of work evaluating the performance of UPC programs: At GWU, El-Ghazawi et al. [75] demonstrate the potential of UPC as a viable programming language and show their potential performance advantages. Coarfa etal. [52] evaluate the effectiveness of these global address space languages and highlight their limitations.

Berkley UPC[44] implements a source to source (UPC-to-C) compiler. Its companion runtime system is built on top of GASNet. While source-to-source translation scheme improves portability, it incurs optimization limitations for accommodating the impact to different back-end compilers. The shared address space in the Berkeley UPC runtime system is limited by the machine address space of a single node [73]. This is a serious limitation when scaling to large scale machines with 32 bit architectures, because the amount of memory on the machine is much larger than what a single node can address. Bell et al. [19] show how the performance advantages of one-sided communication models and overlap can be applied to improve performance of bandwidth limited problems such as 3DFFTs. They have work in scalability through the use of two phases communication [46]. However, in the bibliography scalability is evaluated up to hundreds of processes [99, 20].

As far as we know, our study is the first one evaluating the scalability of UPC to hundreds of thousands of processors.

2.3.2 Reducing communication

Other significant amount of research effort has been focused on aplying compilation techniques to reduce communication for data parallel programs [41, 90, 177]. Chen *et al.* implemented redundancy elimination, split-phase communication and message coalescing in the Berkeley UPC Compiler [45]. For communication analysis and optimization, Zhu and Hendren [202] use compiler analysis to select the “best” place for inserting communication, reduce redundant remote access and message aggregation.

Data distributions

There is a significant body of work on data distributions in the context of High Performance Fortran (HPF) and other data parallel languages. Numerous researchers have tackled the issue of reducing communication on distributed memory architectures by either finding an appropriate distribution onto processors [13, 115] or by determining a computation schedule that minimizes the number of message transfers [91, 162]. By contrast to these works, we do not try to optimize the communication, but rather allow the programmer to specify at very high level an appropriate distribution and then eliminate the need for communication all together using compiler analysis. We do not attempt to restructure or improve the data placement of threads to processors in order to minimize communication. While these optimizations are certainly possible, we leave them as future work.

Tiled and block distributions are useful for many linear algebra and scientific codes [24]. HPF-1 provided the ability to choose a data distribution independently in each dimension if desired. Beside HPF, several other languages, such as ZPL [43] and X10 [198] provide them as standard distributions supported by the language. In addition, libraries such as the Hierarchical Tiled Arrays library [24] provide tiled distributions for data decomposition. ScaLAPACK [117] provides a 2 dimensional block-cyclic distribution for matrices which allows the placement of blocks over a 2-dimensional processor grid. And MPI provides Derived Datatypes [197]. Both are more general than our presented distribution.

2.4 Speed up communication

Our third contribution has the ultimate goal of speeding up communication. It reduces control-data messages between processes, it allows exploitation of the RDMA capabilities of the hardware and a true overlap between communication and computation. All is achieved by means of a remote address cache that caches remote addresses avoiding control-data messages, otherwise necessary to translate the address in the target remote node and send it back to the origin.

2.4.1 Optimizing data transfer

Along the years there has been a lot of research to speed up data transfer. Some of them aimed to provide a zero-copy mechanism that reduces memory copies inside the communication system by different means: prediction mechanisms [4], exploiting the hardware capabilities such as the NIC [174] or the RDMA [122, 121, 187].

The RDMA has highly been exploited in Infiniband: for short MPI message transfers by using a persistent buffer association [123], to speed up collective communication [114] and to optimize MPI communication [121, 109].

Although, some implementations try to exploit the hardware RDMA capabilities in two-sided communication, like Liu et al. [121], this optimizations are limited to long message sizes, while our optimization targets very short messages. Optimization of short message transfer through the use of RDMA cannot be achieved in conventional two-sided messaging systems due to design limitations, MPI relies on message matching on the receiver, which rules out RDMA transfers and the true overlap.

Considering one-sided communication in MPI-2, one common way to implement it is to use existing MPI two sided communication operations such as MPI_Send and MPI_Recv. This approach has been used in several popular MPI implementations such as MPICH. Although it achieves good portability, it suffers high communication overhead and dependency on remote process for communication progress.

Jian et al. [109], aware of the limitations, attempted an implementation on top of Infiniband native RDMA operations. Memory registration is done in the initialization period when the *exposed window* is created. This implementation outperforms others based on two-sided communication. However, it is limited by the total amount of registrable contiguous memory at a single call, if the *window* is bigger than the registrable memory, the RDMA could not be used. Also, only one *window* can be effective at a given moment, if another *window* is required to be exposed, the first one needs to be freed incurring additional registration and deregistration costs. The problem could be solved by applying our proposed address cache with the on-demand registration strategy, covered in section6.

The Rendezvous protocol has also been optimized over InfiniBand by using selective interrupts to achieve better overlap [181].

The NIC capabilities have also been explored [175, 31, 166, 167, 155]. Some works offload the management of the MPI queues and the matching [189, 193], the control-data communication [124], or use it to perform collective operations [158, 125, 64].

In a more theoretic way, White et al. [107] analyze applications to find the potential overlap, and studies the potential benefits [106] of hiding the latency by overlapping communication and computation.

Concerning works done in the UPC environment, we should mention: Iancu *et al.* optimize communication by demand driven synchronization [101]. Their runtime system uses virtual memory support to determine the dynamic program point before which the communication should complete.

2.4.2 Remote Address Cache

Previously existing UPC implementations, such as the Berkeley UPC compiler [22] or MuPC [201] from Michigan Technological University [137], map UPC threads to O/S processes, and each thread maps the entire memory space to the same virtual address, forcing virtual addresses to be identical on all threads, and shared remote addresses are already known. However, this solution results in memory fragmentation (especially when individual threads allocate memory). The XLUPC runtime system prevents this problem by means of the SVD, which allows virtual addresses to be different on each node and allowing UPC threads to be mapped to Pthreads that share memory directly.

The idea of keeping the remote address is also used by Cantonnet *et al.* , they propose a technique that resembles the Translation Look Aside Buffers (TLBs) to reduce address translation overhead on Non-Uniform Memory Access Architectures [38]. The idea of a cache has also been used in UPC for remote data [194].

In MPI-2 one-sided communication is performed over *windows*, similar to the UPC shared memory areas, and the remote address is known because *windows* are initialized with a collective operation where every process defines the “shared” memory area and the remote address of every other process is exchanged at this point.

Memory Registration

Another problem affected by shared variable directories and remote address caching is the handling of memory registration costs on pinning-based networks like Myrinet [140], VIA or Infiniband [65]. MPI implementations like OpenMPI [149] and MVAPICH [186] as well as one-sided messaging systems like ARMCI [144, 142] follow a differential approach based on message size, switching between preallocated registered memory buffers (Bounce Buffers) for short messages and dynamic memory registration and de-registration as part of each data transfer (Rendezvous) for large ones. The crossover point between the protocols is dependent on the underlying network hardware and software, requiring tuning for each machine.

Since on Myrinet/GM the de-registration cost is the most expensive, most existing Myrinet/GM communication layers, including Myricom’s own MPICH-GM [140], use Rendezvous and omit the de-registration step.

Another solution to handling memory registration costs is the Pin-Down cache [185] used in PM [184] and Sockets-GM [140].

Berkeley UPC’s strategy for on-demand registration of shared memory regions, called Firehose [21] [18], distributes the largest amount of memory that can be registered among all nodes. Every node keeps track both of remote regions in other nodes it is using and its own areas used by other nodes. Synchronization required to update this information is minimized based on access locality assumptions. It has been implemented on a number of platforms, including Myrinet/GM. Memory access, however, is not one-sided, instead an Active Message is sent which initiates a put back to the requester.

The Shared Variable Directory concept and remote addresses caching could potentially be applied to every shared-memory programming model that wants to run on a distributed memory system (e.g. OpenMP on a Software Distributed Shared Memory system). Several combinations of OpenMP runtime plus SDSM systems have been implemented [96]. The most significant ones are the OpenMP translator developed by Hu et al. [97] on top of Treadmarks [112], OpenMP on the SCASH system [92], and ParADE [111]. There is also NanosDSM [55] which uses sequential-semantic memory consistency.

Chapter 3

Prediction to scale up the communication system memory management

3.1 Introduction

Looking toward the near future, system sizes have grown considerably over the last few years [188], and this trend is sure to continue. As system size grows, inter-node communication increases, stressing the communication system, which becomes crucial for overall performance and scalability.

MPI (Message Passing Interface) is the most popular approach for HPC. It is a specification for a standard library to address the message-passing model of parallel computation [131].

Despite of the fact that MPI was designed with scalability in mind and many implementations have been developed, MPICH [132], CHIMP [47], LAM-MPI [118], OpenMPI [149], IBM MPI [103, 169], Cray MPI [56], MPICH-GM [140], they have been designed for hundreds of nodes and they are not able to scale beyond. One of the weaknesses of current MPI implementations is scalability. The main problem of current implementations is that performance is more important than scalability and thus some assumptions about resources are taken that do not scale well.

In the MPI model, applications are divided into different tasks (or processes) that communicate by sending and receiving messages among them. Any message arriving before being required by the application (hereafter `unexpected message`) needs to be stored somewhere into the library. The scalability problem comes by the fact that enough memory is assumed to be available anytime for message reception, however this is no longer true for thousands of nodes, and message reception is not guaranteed.

In order to guarantee message reception sender process needs to assure enough memory at receiver's side. For this reason, some kind of control flow is needed. For instance, the sender should first ask permission to send the message, then wait for a confirmation to finally send a message (*rendezvous protocol*). The problem is that this mechanism increases latency since three messages are sent and only one has user's data. It also requires a handshake (synchronization) between the two processes preventing sender process to continue its computation until the confirmation message reception.

Some solutions have been provided to guarantee message reception, with the drawback that they do not scale well. For instance, in IBM MPI [103] static preallocation of buffers is done. Each process keeps a buffer for each one of the other tasks in the communication rank, and messages are received into these temporary buffers. Preallocation of buffers allows messages to be sent avoiding any kind of handshake (*eager protocol*) and reception is guaranteed. Although it speeds up communication, it is not feasible if we plan to scale to thousands of nodes, just imagine the amount of memory it would take.

MPICH takes a different approach [132, 47, 140]. It dynamically allocates temporary storage at the time the message is received, assuming enough resources on the receiver's side. But this assumption is no longer valid if thousands of processes send messages to the same process.

There would be a simple scalable solution if we could predict which **nodes** are going to send messages and which **sizes** they will be, we could pre-allocate the buffers according to the application needs and inform the senders before the message is sent. Sender will be able to avoid control flow (handshake) for those messages that have been predicted. As a result correct message reception is guaranteed and memory usage is reduced.

On the other hand, memory management for data reception not only affects scalability but also performance by highly influencing communication latency. On the receiver side, message-copying operations contribute to these communication latencies. In the situation where a data message arrives `unexpectedly`, it is kept into a library buffer and when the application requests the message, it is copied from the library buffer to the final application's buffer. This copy may be expensive, depending on the message size and the architecture. A zero-copy mechanism reducing this copies would reduce communication latency. If we could predict the address of the final destination buffer beforehand, we could reduce the impact of this copies. Thus, communication latency will be reduced.

Another memory management issue is to be aware of the control data: For every message arrival, control-data needs to be saved in order to manage the message. When memory is short thousands of control-data messages can flood up the receiver's memory.

Our **first contribution** is focused on MPI memory management, dealing with memory shortage in two different scenarios: user data and control-data reception.

In this chapter we address the first scenario, user's data reception, focusing on **improving the memory management in the communication system** to guarantee user's data message reception. Concerning scalability, two solutions are proposed and discussed: (i) A solution based on prediction is proposed: the communication pattern predictability is studied showing high predictability rates. And a mechanism to guarantee correct message reception by predicting the communication pattern and allocating the required resources in-advance is introduced and evaluated. (ii) Another solution based on the *rendezvous protocol* is also discussed and compared to the previous one.

And finally, to focus more on performance, we also present a prediction mechanism that would reduce memory copies into the messaging system, resulting in a communication latency reduction. Although the mechanism has not been implemented, our predictability study shows high prediction rates that would make it feasible.

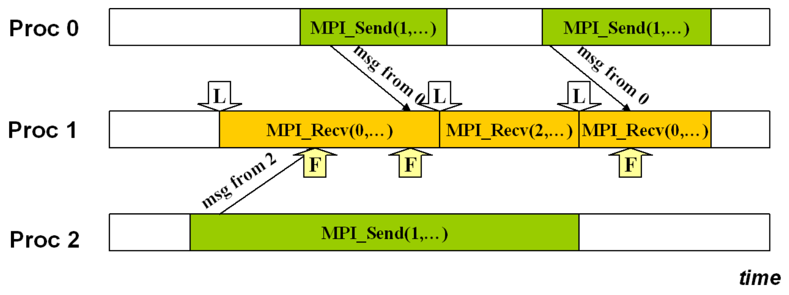


Figure 3.1 Tracing Points for Logical and Physical communication. L-labeled arrows show Logical Communication and F-labeled arrows show the Physical Communication. The Logical sequence of senders for Process 1 is $0, 2, 0$. The Physical sequence of senders for Process 1 is $2, 0, 0$.

3.2 Exploring the predictability of MPI messages

In this section we explore MPI message predictability. In this study, we are interested in the message size and the sender process for every message reception. We present the set of benchmarks used, our predictor scheme and finally the results showing prediction accuracy.

We want to decide if message reception is predictable enough to make feasible a scalable memory management mechanism based on prediction.

3.2.1 Experimental framework

In order to characterize the message stream behavior we instrument MPICH 1.2.4 [132] at two levels: the logical and the physical communication level.

Logical and physical communication

To obtain the logical communication data we trace the MPI calls from the application code to the top level of the MPI library. These calls directly reflect the application structure. The execution of loops in the application code will be seen as iterative patterns in the calls to the MPI library. We can describe the logical communication data as a function of the application code.

To obtain the physical communication data we trace at the low level of the MPI library implementation. Our tracing points show at what time messages are actually received, from which sender process and which is the message size. These calls reflect the application structure, but also random effects in the physical data transfer between processes, load balance, network congestion, and so on.

Table 3.1 MPI benchmarks analysis.

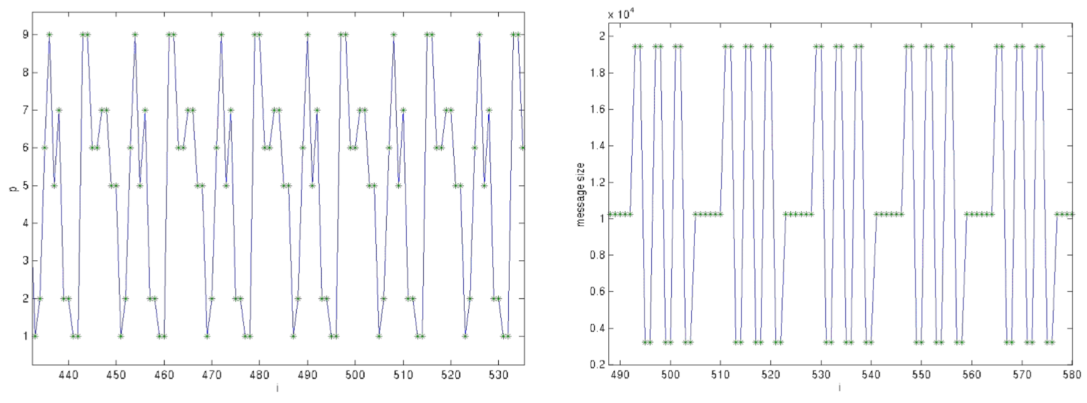
Application	Procs	P2P msgs	Coll msgs	Msg Sizes	Senders
NAS BT	4	2416	9	3	3
	9	3651	9	3	7
	16	4826	9	3	7
	25	6030	9	3	7
NAS CG	4	1679	0	2	2
	8	2942	0	2	2
	16	2942	0	2	2
	32	4204	0	2	2
NAS LU	4	31472	18	2	2
	8	31472	18	4	2
	16	31472	18	2	2
	32	47211	18	4	2
NAS IS	4	11	89	3	4
	8	11	177	3	8
	16	11	353	3	16
	32	11	705	3	32
Sweep 3D	6	1438	36	2	3
	16	949	36	2	2
	32	949	36	2	2

Figure 3.1 shows the points where logical and physical communication is traced in Process 1. In the figure, a message from process 2 overcomes another message from process 0, and the physical communication pattern, $2, 0, 0$, differs from the logical pattern $0, 2, 0$.

Benchmarks used

In our experiments we use applications from the NAS and the ASCI benchmark suites. We select the NAS BT, CG, LU, IS [132] [14], and the ASCI Sweep3D application [119] due to its communication pattern characteristics.

In Table 3.2 several characteristics of the benchmarks used are given. Column two shows the number of processes we executed the applications with. In the third, fourth and fifth column the number of point-to-point, collective, and total number of messages received by a process are shown. The remaining columns indicate the number of different message sizes and different sender processes appearing in the MPI message stream received by a process. Class A problem size is used for the NAS benchmarks.



(a) Sender (b) Message Size

Figure 3.2]
Pattern of process 3 in NAS Bt benchmark class A, 9 processes. The streams are depicted with continuous lines in order to visualize the iterative pattern. (a) Shows Sender processes and (b) Message sizes.

3.2.2 Predictor design

As a preliminary study, the message stream behavior has been analyzed and shows that the data streams we are interested contain repetitive patterns and thus this kind of prediction seems feasible.

Figure 3.2 shows a portion of the data streams of MPI receives and it is a sample of repetitive sender and message size patterns. Pattern of process 3 in NAS Bt benchmark class A with 9 processes is shown. Figure 3.2a shows that senders are processes 1, 2, 5, 6, 7, and 9. The periodicity in the data stream is 18. And 3.2b shows message sizes are: 3240 bytes, 10240 bytes, and 19440 bytes. It can be seen that the order in which these message sizes occur is iterative. The message size pattern repeats every 18 messages.

After the observed behavior we have decided to push the idea further and propose the predictor scheme.

The Prediction scheme

The objective of the predictor is to dynamically detect the iterative pattern in the MPI message streams and to predict future values. First, the predictor has to determine the periodicity of the data stream. Second, knowing the periodicity (the size of the iterative pattern) allows knowing several future values. The predictor should indicate whether periodicity exists in the data stream, give the length of the iterative pattern, and predict the next future values.

In order to perform the prediction task, we modify the dynamic periodicity detector

(DPD) from [83] to enable the prediction of the data streams. The DPD allows segmenting the data stream into repetitive patterns, thus it is able to capture the periodicity of the data stream. The knowledge of the periodic pattern allows predicting several future values. The predictor returns an indication whether periodicity exists in the data stream, about the length of the iterative pattern, and the prediction of the next future values.

Prediction by means of periodicity detection is a powerful technique, since the mechanism learns fast and makes use of the knowledge on the short-term temporal structure of a data stream. In contrast, predictions made by statistical models such as Markov models require more training time and although the temporal structure of the data stream can be considered to be embedded in the probability matrices, these models usually do not detect periodicities and are not prepared to predict several future values.

The algorithm used by the periodicity detector is based on the distance metric given in equation 3.1.

$$d\left(m\right) = sign \sum_{i=0}^{N-1} \left|x\left(i\right) - x\left(i-m\right)\right| \tag{3.1}$$

In equation 3.1 N is the size of the data window, m is the delay $(0 < m < M)$, $M <= N$, $x\left(i\right)$ is the current value of the data stream, and $d\left(m\right)$ is the value computed to detect the periodicity. It can be seen that equation 3.1 compares the data sequence with the data sequence shifted m samples. Equation 3.1 computes the distance between two vectors of size N by summing the magnitudes of the L1-metric distance of N vector elements. The sign function is used to set the values $d\left(m\right)$ to 1 if the distance is not zero. The value $d\left(m\right)$ becomes zero if the data window contains an identical periodic pattern with periodicity m.

The implementation of the predictor is done with circular lists, which reduces the overhead of the predictor. To have a small overhead is important since prediction has to be done at runtime. It was shown in [83] that the overhead of such an implementation is small.

3.2.3 Evaluation of the MPI message predictability

We explore the predictability of MPI message behavior with our prediction scheme. We are interested in concluding how well the logical and physical communication of MPI can be predicted. The task is to predict the next five processes, which send to a particular process and the message size of the next five messages received by a particular process. In the graphics we denote these five future values of the senders and message sizes $+1$ ... $+5$.

Prediction of logical MPI communication

In these experiments, the logical communication stream (see section 3.2.1), as seen by the top level of the MPI library, is the input to the predictor.

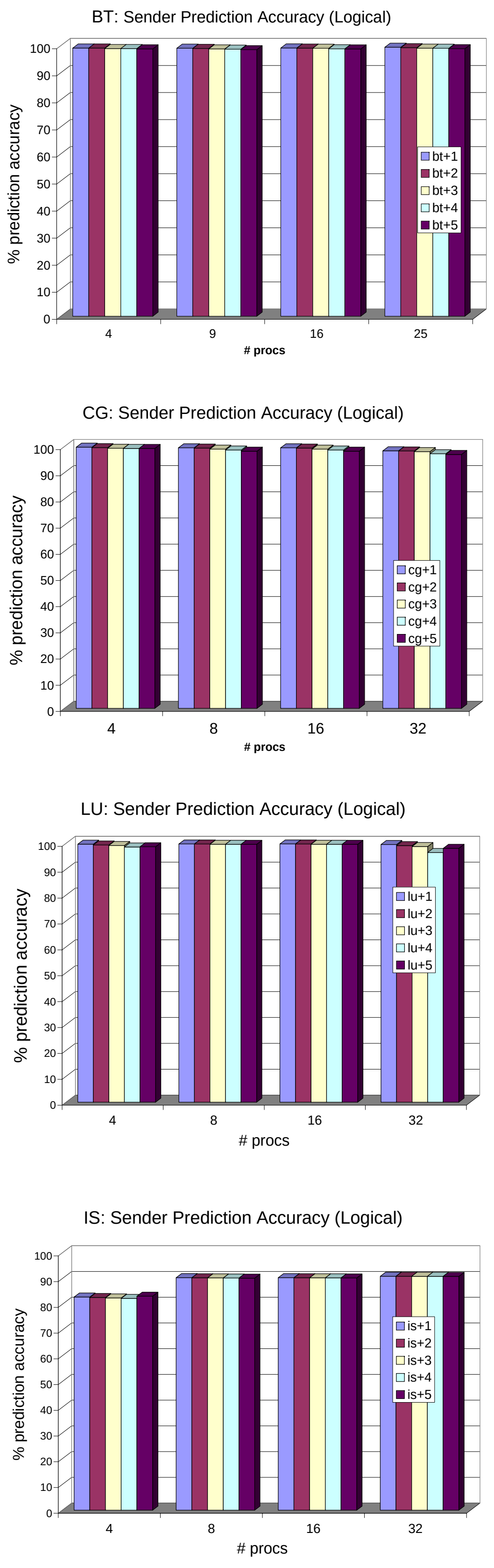
Figures 3.3 and 3.4 show the prediction accuracy obtained when predicting the next five future values of the sender (column (a) in the figure) and message size (column (b)) stream for the NAS and ASCI benchmarks, respectively. We can see that the logical communication of MPI is predicted with very high accuracy in all used benchmarks. The reason for this high predictability is that the message streams exhibit regular iterative patterns, which are captured by our prediction scheme. Results prove that, as we expected, it is easy to predict several future values once the periodicity is detected. The obtained prediction rates are higher than 90 %, mostly close to 100%.

The only exception is the NAS IS.4 benchmark for which we get around 80samples is very short. Some data samples cannot be predicted since a sample of the pattern has to be seen by the predictor for learning. When the NAS IS is executed with more processes and more messages are sent, high prediction rates are obtained. We can conclude from this experiment that the message stream behavior of both MPI applications with collective and point-to-point communications can be predicted very accurately at the logical level.

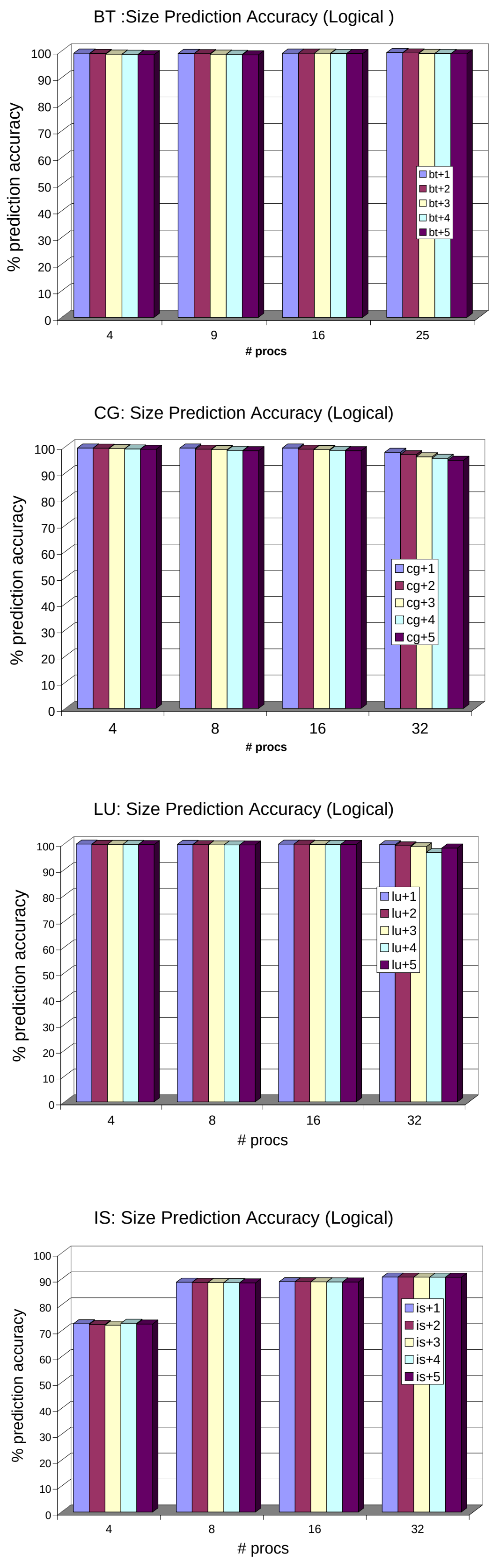
Prediction of physical MPI communication

In these experiments, the physical communication stream (see section 3.2.1), as seen by the low level of the MPI library, is the input to the predictor. Figures 3.5 and 3.6 show the prediction accuracy obtained when predicting the next five future values of the sender (column (a)) and message size (column (b)) stream.

We can see that the physical communication of MPI is predicted with less accuracy. The reason that we do not achieve the same high prediction rates as with the prediction of the logical communication is that the message streams suffers from random effects, which make the iterative pattern less visible. Each random change of the message pattern leads to a failure in the prediction. In some applications like the NAS LU and Sweep3D we still obtain high predictability. One of the reasons is that the message and sender streams have only a few different elements, which hide the random effects. Once we have more different elements in the message streams, like in the BT benchmark, the prediction rate decreases. In the IS benchmark, prediction is very hard. This benchmark uses many collective communications, implemented on top of point-to-point primitives in our implementation in a way that a process receives from any other process. Random changes in the temporal order of messages received are frequent, for which the prediction rates are less than in the other benchmarks. We can conclude from this experiment that

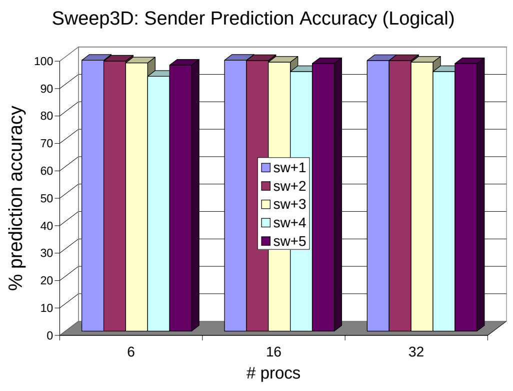


(a) Sender Process

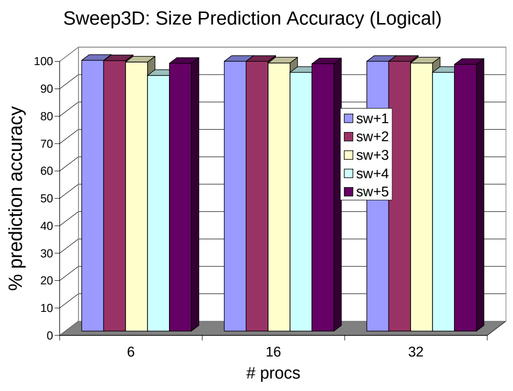


(b) Message Size

Figure 3.3 Logical MPI Communication Prediction Accuracy in Class A NAS Benchmarks.



(a) Sweep3D Sender



(b) Sweep3D Message Size

Figure 3.4 Logical MPI Communication Prediction Accuracy in Sweep3D ASCI Benchmark.

the prediction at the physical level of MPI communication is less accurate

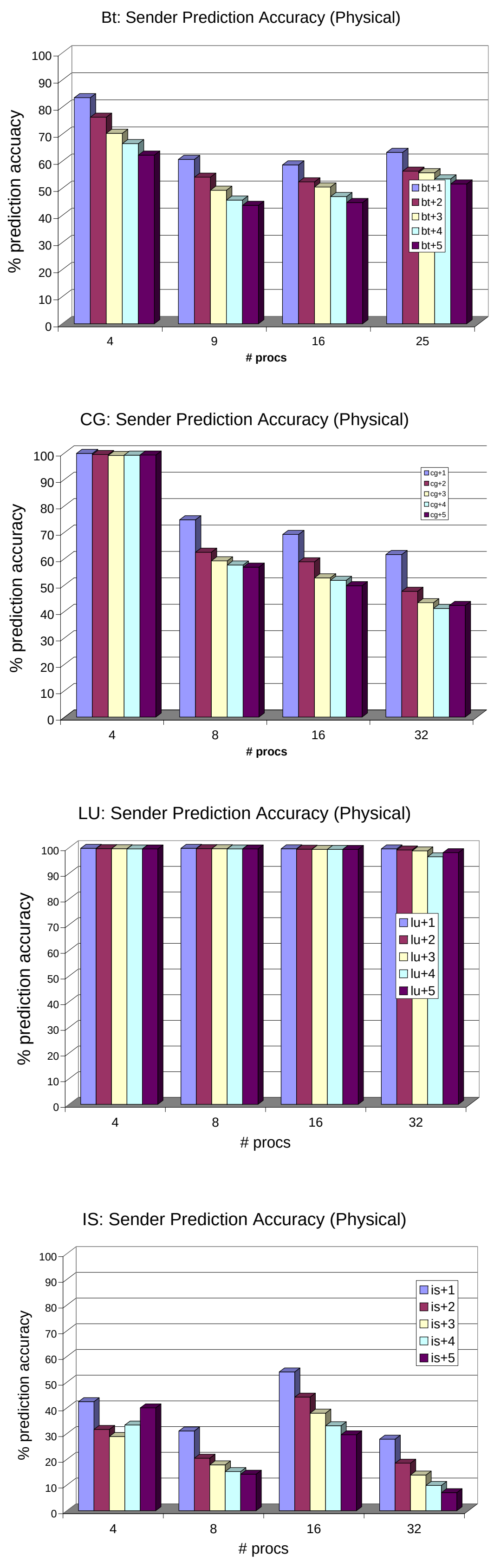
Results discussion

The prediction of the logical communications proved to be very successful, achieving very high prediction rates. The prediction of the physical communication suffers from randomness, which reduces the prediction rates.

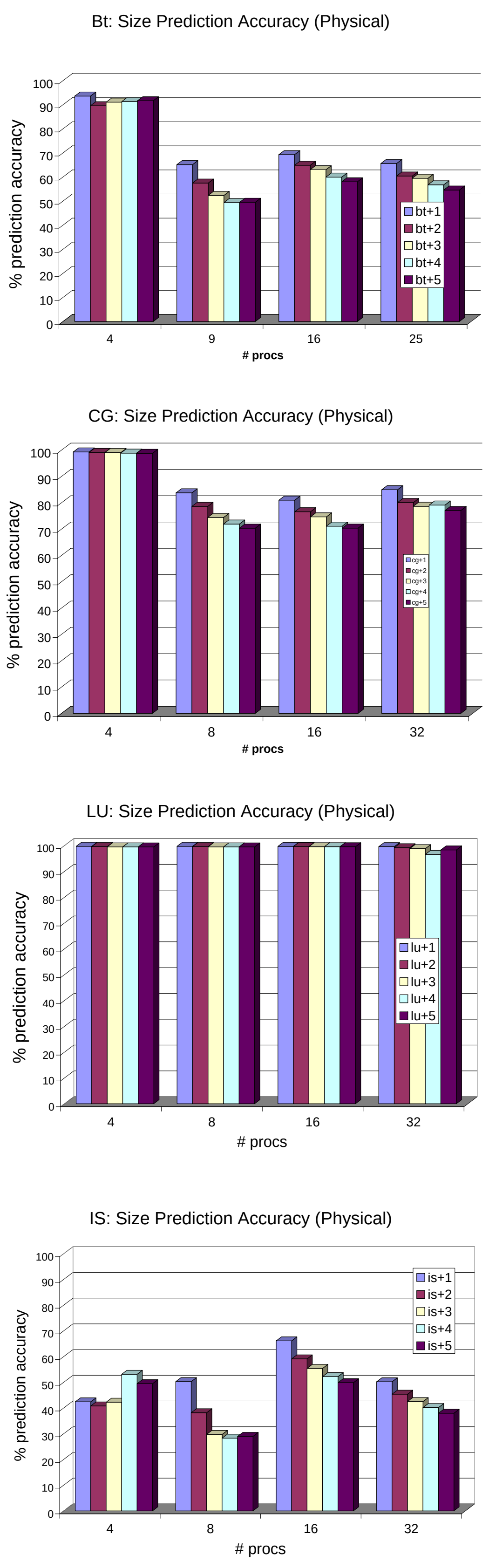
In Figure 3.7 we show by means of the BT application how the stream of message behavior can differ in the logical and physical communication. We can observe in the logical communications the regular pattern of sender processes given by the sequence 002211. In the physical communications this pattern can appear in the same order, or in a different order, such as indicated by the circles in Figure 3.7. Our predictor has no rule for these changes in the pattern and cannot predict correctly, since it expects the pattern 002211 in the given order.

Similarly, the message size stream on the physical level suffers from random changes, which decrease the prediction rates.

Depending on the application of the prediction, predicting the exact order of the future sender and message size may not be required. For instance, if the prediction will be used to allocate memory for message buffers of predicted senders, knowing the exact temporal order may not be necessary. Rather, knowing the next senders and their message size may be useful. This information is available with high accuracy also on the physical level, if we make use of the particular feature of this predictor, which allows several future values to be known.

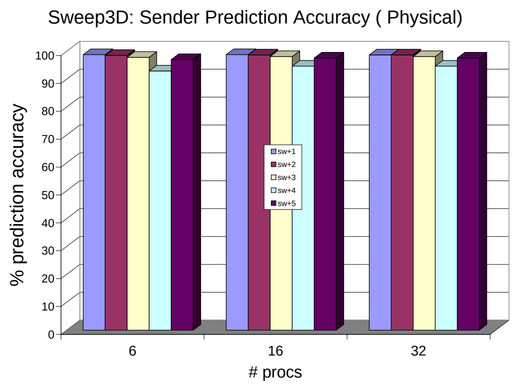


(a) Sender Process

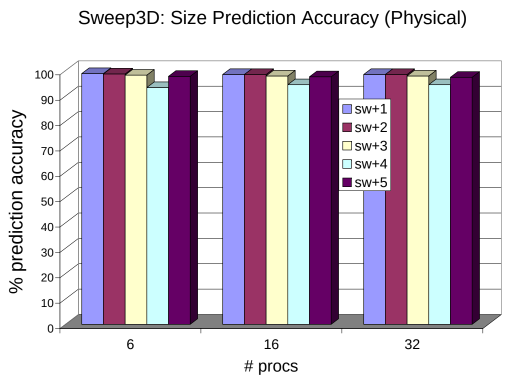


(b) Message Size

Figure 3.5 Physical MPI Communication Prediction Accuracy in Class A NAS Benchmarks.

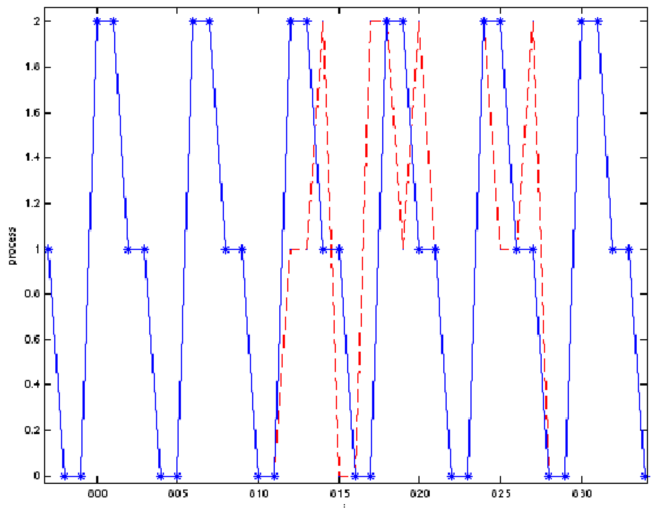


(a) Sweep3D Sender



(b) Sweep3D Message Size

Figure 3.6 Physical MPI Communication Prediction Accuracy in Sweep3D ASCI Benchmark.



(a) Sender

Figure 3.7 Pattern of senders to process 3 in NAS Bt benchmark class A with 9 processes. Logical communication stream is shown in solid line, and physical communication is shown in broken line.

3.3 Towards a zero-copy mechanism by predicting buffer addresses

As it is well known, besides scalability, the communication system memory management can considerably reduce overall performance by increasing communication latencies.

Communication latencies have been identified as one of the performance limiting factors of message passing applications in distributed memory systems. On the receiver side, message-copying operations contribute to these communication latencies.

In the previous section we have proved that prediction of several future messages is possible for the sender and the message size. In this section, we propose long-term prediction as part of the design of a zero message-copying mechanism.

First, the proposed zero message-copying mechanism is described, then we explore long-term prediction of MPI messages for the proposed design. To achieve long-term prediction we evaluate two prediction schemes, the first based on graphs, and the second based on periodicity detection. Our experiments indicate that with both prediction schemes the buffer addresses and message sizes of several future MPI messages (up to +10) can be predicted successfully.

3.3.1 Our zero message-copying approach

Requirement for a zero message-copying mechanism

Previous research [4] has proposed the idea of predicting the user buffer, where the data will be read from, and to place the data directly to its final location. This would avoid copying first into an MPI buffer and then copy it again to the user buffer. This idea, although interesting in its concept, is not implementable. Predicting the destination buffer to use it by the MPI library is too dangerous because an error in the prediction may modify a memory location that has useful data for the application. It is also important to notice that previous work only predicted the location of the buffer for the next message, and this is not enough. Messages do not arrive in the same order they are requested and thus, the system has to be ready to receive messages out of order and still be able to implement the zero-copy mechanism with them. Otherwise, the applicability of the improvement will be very restricted.

Let us assume now that we are able to predict the buffers for the future messages (which we will show along this work), the challenge is how we can use this knowledge to avoid extra copies. As we have said, using the exact buffer is not a possibility, but the

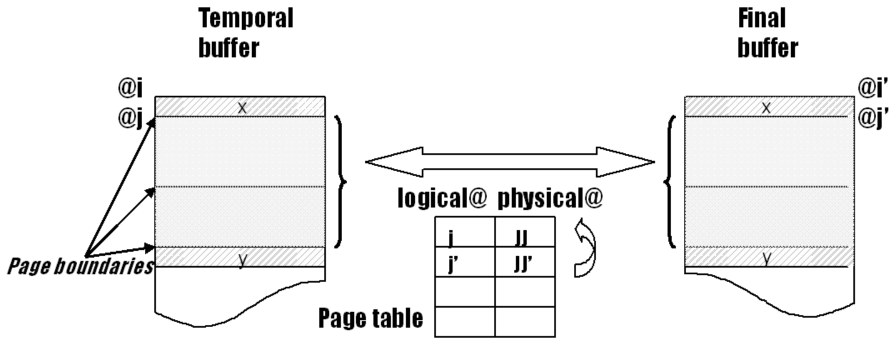


Figure 3.8 Message copied from the temporal buffer into the final buffer using our zero message-copying approach.

MPI library can place the data aligned in the same way as in the final destination buffer. If we hit in the prediction, then we can change the mapping of logical pages to physical pages to move the data to the user address space without a copy.

This proposal could avoid copies of full pages, but not the portions of the message that do not fill in a full page. In the latter case, we will have to copy this information, but hopefully, long and costly messages will avoid the long copies. All messages longer than 2 pages, will at least avoid the copy of one page. Regarding miss predictions, they do not cause any problem because the message will be copied as if no prediction had been done. A miss prediction will mean that the improvement will not be possible but no malfunction will appear.

Figure 3.8 shows the situation, in the example shadowed areas will be exchanged by switching entries in the page table, while portions x and y of the buffer (dashed) will be copied.

The only requirement we have is a system call that allows us to switch the binding of logical pages to their physical pages (of course only among the pages of the process). Once proven its necessity, this implementation is simple and could be easily incorporated in new versions of the OS.

Graph-based predictor

Our first solution for long-term prediction is a graph-based predictor as described in [60]. This predictor is similar to the prediction heuristics used in [4]. The second prediction mechanism, which we evaluate for this task, is the same periodicity-based predictor [83] used in the previous section.

Graph-based predictors describe an observed sequence through a number of states with transitions between them. Each state represents an observation. A probability or counter value is associated with the transition between the states. The graphs are trained (and built) on the observed data. The number of states increases with the increase of different symbols in the observed sequence. The observations contribute to form the transition probability from one state to another. Cyclic behavior in a data sequence, for instance, can be easily represented with such graphs.

The graph-based predictor works as follows: each state represents a sequence of three observations. The value of the transition between states is computed according to the observed sequence. Prediction can be achieved by selecting the most likely successor of a current state. We predict several future values by repeating this process on the predicted states.

Periodicity-based predictor

The approach of the periodicity-based predictor is different to the graph-based predictor, since prediction is based on the detection of iterative patterns in the temporal order of the sequence. Results obtained in previous section show that MPI messages contain repetitive sender and message size patterns. The knowledge of the periodic patterns allows predicting future values. We use the dynamic periodicity detector (DPD) to capture the periodicity of the data stream and modify it to enable the prediction of data streams. The algorithm used by the periodicity detector is based on the distance metric given in equation 3.1 explained in previous section.

3.3.2 Evaluation

Benchmarks used

In order to evaluate the long-term predictability of MPI buffer addresses we run several experiments with the NAS benchmark suite programmed with MPI. We use the class A problem size of the NAS Bt, Cg, Ft, Is, Lu, and Sp benchmarks [62]. As in the previous study the MPI implementation we used is MPICH 1.2.4. The applications are run with different numbers of processes up to 32.

In order to obtain the communication behavior of the applications, we instrument the MPICH implementation. To obtain the communication data we trace the MPI calls from the application code to the top level of the MPI library. We trace point-to-point and collective calls. Collective calls are represented in the trace as point-to-point calls from the different senders. The traces we extract correspond to the receiver side. We extract the buffer address, message size, and sender processes.

Table 3.2 Evaluated benchmarks and communication characteristics.

Application	Procs	Num Msgs	Buff Addr	Msg Sizes	Pattern Sz
NAS BT	4	2425	7	4	18
	9	3660	7	3	12
	16	4835	7	3	24
	25	6039	7	5	30
NAS CG	4	1679	2	2	4
	8	2942	3	2	7
	16	2942	3	2	10
	32	4204	3	2	12
NAS FT	4	998	6	3	12
	8	70	8	1	8
	16	8992	6	7	24
	32	262	32	32	32
NAS LU	4	31490	2	2	126
	8	31490	2	4	126
	16	31490	2	2	126
	32	47230	2	4	126
NAS IS	4	100	9	6	9
	8	188	17	9	17
	16	364	33	17	33
	32	716	65	34	65
NAS SP	4	4823	6	3	12
	9	7226	6	6	18
	16	9000	6	7	24
	25	12000	6	12	30

In Table 3.2 we summarize the characteristics we observed in these traces. Column 2 indicates the number of processes the application is executed with. Column 3 gives the number of messages received per process. Column 4 indicates the number of different buffer addresses in the pattern. Column 5 shows the number of different message sizes in the pattern. Column 6 indicates the size of the observed temporal pattern. The pattern size refers to the pattern formed by point-to-point and collective calls. We have obtained the size of the periodic patterns using the DPD. We can see that the results given in columns 4 and 5 confirm the data locality in MPI messages described in [113]. Column 6 indicates the existence and size of temporal patterns in MPI messages.

Long-term buffer address predictability

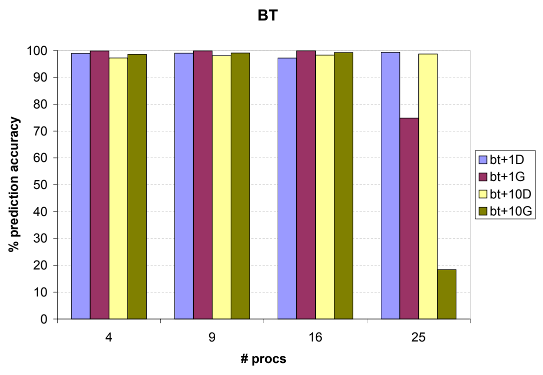
Our proposed zero message-copying mechanism requires the prediction of several messages. Therefore, we are interested in evaluating the long-term predictability of the buffer address together with the message size. The task is to predict the next $(+1)$ and the tenth $(+10)$ future buffer address and message size. The tenth message in the future $(+10)$ is chosen as upper bound. The chosen mechanism for zero message-copying may require to advance less. In our experiments, the input stream of both predictors is a linear combination of the buffer address, messages size and sender process, such as used in previous section. In Figure 3.9 the prediction results are shown. In the graphics, we denote the prediction of the future values with $+1$, and $+10$. The letter D indicates the periodicity-based predictor and the letter G the graph-based predictor. The prediction accuracy for messages larger than two pages is shown, for our experiments we assume a page size of 4K, therefore messages larger than 8K are considered.

As described previously, the zero-copying mechanism is effective for messages larger than two pages (8K). It can be observed that the prediction accuracy is generally very high (many times > 90%) with both predictors and in the $+10$ scenario. We can see very similar performance of both predictors in the Bt, Cg, Ft, Is, and Sp benchmarks. An exception is the Bt executed with 25 processes. In this case the performance of the graph-based predictor decreases when performing the $+10$ prediction task. A special behavior can be observed in the Lu benchmark for the graph-based predictor, when predicting messages larger than 8k on $+10$, correct prediction is not achieved. The reason for failing in this prediction is discussed in detail in the next section.

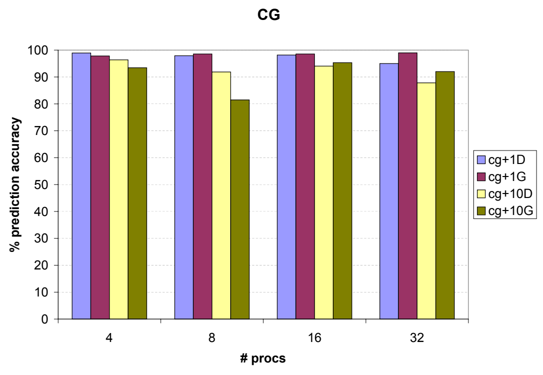
Comparison of predictors: Accuracy and overhead

We observed very high prediction rates including long-term prediction $(+10)$ both with the graph and the periodicity-based predictor. In many benchmarks, the rates of both predictors are similar. These benchmarks include the Bt, Cg, Ft, Is, and Sp, which all showed regular patterns of a rather small temporal size. Furthermore, the sequences have many different elements, which is beneficial for the performance of the graph-based predictor. Differently, the Lu benchmark showed a large pattern, of size 126 within which smaller (nested) pattern are repeated. In the Lu, a large message appears after observing a long sequence of small messages with identical values. In terms of prediction rates, the prediction of $+10$ in the Lu with the graph-based predictor goes down to zero, as it can be seen in Figure 3.9.

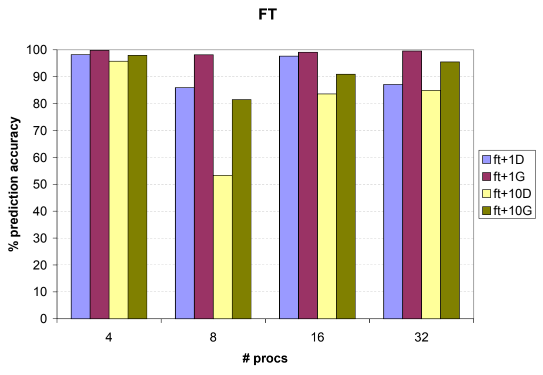
The periodicity-based predictor, however, could predict such a message, even after having observed identical messages during a long time. The reason for this capability



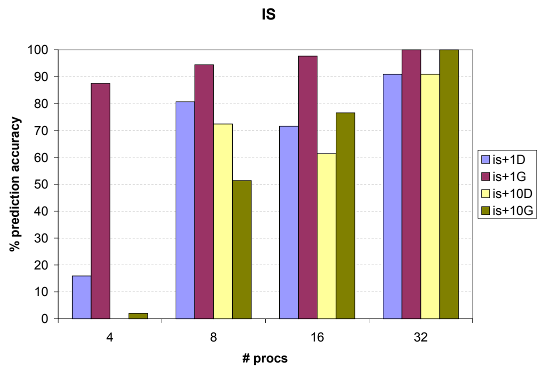
(a) BT



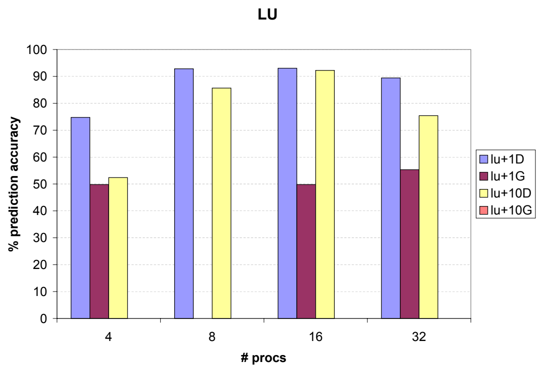
(b) CG



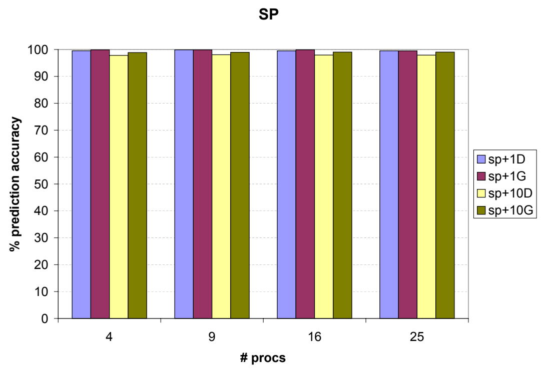
(c) FT



(d) IS



(e) LU



(f) SP

Figure 3.9 Long-term buffer address and message size prediction rates (messages > 8Kb)

is that it captures the periodicity of 126 in the message stream. We found that achieving the knowledge of such long periodicities with the periodicity-based predictor is computationally more expensive than using the graph. In our current implementation, the graph-based predictor is much faster than the periodicity-based predictor. In the periodicity-based predictor, the length of the history, which enables to compute the periodicity, strongly affects the execution time. In our study, we have used the default value of the periodicity-based predictor, which is a history of 256 samples (which allows to capture periodicities up to 256).

Although graphs are usually not used to predict several future values, we saw that predicting them by walking along the built graph provided accurate results for long-term prediction in most of the cases. We found that predictions of this type of sequences by statistical models such the graphs are computationally efficient combined with high prediction rates. On the other hand, the periodicity-based predictor showed its strength capturing large patterns such as observed in the Lu. This achievement, however, also involved a higher computational cost.

3.4 Predictability to improve the memory management

In the previous sections, the existence of an iterative pattern in the MPI message stream has been proven, which is a strong argument in favor of using prediction to improve the memory management in the communication system.

A characterization of the message stream behavior, in terms of message size and sender, has been done by instrumenting the MPICH implementation at two levels: the logical and the physical communication level. The results achieved have shown that logical communication is more predictable with an accuracy of over 90% (even in application with many collective operations). The predictor used for achieving those high-prediction rates is based on dynamic periodicity detection (DPD).

We have also studied long-term predictability of final buffer addresses, we have evaluated two prediction schemes, the one based on periodicity detection and a second one based on graphs. Our results indicate that the accuracy of long-term prediction is very high with both predictors. We identified an advantage of the periodicity-based predictor in capturing large patterns as in the Lu, but observed also a larger computational cost than in the graph-based predictor. If patterns are small and consist of different elements, the graph-based predictor showed to be computationally efficient combined with high prediction rates.

In this section we present a prediction mechanism that attempts to solve the scalability problems mentioned in section 3.1. Logical communication will be considered, we are interested in the prediction of message size and sender, and the predictor is called from the MPI_Recv calls inside the MPI library. These calls directly reflect the application structure. Finally, the dynamic periodicity detection (DPD) predictor has been used in our experiments.

3.4.1 The prediction mechanism

The mechanism was implemented inside MPICH 1.2.4. In this concrete implementation of MPI, the message information is divided into two parts: the message envelope, containing information about the message, and the data. The envelope is relatively small, and is delivered to the destination immediately. The data may or may not be delivered at once. There are three possibilities (figure 3.10): *Short protocol*, where the data is delivered within the message envelope. The *eager protocol*, where data is delivered without waiting for the receiver to request it, and *rendezvous protocol*, where the data is not delivered until the receiver confirms it.

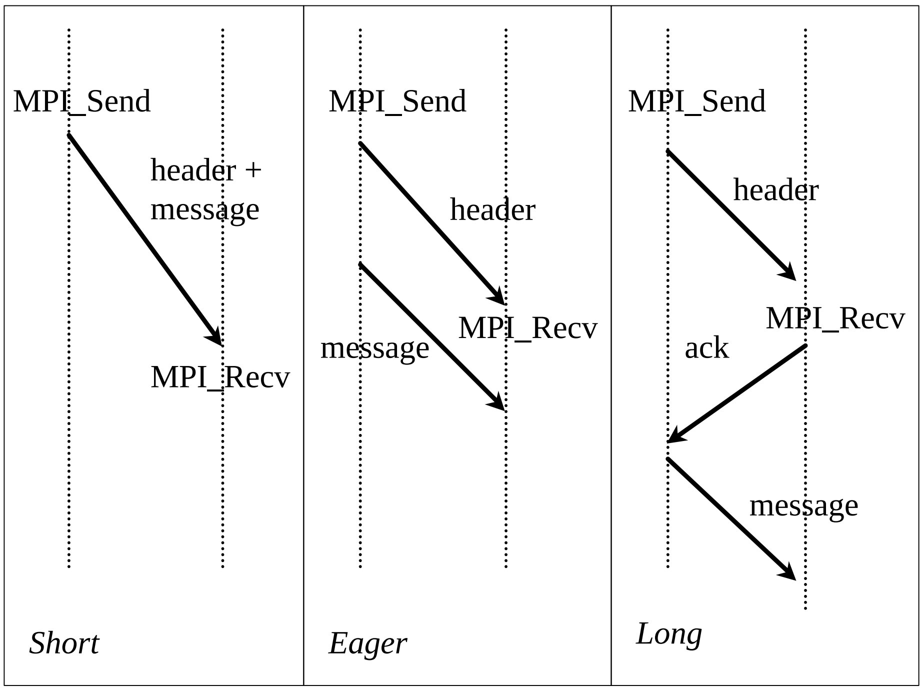


Figure 3.10 MPICH 1.2.4 Protocols depending on message size.

No single choice of protocol is optimal. In MPICH 1.2.4 the protocol is chosen depending on the message size. The idea is that for messages of intermediate length, the eager protocol may offer the best combination of performance and reliability, while for very long data, confirmation to receive the message is required to guarantee correct message reception therefore the rendezvous protocol is used.

Notice that, in this implementation, no pre-allocation of memory is done. Again, the problem is related to scalability. From the receiver point of view, if the matching receive call has been posted there is no problem because the message will be directly stored in the user's buffer. Otherwise, the message will be *unexpected*, and this is a problem because we need to partially store that message somewhere and neither the eager nor the short protocols guarantee that the receiver will have enough memory to receive this message. Thousands of nodes sending messages can flood up the receiver memory.

The prediction algorithm

Predicting the next messages would allow us to pre-allocate buffers at the receiver's side and send the permissions (credits) to the senders in advance. This gives us two advantages: first, we will control the flow of messages with potentially no performance loose. And second, we obtain a better control of the resources, we will avoid a waste of memory since memory will be allocated according to the application needs.

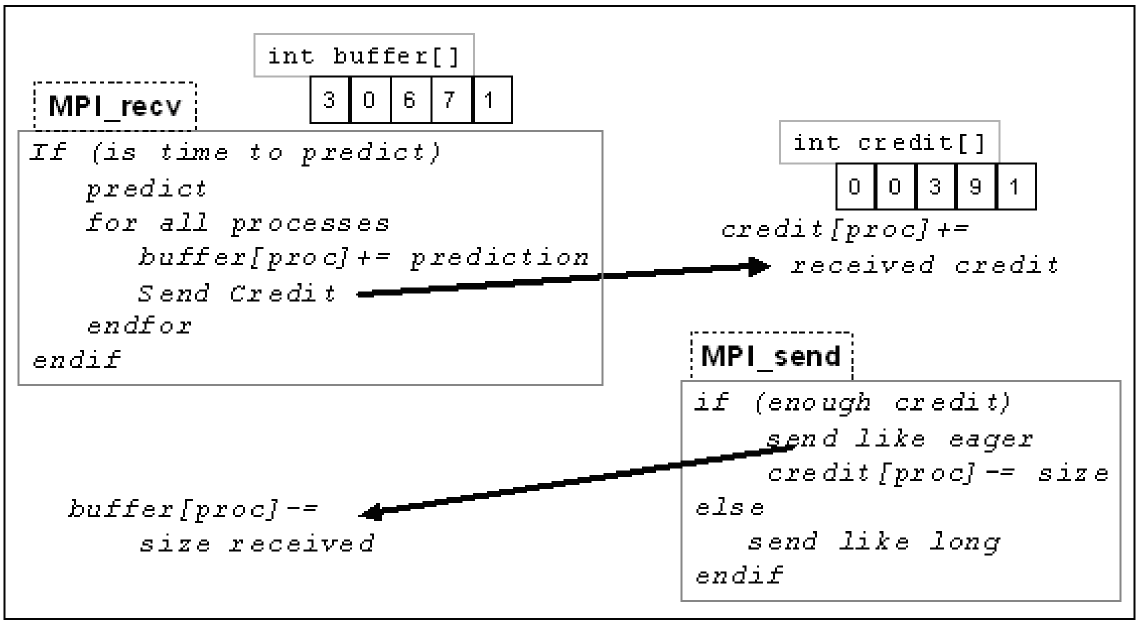


Figure 3.11 Prediction Mechanism.

Figure 3.11 shows the predictor mechanism. Every time a receive call is posted the predictor is called to update its history and only once every M messages, the future M messages to arrive are predicted; concretely we predict the sender and the size of the message. First, buffers are allocated on the receiver's side in a way that each process has a structure of one buffer per process and the size of these buffers is changing according with the prediction. After that, a message is sent to each predicted sender, with the size of the predicted messages, which acts as a credit, it means the sender is allowed to send messages up to this size eagerly. If credit is not enough the rendezvous protocol will be used.

Implementation considerations

Prediction is done with a lookahead of M messages in advance (configurable). This is required because logical prediction is used. It was chosen instead of physical prediction because of the higher prediction rates (section 3.2.3). Since our mechanism reacts to the physical reception of messages, predicting M messages in advance allows us to tolerate out-of order arrival of messages according to our logical predictions. Also there are other beneficial side effects: credit messages are reduced, the predictor overhead is reduced because it is asked for predictions once every M messages and the delay of credit message reception is hidden, softening the problem of credits arriving later than the predicted message reception.

In relation with the predictor, the dynamic periodicity detector (section 3.2.2) was used. To deal with the overhead introduced by the DPD due to the length of its historic, a *short DPD* was implemented. The short DPD is called when the pattern is considered to be stable, it only takes the pattern and returns it. The DPD is called again when there is a miss prediction. The number of times DPD is called is reduced and so is its overhead.

Considering the buffers, a single global buffer was used instead of a buffer for every other process reducing the time spend in managing the memory. Also, we allow the buffer to grow up to a limit, that can be adapted by means of an environment variable, to have better control of the memory resources.

Regarding the credit message distribution, they are piggybacked whenever possible. They are not sent immediately, instead we wait until a threshold (in this case we wait until M/2 messages are received) and piggyback credits into data messages. If the threshold is surpassed and there are still credit messages pending, these messages are sent explicitly.

And finally, messages up to a certain size (configurable) are considered as candidates to be sent eagerly. The predicted pattern takes into account these messages. There is a message size limit over which the time spend in the memory copy from the unexpected buffer to the user's buffer does not pay off, and the rendezvous protocol is faster.

3.4.2 The rendezvous protocol

We have always assumed that the *rendezvous protocol* provides a control flow and therefore it guarantees the correct message reception. Nevertheless, this is not true for every implementation of the `rendezvous` protocol. Again with the purpose of performance gain, in some MPI implementation the rendezvous protocol has been optimized to the detriment of control flow [7].

The `rendezvous` protocol relies on a `header` message (figure 3.10), a packet to ask for permission to send a data message. In these MPI implementations, upon the arrival of this `header` packet the *rendezvous* protocol reacts as follows:

- If a receive call matching control-data carried by the `header` has already been posted, no extra memory is needed because the message will be received into the user buffer of the posted request. The receiver replies with an *Acknowledgement* (`ACK` packet) permitting the sender to proceed with sending data packets.
- If the matching receive call has not been posted, but the receiver has enough memory, an MPI request and a message buffer are allocated dynamically, to be matched against a future posting. An `ACK` packet is sent to the sender. Data is accumulated in the unexpected buffer.
- If the matching receive has not been posted and the receiver is short on memory, MPI aborts execution and prints a message about insufficient memory.

The matching algorithm described above is insufficient because it aborts as soon as it runs out of memory, and message reception is not guaranteed.

We have modified the `rendezvous` protocol to guarantee control flow and correct user's data message reception as follows:

- If a receive call matching control-data carried by the `header` has already been posted, it works the same way. The receiver replies with an `ACK` packet, permitting the sender to proceed with data packets.
- If the matching receive call has not been posted, even if the receiver has plenty of memory, receiver waits for the matching receive call to be posted before sending the `ACK` packet to the sender.

Notice that with the version of the `rendezvous` protocol used in this chapter, temporary buffers are never required because data will always be received into the user's buffer, once the matching has been performed.

This solution based on the `rendezvous` protocol (*scalable rendezvous*) will also be considered and evaluated since it constitutes an easy to implement solution where user's data message reception is guaranteed.

3.4.3 Evaluation

Experimental framework: IBM RS-6000 SP machine

In our experiments we use applications from the NAS benchmark suites. We use the NAS BT, CG, LU, MG and SP, Class B problem size.

The first set of experiments was taken using MPICH 1.2.4 running on top of an IBM RS-6000 SP machine with a total of 128 processors (8 nodes with 16 processors per node), all nodes are connected through a SP Switch2 in a dedicated environment. More details about the overall machine architecture can be found in appendix A. We use the class B problem size of the NAS benchmarks.

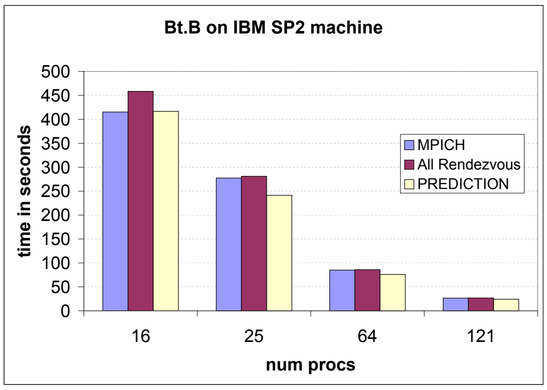
The default configuration of MPICH 1.2.4 sets thresholds as follows: short protocol is used for messages up to 1024 bytes, eager protocol up to 128 Kb and rendezvous protocol is used for longer messages.

Figure 3.12 compares application execution time of the NAS benchmark in three situations. First, the regular MPICH configuration (default). Second, MPICH is configured to send all messages with the `scalable rendezvous` protocol, where control flow is guaranteed, a handshake is performed for every communication and temporary space is not required. And finally our proposal based on the `prediction mechanism`. Parameters of the prediction mechanism were conveniently adjusted.

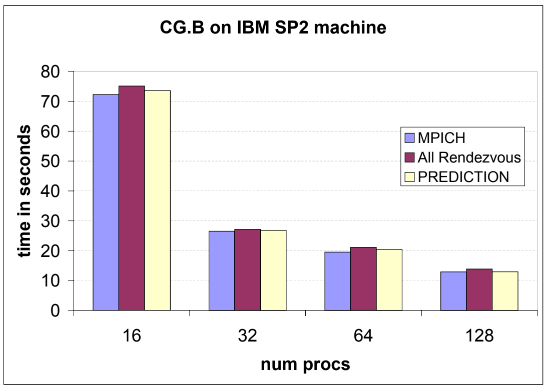
Figure 3.13 shows a different view of the same situation. It shows performance gain of the prediction mechanism and the scalable rendezvous approaches taking the default MPICH implementation as a base line.

It is observed that the price to pay for control flow is a performance degradation that increases with the number of processors. If control flow is guaranteed by using the `rendezvous protocol` degradation reaches 40% in the worse case in LU benchmark with 128 processors. The prediction mechanism guarantees control flow and performance degradation is softened, up to 18% in the worst case. In some cases, like BT prediction mechanism outperforms the default MPICH configuration.

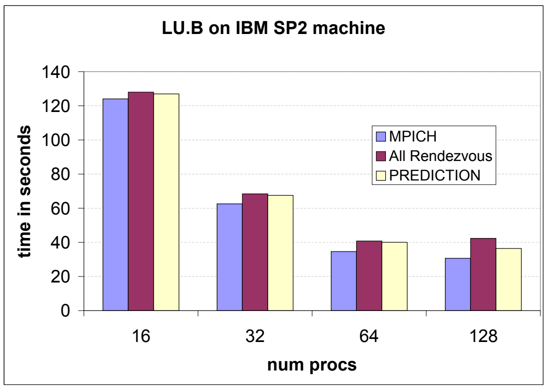
Concerning our Prediction mechanism, the source of improvement relies on converting messages sent through the *rendezvous* protocol to *eager* messages, which are believed to be more performant because: first, there is not any communication other than the data message and also, the handshake is avoided allowing sender process to



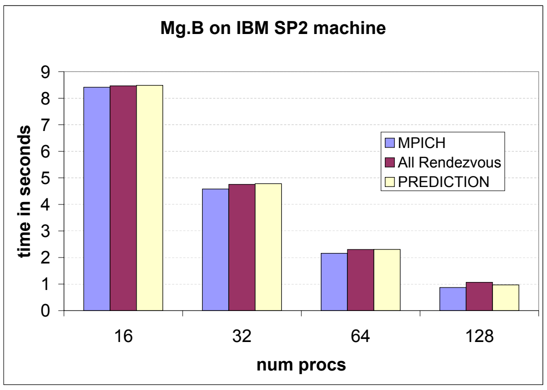
(a) BT



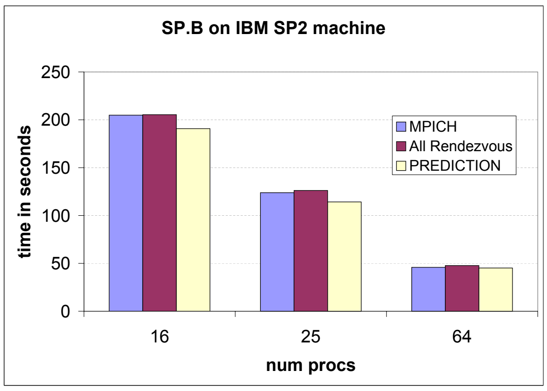
(b) CG



(c) LU

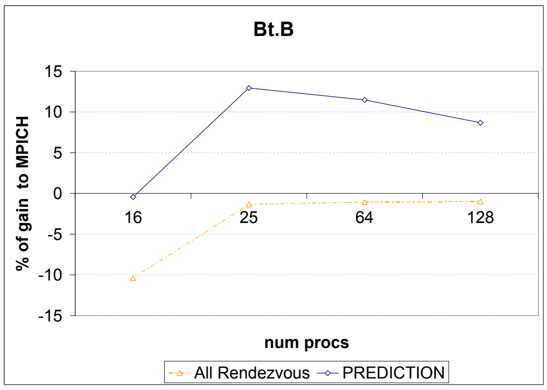


(d) MG

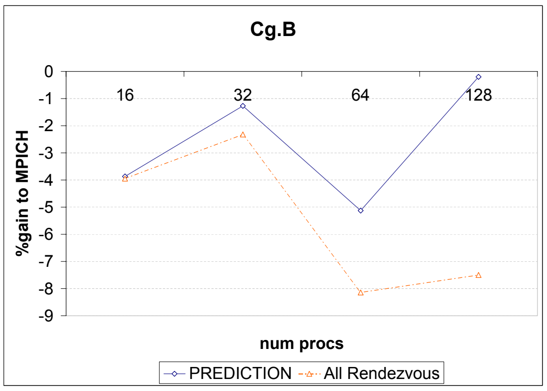


(e) SP

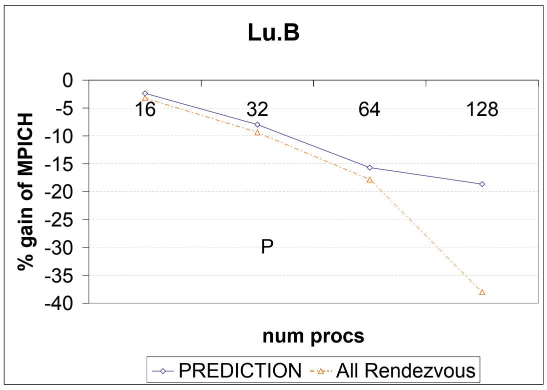
Figure 3.12 Performance evaluation of prediction mechanism using the NAS benchmarks in an IBM RS6000. X axis represent the number of processes and Y axis is the execution time in seconds. MPICH default configuration (first column) is compared with MPICH using rendezvous protocol for all messages (second column) and MPICH with our prediction mechanism (third column).



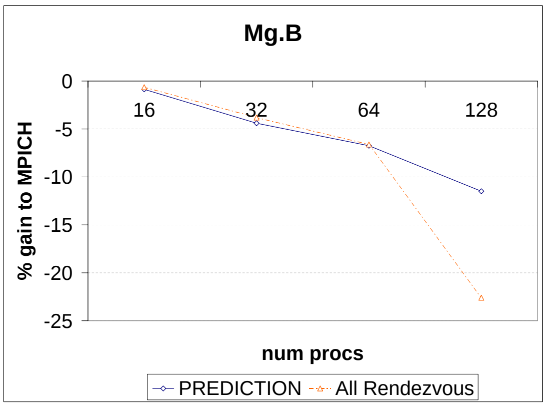
(a) BT Performance Gain



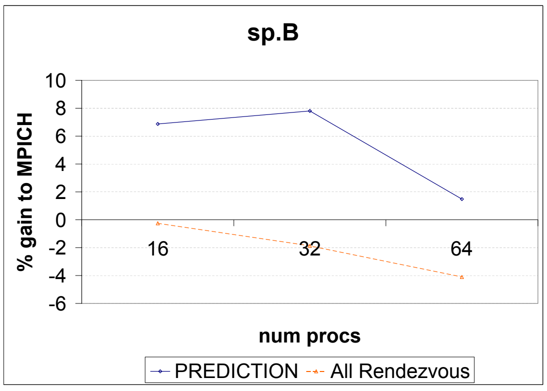
(b) CG Performance Gain



(c) LU Performance Gain



(d) MG Performance Gain



(e) SP Performance Gain

Figure 3.13 Performance benefit of prediction mechanism using the NAS benchmarks in an IBM RS6000. MPICH default configuration is taken as a baseline and the percentage of gain is computed for both `rendezvous` and `prediction`. `X` axis represent the number of processes and `Y` axis is the percentage computed as: $\frac{100(Z-W)}{Z}$, with `Z` being the default execution time, and `W` being the execution time with all rendezvous or with prediction.

make progress immediately. However, two main sources of overhead were detected: (i) when the message is received *unexpectedly*, *eager* protocol implies a memory copy that, depending on the message size, may be expensive; (ii) depending on the application, the communication pattern is complex and the predictor overhead increases.

To face (i), a zero-copy mechanism has been proposed based on prediction. It is explained in section 3.3. As a summary, the idea is to predict the final user's buffer address and allocate the temporary buffer aligned with it. This will allow to perform the memory copy from the temporary buffer to the user buffer by simply switching entries in the page table. Despite the high predictability rates, the need of a kernel modification prevents us from implementing the idea.

Concerning (ii) a graph-based predictor 3.3 has been studied. It reduces the overhead and obtains high prediction rates, if patterns are small and consist of different elements. However, the accuracy is damaged in the case of LU, where the DPD predictor overhead is higher, which discourages us from changing predictor.

According to our experiments up to 128 processes, the prediction mechanism improves the robustness of MPI memory management outperforming the *all rendezvous* solution. Nevertheless, the downward trend followed by both lines casts doubt on the reliability of this solution when extended to thousands of nodes. Some more testing would be needed with a larger number of nodes.

Experimental framework: MareNostrum supercomputer

A subset of 512 nodes of the MareNostrum [32] supercomputer were used in our experiments, each node with two dual core, interconnected through a 2Gb Myrinet network (Appendix A contains more detailed information about the overall machine architecture). The MPI implementation used is MPICH-GM 1.2.6 [140] which is based on MPICH 1 over the GM driver [89].

Class C problem size of NAS Benchmarks Suite was used to compare the MPICH default configuration against the scalable *rendezvous*, where control flow was guaranteed. Figures 3.14 and 3.15 shows the results: For all the benchmarks, except LU, *rendezvous* protocol performs acceptably good, with a 10% performance degradation in the worse case, and outperforming the default configuration in IS or CG. LU shows a different situation, performance degradation increases with the number of nodes up to a 100%.

Comparing with our results in the IBM SP-R6000 it is noticed that the overhead added by the *rendezvous* protocol has been reduced, because network latency is lower in MN, and in some cases the rendezvous protocol outperforms the default configuration.

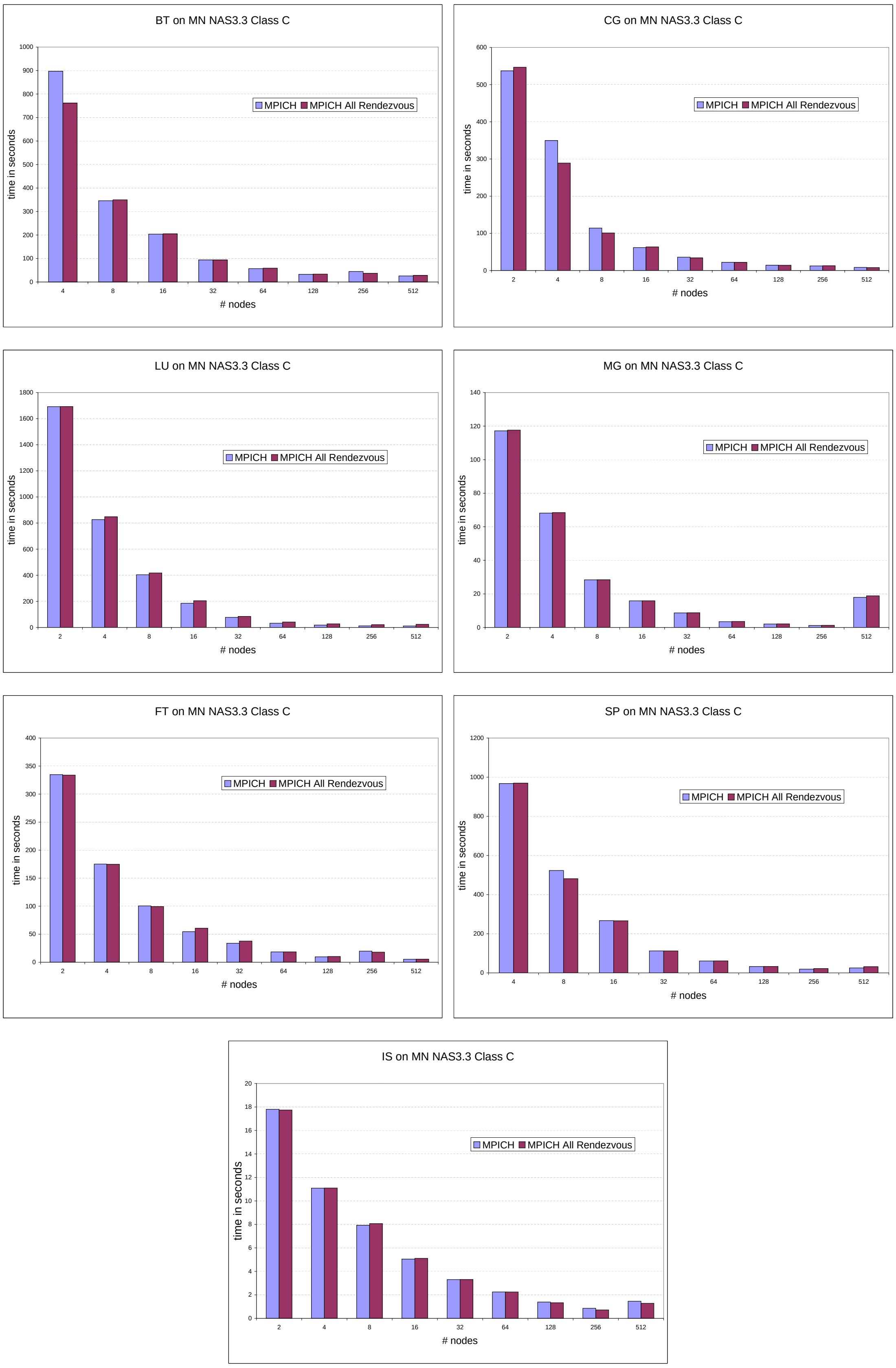


Figure 3.14 Performance evaluation of rendezvous protocol using the NAS benchmarks in MareNostrum. X axis represent the number of processes and Y axis is the execution time in seconds. MPICH default configuration (first column) is compared with MPICH using rendezvous protocol for all messages (second column).

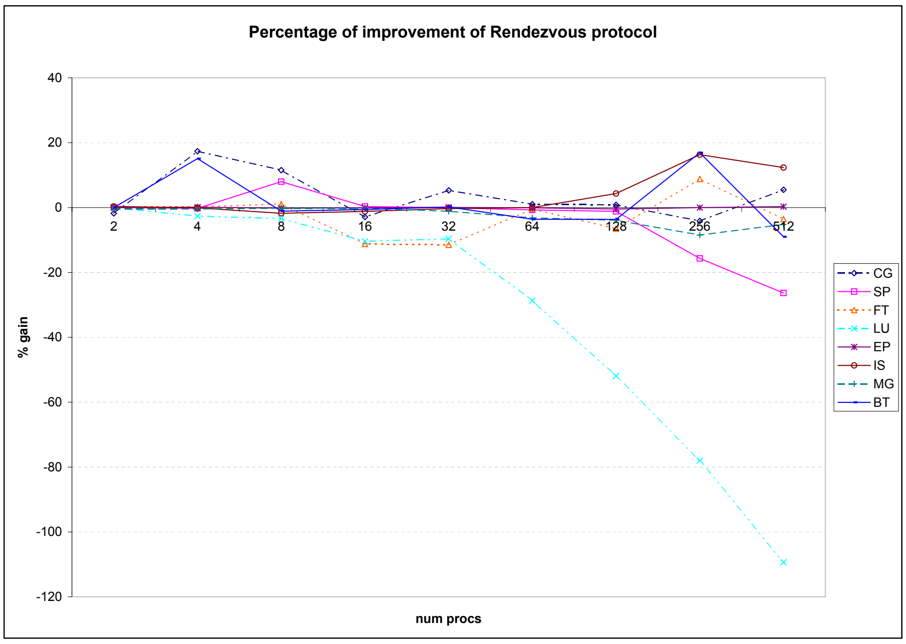


Figure 3.15 Performance benefit of rendezvous protocol using the NAS benchmark suite in MareNostrum: a line represents a benchmark. MPICH default configuration is taken as a baseline and the percentage of gain is computed for `rendezvous`. X axis represent the number of processes and Y axis is the percentage computed as: $\frac{100(Z-W)}{Z}$, with Z being the default execution time, and W being the execution time with all rendezvous.

Experimental framework: BlueGene/L supercomputer

This measurements were made on a BlueGene/L rack (2048 processors) using the BlueGene/L communication library. BlueGene/L [86] overall architecture is covered in appendix A.

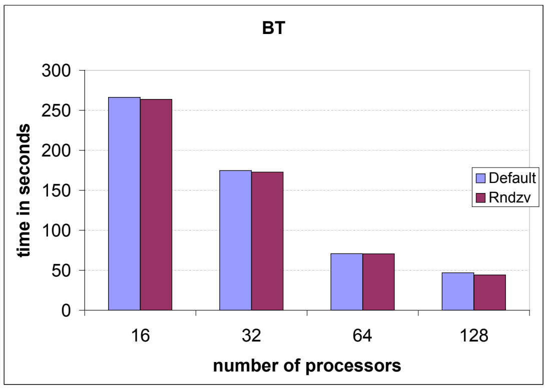
Class C problem size of NAS Benchmarks Suite was used to compare the MPICH default configuration against the scalable *rendezvous*. Figure 3.16 shows the results: for all the benchmarks *rendezvous* protocol performs acceptably good, it does outperform the default configuration in some cases (BT,CG or SP) and even in the case of LU, it performs reasonably good.

There are a couple of reasons for this behavior. First, the network: one important consideration about the BlueGene/L network is that due to the way packets are routed (minimal adaptive routing), ordinary data packets can arrive out of order. In any data transfer, the first packet contains information not repeated in other packets such as the message length and MPI matching information, if this packet is delayed in the network the receiver is unable to handle the subsequent packets, and has to allocate temporary buffers or discard the packets, with obvious undesirable consequences. Although packets can be forced to arrive in order, doing so with a large number of packets tends to create hot-spots in the network, decreasing overall throughput.

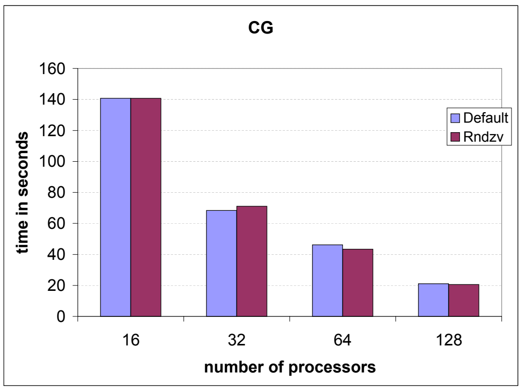
In practical terms this means that any data transfers that require more than one packet of data sent thought the eager protocol needs to use ordered packets. If the first packet is acknowledged (i.e. in the rendezvous protocol), further packets can be dynamically routed. The network routing characteristics penalize eager protocol in favor of rendezvous.

And second, memory copies are inordinately expensive on BlueGene/L, mainly due to its slow processor (Power-PC at 700MHz of clock frequency). A processor with modest performance was chosen for the BlueGene family because of the clear performance/power advantage of such a core, which was one of the goals for the Blue Gene design. Since the eager protocol may imply a memory copy from the *unexpected* to the user's buffer, it is being penalized by this expensive memory copies on BlueGene/L.

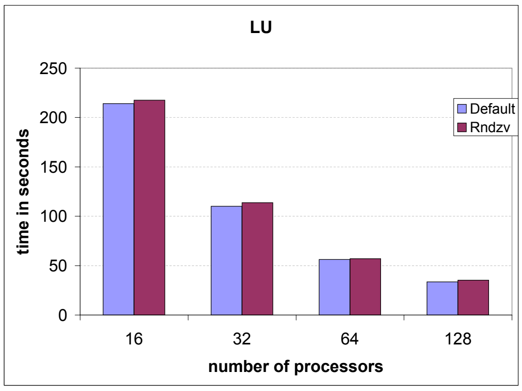
This results obtained in two supercomputers, such as MareNostrum and BlueGene/L, justify that most current supercomputers have disregarded the *eager* protocol in favor of the *rendezvous* for every message size. Control flow is guaranteed and with it the correct user's data reception, performance is improved in some cases and slightly degraded in the worst case.



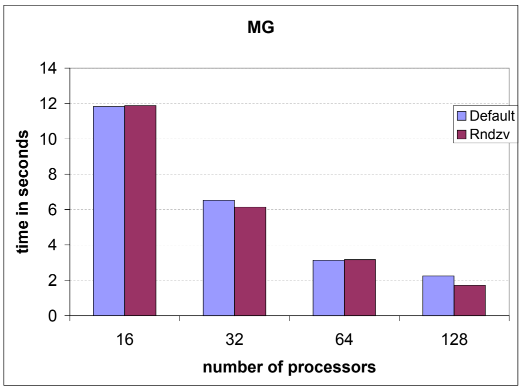
(a) BT



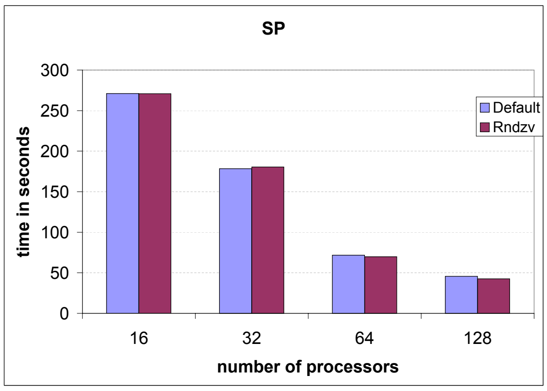
(b) CG



(c) LU



(d) MG



(e) SP

Figure 3.16 Performance evaluation of rendezvous protocol using the NAS benchmarks in the BlueGene/L. X axis represent the number of processes and Y axis is the execution time in seconds. MPICH default configuration (first column) is compared with MPICH using rendezvous protocol for all messages (second column).

3.5 Conclusions

We have investigated the predictability of MPI communication streams at both the physical and logical levels. We have concluded that communication streams are very predictable at the logical level, with an accuracy of over 90% (even in application with many collective operations). We have also seen that prediction at the physical level is not as accurate in some cases, due to the randomness of the environment.

We have proposed a memory management mechanism based on prediction to solve the scalability problems detected by taking advantage of the high-predictability observed. The mechanism has been implemented and evaluated in an IBM RS/6000 machine resulting in a more reliable implementation of the library with a little overhead, but not good enough due to the cost of memory copies. We identified the problem and found a solution, but it needs kernel support to be implemented.

We have also shown how in architectures with large number of nodes and high-performance networks, like MareNostrum or BlueGene/L supercomputers, a simpler solution consisting of using a slightly modified version of the *rendezvous* protocol for all messages can also be applied to guarantee control flow and user's data message reception with a reasonable overhead. Thus, the prediction-based mechanism seems to be unnecessary for modern architectures.

The added value of our prediction mechanism is that it eliminates the handshake between the two processes involved in the communication and reduces the amount of control-data which may be very beneficial in high-latency networks.

The trend followed by current supercomputers goes for thousands of ordinary CPU connected through increasingly faster high-performance networks which makes the rendezvous protocol a good choice to guarantee control flow. However, there are applications (we have shown LU) and architectures (more CPU power and lower performance network) where this protocol is not suitable and our proposal would benefit.

Nevertheless, we are also confident that the research community will be able to use the high predictability demonstrated here to solve many other problems. Prediction could still be applied by other means.

To summarize, the contribution of this work is first the characterization of the temporal communication patterns of MPI applications. Second, we proposed two prediction schemes, one based on periodicity detection and another based on a graph. Both achieve high prediction rates. And third, we proposed a memory management mechanism based on prediction that solved the scalability problems detected with little overhead and compare it against a control flow solution based on the `rendezvous` protocol, also interesting due to its simplicity.

3.5.1 Our Prediction mechanism in UPC

Finally, this study has been carried on in MPI because this problem does not necessarily exist in UPC. Unlike MPI, UPC relies on one-sided communication, message matching is not required therefore a message can not be *unexpected* and temporary buffers are not required. GET and PUT operations take place over shared memory locations and the user is responsible for the allocation of shared memory areas before any data transfer involving these areas occurs. Communication takes place directly into the user's memory area. However at a lower level, temporary space could be used to optimize short message reception but even in this case the problem does not exist because the temporary space is not tight to a matching receive call. Therefore, two buffers for every message size we want to speed up would be enough. For UPC Runtime implementations on top of a two-sided messaging system, such as MPI, the same proposed solution applies.

Chapter 4

Control flow for control data

4.1 Introduction

This chapter covers the second part of our first contribution, which focuses on control data reception in short-memory MPPs (such as BG/L). This work was carried out in MPI [130] because it is a highly popular programming model for today's parallel machines.

In the MPI programming model, the receiver process explicitly demands to receive a message (posting i.e. a MPI_Recv call) and arriving messages need to match with posted receives. When a new message arrives, some information needs to be stored somewhere to manage the message reception (hereafter `control-data`). This control-data is kept in the MPI library until a matching receive call is posted. Once the matching has been done, the space for control-data can be freed, but if a process receives lots of `unexpected messages` (messages received before the matching receive call has been posted), even if only control-data is kept, receiver process will be unable to handle all message reception and the application would crash. Notice that when a receive call is posted, space for metadata is provided by the user, while when a message just arrives at receiver's side unexpectedly, space for its metadata has to be provided by the MPI library itself.

The memory needed increases with the number of messages, which at its turn increases with the number of nodes. It can be seen in the top500 [188] list that current parallel computers tend to increase the number of nodes, today we talk about hundreds of thousands of processes. Thus, this situation is not unlikely.

Most current MPI implementations, assume to have enough memory to allocate control-data dynamically. However current MPPs trend to have short memory per node and large number of nodes, which makes memory a limitation, and control-data a problem that most current MPI implementations do not face.

As we have shown in previous chapter, when a message is unexpected, the message itself, the user's data, may also flood up receiver's memory. This problem has been widely discussed in the previous chapter and one of the proposed solutions will be adopted here.

Finally, to test a real system where MPI has the above-mentioned scalability problems, let us use BGL supercomputer [120] [61]. It provides a simple, flat, fixed-size 512Mb address space, with no paging, which limits the memory resources. This limitation requires a better control of the memory management, not only for data but also for control-data.

Although our work has been focused on BGL machine, supercomputers design gives high priority to efficiency and page fault management is very expensive in terms of performance, thus supercomputers tend to disable the virtual memory mechanism, stressing even more the memory management system. This is the case of BG/L running blrst [23], RedStorm [57] running a lightweight kernel called Catamount [182], ASCI Red [10] [164] among others [30]. Therefore our solution may be applied to these other large scale MPPs supercomputers sharing similar issues and to future designs. Moreover, some of our proposed solutions may be also reused in architectures based on embedded processors.

In this chapter we describe the mechanism we have used to solve the above-mentioned problems thus make MPI scalable to any number of nodes. We present one algorithm that improves the robustness of MPI implementations for short-memory MPPs, taking care of data and control-data reception. The proposed solution achieves this goal without any observable overhead when there are no memory problems. Furthermore, in the worst case, when memory resources are extremely scarce, the overhead will never double the execution time (and we should never forget that in this extreme situation, traditional MPI implementations would fail to execute).

4.2 Framework

In order to clearly understand the proposed solutions, we must first see the complete framework where we work. First we present a quick overview of the machine Blue Gene Light, that will show the most important characteristics for our proposal. Then a brief overview of the design of MPI over the BlueGene/L machine and finally some relevant comments related to scalability and memory management.

4.2.1 BlueGene/L

The BlueGene/L supercomputer [141] is a 65,536-node machine designed around embedded Power-PC processors. The machine was later upgraded up to 106,496 nodes. Each BlueGene/L node carries 512 MBytes of DDR memory. And nodes are connected through a high speed three-dimensional torus network.

Whereas existing large scale systems range in size from hundreds (ASCII White, Earth Simulator) to a few thousands (Cplant, ASCII Red, Mare Nostrum) of compute nodes, BlueGene/L makes a jump of almost two orders of magnitude. The problem of scalable resource management is addressed in BlueGene/L through a hierarchical approach. More details concerning the overall machine architecture are covered in the appendix A).

The large configuration and memory shortage of BlueGene/L makes it the perfect platform to carry out our experiments.

System software overview

The system software for BlueGene/L [85] is a combination of standard and custom solutions. Certain nodes called the I/O nodes execute a version of the Linux kernel for embedded processors but no user code directly executes on the I/O nodes. User code runs on compute nodes, which execute a single user, single process minimalist custom kernel, and are dedicated to efficiently run user applications.

The control software running on compute nodes, it is a minimalist POSIX compliant compute node kernel (CNK) which provides a simple, flat, fixed-size address space with no paging. The kernel and application program share the same address space, with the kernel residing in protected memory. The kernel protects itself from the application by appropriately programming the PowerPC MMU.

4.2.2 MPI on BG/L

The BlueGene/L communication software architecture [88] is divided into three layers. The torus/tree **packet layer** is a thin layer of software designed to abstract and simplify access to hardware by means of three main functions: initialization, packet send and packet receive.

The **message layer** is an active message system [191], built on top of the packet layer, that allows the transmission of arbitrary buffers among compute nodes.

The third component of the MPI software stack is **MPICH2**, an interface of MPI developed at Argonne National Laboratory. Figure 4.1 shows the software architecture of the MPI software stack.

A well known issue in the design of communication protocols is choosing at which level the different protocol functionalities are best implemented; in particular, the memory management mechanism has been the focus of considerable attention in previous projects (AM[191], U-Net [183], xKernel [100], [195], FM[150])

Because most of the needed information is in the abstract device implementation layer of the MPICH2 library (denoted `bgltorus` in Figure 4.1), the mechanism has been implemented in this layer, and only slight modifications have been done in the message layer.

Memory management: preliminaries

The design of the message layer was influenced by specific BlueGene/L hardware features, such as network reliability, packetization and alignment restrictions, out-of-order

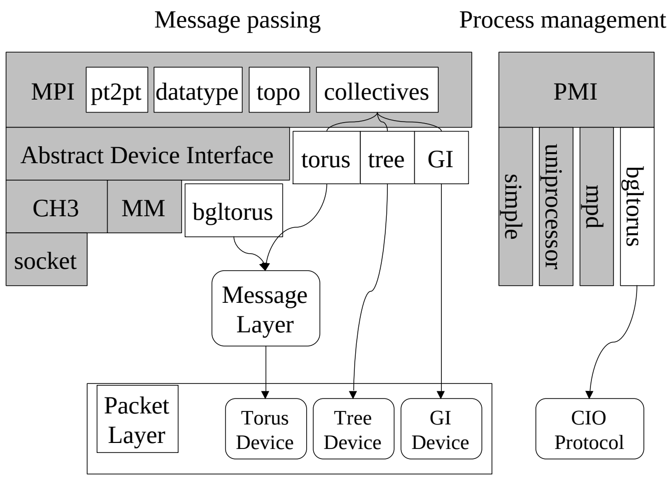


Figure 4.1 The BlueGene/L MPI architecture

arrival of torus packets, and the existence of non-coherent processors in a chip. Together with these hardware features, there is the requirement for a low overhead scalable solution.

Scaling issues and virtual connections: In MPICH2, point to point communication is executed over virtual connections between pairs of nodes. A factor limiting scalability is the amount of memory needed by an MPI process to maintain state for each virtual connection. The current design of the message layer uses only about 50 bytes of data for every virtual connection for the torus coordinates of the peer, pointer sets for the send and receive queues, and state information. Even so, 65,536 virtual connections add up to 3 MBytes of main memory per node, not more than 1% of the available total (512 MBytes), just to maintain the connection table.

Communication protocol in the message layer: The protocol design is crucial because it is influenced by virtually every aspect of the BlueGene/L system: the reliability and out of order nature of the network, scalability issues, and latency and bandwidth requirements.

For every message, the transmitted information consists of a message envelope, containing control-data, and the data. The envelope is relatively small, and is delivered to the destination immediately. The data may or may not be delivered at once. As you can see in figure 4.2, BlueGene/L message layer offers three possibilities: eager, short or rendezvous protocols.

Because of the reliable nature of the network no acknowledgments are needed. A simple "fire and forget" eager protocol is a viable proposition. Any packet out the

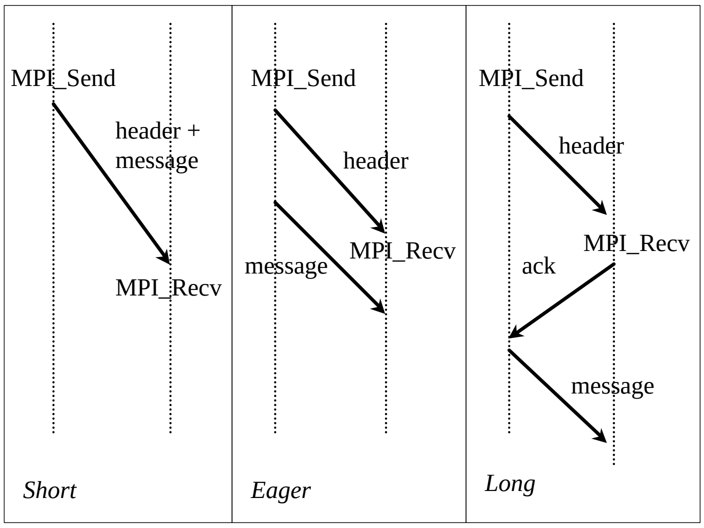


Figure 4.2 MPI Protocols.

send FIFO can be considered safely received by the other end. A special case of the eager protocol is represented by single-packet messages, which should be handled with a minimum of overhead to achieve good latency.

The message protocol is also influenced by out-of-order arrival of packets. The first packet of any message contains information not repeated in other packets, such as the message length and MPI matching information. If this packet is delayed on the network, the receiver is unable to handle the subsequent packets, and has to allocate temporary buffers or discard the packets, with obvious undesirable consequences. This problem gets solved if the first packet is always acknowledged (i.e. in the rendezvous protocol) and next packets can be dynamically routed.

Memory usage issues: In the BlueGene/L MPI library no pre-allocation of memory is done, so unexpected messages, have to be received into a temporally buffer (called the *unexpected message buffer*) created dynamically. This is true for both eager and rendezvous protocols. Simultaneous unexpected messages from thousands of nodes can flood the receiver memory.

Moreover, there is another problem: even if the size of theses messages is 0 bytes, or the body of the message is not send, they would consume some memory space because control-data, which is delivered immediately into the message envelope, needs to be saved until it matches a posted receive.

The outstanding control-data messages we need to support at any time increases with the number of processors, regardless of the size of the message, as memory is short, the control-data alone can flood the receiver memory.

Application developers typically think of limits in terms of message sizes rather than message counts. To an application developer, a message of size 0 shouldn't consume any buffer space at all.

The problem of simultaneous unexpected data messages has been faced before [150, 37, 116, 29], and it has been discussed in the previous chapter 3. We need to solve this problem as well, the more suitable approach was selected to be implemented as a base for our memory management protocol. A credit-based flow control scheme was considered [37, 116]. It would allow the receiver to control incoming traffic, overcoming eager protocol's main limitation and avoiding the 3-handshake protocol which usually increases latency. On the other side, an acknowledgment-based scheme of control flow [65, 135] 3.4.2 is more simple to implement with dynamic routing of data packets since it does not have to deal with the out-of-order arrival of packets. Therefore, an acknowledgment-based scheme of flow control was chosen for our memory management mechanism.

4.3 The proposed memory management algorithm

At this point we have shown that the performance and scalability of an MPI implementation may depend on the memory management techniques used by the library. Naive implementations of the MPI standard require more and more memory as the number of processors increases.

Our approach allows the MPI library to reside in a fixed amount of memory. We use a variant of the rendezvous protocol to implement memory control 3.4.2. The implementation must not degrade performance (e.g. by increasing latency) in the common case when memory is plentiful.

We start out with a short description of a simple rendezvous protocol that we used as the base case for our optimized implementation, a similar approach has already been presented in section 3.4.2. This will solve the problem of user's data, and **then modify it to control memory usage for control-data, which is the main contribution of this work**.

4.3.1 High level overview of algorithm

The algorithm relies on an *Request To Send*, or RTS, packet to ask for permission to send a data message. Upon the arrival of an RTS packet the receiver is faced with one of multiple situations. A naive implementation of MPI reacts as follows:

- If a receive call matching control-data carried by the RTS already exists in the *posted request queue*, no extra memory is needed because the message will be received into the user buffer of the posted request. The receiver replies with a *Clear To Send* packet (*CTS*), permitting the sender to proceed with data packets.
- If the matching receive call has not been posted, but the receiver has plenty of memory, an MPI request and a message buffer are allocated in the *unexpected request queue*, to be matched against a future posting. A CTS packet is sent to the sender. Data is accumulated in the unexpected buffer.
- If the matching receive has not been posted and the receiver is short on memory, MPI aborts execution and prints a message about insufficient memory.

The matching algorithm described above is insufficient because (a) it uses heap memory for unexpected messages and (b) aborts as soon as it runs out of the heap.

Our optimized algorithm leaves the system's behavior unchanged in situations where memory is plentiful, but deals with low memory situations differently. Our aim is for MPI to make progress in these situations, avoiding deadlock or livelock while respecting MPI ordering semantics. We make use of the fact that MPI ordering semantics requires message *matching* to be in the order in which messages were sent but does not require message *delivery* to be in the same order.

Not enough memory: The first case we consider is when the matching receive has not been posted at the receiver and there is not enough memory to allocate an unexpected buffer, but there is enough memory to allocate an MPI request in the unexpected message queue. In other words, memory at the receiver is enough to perform the message matching but not enough to hold the actual message. The receiver does not allocate buffer memory and puts message delivery from the sender on hold.

The sender has to hold on to send data until a `CTS` packet arrives from the receiver. However, since message matching has been performed normally and in order, the sender is free to attempt to send subsequent messages to the receiver. Even though these later messages may actually complete sooner than the original refused message, this is not a violation of MPI semantics – after all message *matching* has been performed in order.

The message put on hold by the receiver will be delivered when the matching MPI receive is finally posted (which in a correct MPI program is bound to happen before the program ends).

Very little memory: The second case is more complex. Here is the novelty of our work. It occurs when the incoming message finds no matching receive, and the receiver is so short on memory that it cannot post the MPI request in the unexpected queue.

In this situation the receiver cannot perform the matching. It sends a *No Clear To Send Packet (NCTS)* and drops the message. The sender saves the message order by holding the message in the *pending send queue*. The sender is now responsible for keeping the order of the messages it is trying to send, and to retry them in order when prompted by the receiver. In effect we are extending the receiver's *unexpected* queue to the senders.

4.3.2 Data structures

Before we proceed to describe the algorithm implementation we need to introduce the data structures in our algorithm. The well known MPI message matching algorithm provides two request queues, one for messages *posted* by the receiver that have not yet arrived, and another queue for messages that were *unexpected* at the receiver but have arrived anyway.

Our algorithm is designed to cope with low memory situations at the receiving end. The intuitive idea is to extend the *posted* and *unexpected* queues to the senders in order to relieve the receiver from the burden of maintaining them in a low memory situation.

Thus we introduce an ordered *pending send queue* at every sender. Any MPI messages intended to be sent is first inserted into the pending send queue and is only removed when accepted into the receiver's own *posted* or *unexpected* queue.

The *failed match bit array* is an $O(n)$ array of bits (where n is the number of nodes in the MPI job). The bit corresponding to a process is set if message matching failed due to memory shortage at the destination.

The *age* array is another $O(n)$ array that is used to count the number of pending send queue restarts that have happened so far to a particular process. Every time the receiver recovers from a very low memory situation, communication is restarted with a different age to disambiguate message ordering.

Emergency memory is used by the receiver to communicate to the senders of pending messages when memory is low $O(1)$.

4.3.3 Detailed description

In this section we describe the algorithm in detail by following the path of a point-to-point message in our MPI protocol.

The message originates as an MPI request object at the sender, where it is enqueued into the pending send queue. Negotiation starts when an `RTS` (Request To Send) packet is sent from the sender to the receiver. The RTS packet carries control-data with the current age value of the sender. Upon arrival to the receiver a match against the *posted* queue is attempted. Multiple possibilities arise depending on the result of the match, available memory at the receiver and age discrepancy between sender and receiver.

Successful message match: Figures 4.3 and 4.4 describe the relatively simple case when the incoming message at the receiver is either matched in the posted queue or else there is enough memory to hold the incoming message until it is posted (in which case it gets inserted into the unexpected queue). The age value carried by the RTSăpacket must match the receiver's current age. Both these situations cause the receiver to reply with a CTS (Clear To Send) packet carrying the address of the (found or allocated) message buffer.

At the sender's site the MPI Request object is removed from the top of the pending send queue. This marks the end of the negotiation phase for this message. The sender proceeds to send the message data to the receiver in the usual way.

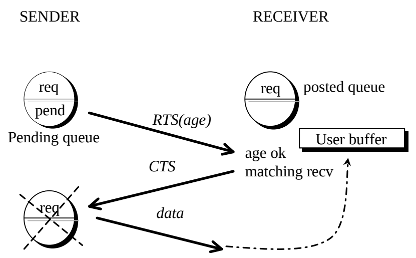


Figure 4.3 Age OK and matching MPI_recv

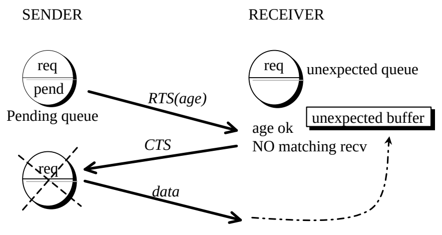


Figure 4.4 Age OK, no matching MPI_recv and enough memory

Age mismatch: Age mismatch means that memory conditions have changed since the message was sent, and so the message may be received out of order because previous messages may have been rejected. The protocol deals with this situation as follows: if the age value carried by the RTS packet does not match (figure 4.5), the receiver replies with a NCTS (Not Clear To Send) packet. At the sender this will cause the send request to stay in the pending send queue.

However, even in case of an age mismatch the receiver still checks the RTS packet against the posted queue. If there is a match (Figure 4.6), the receiver sends off a resendAll packet as well. The sender copies the age value from the packet and resends all the messages currently in the pending send queue, in the correct order, with the new age value.

Receiver low on memory: Figure 4.7 shows the situation where the age matches and the receiver has enough memory to create a matching record in the unexpected queue,

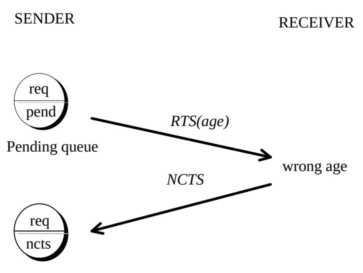


Figure 4.5 Age NOT OK and no matching MPI_recv

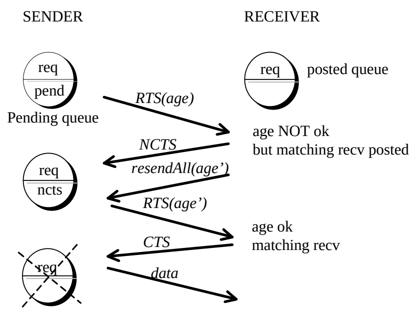


Figure 4.6 Age NOT OK and matching MPI_recv

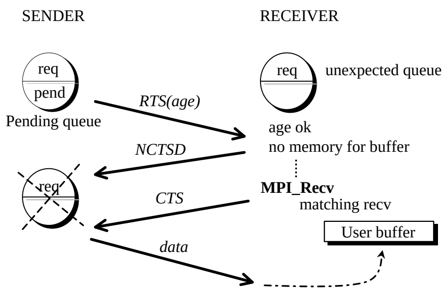


Figure 4.7 Enough memory for request but NOT for the buffer: delay data transmission

but not enough memory to hold the incoming message buffer. This causes the receiver to create an entry in the unexpected queue, using some memory from the heap. A `NCTSD` (Not Clear To Send Data) reply packet is sent back to the sender.

Upon receipt of a `NCTSD` packet the sender is allowed to remove the send request from the pending send queue (reader should notice that it differs from `NCTS` on this point). The sender is not allowed to send the actual data because there is no assigned buffer for it at the other side. However, since a record of the attempted match now exists at the receiver the sender is free to send other MPI send requests to the receiver.

Receiver very low on memory: Figure 4.8 displays the case where the receiver does not have enough memory to allocate a record of the missed match in the unexpected queue. The sender has queued the request into the pending queue before sending off the `RTS`. The receiver has to resort to *emergency memory* to send a `NCTS` reply. The receiver also sets the *match failed* bit for the sender.

The `NCTS` packet causes the sender to keep the current message in the pending send

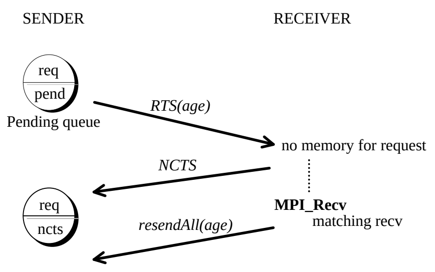


Figure 4.8 Not enough memory for request on receiver's side

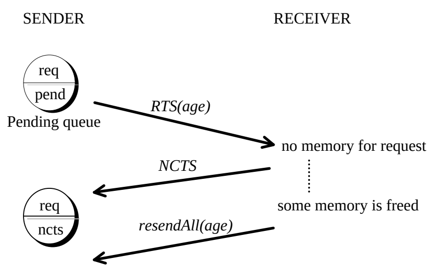


Figure 4.9 Some memory is freed

queue. This is a remedial action for making a record in the receiver's unexpected queue (the local pending send queue becomes an extension of the receiver's unexpected queue).

To allow the pending send queue to drain, one of two things has to happen at the receiver: either a new MPI request arrives into the *posted* queue or heap memory is freed through the `free()` library call (Figure 4.9). When either of these events happens, the senders have to be notified (by means of a `resendAll` packet) to re-try their message queues. The `resendAll` packet also carries a **new** age value to the senders. This ensures that stale data from the senders carrying the old age value will not be matched by the receiver, preserving MPI ordering semantics.

In order to ensure that the pending send queues are restarted when memory frees up in the receiver, our implementation intercepts calls to the standard `free()` library function, age is updated and `resendAll` message is sent.

Restarting the senders' pending send queues may require further use of the emergency memory by the receiver; in the situation where memory is very low but an MPI_Recv request is posted at the receiver communication has to restart without any additional memory resources. In this situation the `resendAll` packet is sent from emergency memory.

Emergency memory is not needed when communication is restarted because memory has been freed. If not enough memory has been freed to allow the sending of a `resendAll` packet it is likely not worth restarting communication.

Reacting to a `resendAll`: The sender starts sending all messages in the pending send queue whenever a `resendAll` packet arrives. The sender has to stop sending `RTS` packets when it reaches a send request that has not received an answer from the receiver yet. Valid answers are `CTS`, `NCTS` and `NCTSD`. If `CTS` or `NCTSD` is received the message is de-queued and thus it will not be resent. The sender is forbidden from sending an `RTS` packet for any message in the queue that has not been replied to yet, because that would potentially violate MPI ordering semantics.

4.3.4 Deadlock issues

In this section we discuss how our improved algorithm differs from naive MPI message matching in terms of hanging behavior and deadlocks. Progress in an MPI program stops when the sender attempts to send a message but the receiver ignores or drops the data. This can happen in a number of situations.

- One BlueGene/L messages *cannot* be lost during network transport because the network is reliable and guarantees the arrival of each packet.

- A situation in which a node stops draining the network *can* occur if an **incorrect** MPI program stops calling MPI routines. A BlueGene node will never stop draining incoming network packets as long as MPI routines are called. Thus, in a correct MPI program deadlock due to nodes sending data but not processing incoming packets cannot occur.

- **Incorrect** MPI programs can cause communication to be stalled permanently by not posting a matching receive for each sent message. Our algorithm ensures that deadlock does not occur if the program we are executing is correct. Although communication can temporarily stall because of lack of resources, a matching receive for *every* send will eventually be posted, causing communication to be restarted and the pending send queues to be systematically emptied. Therefore, deadlock due to insufficient memory resources in an MPI process cannot occur in a correct program. As long as receives are posted sender/receive stalls will be dissolved and communication will get restarted.

  We want to point that according to MPI specification [131] a correct MPI application should not rely in the underlaying implementation for its correctness, therefore using a acknowledge-based protocol for sending all messages does not change MPI semantics.

  However, we have to note that while **incorrect** MPI programs would cause the regular MPI implementation to overflow at the receiver and to abort execution, the advanced message matching algorithm may cause MPI to hang. We believe that this slightly increased risk of deadlock is worth taking in order to further scalability.

- *Livelock* (i.e. churning without making progress) is impossible in our optimized algorithm for similar reasons. In a situation where message matching fails due to lack of memory, and the receiver subsequently posts a new Receive request, the algorithm ensures that the pending sends from each sender will attempt to match the just posted receive. This is enough to ensure progress, because in a correct program the matching receive request *will* be posted.

4.4 Evaluation

The evaluation of the memory management protocol, will be done in two steps. First, we want to evaluate, how the mechanism works in an extreme situation, meaning an application that would not finish properly (crash, error) without the memory management mechanism. In order to do this testing we implement a microbenchmark.

Afterwards the mechanism will be tested under normal conditions. The NAS benchmarks will be used on this step because they do not show memory problems and we could measure the overhead added in the situation where the memory management protocol is not needed.

The NAS benchmarks are not usable to test the extreme case, because they are well balanced, meaning that the communication pattern is symmetric, so if a process is not able to receive it will not be able to send neither.

4.4.1 The killer microbenchmark

We will use a microbenchmark that recreates memory shortage conditions to prove the reliability of our mechanism and its behavior under this conditions.

```
if rank == 0
  for i = 1,N
    for j =1,number of processes
      MPI_Recv(from=j , tag=N-i )
else
  for i = 1,N
    MPI_iSend(to=0, tag=i )
  for i = 1,N
    MPI_Wait()
```

Figure 4.10 Killer benchmark structure

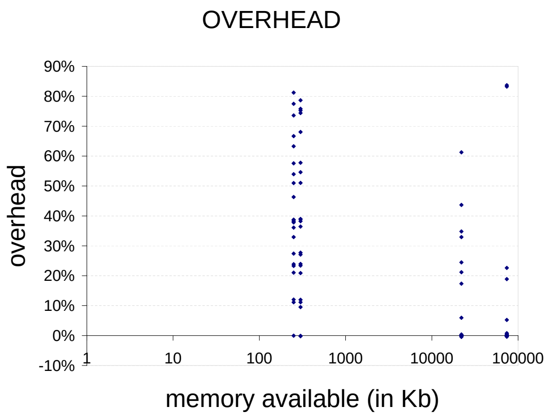


Figure 4.11 Overhead when memory management protocol is activated

The application used, figure 4.10, runs with *n* processors and all processors send a number of messages *m* (line 7), to the process number 0, this process is receiving the messages in the reverse order they have been sent (notice the *tags* in lines 4 and 7). As a result, most of the messages either have to be received as unexpected messages or they have to wait for the matching MPI_recv to be posted.

Memory shortage condition is recreated by reducing the memory available. Figure 4.11, shows the overhead for different amounts of available memory (250Kb, which is the minimum amount of memory the application needs to run, 300Kb, 22Mb, 74Mb), the situation without memory problems, with all memory available, was taken as a baseline. The application execution time was measured for different message sizes (1024, 10240, 102400), different number of messages (3 and 5) and different number of processors (16, 32, 64 up to 128).

We obtained a mesh of points and we can see that different overheads are not really related to the amount of memory available. Actually, once the memory management algorithm gets activated, factors such as restarts per message, message size, unexpected messages, etc affect the performance significantly and even in cases with very little memory available, a wide range of different overhead values is observed.

Nevertheless, the main conclusion is that the overhead is never more than twice the execution time without memory problems, which is not a hight price to pay to make your application run without problems.

We would like to point out that our killer microbenchmark does not work neither in the standard MPI implementation running on BG/L nor in MPICH-MX. The implementation on BG/L and its problems have been deeply discused. About MPICH-MX, sender process is stoped when a memory problem occurs, hopping that receiver will free some memory but it may happen that, like in our benchmark, memory is not freed until it resolves the first receive and some memory needs to be allocated in the receiver process for that. A deadlock situation is reached, therefore the application hangs.

Moreover, this result is specially significant because the application is message bound and no computation is done. If it had more computation, the effect of the message overhead would be even smaller.

4.4.2 NAS benchmarks

In order to evaluate the overhead in applications without memory problems, the NAS-MPI benchmark suite was used. We use class B problem size of the NAS Bt, Cg, Ft, Ep, Mg, Lu, and Sp benchmarks [62].

Figure 4.12 shows the execution of each benchmark, with 16, 32, 64 and 128

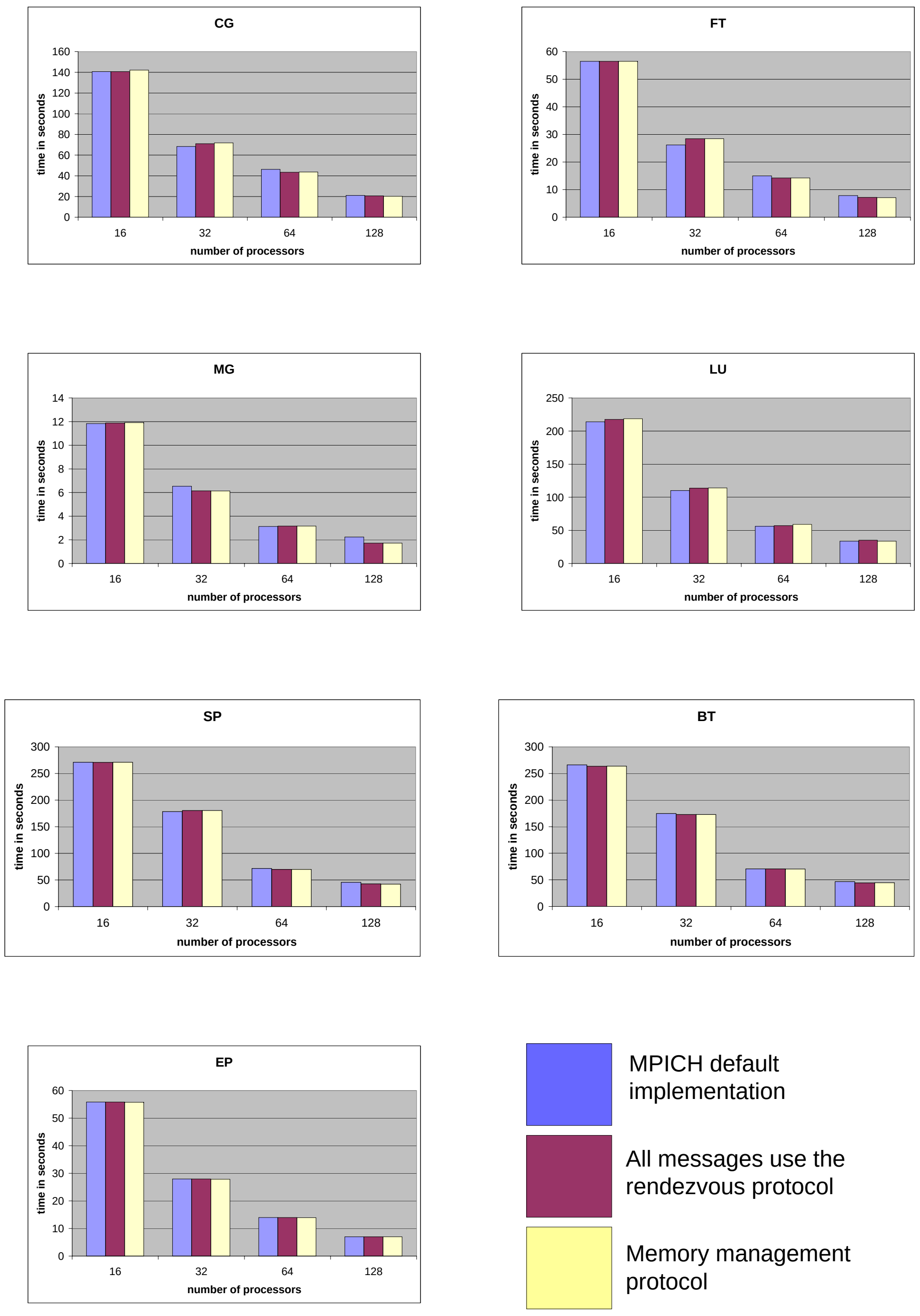


Figure 4.12 Time in seconds of the class B of NAS benchmarks Cg Ft Mg Lu Sp Bt and Ep, comparing the standard solution, the standard sending all messages with the rendezvous protocol and the solution with the memory management protocol

processors. First column is the default MPICH solution, the messages are sent using different protocols (short, eager or rendezvous) depending on the message size. The second column shows the execution of each benchmark sending all messages using the rendezvous protocol regardless of the message size. And the third execution is our memory management protocol, where all messages are also sent using the rendezvous protocol but with the memory management mechanism.

As it is shown in figure 4.12, most of the times there is no overhead, this is the case for Bt, Ep and Sp benchmarks, where the proposed solution is as performant as the default solution, we also notice that for these benchmarks the execution sending all messages using the rendezvous protocol has good performance as well. Regarding the rest of the benchmarks, Cg, Ft, Lu and Mg, figure 4.12, we notice an overhead of 1% in the worst case, we realize that most of the overhead is due to the use of the rendezvous protocol. And another good news is that this slight overhead is decreasing with the number of processes, we observe it in Cg, Ft, and Mg benchmarks.

4.5 Conclusions

We have shown the necessity of controlling the correct arrival of any message, containing either data or control-data, for current trend MPP machines like BG/L.

We have proposed a memory management protocol for BG/L overwhelming the limitation of short-memory MPPs in order to gain scalability. Our solution allows MPI to support as outstanding unexpected messages as memory is available not only in the receiver node but also in all senders. And we have shown that this mechanism works, without any overhead under normal conditions (no memory problems) and within less than twice the execution time, for an application that would crash without the proposed memory management protocol.

4.5.1 Our Memory management protocol in UPC

Our memory management protocol was implemented in the Blue Gene message layer with some changes in MPI. The protocol is generic for any messaging system, including messaging systems supporting PGAS languages, and so it does apply also for UPC.

In messaging systems supporting PGAS languages, data transfer always occur within a shared memory area in the remote process, therefore the problem of *unexpected* arrival of data messages does not exist and a couple of temporary buffers for every message size we want to receive eagerly would be enough to avoid the handshake because data will be immediately copied into the final shared memory area.

However, failure in control-data reception is less likely to happen because control information does not have to be kept until a matching is performed (application dependent) but until the message transfer has finalized (network dependent). Nevertheless, the problem can still appear if a process receives requests from every other process in the system in a short period of time, this may flood up receivers memory and the messaging system would crash. During the development of this thesis the UPC messaging system has been implemented over different networks/message drivers: over LAPI, over GM driver and over BG/L 3D torus message layer, and the problem of control-data reception potentially appears on every platform.

In PGAS languages, for every message transfer, some control information needs to be kept in the remote process until the transfer is finalized. During this time several requests may be received and if the receiver is short in memory the control information could not be held and a naive implementation would not know how to react and crash. To solve the problem our proposed control-flow mechanism could be applied. As in MPI, the order of message transfers between every two processes needs to be maintained in UPC, if the target process is short in memory a `NCTS` message be sent to the sender that will start keeping requests in a `pending send queue`.

For UPC the proposed protocol could be optimized and simplified. As opposed to MPI new message arrival will not free the memory, therefore when a process has received a `NCTS` message it can stop sending messages until a `resendAll` message is received. The protocol is simplified since an age mismatch will never occur and keeping the age is not needed anymore.

Chapter 5

Multidimensional blocking: a language extension to reduce communication

5.1 Introduction

With the advent of Petascale computing, programming for large scale machines is becoming evermore challenging. Building solutions for real-life applications, understanding the problem and designing an algorithm that scales to a large number of processors is a challenge in itself. Thus, adequate programming tools are essential to increase the programming productivity for scientific applications. Several initiatives, such as the DARPA High Productivity Computer Systems (HPCS) program, are encouraging industry and academia to take a fresh look at the issue of programming large scale machines.

Partitioned Global Address Space (PGAS) languages, such as UPC [77], Co-Array Fortran [146], and Titanium [200], extend existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distributions. They are based on languages that have a large user base and therefore there is a small learning curve to move codes to these new languages.

A long standing issue in high-performance computing is the productivity of efficient software development for high-end parallel machines. The expected increased dissemination of machines built on a hybrid memory-access model compounds this problem. A hybrid memory-access model that consists of a collection of multi-processor shared address processing nodes connected through a message-passing fast network is likely to be dominant for high-performance computing in the near future. A programming language that is designed under a PGAS programming model, such as Unified Parallel C (UPC), facilitates the encoding of data partitioning information in the program. Closing the gap between the programming and the machine models should increase software productivity and result in the generation of more efficient code.

UPC [40, 77] is a shared memory programming extension of C that provides a few simple primitives to allow for parallelism. The first UPC language specifications appeared on 2001, UPC community emphasized performance and scalability when the language was designed, and there are promising preliminary results [75]. However UPC scalability and efficiency still needs to be proven.

In our *second contribution* a preliminary study evaluates the use of UPC, a shared memory programming language on large scale machines. We prove that UPC language applications are able to scale up to hundreds of thousands of nodes with performance comparable to MPI implementations. Section 5.2 covers this study.

However, several issues with the current language definition were identified, such as: rudimentary support for data distributions (shared arrays can be distributed only block cyclic), flat threading model (no ability to support subsets of threads), and shortcomings in the collective definition (no collectives on subsets of threads, no shared data allowed as target for collective operations, no concurrent participation of a thread in multiple collectives).

Tackling some of these issues, we propose a new data distribution directive, called **multidimensional blocking**, that allows the programmer to specify n-dimensional tiles for shared data resulting in a better and easier exploitation of data locality and therefore a reduce in the amount of communication.

Finally, all the work presented in this chapter was done in collaboration with the UPC team at IBM, which is the compiler team at Toronto IBM Laboratory and the runtime team located at IBM Watson T.J. Laboratory. The final result is product of a collaborative work that comprises: (i)the proposed language extension and the necessary support in the UPC RTS. Which is part of this thesis and it is covered in this chapter. And (ii) the compiler support to recognize the language syntax (Front End FE) and a compiler optimization that was crucial to obtain good performance results. These are not contributions of this thesis and are briefly outlined here, more information can be found in [17].

5.2 Shared Memory Programming for Large Scale Machines

This section explores the use of Unified Parallel C (UPC) [39, 77] on the BlueGene/L machine.

The UPC memory model is that of a Partitioned Global Address Space (PGAS) with each thread having access to a private, a shared-local, and a shared-remote section of memory. It can be mapped to either distributed memory machines, shared memory machines or hybrid (clusters of shared memory machines).

This section describes the design and implementation of a Run-Time System (RTS), and a compiler for UPC on the BlueGene/L machine.

BlueGene/L [86] overall architecture is covered in appendix A. It is important to underline here that BlueGene/L featured as many as 65,536 dual-processor compute nodes (at the time this research was done, it was later upgraded), each operating at a very low power, and hence at a relatively low frequency of 700 MHz. The strong point of BlueGene/L is its network - a $64 \times 32 \times 32$ 3D torus that spans all compute nodes. The default software installation includes a port of the MPI library, which augments the standard Fortran, C and C++ compiler. It has been shown that careful programming and judicious use of the MPI primitives (and in some cases re-engineering of applications) allow scaling to the full extent of the machine.

UPC implementation prior to ours were designed for SMPs of clusters composed of several hundred processors. Such a design point was eminently justifiable until recently given the lack of availability of parallel machines at the scale of BlueGene/L. Given that our implementation targets much larger systems, scalability is a prime concern in our design, which required us to deviate from certain design principles used in prior implementations. For example, we use a Shared Variable Directory (SVD) to keep track of the shared data, and thus avoid the possible memory fragmentation which would occur if shared data had to be mapped to the same address location in each node.

All improvements in the runtime are contingent of the ability of the compiler to exploit them efficiently, thus a collaborative work has been performed. The main goals in this section is to demonstrate that productivity through the use of PGAS languages for large scale machines is possible. The main contributions presented in this section are:

- the UPC compiler and the UPC Run-Time System (RTS) that allow scaling of UPC programs to more than a hundred thousand processors;
- the design and implementation of a distributed shared variable directory that solves the problem of addressing shared data in very large scale PGAS systems;
- to show that, with the right kind of support from the compiler and run-time environment, scaling to hundreds of thousands of threads in a PGAS programming model is possible; we demonstrate productivity and scalability using very simple naive implementations of two of the HPC Challenge benchmarks, we won the HPC Challenge Productivity Award [108] receiving the community endorsement for this work.

5.2.1 UPC programming model considerations

UPC extends the ANSI C language with constructs for manipulating shared memory variables, thread synchronization operations (locks and barriers) and memory consistency models (strict and relaxed). UPC can be implemented on different architectures: shared memory, distributed memory or a combination of the two. Existing implementations do not scale above 64-256 threads. We propose a solution that will scale to tens of thousands of nodes, such as the Blue Gene/L machine (65536 nodes).

The UPC programming model is Single Program Multiple Data (SPMD). In the UPC execution model all the threads are started before the user code begins. Threads are synchronized using barriers and locks. The UPC memory model is shown in Figure 5.1. The PGAS memory model gives each thread access to a private section, a shared-local section, and a shared-remote section of memory. The shared data objects have affinity to

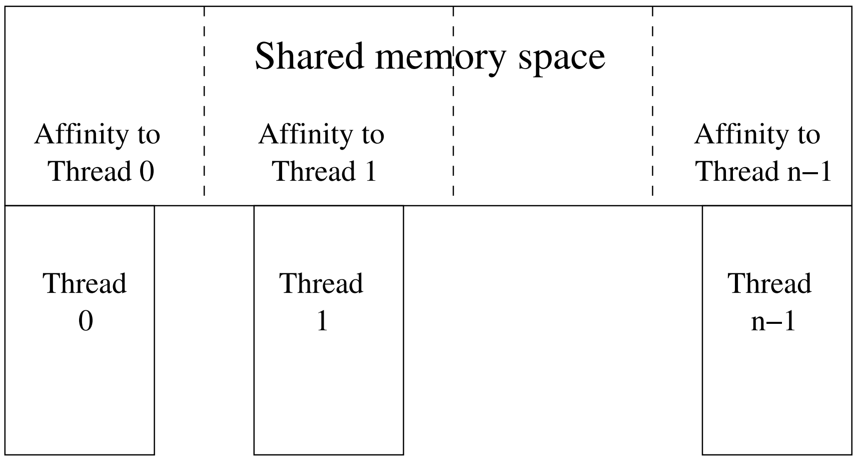


Figure 5.1 UPC Memory View

threads. Typically the latency to access the shared-local section is lower than the latency to access the shared-remote section because communication is required in distributed memory machines.

The parallel programming model exposes two consistency models: a strict model and a relaxed model. Strict consistency can be used to guarantee memory references ordering at thread level. Relaxed consistency can be used for performance. The consistency model can be specified globally or on a per access basis.

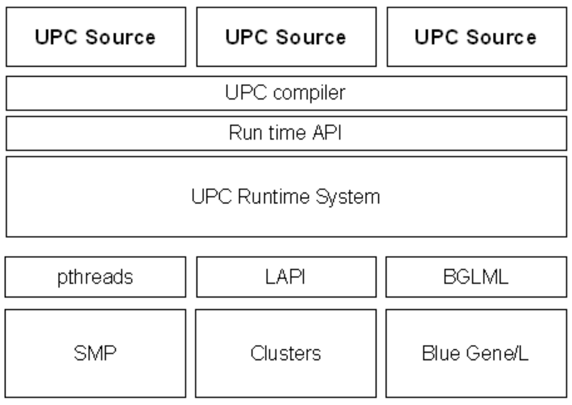


Figure 5.2 XL UPC Compiler and Runtime System

Memory layout of shared arrays in UPC

Shared arrays are distributed in block-cyclic fashion. The language allows to specify a blocking factor b, and each thread " owns " blocks of b adjacent elements.

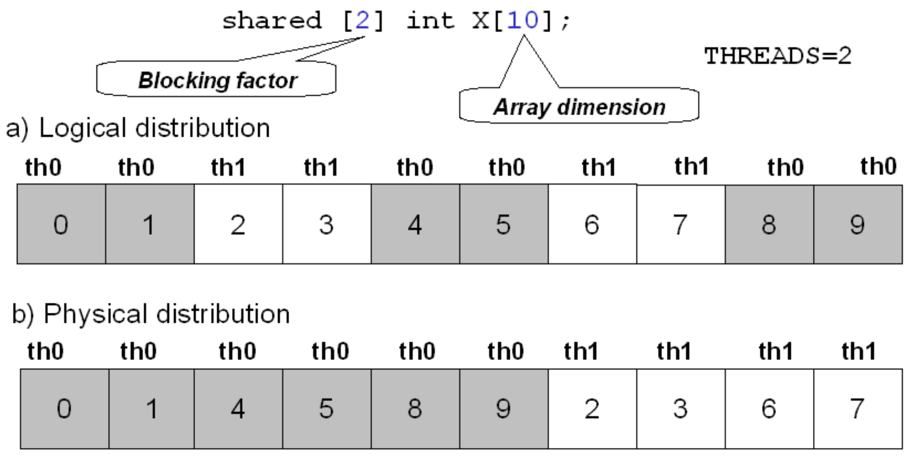


Figure 5.3 Example of UPC Array distribution

Figure 5.3 shows an example of logical and physical layout distribution for 2 THREADS, corresponding to the array declaration:

```
shared <type> [b] A[M][N];
```

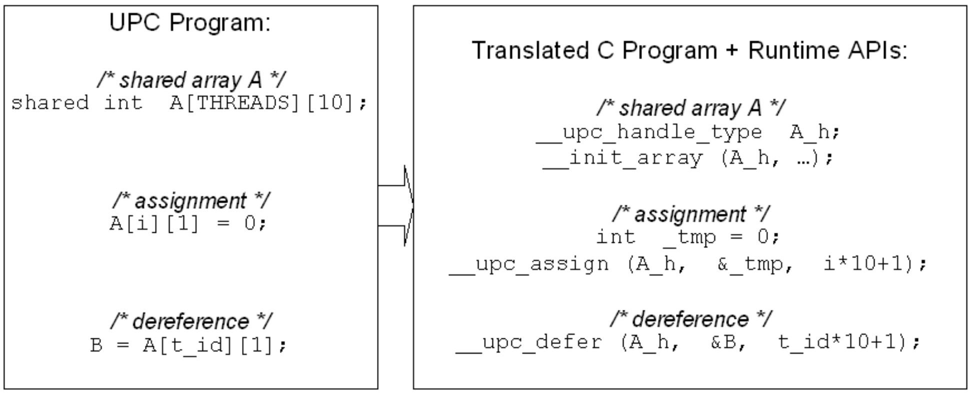
Next section presents the IBM design for the UPC framework, it is a layered approach, with a compiler that parses UPC code and transforms the UPC constructs into calls to a scalable run-time system (UPC RTS)(Figure 5.2).

5.2.2 The Runtime System

The UPC RTS exposes a few simple abstractions to the compiler. This makes the translation of UPC programs relatively easy. Figure 5.4 shows a simple translation of a UPC call to appropriate runtime calls.

The runtime system consists of a set of data structures and functions that operate on these data structures. The runtime system implements the UPC semantics on top of the hardware and OS primitives. The platform-independent interface exposed by the runtime system can be implemented on a variety of platforms and is applicable to both distributed-memory and shared-memory machines. A similar approach was followed by GASNet [27].

At the stage of this work, we have implemented the UPC RTS interface on three different platforms: (1) shared-memory multiprocessors (SMP), uses the Pthreads library [33]; (2) clusters of workstations connected either through Ethernet or through specialized networks for which a Low-level Application Programming Interface (LAPI) [170] implementation was available; and (3) BlueGene/L, using the BlueGene/L message



UPC Program:

```
/* shared array A */
shared int  A[THREADS][10];

/* assignment */
A[i][1] = 0;

/* dereference */
B = A[t_id][1];
```

Translated C Program + Runtime APIs:

```
/* shared array A */
__upc_handle_type  A_h;
__init_array (A_h, …);

/* assignment */
int  _tmp = 0;
__upc_assign (A_h,  &_tmp,  i*10+1);

/* dereference */
__upc_defer (A_h,  &B,  t_id*10+1);
```

Figure 5.4 Example of UPC source code and its correspondent C code with the added runtime calls.

layer [7]. In this study we will discuss results only on the BlueGene/L machine. Most of the optimizations presented here are applicable to the other implementations.

At the stage this work has been performed the UPC RTS provides two modes of operation: *shared* mode of operation on SMP nodes, the Pthreads library is used to spawn multiple UPC threads. And *distributed* mode, implemented over LAPI [170], and the BlueGene/L messaging framework [7]. From now on we focus on the distributed mode of operation.

The UPC RTS has multiple roles: it spawns and collects UPC threads, implements accesses to shared data, performs pointer arithmetic on pointers to shared objects and implements all the UPC intrinsic function calls (such as `upc_phaseof`, `upc_barrier` and `upc_memget`). It defines an external API that is used by the UPC compiler when generating code.

Shared objects are an important abstraction in UPC programs. The UPC RTS recognizes several kinds of shared objects: *shared scalars* (including structures/unions/enumerations), *shared arrays*, *shared pointers* (with either shared or private targets) and *shared locks*. A transparent handle is used to refer to a shared object. The single entry point to access shared data is the opaque handle type *SVD handle*. That is, corresponding to every shared object in the program, *each* UPC thread at runtime contains a variable of type *SVD handle* with an appropriately mangled name. These handles are kept internally, by the RTS, in a Shared Variable Directory (SVD), described later in this section. The UPC RTS provides routines to initialize and manipulate these handles. It is the responsibility of the compiler to manage the SVD entries when variables are created or go out of scope.

Additionally, the RTS exposes one other mechanism typically used for accessing shared data, what we call *fat pointers*, the `upc_shared_addr_t` . A fat pointer is a structure representation of a shared address that allows the program to reference shared object anywhere in the partitioned global address space.

For every shared access the runtime is responsible for translating this fat pointers to the data location. It has to know the *owner* thread and the node where this thread is being executed, and convert it to a traditional C pointer with the right address in the *owner*'s address space. Communication may be required to disambiguate the fat pointer.

Note that the only information that the RTS maintains about the base C type of a shared object is the size in bytes of that type. This information is necessary for correct handling of arithmetic and dereferencing of shared pointers.

And finally, all UPC shared objects have an *affinity* property. A shared object is affine to a particular UPC thread if it is local to that thread's memory. The language semantics strictly defines the affinity of shared objects. All non-array shared-qualified objects have affinity with thread zero. While shared arrays are distributed in a block-cyclic fashion among the threads, so different pieces of the array have affinity to different threads (section 5.2.1 shows the memory layout of shared arrays in UPC).

Note that most of the affinity values that are non-zero can not be determined until the program has begun execution, because the numerical value of `THREADS` may not be known until program execution time. The RTS infers the affinity value from the routine used to initialize or allocate the shared variable, and the compiler is not required to provide this value as an argument.

Shared Variable Directory (SVD)

The UPC RTS was designed with scalability in mind. The Shared Variable Directory (SVD) is a partitioned data structure that the RTS uses to manage allocation, deallocation, and access to shared variables. It is designed to scale to a large number of threads while allowing efficient manipulation of shared data.

As opposed to other UPC implementations, we do not require that local sections of arrays be mapped to the same memory location in all the threads. Rather, like Titanium [199], we allow the local sections of a shared array to be of arbitrary length and rely on the RTS to do the bookkeeping. The SVD has the following design principles:

1. threads must be able to create shared variables independent of each other and keep the SVD consistent with a minimum of communication;

2. for collective operations, such as `upc_all_alloc`, when all the threads execute the same operation, no locking should be required;

3. no structure should keep pointers or references based on the number of processors; rather, if remote information about a variable is required, the requester should get the information through message exchange.

As far as we know, no other PGAS language implementation is able to scale to more than one or two thousand processors, in part because of limitations on their fat-pointer implementations.

The SVD implementation is presented in Figure 5.5. In this figure we assume a Partition Global Address Space in a distributed memory machine. Each thread owns a section of the memory (the shared local portion) and also has a private section of the memory.

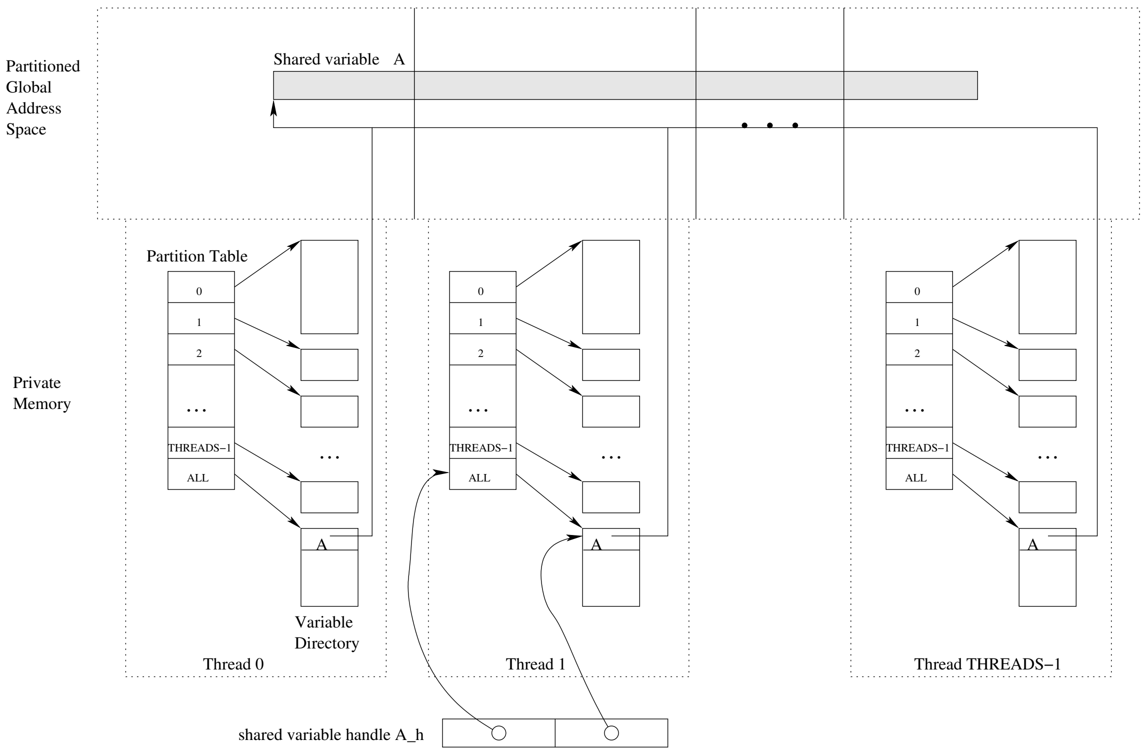


Figure 5.5 Shared Variable Directory in a PGAS distributed memory machine.

Logically, the SVD consists of a two-level data structure: at the first level there is an array with `THREADS`+1 entries, where `THREADS` is a constant defined in the UPC language – the number of threads in a UPC program. Each entry points to a partition that stores handles to shared variables that have affinity to the thread with `MYTHREAD` equal to the partition number. The partition with number `THREADS`, we call it the `ALL` partition, that is used for statically declared array objects. The reason for this separation is that the `ALL` partition has a fixed size, while the other partitions are resized as threads allocate shared data dynamically.

Shared objects are placed into its partition according to the following rules:

1. Statically declared non-array shared variables, and statically declared arrays with indefinite block size go to partition 0;
2. All other statically declared arrays go to `THREADS`, as do arrays that are dynamically allocated using the `upc_all_alloc` routine;
3. Arrays that are dynamically allocated using the `upc_global_alloc` or
4. `upc_local_alloc` routines are assigned to the partition corresponding to the thread that called the routine, i.e., to `MYTHREAD`;

Each different thread uses a mutually exclusive partition of SVD. Each partition is an independent, resizable array of pointers to control structures, such that, if one thread declares a large number of shared variables, only its partition will grow.

For distributed-memory machines (Figure 5.5), the SVD is kept in the private memory of each thread and it is replicated across all threads. Because most of the operations on the SVD are global operations (all threads participate), each thread's copy of the SVD can be updated without communication, in a consistent manner (atomically). Communication is required in the case of non-global operations such as `upc_global_alloc` or `upc_local_alloc`. Even in this case, the communication is non-blocking, as our design of the SVD guarantees that only one thread has "write" access rights to its SVD partition.

Access to shared data

Shared variable are accessed using handles, as shown by A_h in Figure 5.5. The handles address a variable by its *partition number* and the *index* in the partition. Depending on the type of the shared variable, we obtain the address of the variable or we need through another indirection level. When a thread in a distributed memory system request data from a remote thread it passes only the shared variable handle, and the remote thread will determine the local address. Optimizations, such as caching the values of shared variables and the addresses of shared objects, are discussed in the next chapter.

Figure 5.6 shows and example of how shared data is accessed in a shared array. To access a shared array given a handle (*SVD handle* in the figure), the thread requesting access dereferences the SVD partition for which the variable has affinity. The SVD entry of a shared array points to the shared variable control block (`upc_shr_var_ctrl_block` in the figure) that in turn points to a structure dependent on the variable type (`upc_shared_array_t` for arrays). On a distributed memory machine, each thread has a copy of the both structures that point to the locally allocated part of the shared array, and contain enough information, such that each thread can locally compute the affinity of every element in the array.

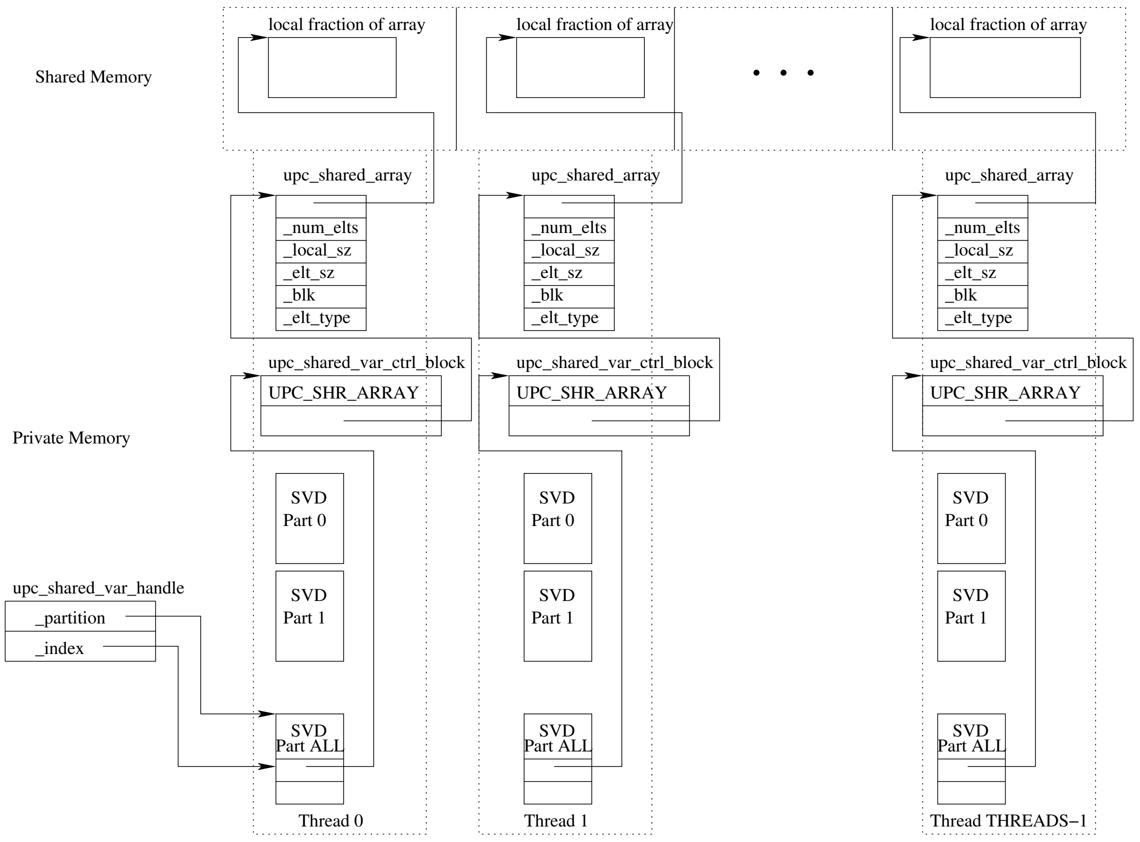


Figure 5.6 Shared array access through the SVD in Distributed Memory

In general, for an array index expression $\mathbf{i} = f(i_1, i_2, \ldots, i_n)$, the affinity or owner thread of element $A[\mathbf{i}]$ can be defined as:

$$thread(A, \mathbf{i}) ::= \left\lfloor \frac{(f(i_1, i_2, \ldots, i_n)}{b} \right\rfloor \bmod \mathcal{T} \qquad (5.1)$$

Note that b represents the block size of the shared array or the shared pointer, and it is known statically since it has been specified in the array declaration. And $\mathcal{T}$ is the total number of threads.

At this point, the thread accessing the data (hereafter `requestor thread`) sends a message to the `owner thread` ($thread(A, \mathbf{i})$) containing the SVD, the array index i, and the `length` of the requested shared data. The *shared array* structure within the variable SVD entry, points to the local portion of the array base address $A_{address}$. And the appropriate *offset* from the base address can be locally computed at any thread (more details about the offset computation well be presented in section 5.3.1). The `owner thread` computes the final memory location as $A_{address} + offset$ and from this address `length` bytes are returned to the `requestor thread`.

Allocation and Deallocation of Shared Variables

The major limiting factor of scalability for current implementations of the UPC runtime is the fact that threads are mapped to processes, and each thread has to map the entire memory space, at the same virtual address, such that static data are implicitly shared by virtue of being located at the same address on all the threads. We overcome this limitation by using the SVD.

Beside the statically declared shared arrays, UPC provides routines for dynamically allocate data (`upc_global_alloc`, `upc_all_alloc`, and `upc_local_alloc`). Some of these routines require synchronization/communication between threads. These interactions are non-trivial in distributed memory systems where messages are not guaranteed to arrive in order. Our implementation resolves this issue by partitioning the SVD. Essentially, there is no requirement on the message ordering because operations are "atomic". Each thread is responsible for managing its own partition, and a remote thread will not see the shared variables owned by another thread until its copy of the SVD is updated by the owning thread. All the variables stored in the all partition are allocated through collective operations, therefore guaranteed to be consistent.

Messaging System

As mentioned before, the UPC RTS is written to leverage multiple types of hardware, e.g. shared-memory machines as well as the LAPI library; however, all measurements presented in this section were made on BlueGene/L, using the BlueGene/L communication library.

The BlueGene/L communication library is designed to support both MPI as well as other, more light-weight, communication paradigms, such as UPC, Global Arrays [143] and Hierarchically Tiled Arrays [8]. Our design target was low overhead and high scalability. This is how we achieved these goals:

- **Messaging library choice:** The relatively low ratio of CPU speed versus network speed makes it imperative to send and process messages in as few CPU cycles as possible. Low overhead communication is very important for UPC performance, because non-optimized UPC code typically performs individual remote variable dereferences. These references result in very short network communications that are latency bound.

Implementing the UPC runtime on top of the standard BlueGene/L MPI library would have caused unacceptably high latency because of the software overhead of starting, monitoring, and receiving an MPI message. Many of our communication library optimizations address this problem.

- **Packetization and network order:** The BlueGene/L network is packet based, with packets from 32 to 256 bytes long. Due to the way packets are routed in the BlueGene/L network, ordinary data packets can arrive out of order. Packets can be forced to arrive in order, but doing so with a large number of packets tends to create hot-spots in the network, decreasing overall throughput.

  In practical terms this means that any data transfers in UPC that require more than one packet of data have to be accomplished by handshake, resulting in long latencies. We use ordered packets for very short data communications, e.g., single value get/put operations to avoiding the need to hand-shake with the receiver for the transfer of a 4-byte value.

- **Alignment issues:** The CPU-network interface is accessible only in 16 byte chunks, with each access being 16-byte aligned. Thus, when UPC buffers are arbitrarily aligned the messaging library is forced to copy data to and from aligned buffers during send/receive. This copying results in CPU overhead –memory copies are inordinately expensive on BlueGene/L– that translates into higher latencies for short transfers and into potentially decreased bandwidth when a node transmits and receives simultaneously. We mitigate the bandwidth problem by employing techniques already described in the context of the MPI library implementation [7]. For short data transfers we employed pre-allocated message objects that have alignment guarantees.

- **Overlapping computation and communication:** The BlueGene/L network hardware does not directly support one-sided communication. All network packets are insert into/extracted from the network by the processor(s). The communication library is designed to be operated by polling, restricting the overlap of communication and computation. Moreover, the same set of double-wide floating point registers are used by the machine to perform computation and to talk to the network device, further restricting overlap.

Each BlueGene/L node features two processors. The original design point of BlueGene/L called for one of the processors to act as a communication processor, while the other performs computation. However, co-operation between the processors is limited by a lack of coherent view of the memory, making low-latency communication using a dedicated communication processor impractical.

Another way to achieve the effect of overlapping computation and communication would have been to interrupt the computation processor when a network packet arrives. However, switching to a network handler for every packet involves at least a context switch, with the added burden (compared to other machines) of saving and restoring a relatively larger number of registers, again causing performance loss.

Ultimately, while running programs on BlueGene/L we noticed that most applications that were written to scale to a high number of processors tend to perform synchronized (and most often, collective) communication anyway. In hindsight, the problem of overlapping computation and communication seems not to be as important as it seemed.

- **Memory and scaling issues:** Because the network hardware does not support one-sided communication, the remote get operation has to be implemented by sending a request to the processor that owns the data. This processor then has to send a reply to the processor that originally requested it.

  Therefore a remote get operation involves the allocation of resources at the passive target. This allocation causes two problems. First, memory allocation on the passive target constitutes overhead. We mitigate this overhead by allocating and maintaining a pool of pre-allocated requests. The second problem occurs when a processor is the target of too many remote get requests. This problem has already been covered in the previous chapter of this thesis in the context of the MPI library implementation (chapter 4) and the same algorithm should be used. However, applications written for scalability typically do not exhibit such patterns, and thus the implementation of the control flow algorithm in the RTS is left as a future work. In this preliminary implementation we followed the decision of shifting the burden of managing high volume of communication to the programmer.

5.2.3 Optimizations

In this section we present two compiler plus runtime optimizations that are essential to gain performance in UPC codes. These optimizations are: transforming shared variable accesses that have affinity to the accessing thread into local accesses at compilation time, and identifying and exploiting the update primitives.

Local Memory Optimizations

Accesses of shared arrays are optimized converting *fat* pointers into *thin* pointers when the location of the reference allows. A fat pointer is an aggregated data structure, used by the RTS, that identifies a shared variable, while its thin counterpart is a standard C pointer. As we have seen in the previous section, dereferencing a fat pointer requires several levels of indirection in the SVD and the shared variable control block. Thus thin pointer dereferences are much less costly. In a distributed memory architecture, pointers that are known to be non-local must remain a fat pointer, it is necessary to use functions defined in the RTS to perform the memory access because communication is required. But pointers that are identified as local pointers can be converted.

The detection of remote accesses relies on the affinity clause of the `upc_forall` loop. In general, for an affine array index expression $\mathbf{i} = f(i_1, i_2, \ldots, i_n)$, and a `upc_forall` affinity expression g, the necessary condition to ensure the array element $A[\mathbf{i}]$ is local is that the `owner thread` 5.1 equals g:

$$thread(A, \mathbf{i}) = g.$$

For any array reference that satisfies this condition, which in many cases can be statically determined, the compiler will transform the shared array access using fat pointers into traditional C pointers. The same equations used in the RTS apply to compute the offset 5.2.2, and the base address of the array is hoisted out of the innermost loop. Array references for which the affinity can not be determined statically will remain fat pointer accesses. A detailed explanation of the algorithm can be found in [17].

The UPC language supports cast operations that an experienced programmer could use to convert fat pointers pointing to local data to thin C pointers and our RTS would report similar performance 5.2.4. However, array layouts often depend on compiler time defined variables and we believe converting the pointers at compilation time is more convenient allowing simpler UPC codes.

Update Optimizations

The RandomAccess benchmark (see section 5.2.4) is part of an important set of applications that use read-modify-update operations. The BlueGene/L messaging library supports an active message paradigm, which enables the following optimization: when the data is not used by the local thread, the update can be performed by the thread that owns the data, remotely.

In the RTS, optimized remote updating operations have been implemented. The compiler identifies updates of memory locations that use a binary logical operator (logical *and*, *or* and *xor*). The read-modify-update operation for a memory reference R_s is defined as $R_s = R_s\texttt{OP}B$, where `OP` is a binary logical operator and B can be either shared or private). The updates detected consist of instances of logical binary operators that both define and use the same shared reference.

The statement containing R_s is replaced with a call to the *update runtime function* passing in the SVD handle for R_s, the logical operation, the data type for the operation, and the value used in the logical operation (B).

If R_s has affinity with the thread P performing the operation, no communication is required, and the update is done in place, by P. Otherwise, an asynchronous message is sent from P to the owner of R_s. This `update` message is an active message which trigges an action when received by the owner of R_s.

When the `update` message is received by the owner of R_s it triggers the appropiate handler. In the handler the SVD handle is used to locate the underlying memory for R_s. The operation specified in the message is performed using the data value and the result is assigned to the shared memory location atomically (*i.e.* $R_s = R_s\texttt{OP}B$).

Since the message is asynchronous the sender will not receive a confirmation. The thread performing the remote update does not know when the update has completed. Thus, the remote update optimization is limited to relaxed shared accesses. Remote updates to strict shared accesses are performed using the traditional approach (get the current value of the shared memory location, perform the update and write the new value back to the shared memory location) because execution cannot proceed until the update has finished.

The main benefit of the update optimization is the reduction of inter-node communication from potentially three messages to a single one.

5.2.4 Evaluation

In this section we present the environment we ran our experiments in, the benchmarks we used to evaluate the UPC compiler and the actual performance results we obtained.

Hardware

The benchmark runs for this section were done on a number of BlueGene/L installations. Most of the development work was done on free-standing “node cards” (64 processors) each, and on a single rack of BlueGene/L (2048 processors). All other runs had to be scheduled in advance, either on the `BG/W` machine at IBM TJ Watson (20 racks, 40960 processors), or at the LLNL installation (64 racks, 131072 processors).

In all the runs we scheduled one UPC thread for each BlueGene/L processor. Therefore we will use threads and processors interchangeably in the following discussion.

Benchmarks

Three relevant benchmarks were used for our experiments: Random Access and EP STREAM (from the set of benchmarks used in the HPC Challenge competition [108]) and CG (from the NAS benchmarks Suite [62])

Random Access Benchmark RandomAccess is one of the four benchmarks that constitute the HPC Challenge Competition [108]. We implemented the UPC version of the benchmark from first principles, following instructions laid out on the HPC Challenge web site. We used the simplest possible algorithm, to keep source code simple; the UPC code has 111 lines.

The main loop in RandomAccess resolves to a number of read-modify-write (RMW) operations to remote locations across the machine. Each remote RMW operation translates to a network packet; hence, in the current form of the UPC RandomAccess code performance is bounded by communication latency. Good runtime and communication library performance are crucial for this benchmark, as is the compiler's ability to generate remote update calls from a read-modify-write sequence in the source code.

Verification: the RandomAccess benchmark can be easily verified by running it twice. All updates are *exclusive or* operations, and restore the original content of the array when executed for the second time. Verification is part of our benchmark implementation.

EP STREAM Triad Benchmark EP STREAM Triad is another of the HPC Challenge benchmarks. As with RandomAccess, we implemented this code from first principles, ending up with 105 lines of code.

In the EP version of the STREAM triad, all the computation is done locally. We obtained this effect in UPC by using the affinity clause of the `upc_forall` loop.

Verification: doing the verification on a single processor for an array of more than 366 billion elements is expensive and would consume all our machine allocation quota. Therefore we chose to do verification by sampling. Each thread randomly selects a set of indexes (the set size being the number of threads running the program) and verifies that the array element at that location has the correct value. Note that as opposed to the embarrassingly parallel triad operation, in which each node operates on local data exclusively, the verification step involves communication across the machine.

NAS Conjugate Gradient Benchmark For this benchmark we used the NAS CG code as implemented by El-Ghazawi and F.Cantonnet [76], with a few changes – we privatized a number of shared variables in the benchmark implementation that need not be shared, for purposes of code clarity and performance.

The resulting code looks similar to the MPI version of the benchmark. A butterfly pattern is set up by the code to aid in the execution of what are really `Allreduce` operations, but are executed by MPI point-to-point primitives. In the UPC version of the code these primitives are replaced by calls to `upc_memget`, `upc_memput` and `upc_barrier`. We ended up using barrier calls because point-to-point synchronization primitives are not yet available in the runtime and in the communication library. NAS CG has built-in verification.

Optimization Evaluation

First we use Random Access and EP STREAM Triad benchmarks to discuss the effect of the optimizations presented in Section 5.2.3. Table 5.1 shows the performance obtained by enabling each optimization in isolation. The optimizations presented are as follows: *No opt* – any optimization is enabled, *locality* – the local memory optimization discussed in section 5.2.3, *update* – the update optimization presented in Section 5.2.3, *pwr2* specifies to the compiler that the number of threads is a power of 2, *all - pwr2* are all optimizations except *pwr2*, and finally *all* – all optimizations combined. The results in the *all* column also include the `upc_forall` loop simplification [17] [59]

There are few observations that we make:

- the *locality* optimization affects mainly the STREAM benchmark, because all accesses are local, as opposed to Random Access where most accesses are remote;
- the *update* optimization improves Random Access by as much as 200%, because we are essentially replacing three messages (two for get plus one for put) with one message (the update);
- the *pwr2* optimization (which essentially enables the compiler to replace an integer division with a shift operation) has no effect on its own – there is far too much overhead in dereferencing shared structures for its effect to show up;

The most interesting observation is that while each of these optimizations show modest gains, by combining all of them together, we obtain speedups of 7 for Random

Benchmark	Measure	No opt	Optimizations					
			locality	update	pwr2	all-pwr2	all
Random Access	GUPS	0.00270	0.00272	0.00561	0.00270	0.01815	0.01918
	Time (sec)	198.492	197.033	95.729	198.661	29.580	27.987
	Speedup	1.00	1.01	2.07	.999	6.71	7.09
Stream	GB/s	0.1343	0.1769	0.1343	0.1343	0.5978	32.3609
	Time (sec)	35.730	27.129	35.730	35.730	8.029	0.148
	Speedup	1.00	1.32	1.00	1.00	4.45	240.77

Table 5.1 Optimizations effects on Random Access and Stream Benchmarks, running on 64 threads. Speedups are measured relative to the no opt case.

Access and 240 for STREAM. As we hinted before, the compiler is able to transform most of the fat pointers into standard C pointers (local references), enabling the code generation step to optimize the code as for a sequential program. This is illustrated by the effect that the *pwr2* optimization has on STREAM after all the other optimizations were performed.

Scalability Evaluation

Random Access Benchmark Table 5.2 shows the absolute and scaling performance of Random Access benchmark as measured on up to 64 racks of BlueGene/L. The RandomAccess benchmark is designed to scale weakly (the memory required by the program as well as the total number of updates is directly proportional to the number of processors). To measure scaling performance we define efficiency for N processors as $\frac{T_{single}}{T_{parallel}}$.

We arranged for 50% of the memory to be used. With perfect scaling, a RandomAccess run should take about 300 seconds regardless of the number of processors it is running on. Since performance does not scale linearly (see the efficiency column in Table 5.2), the total runtime increases on larger runs.

The benchmark is affected by two performance limiting effects. At low numbers of processors the gating factor is communication latency. For large numbers of processors the gating factor becomes the torus network's cross-section bandwidth. The cross-section bandwidth of a booted BlueGene/L partition is determined by its longest torus dimension: cubic partitions have the highest cross-section bandwidth relative to the number of nodes they contain.

The largest machine configuration we ran RandomAccess on (128K processors), has an effective cross section of:

$$32 \times 32 \times 2 \times 2 = 4096$$

network links. This results from the 32×32 geometry of the cross-section and two doubling factors: each link is bi-directional and the machine is a 3D torus, not a mesh.

Thus, cross-section bandwidth for the 128K processor machine configuration can be determined as the product of the wire speed, 175 MBytes/s, and the number of links in the cross-section, yielding $175 \times 4096 = 716,625 MBytes/s$, of aprox. 716.6 GBytes/s.

Given that Random Access update packets end up as 42 bytes each on the wire, and that only half of all Random Access updates have to travel through the cross section, the maximum theoretical GUPs number for the benchmark on this configuration can be calculated as:

$$\frac{2 \times 716.6}{42} = 34.12 GUPS$$

As table 5.2 shows, the actual measured benchmark performance is very close to this theoretical peak.

Threads	Performance	Memory TBytes		efficiency
	(GUPS)	used	total	(%)
1	5.4E-4	0.000128	0.000512	100
2	7.8E-4	0.000256	0.000512	72
4	1.3E-3	0.000512	0.001	61
64	0.02	0.008192	0.016	61
2048	0.58	0.250000	0.500	51
4096	1.15	0.500000	1.000	50
8912	2.28	1.000000	2.000	49
16384	4.49	2.000000	4.000	49
32768	8.83	4.000000	8.000	48
65536	14.80	8.000000	16.000	43
131072	28.30	8.000000	16.000	38

Table 5.2 Random Access performance results. Performance is measured in terms of Giga updates per second.

Our benchmark beats the absolute performance of RandomAccess measured on any machine other than BlueGene/L , and achieves about 50% of the best known hand-coded optimization written for the same machine [93].

EP STREAM Triad Benchmark The memory requirements of STREAM are dictated by 3 shared arrays: the HPC Challenge requirement is that the size of these arrays has to be more than a quarter of the main memory and may not fit in the cache. Thus, STREAM scales weakly. We chose to be conservative and selected the arrays to fill half the memory of the machine for every machine size that we ran STREAM on.

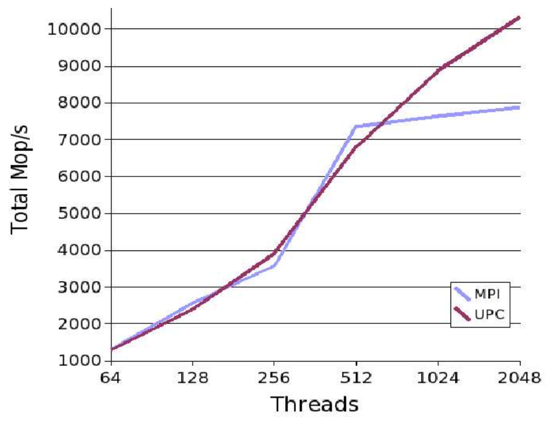


Figure 5.7 UPC vs MPI scaling on CG class C.

STREAM performance results are shown in Table 5.3, as the benchmark is embarrassingly parallel, there is no scaling drop. In the table we left out the intermediate results because they contribute no information.

Threads	Performance	Memory	efficiency
	(GB/s)	TBytes	(%)
1	0.73	0.000128	100
2	1.46	0.000256	100
4	2.92	0.000512	100
64	46.72	0.008192	100
65536	47827.00	8.000000	100
131072	95660.77	8.000000	100

Table 5.3 STREAM Triad performance results.

NAS Conjugate Gradient Benchmark Figure 5.7 compares the scaling of the UPC version of the CG benchmark with the NAS NPB MPI version, on input size Class C. Up to about 512 processors the performance of both UPC and MPI is equivalent. However, for more than 512, since the problem size remains constant (strong scaling), message sizes become too small to hide MPI overheads for two-sided communication. In the UPC implementation, due to the use of one-sided communication, the overheads are smaller and the benefits appear at 1024 processors and up. The scaling trend in the Figure suggests that CG will not scale much beyond 2048 processors.

5.2.5 Summary

At this stage we have shown that shared memory programming for large scale distributed memory machines is not a myth. Scaling non-trivial shared-memory programs to hundreds of thousands of threads is possible with the right support from the compiler and from the run-time system. We have described our XL UPC compiler infrastructure and the UPC Run-Time System; we have presented the essential optimizations that contributed to high performance. We have illustrated our work with three benchmarks, two of which we scaled to more than a hundred thousand processors on the BlueGene/L machine.

In the course of this evaluation, we encountered several challenging problems, which we will continue to address. One of these challenges was the lack of benchmarks and algorithms written in UPC that can scale to the size of a BlueGene/L computer. Existing efforts, such as the DARPA HPCS program, to provide scalable algorithms and applications for Petaflops computing are the right approach. Using PGAS languages to develop these applications will enable programmers to be more productive, while not sacrificing performance. We have shown this is possible.

Facing the challenge, we implemented several parallel algorithms — High Performance Linpack (HPL), stencil computation and linear algebra operations such as matrix-vector and Cholesky factorization — in the UPC programming language. During this effort we identified several issues with the current language definition, such as: rudimentary support for data distributions (shared arrays can be distributed only block cyclic), flat threading model (no ability to support subsets of threads), and shortcomings in the collective definition (no collectives on subsets of threads, no shared data allowed as target for collective operations, no concurrent participation of a thread in multiple collectives). This issues have motivated the work presented in the next section 5.3

5.3 Multidimensional blocking in UPC

Tackling some of above mentioned language limitations, we propose a new data distribution directive, called multidimensional blocking, that allows the programmer to specify n-dimensional tiles for shared data (section 5.3.1); In addition, while implementing a compiler and runtime system we found that naively translating all shared accesses to runtime calls is prohibitively expensive. While the language supports block transfers and cast operations that could alleviate some of the performance issues, it is more convenient to address these problems through compiler optimizations. We also present several benchmarks that demonstrate the benefits of the multidimensional blocking features; these results were obtained on a cluster of SMP machines (section 5.3.4).

We claim that this extension allows for better control of locality, which implies that communication is reduced, and therefore performance and scalability are improved in the UPC language.

5.3.1 Multidimensional Blocking of UPC arrays

In this section we propose an extension to the UPC language syntax to provide additional control over data distribution: tiled (or *multiblocked*) arrays. Tiled data structures are used to enhance locality (and therefore performance) in a wide range of HPC applications [24]. Multiblocked arrays can help UPC programmers to better express these types of applications, allowing the language to fulfill its promise of allowing both high productivity and high performance. Also, having this data structure available in UPC facilitates using library routines, such as BLAS [69], in C or Fortran that already make use of tiled data structures.

Consider a simple stencil computation on a 2 dimensional array that calculates the average of the four immediate neighbors of each element.

```
shared double A[M][N];
...
for (i=1..M-2,j=1..N-2)
  B[i][j] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
```

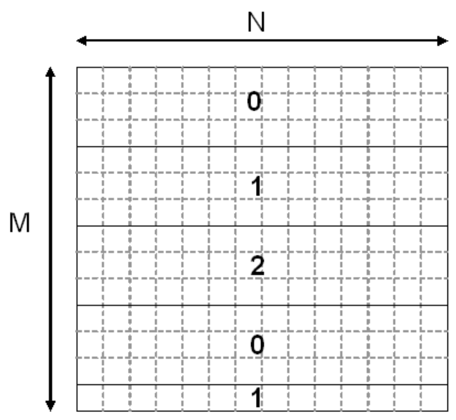
Since it has no data dependencies, this loop can be executed in parallel. However, the naive declaration of `A` above yields suboptimal execution, because e.g. `A[i-1][j]` will likely not be on the same UPC thread as `A[i][j]` and may require inter-node communication to get to. A somewhat better solution allowed by UPC is a striped 2D array distribution (Figure 5.8a).

```
shared double [M*b] A[M][N];
```

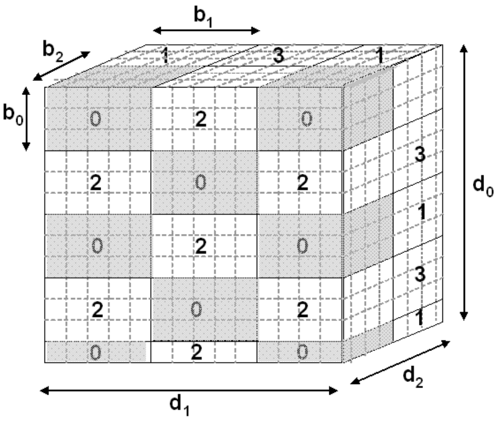
$M \times b$ is the *blocking factor* of the array; that is, the array is allocated in contiguous blocks of this size. This however, limits parallelism to $\frac{N}{b}$ processors and causes $O(\frac{1}{b})$ remote array accesses. By contrast, a tiled layout provides $\frac{M \times N}{b^2}$ parallelism and $O(\frac{1}{b^2})$ of the accesses are remote. Typical MPI implementations of stencil computation tile the array and exchange "border regions" between neighbors before each iteration. This approach is also possible in UPC:

```
struct block { double tile[b][b]; };
shared block A[M/b][N/b];
```

However, the declaration above complicates the source code because two levels of indexing are needed for each access. We cannot pretend that `A` is a simple array anymore. We propose a language extension that can declare a tiled layout for a shared array, as follows:



(a) Striped 2D distribution



(b) 3D tiled distribution

Figure 5.8 (a) Example of a striped 2D array cyclic distribution with 3 THREADS. (b)Example of a 3-dimensional tiled array distribution. d_n represent the array dimensions and b_n the blocking factor in each dimension. The example shows a block-cyclic distribution with 4 THREADS.

```
shared <type> [b0][b1]...[bn] A[d0][d1] ... [dn];
```

Array A is an n-dimensional tiled (or "multi-blocked") array with each tile being an array of dimensions $[b0][b1]...[bn]$. Tiles are understood to be contiguous in memory (Figure 5.8b).

UPC array layout

To describe the layout of multiblocked arrays in UPC, we first need to discuss conventional shared arrays. A UPC array declared as below:

```
shared [b] <type> A[d0][d1]...[dn];
```

is distributed in memory in a block-cyclic manner with blocking factor b, where b is an optional constant-expression. Given an array index $\mathbf{v} = v_0, v_1, ...v_{n-1}$, to locate element $A[\mathbf{v}]$ we first calculate the linearized row-major index (as we would in C):

$$L(\mathbf{v}) = v_0 \times \prod_{j=1}^{n-1} d_j + v_1 \times \prod_{j=2}^{n-1} d_j + ... + v_{n-1} \tag{5.2}$$

Block-cyclic layout is based on this linearized index. We calculate the UPC *thread* on which array element $A[\mathbf{v}]$ resides. Within the local storage of this thread the array is kept as a collection of blocks. The *course* of an array location is the block number in which the element resides; the *phase* is its location within the block. Figure 5.9 shows an example.

$$\begin{cases} thread(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \bmod \mathcal{T} \\ phase(A,\mathbf{v}) & ::= L(\mathbf{v}) \bmod b \\ course(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b\times\mathcal{T}} \right\rfloor \end{cases}$$

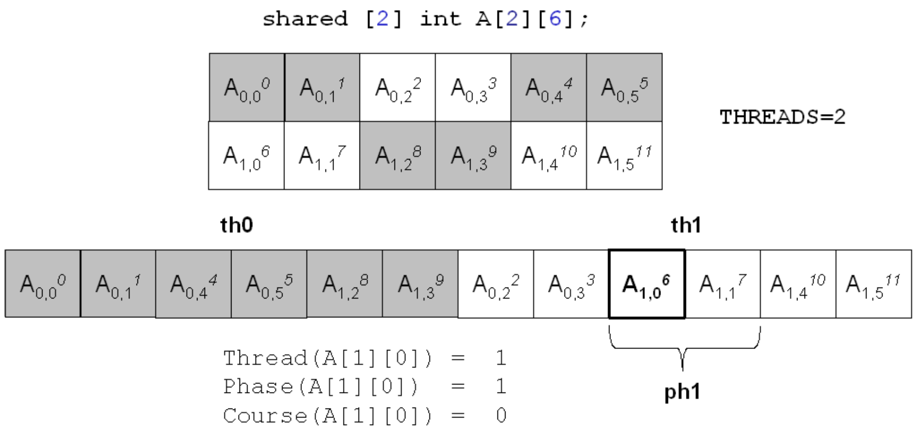


Figure 5.9 Example of UPC Block-cyclic array distribution. The subindex shows the array index $\mathbf{v} = v_0, v_1$ and the superindex shows the linearized row-major index $L(\mathbf{v})$.

Multiblocked arrays: The goal is to extend UPC syntax to declare tiled arrays while minimizing the impact on language semantics. The internal representation of multiblocked arrays should not differ too much from that of standard UPC arrays. Consider a multiblocked array A with dimensions $D = \{d_0, d_1, ...d_n\}$ and blocking factors $B = \{b_0, b_1, ...b_n\}$. This array would be allocated in $k = \prod_{i=0}^{n-1} \left\lceil \frac{d_i}{b_i} \right\rceil$ blocks (or tiles) of $b = \prod_{i=0}^{n-1} b_i$ elements. We continue to use the concepts of *thread*, *course* and *phase* to find array elements. However, for multiblocked arrays two linearized indices must be computed: one to find the block and another to find an element's location within a block. Note the similarity of Equations 5.3 and 5.4 to Equation 5.2:

$$L_{in-block}(\mathbf{v}) = \sum_{k=0}^{n-1}((v_k \bmod b_k) \times \prod_{j=k+1}^{n-1} b_j) \tag{5.3}$$

$$L(\mathbf{v}) = \sum_{k=0}^{n-1}(\left\lfloor \frac{v_k}{b_k} \right\rfloor \times \prod_{j=k+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil) \tag{5.4}$$

The *phase* of a multiblocked array element is its linearized in-block index. The *course* and *thread* are calculated with a cyclic distribution of the block index, as in the case of regular UPC arrays.

$$\begin{cases} thread(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i} \right\rfloor \bmod \mathcal{T} \\ phase(A,\mathbf{v}) & ::= L_{in-block}(\mathbf{v}) \\ course(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i \times \mathcal{T}} \right\rfloor \end{cases} \tag{5.5}$$

Array sizes that are non-multiples of blocking factors: The blocking factors of multiblocked arrays are not required to divide their respective dimensions, just as blocking factors of regular UPC arrays are not required to divide the array's dimension(s). Such arrays are padded in every dimension to allow for correct index calculation.

Multiblocked arrays and UPC pointer arithmetic

The address of any UPC array element (even remote ones) can be taken with the `upc_addressof` function or with the familiar `&` operator. The result is called a *pointer-to-shared*, and it is a reference to a memory location somewhere within the space of the running UPC application. In our implementation a pointer-to-shared identifies the base array as well as the thread, course and phase of an element in that array.

UPC pointers-to-shared behave much like pointers in C. They can be incremented, dereferenced, compared etc. The familiar pointer operators (`*`, `&`, `++`) are available. A series of increments on a pointer-to-shared will cause it to traverse a UPC shared array in row-major order.

Pointers-to-shared can also be used to point to multiblocked arrays. Users can expect pointer arithmetic and operators to work on multiblocked arrays just like on regular UPC shared arrays.

Affinity, casting and dynamic allocation of multiblocked arrays: Multiblocked arrays can support affinity tests (similar to the `upc_threadof` function) and type casts the same way regular UPC arrays do.

Dynamic allocation of UPC shared arrays can also be extended to multiblocked arrays. UPC primitives like `upc_all_alloc` always return shared variables of type `shared void *`; multiblocked arrays can be allocated with such primitives as long as they are cast to the proper type.

5.3.2 UPC RTS Implementation Issues

A shared variable has a corresponding *fat* pointer. The fat pointer is an aggregated data structure that is used by the RTS to identify the shared variable in the SVD, it contains all the necessary information to locate the element. This is: *handler*, *thread*, *course*, *phase* as computed in 5.5, the *block size* and *element size*. A naive compiler transforms accesses to shared variables to calls to the RTS that use the fat pointer to determine the location of the shared variable. If the shared variable is located in the shared memory domain of the accessing thread, the access is performed using `memcpy`. If the shared variable is located in a different shared memory domain, a message is sent from the `requester` thread to the `owner`.

Hybrid memory layout: Our UPC runtime implementation has significantly evolved since 5.2.2. At this point it offers three modes of operation: in addition to the *shared* mode of operation on SMP nodes and the *distributed* mode, implemented over LAPI [170], and the BlueGene/L messaging framework [7], the RTS is capable of running in mixed multithreaded/multinode environments, (aka. *hybrid* mode). Hybrid mode is designed for operation on clusters of SMP. UPC threads communicate through shared memory when available, and send messages through one of several available transports when necessary. The Pthreads library is used to spawn multiple UPC threads on systems with SMP nodes.

In such an environment locality is interpreted on a per-node basis, but array layouts have to be on a per-UPC-thread basis to be compatible with the specification. This is true for both regular and multiblocked arrays.

For the hybrid environment, the SVD (Figure 6.2) is replicated across nodes. Threads sharing a node share an instance of the SVD, to keep the affinity a partition is still maintained for every UPC thread.

Access to conventional shared arrays in an hybrid environment, in addition to the equations in 5.3.1, we should also consider the `node` and the `local thread` (thread number within the node).

$$\begin{cases} node(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \bmod \mathcal{N} \\ lthread(A, \mathbf{v}) & ::= \frac{L(\mathbf{v})}{b} \bmod \left(\frac{\mathcal{T}}{\mathcal{N}}\right) \end{cases} \tag{5.6}$$

Where $\mathcal{N}$ is the number of nodes, and $\frac{\mathcal{T}}{\mathcal{N}}$ the threads per node. A thread r (`requester`) accessing array element $A[\mathbf{v}]$, will send a message to the owner n ($node(A, \mathbf{v})$ 5.6), containing svd, $\mathbf{v}$, and length. Node n will compute the `offset` to locate the array element $A[\mathbf{v}]$ as in 5.7. And finally, t will send the requested data to r.

$$offset(A, \mathbf{v}) = (A_{lsize} \times lthread(A, \mathbf{v})) + e_{size} \times ((b \times course(A, \mathbf{v})) + phase(A, \mathbf{v})) \tag{5.7}$$

A_{lsize} is the size of the local portion of the array with affinity to a UPC thread, and e_{size} is the size of an array element $A[\mathbf{v}]$.

Access to Multi-Dimensional Blocked shared arrays is analogous to shared arrays.

Defining the owner `node` and `local thread` as follows, and the same equation for the offset is applied 5.7:

$$\begin{cases} node(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i} \right\rfloor \bmod \mathcal{N} \\ lthread(A,\mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i} \right\rfloor \bmod \left(\frac{\mathcal{T}}{\mathcal{N}}\right) \end{cases} \tag{5.8}$$

Processor tiling: Another limitation of the current implementation is related to the cyclic distribution of blocks over UPC threads. An alternative would be to specify a processor grid to distribute blocks over. Equation 5.4 would have to be suitably modified to take thread distribution into consideration. We have not implemented this yet in the UPC runtime system, although performance results presented later clearly show the need for it.

5.3.3 Locality Analysis

The detection of local accesses relies on the affinity clause of the `upc_forall` loop. In general, for an array element $A[\mathbf{v}]$, and a `upc_forall` affinity expression g, the necessary condition to ensure locality of $A[\mathbf{v}]$ is that the `owner node` equals g: $node(A,\mathbf{v}) = g$.

The locality can be exploited by the programmer by using the `upc_threadof` function provided by the RTS, an `if` clause would need to be added in the inner-most loop plus a cast to local pointer, we call it a *User-privatized shared access* and it generates long and uncomfortable codes.

	RTS Shared Access	Compiler-privatized Shared Access	User-privatized shared access	Private Access
Assign (Single Block)	0.40966	0.00871	0.00108	0.00109
Assign (Multi-Block)	0.74690	0.036619	0.0012	0.00117

Table 5.4 Access times in microseconds for shared array elements

If the compiler can prove that a shared variable is in the shared memory domain of the accessing thread, the compiler can bypass the function call to the RTS and directly access the location of the shared variable. Table 5.4 show the times to access local shared variables using the functions provided by the RTS and privatized accesses generated by the compiler. The time to complete a private access in UPC is also shown.

These results show the potential for performance improvements when the compiler can privatize shared accesses. Accessing shared variables using the functions in the RTS involves traversing several data structures maintained by the SVD, resulting in several levels of indirection. In addition to these levels of indirection, computing the actual location of the data in memory involves integer divisions and modulo operations. When the accesses are privatized, the traversal of the SVD data structures is eliminated. The overhead of the data structure traversal is observed in the time differences between the RTS Shared Access and the Privatized Shared Access columns in Table 5.4. However, the integer division and modulo operations are still necessary to compute the exact location. The overhead of these operations is observed in the time differences between the Privatized Shared Access and Private Access columns in the table.

Compile-time Locality analysis for multi-dimensional blocked arrays In Figure 5.10 we present a loop nest where the shared array element in the affinity test — the last parameter in the `upc_forall` statement — is formed by the current loop-nest index, while the single element referenced in the loop body has a displacement, with respect to the affinity expression, specified by the distance vector $\mathbf{k} = [k_0, k_1, \ldots, k_{n-1}]$. Any loop nest in which the index for each dimension, both in the affinity test and in the array reference, is an affine expression containing only the index in the corresponding dimension can be transformed to this canonical form.[1] And the compiler is able to perform the locality analysis. The goal of the locality analysis is to compute symbolically the node ID of each shared reference in the loop and compare it to the node ID of the affinity expression. All references having a node ID equal to the affinity expression's node ID are local.

The analysis considers loop nests that contain accesses to UPC shared arrays and finds shared array references that are provably local (on the same UPC thread) or shared local (on the same node in shared memory, but on different UPC threads). All other shared array references are potentially remote (reachable only via inter-node communication).

The analysis enables the compiler to re-factor the loop nest to separate local and remote accesses. Local and shared local accesses cause the compiler to generate simple memory references; remote variable accesses are resolved through the runtime with a significant remote access overhead. We consider locality analysis crucial to obtaining good performance with UPC.

The details of how these algorithms work are out of the scope of this thesis and can be found in [16].

[1] An example of a loop nest that cannot be transformed to this canonical form is a two-level nest accessing a two-dimensional array in which either the affinity test or the reference contains an expression such as `A[`$v_0 + v_1$`][`v_1`].`

```
shared [$b_0$][$b_1$]$\cdots$[$b_{n-1}$] int A[$d_0$][$d_1$]$\cdots$[$d_{n-1}$];
for($v_0$=0 ; $v_0 < d_0 - k_0$ ; $v_0$++)
    for($v_1$=0 ; $v_1 < d_1 - k_1$ ; $v_1$++){
        ...
        upc_forall($v_{n-1}$=0 ; $v_{n-1} < d_{n-1} - k_{n-1}$ ; $v_{n-1}$++ ; &A[$v_0$][$v_1$]...[$v_{n-1}$])
                A[$v_0 + k_0$][$v_1 + k_1$]...[$v_{n-1} + k_{n-1}$] = $v_0 * v_1 * \ldots * v_{n-1}$;
    }
```

Expression	Description
n	number of dimensions
b_i	blocking factor in dimension i
d_i	array size in dimension i
v_i	position index in dimension i
$\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$	Index of an array element
$\mathcal{T}$	number of threads
k_i	displacement in dimension i

Figure 5.10 Multi-level loop nest that accesses a multi-dimensional array in UPC. And a summary of the meaning of each expression.

5.3.4 Evaluation

In this section we evaluate the claims we have made in this chapter namely the usefulness of multiblocking for a better exploitation of data locality, to reduce communication, to allow the use of linear algebra libraries and for better code scalability. In a word it allows better performance while simplifying the code (*sintax sugar*).

We evaluate: (i) the performance gains of privatize local shared memory accesses (locality analysis). We use **Dense matrix-vector multiplication** and **5-point Stencil** benchmarks, and run them on a an IBM Squadron$^{\text{TM}}$cluster. Each node has 8 SMP Power5 processors running at 1.9 GHz and 16 GBytes of memory. (ii) productivity of the multiblocking. Three benchmarks were written to showcase multi-blocked arrays: **Cholesky** and **Matrix multiply** were evaluated on the same IBM Squadron$^{\text{TM}}$cluster and **HPL** was run on the BlueGene/L supercomputer to show scalability.

Dense matrix-vector multiplication: This benchmark multiplies a two-dimensional shared matrix with a one-dimensional shared vector and places the result in a one-dimensional shared vector. The objective of this benchmark is to measure the speed difference between compiler-privatized and unprivatized accesses.

The matrix, declared of size 14400×14400, the vector as well the result vector are all blocked using single dimensional blocking. The blocking factors are equivalent to the [*] declarations. Since the vector is shared, the entire vector is first copied into a local buffer using `upc_memget`. The matrix-vector multiplication itself is a simple 2 level nest with

Matrix-vector multiply

Naive	1 node	2 nodes	3 nodes	4 nodes
1 TPN	27.55	16.57	14.13	9.21
2 TPN	16.57	8.59	7.22	4.32
4 TPN	8.57	4.3	3.63	2.18
6 TPN	7.2	3.62	2.43	1.89
8 TPN	4.33	2.2	1.96	1.28

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 TPN	2.08	1.22	0.78	0.6
2 TPN	1.7	0.85	0.63	0.43
4 TPN	0.85	0.44	0.33	0.23
6 TPN	0.65	0.35	0.25	0.19
8 TPN	0.44	0.23	0.22	0.17

Stencil benchmark

Naive	1 node	2 nodes	3 nodes	4 nodes
1 thread	35.64	24.59	19.04	13.41
2 threads	18.85	13.56	9.82	7.9
4 threads	9.8	13.64	5.58	8.9
6 threads	10.85	8.98	7.53	6.12
8 threads	4.9	5.58	9.52	3.66

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 thread	0.30	1.10	1.41	0.74
2 threads	0.73	0.72	0.75	1.06
4 threads	0.44	1.19	0.39	0.84
6 threads	0.32	0.30	1.11	0.75
8 threads	0.22	0.63	1.07	1.02

Figure 5.11 Runtime in seconds for the matrix-vector multiplication benchmark (left) and for the stencil benchmark (right). The tables on the top show naive execution times; the tables on the bottom reflect compiler-optimized runtime.

```
void update_mb (shared double [B][B] A[N][N], int col0, int col1) {
  double a_local[B*B], b_local[B*B];
  upc_forall (int ii=col1; ii<N; ii+=B; continue)
    upc_forall (int jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {
      upc_memget (a_local, &A[ii][col0], sizeof(double)*B*B);
      upc_memget (b_local, &A[jj][col0], sizeof(double)*B*B);
      dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B, a_local,
             &B, &beta, (void *)&A[ii][jj], &B);
    }
}
```

Figure 5.12 Cholesky. Distributed symmetric rank-k update routine.

the outer loop being `upc_forall` . The address of the result vector element is used as the affinity test expression.

Results presented in Figure 5.11 (left side) confirm that compiler-privatized accesses are about an order of magnitude faster than unprivatized accesses.

5-point Stencil: This benchmark computes the average of a 4 immediate neighbors and the point itself at every point in a 2 dimensional matrix and stores the result in a different matrix of same size. The benchmark requires one original data matrix and one result matrix. 2-d blocking was used to maximize the locality. The matrix size used for the experiments was 5760×5760. Results, presented in Figure 5.11 (right side), show that in this case, too, run time is substantially reduced by privatization.

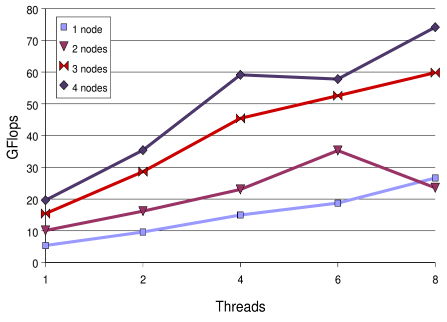
Cholesky factorization and Matrix multiply: Cholesky factorization was written to showcase multi-blocked arrays. The tiled layout allows our implementation to take direct advantage of the ESSL [78] library. The code is patterned after the LAPACK [69] `dpotrf` implementation and adds up to 53 lines of text. To illustrate the compactness of the code, we reproduce one of the two subroutines used, distributed symmetric rank-k update, below. The matrix multiply benchmark is written in a very similar fashion. It

Cholesky Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.37	10.11	15.43	19.63
2 TPN	9.62	16.19	28.64	35.41
4 TPN	14.98	23.03	45.43	59.14
6 TPN	18.73	35.29	52.57	57.8
8 TPN	26.65	23.55	59.83	74.14

Matrix Multiply Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.94	11.30	16.17	22.24
2 TPN	11.76	21.41	29.82	42.20
4 TPN	23.24	39.18	51.05	73.44
6 TPN	31.19	54.51	66.17	89.55
8 TPN	44.20	63.24	79.00	99.71



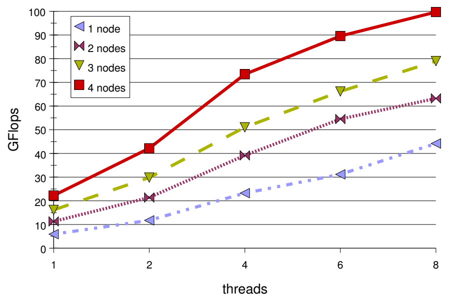


Figure 5.13 Performance of multiblocked Cholesky and matrix multiply as a function of participating nodes and threads per node (TPN). Theoretical peak: $6.9\ GFlops \times threads \times nodes$

amounts to little more than a (serial) `k` loop around the `update` function above with slightly different loop bounds and three shared array arguments `A`, `B` and `C` instead of only one. It amounts to 20 lines of code. Without question, multiblocking allows compact code representation. The benchmark numbers presented in Figures 5.13 show mediocre scaling and performance "hiccups", which we attribute to communication overhead and poor communication patterns. Clearly, multiblocking syntax needs to be extended with a distribution directive. Also, the UPC language could use better collective communication primitives; but this is part of the future work of this thesis.

High Performance Linpack in UPC

HPL (High Performance Linpack) is a well known implementation of the Linpack TPP (Toward Peak Performance) variant of original Linpack benchmark which measures the floating point rate of execution for solving a linear system of equations.

Linpack Benchmark was originally designed to assist users of the LINPACK package [71] by providing information on execution times required to solve a system of linear equations. The first "LINPACK Benchmark" report appeared as an appendix in the LINPACK User's Guide [71] in 1979. Over the years more data was added reporting execution times in different computer systems, and today LINPACK benchmark is a yardstick of computer performance [67, 70].

The TOP500 list [188] contains computers ranked by their performance on the

LINPACK Benchmark in the arbitrary matrix size modality [66, 68].

HPL is also part of the HPC Challenge Benchmark Suite, the Suite has been released by the DARPA HPCS program to help define the performance boundaries of future Petascale computing systems [160]. Once a year an HPC Challenge Awards Competition takes place [108] which includes two classes of awards: (i) Class 1: Best Performance (ii) Class 2: Most productivity: *Most "elegant" implementation of four or more of the HPC Challenge benchmarks with special emphasis being placed on: Global HPL, Global RandomAccess, EP STREAM (Triad) per system and Global FFT. This award would be weighted 50% on performance and 50% on code elegance, clarity, and size.*

We implement HPL in UPC with the HPC Challenge Awards Competition in mind, following the specifications in [108].

Description HPL solves a linear system of equations of order n:

$$Ax = b; A\epsilon\mathcal{R}^{n\times n}; x, b\epsilon\mathcal{R}^{n}$$

By first computing the LU factorization with row partial pivoting of the n-by$(n-1)$ coefficient matrix coefficient matrix:

$$[A, b = [[L, U], y]$$

Since the row pivoting and the lower triangular factor L are applied to b as the factorization progresses, the solution x is obtained by solving the the upper triangular system:

$$Ux = y$$

The lower triangular matrix L is left unpivoted and the array of pivots is not returned.

In the bibliography several reports were found with different algorithms implementing HPL and LU factorization, and performance numbers reported for different supercomputers. In most of them data is distributed among processes according to the block-cyclic scheme. The n-by-$(n+1)$ coefficient matrix is first logically partitioned into nb-by-nb blocks that most of the implementations distribute onto a two-dimensional P-by-Q grid of processes to ensure a good load balance as well as the scalability of the algorithm [72]. This is the case of MPI-HPL [9], SCALAPACK LU factorization [50, 48, 117], among many others [63, 190], in Co-Array Fortran [165], other implementations use a block column distribution and apply the look-ahead optimization [159].

Implementation While implementing a UPC version of linpack we realized that tiles were absolutely necessary and we introduced the presented multi-dimensional blocking of shared arrays. This extension allowed us to benefit from the optimized BLAS [25] (or ESSL [78]) routines. Our effort have resulted in a nice and simple code (with only 288 lines, Table 5.5 shows the countage of the HPC UPC code. It has 536 lines including comments, a few compared to more than 20 thousand lines in the MPI version of HPL) but low performance.

Lines	Cmnts	NCSL	File
48	11	30	backsolve.upc
89	26	48	main.upc
52	12	35	matgen.upc
43	25	24	panel.upc
50	13	30	pivot.upc
45	16	23	swap.upc
45	24	15	tri_solve.upc
101	49	55	update.upc
63	22	28	hpl.h
536	198	**288**	Total

Table 5.5 Line countage of the HPL UPC code. First column shows the total number of lines including comments, second column counts the commented lines and third column counts the lines of code.

We declared a shared array using the multi-blocking extension:

```
shared double [b][b] A[M][N];
```

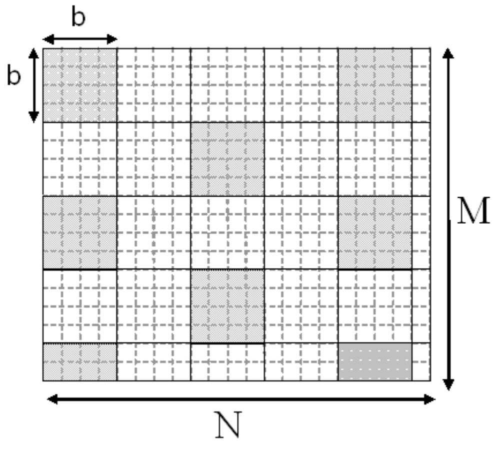


Figure 5.14 Example of UPC Block-cyclic array distribution used in UPC HPL for 4 THREADS, blocks affine to thread 0 are shadowed.

Blocks are assigned sequentially block-cyclic distribution, as shown in figure 5.14, which is a strong limitation to load balance the application.

We choose to implement the right-looking variant of the LAPACK LU factorization [50], and Level 2 BLAS routines were used. LU factorization applies a sequence of Gaussian eliminations from $[A, b = [[L, U], y]$. L is a lower triangular matrix with 1's on the main diagonal, and U is the upper triangular.

At the k_{th} step of the computation A is to be partitioned as follows:

$$\left(\begin{array}{cc|c} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{array}\right) = \left(\begin{array}{cc} L_{11} & 0 \\ L_{21} & L_{22} \end{array}\right) \times \left(\begin{array}{cc|c} U_{11} & U_{12} & y_1 \\ 0 & U_{22} & y_2 \end{array}\right)$$

where the block A_{11} is $b \times b$, A_{12} is $b \times (n - b)$, A_{21} is $(m - b) \times b$ and A_{22} is $(m - b) \times (n - b)$.

At first, our proposed algorithm does a **panel factorization** along the first $(m \times b)$ panel of A (i.e. A_{11} and A_{21}). Once this is complete the matrices L_{11}, L_{21} and U_{11} are known, and we can compute $U_{12} \Leftarrow (L_{11}^{-1})A_{12}$ using the **triangular solve** algorithm and the remaining matrix A_{22} is **updated** $A_{22}^{'} \Leftarrow A_{22} - L_{21}U_{12}$.

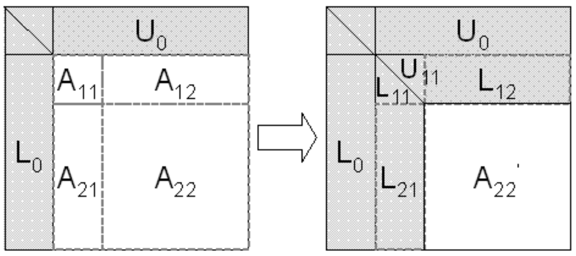


Figure 5.15 A snapshot of block LU factorization. The shared areas represent data for which computations have completed.

The LU factorization is done by recursively applying the steps outlined above to the $(m - b) \times (n - b)$ matrix $A_{22}^{'}$. Figure 5.15 shows a snapshot of the block LU factorization.

Our algorithm involves the following operations, the *panel factorization* is computed as follows:

In line 6 *max_pivot* routine uses `idamax` within the blocks and a reduction to find the absolute maximum along the column. `dscale` is used in the blocks scale the column with the pivot (line 8), and we use `dger` on every block along the panel to perform the outer product (line 10).

After the panel factorization we use `dtrsm` on every block to perform the **triangular solve** on the right part of the panel ($(U_{12}|y_1)$). And finally, `dgemm` was used to perform a matrix multiplication in every block to **update** the rest of the matrix ($(A_{22}^{'}|y_2)$), the code

```
shared int ipvt[M];       // shared array to store pivots
void panel (int col0, int col1, int col2)
{
  for(int k= col0; k < col1; k++)
    {
      int pivotRow = max_pivot(k);          // compute the pivot row
      ipvt[k-col0] = pivotRow;              // update perm. vector
      scale_column(k, pivotRow);            // scale column w/ pivot
      swap_row(k, pivotRow, col0, col2);    // then swap the rows
      outer_product(k, col2);               // update rest of matrix
    }
}
```

Figure 5.16 The routine to perform one block panel factorization. Every call updates A_{11} and A_{12} to compute L_{11} and L_{12}

Procs	Matrix size	GFlops	Efficiency
1	5000	1.47	52.50%
64	44000	47.17	26.32%
256	85000	117.87	16.44%

Figure 5.17 HPL Performance (GFlops)

for the update is very similar to the one used in Cholesky (Figure 5.12).

Since the row pivoting and the lower triangular factor L are applied to b as the factorization progresses y has been solved, the solution x is obtained by solving the upper triangular system $Ux = y$ using backward substitution: we use triangular solve for the diagonal blocks (`dtrsm` in the blocks) and matrix per vector in the others (`dgemv`).

Performance The HPL was executed on a Blue Gene machine. Each BG node has 2 processors, a 4M L3 cache and 512 MB of local memory. Nodes are connected by a 3D torus (175 MB/s/link). For our evaluation platform we use Blue Gene/X: 1 rack, 2048 processors and 512 GB of total memory (Appendix A).

Table 5.17 summarizes the performance results. Notice that the XLUPC compiler was not able to compile the UPC HPL code and local shared memory accesses were privatized directly in the code. To measure scaling performance we define efficiency as: $\frac{T_{serial}}{T_{parallel} \times \mathcal{N}}$. $\mathcal{N}$ is the number of nodes. For 1 node performance is gated by the RTS overheads to access shared data, this would be solved with the locality analysis. Next section analyzes the bottlenecks of our implementation.

Bottlenecks Two main bottlenecks were identified:

- in the update, where the `upc_memget` calls were overloading the CPUs owning $A[ii][col0]$ and $A[jj][col0]$ (see the code in 5.12). Communication is point to point, resulting in communication imbalance and waste of bandwidth. This problem could be solve with the appropriate *data centric collective* routines rather than thread centric. UPC distributed arrays create natural partitions of processes in each of the dimensions of the multi-blocked array, these partitions or subsets of threads are called *teams* in UPC terminology [145], and the closest equivalent to teams are the MPI communicators [130]. With a new set of *data centric* collectives working with *teams* the problem in the update will be solved. Collectives could also be used in *backsolve*, *triangular_solve*, *outer_product* and a reduction could be performed to find the pivot in the panel factorization.

- The other problem is load balancing, in factorization routines, such as the LU and Cholesky factorizations, in which the distribution of work becomes uneven as the communication progresses, a larger block size results in greater load imbalance, but reduces the frequency of communication between processes. There is, therefore, a tradeoff between load imbalance and communication startup cost that can be controlled by varying the block size.

  In addition to the load imbalance that arises as distributed data are eliminated from a computation, load imbalance may also arise due to computation "hot spots" where certain processes have more work to do between synchronization points than others. This is the case, for example, in the LU factorization algorithm in which partial pivoting is performed over rows, and only a single column of the process template is involved in the pivot search while the other processes are idle. Similarly, the evaluation of each block row of the U matrix requires the solution of the lower triangular system which involves only processes in a single row of the process template. The effect of this type of load imbalance could be minimized through distributing data in a PxQ grid of processes.

Some of this issues have been addressed in [145], and the rest will be addressed in our future work.

5.4 Conclusions

In this chapter we have shown that shared memory programming for large scale distributed memory machines is not a myth. **Scaling** non-trivial shared-memory programs to hundreds of thousands of threads is possible with the right support from the compiler and from the run-time system. We have described our XL UPC compiler infrastructure and the UPC Run-Time System; we have presented the essential compiler optimizations and the runtime features that contributed to high performance. We have illustrated our work with three benchmarks (Random Access, EP and CG, two of which we scaled to more than a hundred thousand processors on the BlueGene/L machine [34].

In the course of this evaluation, some benchmarks and algorithms were written in UPC - High Performance Linpack among others - we identified several issues with the current language definition, such as: rudimentary support for data distributions (shared arrays can be distributed only block cyclic) flat threading model (no ability to support subsets of threads), and shortcomings in the collective definition.

Reacting to this problems we proposed a **language extension** for UPC shared arrays that provides fine control over array data layout. This extension allows the programmer to obtain better performance while simplifying the expression of computations, in particular matrix computations. An added benefit is the ability to integrate existing libraries written in C and Fortran, which require specific memory layouts. Shared memory accesses have been optimized by exploiting data locality and the overheads of translating every shared memory access to a runtime call have been reduced.

A number of issues still remain to be resolved, both in the UPC language and more importantly in our implementation. For multiblocked arrays, we believe that adding processor tiling will increase the programmer's ability to write codes that scale to large numbers of processors. Defining a set of collectives that are optimized for the UPC programming model will also address several scalability issues, such as the ones occurring in the Cholesky, Matrix multiply and the High Performance Linpack kernel [94].

Finally, the idea of multidimensional blocking applies to any other PGAS language and also to any serial language such as C, that would benefit from the flexibility to specify the data layout, cache usage could be optimized and it could facilitate data distribution in other parallel environments such as OpenMP or MPI.

5.4.1 Data layout in MPI

Finally, this work was done in the UPC Runtime System, because MPI handles the data distribution explicitly. A typical MPI program wanting to use the total available memory will organize processes such that every process allocates its piece of the array and some additional information is required to logically reconstruct the whole array. For instance, in figure 5.3 every MPI process allocates its physical piece of the array and stores the necessary information to reconstruct the logical position of every block within the global array (the array dimension, the blocking factor in each dimension and information about the grid of processes where the data was distributed). The source code becomes by far more complex than in UPC.

On the other hand, the closest somehow equivalent to tiles in MPI are the *MPI*

Datatypes. MPI datatypes are very flexible allowing the programmer to exploit data locality and minimize communication. The basic MPI communication mechanisms can be used to send or receive a sequence of identical elements that are contiguous in memory. To overcome this limitation Datatypes were introduced. Derived datatypes allow the user to specify more general data layouts, which allows the user to send data that is not homogeneous or that is not contiguous in memory. The fixed overhead of sending and receiving a message is amortized over the transmittal of many elements. Datatypes are constructed by specifying: a sequence of basic types plus a sequence of integers (byte) displacements (*typemap* in MPI terminology). Datatypes are by far more flexible than tiled arrays, but the responsibility is left to the programmer, and again it results in long and complex source codes.

Chapter 6

Remote Address Cache: Scalable RDMA performance for PGAS languages

6.1 Introduction

Parallel programming for both multicores and large scale parallel machines is becoming evermore challenging; adequate programming tools offering both ease of programming and productivity are essential. However, while the productivity of developing applications for these machines is important, the users of massively parallel machines are expecting the same level of performance as obtained by manual tuning of MPI applications.

Partitioned Global Address Space (PGAS) languages, such as UPC, Co-Array Fortran, and Titanium, extend existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distributions. These languages provide a simpler, shared memory-like programming model, with control over the data layout. The performance of these languages relies on two main factors: (i) the programmer to tune for locality by specifying appropriate data layouts; and (ii) the compiler and runtime system to efficiently implement the locality directives.

As a reminder, the memory model of UPC follows the PGAS model, with each thread having access to a private, a shared local, and a shared global section of memory. Threads have exclusive, low latency, access to the private section of memory. Typically the latency to access the shared local section is lower than the latency to access the shared global section. The UPC memory and threading model can be mapped to either distributed memory machines, shared memory machines or hybrid (clusters of shared memory machines, such as MareNostrum [32]).

In this chapter we describe an optimization in the IBM XLUPC runtime system to exploit hardware features, such as Remote Direct Memory Access (RDMA), and improve application performance while maintaining a scalable design. The IBM XLUPC runtime system (hereafter RTS) uses a Shared Variable Directory (introduced in Section 5.2.2) to provide a scalable infrastructure that has been demonstrated to scale to hundreds of thousands of threads on the BlueGene/L computer [17]. However, in this scalable design, nodes keep only local information which prevents a better use of the RDMA. To exploit RDMA, we implemented a cache of remote addresses on two different platforms: the MareNostrum [32] supercomputer, and a cluster of Power5 SMP machines. We measured the performance of this optimization on a set of benchmarks, and we demonstrate a reduction in execution time of up to 40% and 30%, on each of the platforms, respectively.

The effect of using RDMA are the largest for very short messages; these are the kinds of performance improvements that conventional two-sided messaging systems cannot achieve because of design limitations (e.g. MPI relies on message matching on the receiver, which rules out RDMA transfers).

This chapter makes the following main contributions:

- describes the Shared Variable Directory (SVD), underlying the design decisions which are crucial to the scalability of the UPC Run-Time System (RTS).
- presents a runtime optimization that allows the runtime to improve its performance while being scalable and portable.
- introduces a UPC parallel implementation of a subset of the DIS Stressmark Suite and evaluates these benchmarks on our system.

All discussion in this chapter will focus on the UPC language, however, the optimizations apply to any of the PGAS languages.

6.2 The IBM XLUPC runtime

The IBM XLUPC RTS has multiple roles: it spawns and collects UPC threads, implements accesses to shared data, performs pointer arithmetic on pointers to shared objects and implements all the UPC intrinsic function calls (such as `upc_phaseof`, `upc_barrier` and `upc_memget`). The RTS defines an external API that is used by the UPC compiler when generating code.

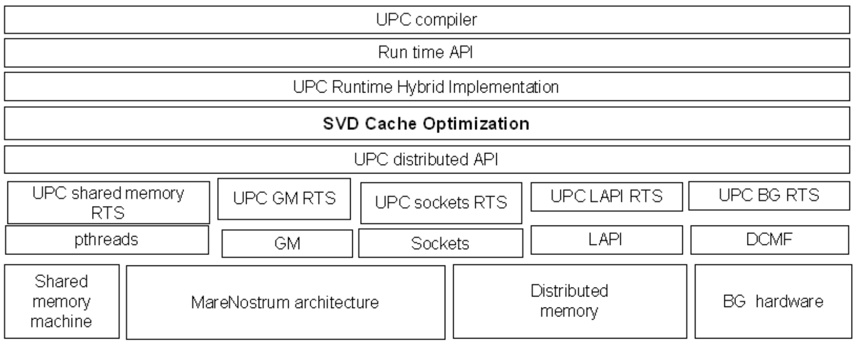


Figure 6.1 Software organization of RTS

The overall architecture of the RTS (Figure 6.1) has evolved since last chapter (Figure 5.2). We have specified a new API, `UPC distributed API` which splits the runtime into its messaging system and the common RTS functionalities, resulting in an easy-to-extend modular framework to accommodate new messaging systems for other machine architectures. It is similar to that of GASNet [27]. It provides a platform-independent interface that can be implemented on top of a variety of architectures, SMP or distributed.

The RTS is designed for a *hybrid* mode of operation on clusters of SMP nodes: UPC threads communicate through shared memory when available, and send messages through one of several available transports when necessary. The Pthreads library is used to spawn multiple UPC threads on systems with SMP nodes. Two messaging systems have been added and currently, implemented messaging methods include TCP/IP sockets, LAPI [170], Myrinet/GM transport [89] and the BlueGene/Lmessaging framework [7].

In this thesis we focus on the hybrid and distributed-memory implementations of the runtime.

6.2.1 The Shared Variable Directory

The Shared Variable Directory (SVD) has already been introduced in section 5.2.2, however some information needs to be either added or reminded due to its relevance on the work presented here.

Shared objects (i.e. data structures accessible from all UPC threads) form the basis of UPC language. The RTS recognizes several kinds of shared objects: *shared scalars* (including structures/unions/enumerations), *shared arrays* (including multi-blocked array [16]), *shared pointers* (with either shared or private targets) and *shared locks*.

All UPC shared objects have an *affinity* property. A shared object is affine to a particular UPC thread if it is local to that thread's memory. Shared arrays are distributed in a block-cyclic fashion among the threads, so different pieces of the array have affinity to different threads.

Access to shared objects presents a scalability problem that all UPC implementations share, namely that the addresses of locally allocated portions of shared objects are needed by other nodes in order to access shared data. There are multiple solutions to address this problem:

- Ensure that shared objects have the same addresses in all nodes. Unfortunately this approach does not work too well with dynamic objects: it tends to fragment the address space and it is cumbersome to implement, sometimes requiring changes to the system memory allocator.

- A distributed table of size $O(nodes \times objects)$ can be set up to track the addresses of every shared object on every node. For a large number of nodes or threads, this can be prohibitively expensive and thus, directly impacting scalability. It also requires extensive communication when the virtual to physical mapping changes on a particular node.

- The third solution involves a distributed symbol table. Shared objects are known by their handles (unique identifiers for every shared object). Translation from shared object handles to memory locations only happens on the home node of the shared object.

In the RTS we opted for the last approach. Shared objects are organized into a distributed symbol table called the Shared Variable Directory (SVD). The SVD manages the life cycle of shared objects (allocation, freeing, use). On a system with n UPC threads the SVD consists of $n + 1$ partitions. Partition k, $0 \leq k < n$ holds a list of those variables *affine* to thread k. The last partition (called the *ALL* partition) is reserved for shared variables allocated statically or through collective operations.

Shared objects are referred to by their *SVD handles*, opaque objects that internally index the SVD and keep the associated memory address in question. An SVD handle contains the **partition** number in the directory, and the **index** of the object in the partition (Figure 6.2).

Multiple replicas of the SVD exist in a running XLUPC system. The SVD often changes at runtime because UPC routines for dynamic data allocation, such as `upc_global_alloc`, `upc_all_alloc`, and `upc_local_alloc`. The SVD has to be kept internally consistent. Partitioning greatly aids this process, because it allows the SVD to be kept consistent with a minimal effort and without any bottlenecks:

1. UPC threads can allocate and de-allocate shared variables independently of each other. Each thread updates its own partition, and sends notifications to other threads;

2. Because the SVD is partitioned, and each partition has a single writer, memory allocation events do not require locks. The *ALL* partition is only updated by collective memory allocation operations that are already synchronized.

6.2.2 The performance compromise of the SVD design

We consider the SVD essential to the scalability of the XLUPC runtime. Unfortunately there is a price to be paid for translating SVD handles to memory addresses only at the target node.

Modern communication transports (like Myrinet [140] and LAPI [102]) have one-sided RDMA communication primitives that require no CPU involvement on the remote end. However, these primitives typically require the *physical* address of the shared object at the target to be known by the initiator of the communication. Figures 6.3 and 6.4

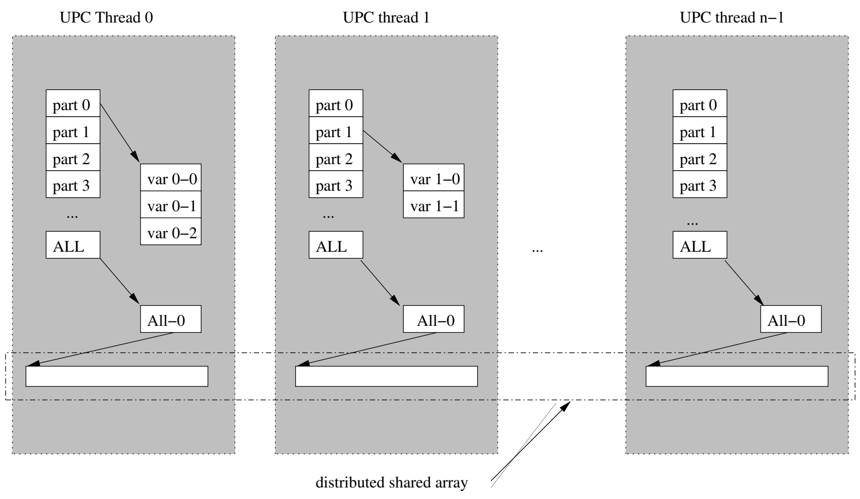


Figure 6.2 Shared Variable Directory in a distributed memory machine. Gray boxes represent the local memory of each of n UPC threads, each with a copy of the SVD. The SVD has $n + 1$ partitions; each partition has a list of control blocks, one each for shared objects known locally. Addresses are only held for the local or ALL partitions. Distributed shared array "All-0" has a different local address on every node.

contrast the implementation of remote GET and PUT protocols when the address of the object is known, and when it is not.

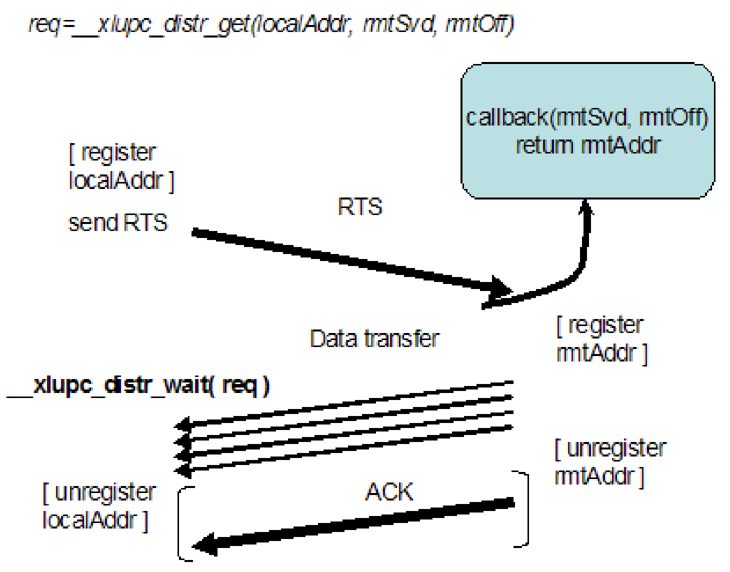
RDMA GET and PUT functions cannot be used in a naive SVD implementation, since SVD lookups require CPU involvement. This both lengthens communication latency and burdens the target CPU with work, contributing to the scalability bottleneck.

In the next section we outline an optimization that allows the use of RDMA operations while preserving the SVD design, by caching the remote addresses of shared objects on an as-needed basis.

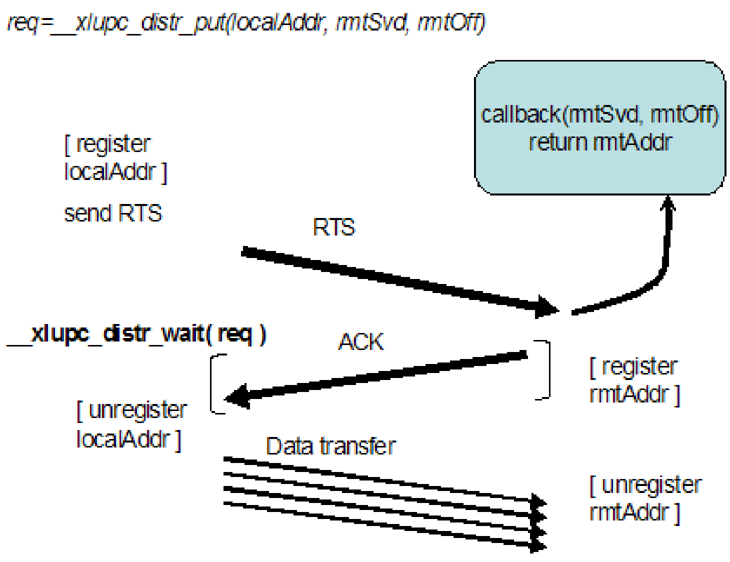
6.3 Remote Address Cache

The goal of the remote address cache (hereafter `address cache`) is to enable small message transfers via RDMA, and thus reduce latency and improve the performance of the RTS. It does this by caching remote addresses as needed by computation. Unlike a full table of remote addresses, the cache's memory requirements can be controlled and offset against performance.

The `address cache` is implemented as a hash table. Each entry in the cache correlates a universal SVD handle and a node identifier ID with the physical base address for the shared variable identified by the SVD handle on the remote node ID. During a GET or PUT operation the initiating node consults the `address cache` for the base

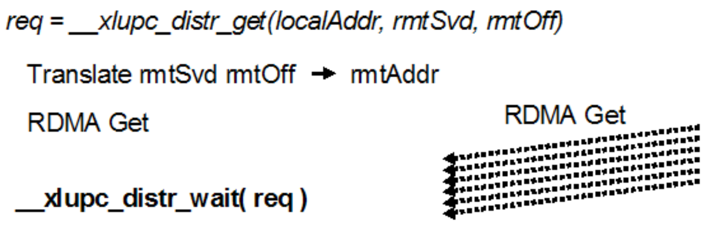


(a) Get

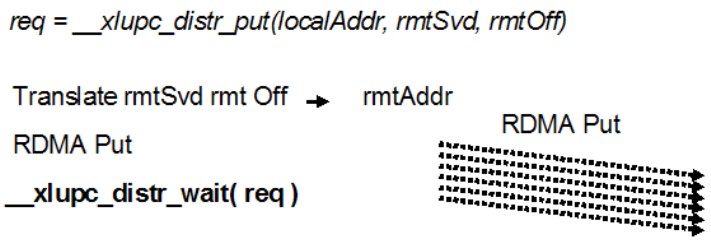


(b) Put

Figure 6.3 Protocols for XLUPC GET and PUT operations when the remote address is unknown. Acknowledge message may be required depending on the implementation (i.e. to signal completion or to carry the remote address if data transfer is performed thought one-sided communication).


req = __xlupc_distr_get(localAddr, rmtSvd, rmtOff)
Translate rmtSvd rmtOff → rmtAddr
RDMA Get
RDMA Get
__xlupc_distr_wait(req)

(a) Get


req = __xlupc_distr_put(localAddr, rmtSvd, rmtOff)
Translate rmtSvd rmt Off → rmtAddr
RDMA Put
RDMA Put
__xlupc_distr_wait(req)

(b) Put

Figure 6.4 Protocol for GET and PUT operations with RDMA operations when remote addresses are known.

address on the target node. A cache hit guarantees that the final remote address (base address + offset) can be calculated on the initiator node, allowing the message transfer to be executed as an RDMA operation.

Conversely, if the address lookup in the cache fails the operation cannot be executed as an RDMA operation. However, the slower operation can be harnessed to retrieve the remote base address for the next operation to the same node. We have modified the default (non-RDMA) one-sided messaging protocol to retrieve the base address of the remote shared object during the transfer by piggybacking it either on the data stream or on the ACK message.

To ensure the correctness of RDMA transfers, the remote node has to guarantee that the remote memory address has not changed between accesses. To ensure that the *physical* address (typically required by RDMA operations) of a shared array is fixed, we need to *pin* (aka. *register*) the array in memory [187] [181]. To this end we augmented the address cache with a table of registered (pinned) memory locations. The *pinned address table* is tagged by **local** virtual addresses and contains physical addresses in the format needed by RDMA operations.

Figure 6.5 portrays a typical runtime snapshot. In the figure the address cache of node A caches two remote addresses on node B. Both these entries exist in the pinned address table of node B. Node C also caches one of these entries, as well as another entry on node A (shaded appropriately to show the correspondence).

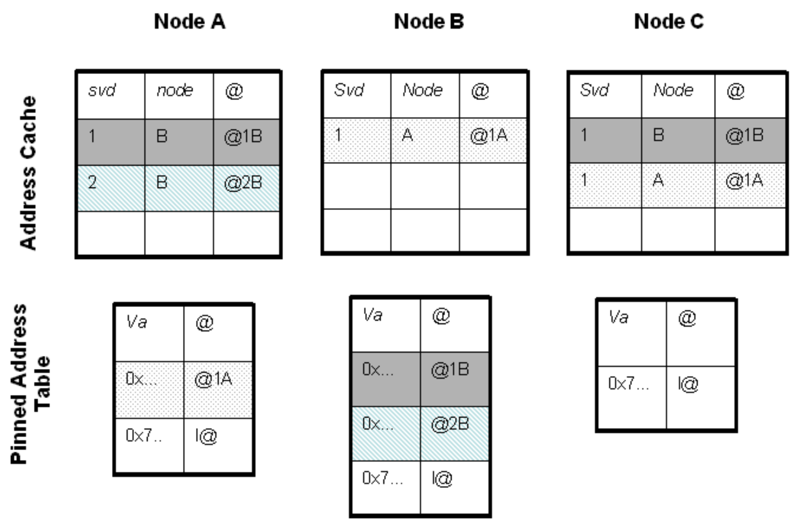


Figure 6.5 Example snapshot of address cache on three nodes.

The `address cache` is implemented as a layer sandwiched between the top layers of the RTS and the distributed messaging API (Figure 6.1 highlights the `address cache`'s location in boldface). This design allows for the cache to be turned on or off transparently, mitigating the difficulties caused by grafting research prototypes onto commercial software.

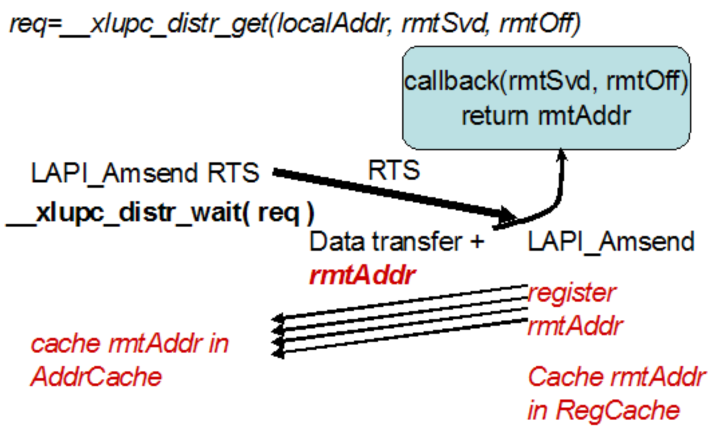
6.3.1 Considerations for a LAPI based implementation

The LAPI implementation of the XLUPC low level messaging API uses Active Messages (`LAPI_Amsend`) (as shown in Figure 6.6). Implementation of the remote address cache required only trivial changes in the messaging library to enable the process of populating the `address cache`. Cache hits result in messages that bypass the standard messaging system completely and use RDMA directly.

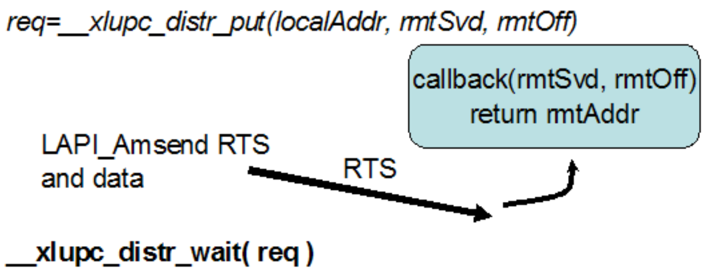
LAPI limits the amount of memory that can be assigned to a single registered memory handle (a configuration parameter, 32 MBytes on our machines).

6.3.2 Considerations for the Myrinet/GM implementation

GM [89] is a message-based communication system for Myrinet. The RTS Myrinet port was implemented on top of GM instead of MX [139], mainly because GM is currently the driver installed on our testing platform (the MareNostrum supercomputer [32]). Moreover, GM provides one-sided support and primitives to directly expose the RDMA capabilities of the hardware.



(a) Get


req=__xlupc_distr_put(localAddr, rmtSvd, rmtOff)
callback(rmtSvd, rmtOff)
return rmtAddr
LAPI_Amsend RTS
and data
RTS
__xlupc_distr_wait(req)

(b) Put

Figure 6.6 Enabling `address cache` population in the standard XLUPC LAPI implementation (changes shown in *italics*). For GET messages, the initiating process sends an Active Message (`LAPI_Amsend`) that triggers the execution of a header handler on the passive target. The header handler performs address translation *and memory registration*. The reply sent back to initiator process contains requested user's data *plus the address*. We did not change PUT messages.

However, three considerations need to be taken into account: first, Myrinet requires memory registration for any data transfer, and it is handled by the programmer in GM. Second, the largest message GM can send or receive is limited by the amount of DMAable memory the GM driver is allowed to allocate by the operating system (1 GByte on our test machines). Finally, memory registration is an expensive operation; memory de-registration even more so.

Our original XLUPC Myrinet port implements multiple transfer protocols depending on message size [144]. Short messages are copied to avoid memory registration costs. Long messages are transferred with an MPI-like rendezvous protocol with memory registration/de-registration embedded into the phases of the protocol. As an optimization a cache of registered memory regions was implemented [185] with lazy memory de-registration.

Again, the changes required in the messaging library to enable the remote address cache were trivial (Get in Figure 6.7 and Put in Figure 6.8) and very similar to the ones in LAPI.

6.3.3 Memory Registration Issues

Depending on their support for page registration we distinguish two categories of network interfaces: Automatic **Hardware-assisted** registration or passive **Pinning-based** registration. With a hardware-assisted approach, the client software is relieved from explicitly managing and registering the memory to be used for RDMA and the NIC hardware is used to perform registration on-demand. It works similarly to a regular paging-based memory system, and there is no restrictions on what memory is made DMA-able, potentially all of the user's virtual memory can be made available to remote DMA operations. This is the case of Quadrics [157], and Infiniband [187] [181]. The network hardware is expensive but in terms of performance and scalability, hardware-assisted registration is the preferred approach for GAS language implementations.

However, most existing high performance NICs, used by the large scale machines currently in operation (e.g., ASCI Purple, MareNostrum, Blue Gene) are not equipped with the necessary hardware that manages memory registration and a pinning-based strategy needs to be used. The programmer needs to explicitly set up the region of memory to be enabled for RDMA operations. The pages are marked as nonpageables in main memory which instructs the OS that the underlying physical pages cannot be swapped out until the application terminates or explicitly unpins them. Due to this restriction the amount of memory that can be pinned at one time is limited by the size of physical memory, it depends on the OS and the NIC hardware.

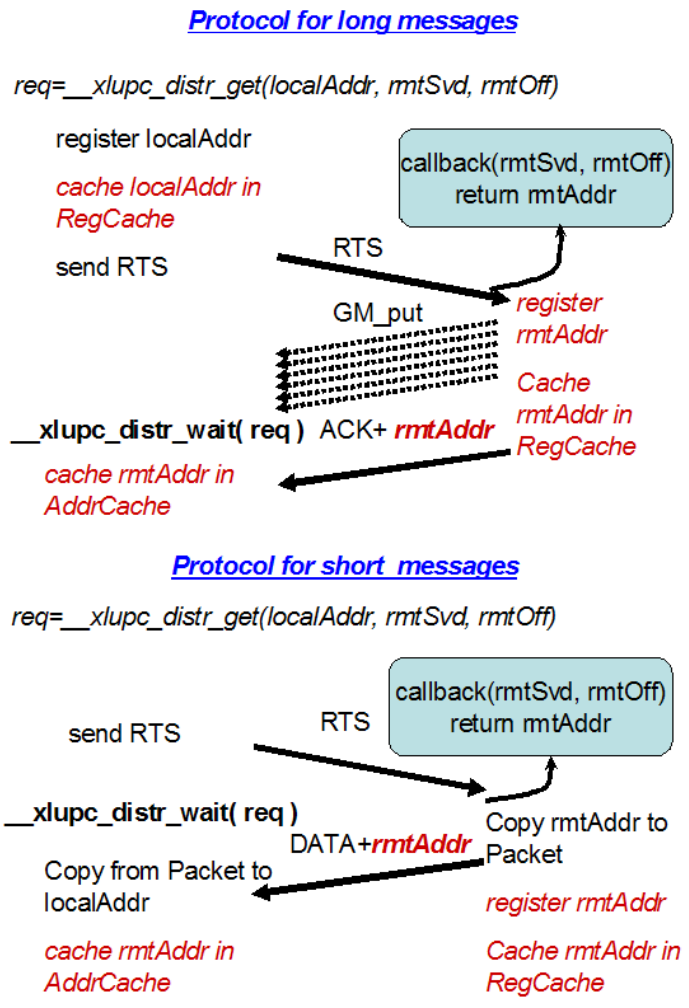

Protocol for long messages
req=__xlupc_distr_get(localAddr, rmtSvd, rmtOff)
register localAddr
cache localAddr in RegCache
send RTS
RTS
callback(rmtSvd, rmtOff) return rmtAddr
register rmtAddr
GM_put
Cache rmtAddr in RegCache
__xlupc_distr_wait(req)
ACK+ rmtAddr
cache rmtAddr in AddrCache
Protocol for short messages
req=__xlupc_distr_get(localAddr, rmtSvd, rmtOff)
callback(rmtSvd, rmtOff) return rmtAddr
send RTS
RTS
__xlupc_distr_wait(req)
Copy rmtAddr to Packet
DATA+rmtAddr
Copy from Packet to localAddr
register rmtAddr
cache rmtAddr in AddrCache
Cache rmtAddr in RegCache

Figure 6.7 GM implementation of GET function with different protocols for different message sizes. Required changes are shown in italics.

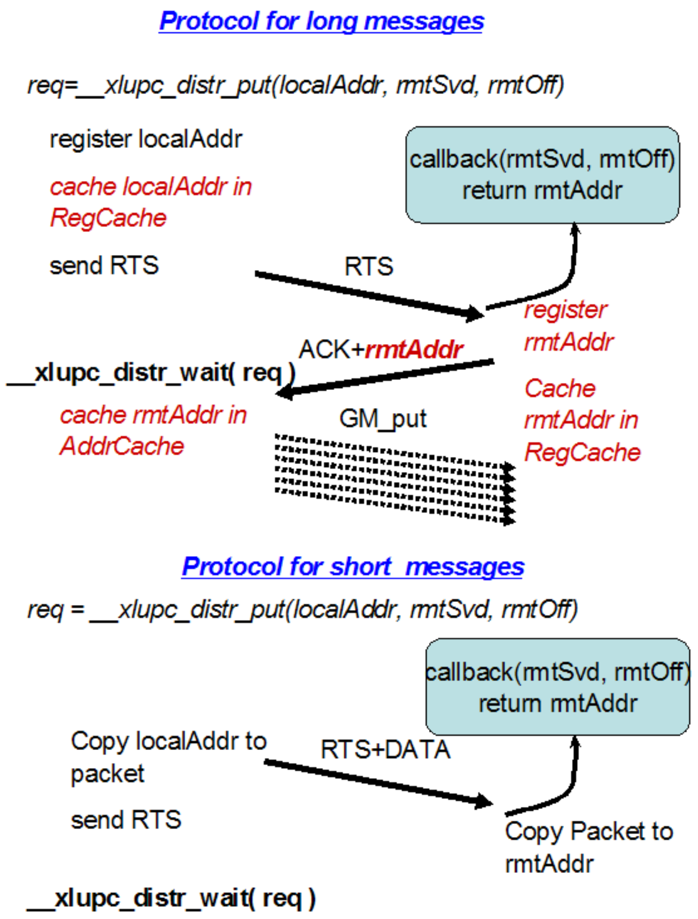

Protocol for long messages
req=__xlupc_distr_put(localAddr, rmtSvd, rmtOff)
register localAddr
cache localAddr in RegCache
send RTS
RTS
callback(rmtSvd, rmtOff) return rmtAddr
register rmtAddr
ACK+rmtAddr
__xlupc_distr_wait(req)
cache rmtAddr in AddrCache
GM_put
Cache rmtAddr in RegCache
Protocol for short messages
req = __xlupc_distr_put(localAddr, rmtSvd, rmtOff)
callback(rmtSvd, rmtOff) return rmtAddr
Copy localAddr to packet
RTS+DATA
send RTS
Copy Packet to rmtAddr
__xlupc_distr_wait(req)

Figure 6.8 GM implementation of PUT function with different protocols for different message sizes. Required changes are shown in italics.

There are many strategies for on-demand pinning and un-pinning [18, 185, 144]. We have implemented our `address cache` using two different registration strategies:

- A **greedy "pin everything"** approach, explained in section 6.3.4. This approach was induced by a desire to get an idea of the results without incurring the time and effort of a system that deregisters memory on demand. Nevertheless, it has some limitations addressed in our second implementation.

- A **Pin on-demand memory registration protocol based on memory *chunks***. It is presented in section 6.3.5.

6.3.4 Pin-everything registration strategy

Our initial implementation uses a **greedy "pin everything"** approach, it works as follows.

Before an address can be tagged in another node's `address cache` it needs to be pinned locally. In this initial implementation the entire memory allocated for a shared object is pinned at once on a particular node. For example, if any element of a shared array A is accessed on node n, *all* of the memory related to array A is pinned on that node, making it all available for RDMA operations. This simplifies the implementation of the cache, because the cache tags can simply be the SVD handles and node identifiers.

Also, once a shared object is pinned it remains pinned until it is freed. The address cache is eagerly invalidated when a shared object is deallocated.

This implementation gives us an idea of the results without incurring the time and effort of a system that deregisters memory on demand. Nevertheless, it is limited in two important respects. First, it does not deal with network transports that limit the amount of contiguous memory registered pinned a single call. And second, we also chose to ignore limits on the total amount of memory pinned.

As a consequence, the `address cache` is limited to be functional only for those shared objects whose local shared memory area is kept under the limit on the amount of registrable continuous memory at a single call. And `address cache` usage is limited to the first accessed shared object until the total amount of registrable memory is reached.

Since we choose to avoid memory deregistration, consistency is not an issue in this implementation and no additional change is required to the already-mentioned data structures shown in Figure 6.5 not to the protocols, Figure 6.6 for the LAPI-based implementation and Figure 6.7 6.8 for GM-based implementation.

6.3.5 Chunk-based Pin on-demand registration strategie

The **Chunk-based Pin on-demand memory registration protocol** aims to address the shortcomings of our *pin everything* strategy.

Two main new concepts are introduced in this implementation:

- Shared memory areas in every node are divided by chunks. Instead of pinning at once the entire local shared memory area in a node, only the appropriate chunk containing the accessed element is pinned. Chunks make the `address cache` usable to any application regardless of the required amount of shared memory area. Also, in most applications, showing communication spatial locality, registering the whole shared memory is a waste and introducing chunks may reduce registration cost.
- Registered memory can be deregistered to allow new registrations. Allowing deregistration makes the `address cache` usable by any shared object.

As a reminder, two tables are used to implement our `address cache`. The table of registered memory locations is for local housekeeping of pinned memory. And the Address Cache correlates universal SVD handles and node identifiers with physical addresses on remote nodes in the format required to issue a RDMA transfer.

Consistency between this two tables was not an issue in our previous implementation because of pin everything approach. By contrast, in our current implementation a cached address can be invalidated and a memory area can be deregistered, therefore consistency needs to be maintained.

The memory registration algorithm

Shared memory on a node is divided by chunks of size CH, and chunk identifiers of an array A are calculated depending on the array index v according to the next formula:

$$chunk(A, \mathbf{v}) ::= \left\lfloor \frac{\mathbf{v} \times e}{CH} \right\rfloor$$

Where e is the size of the array element. The `address cache` structures have been modified to accommodate our registration strategy. And the Address Cache is tagged by SVD handler, remote node and chunk identifier.

As before, a cache hit guarantees that the remote shared data is pinned down, the remote address can be calculated on the initiator and the message transfer can be executed as an RDMA operation. Otherwise, in case of a miss, the default non-RDMA one sided messaging protocol is used.

Table 6.1 Data structure for the address cache.

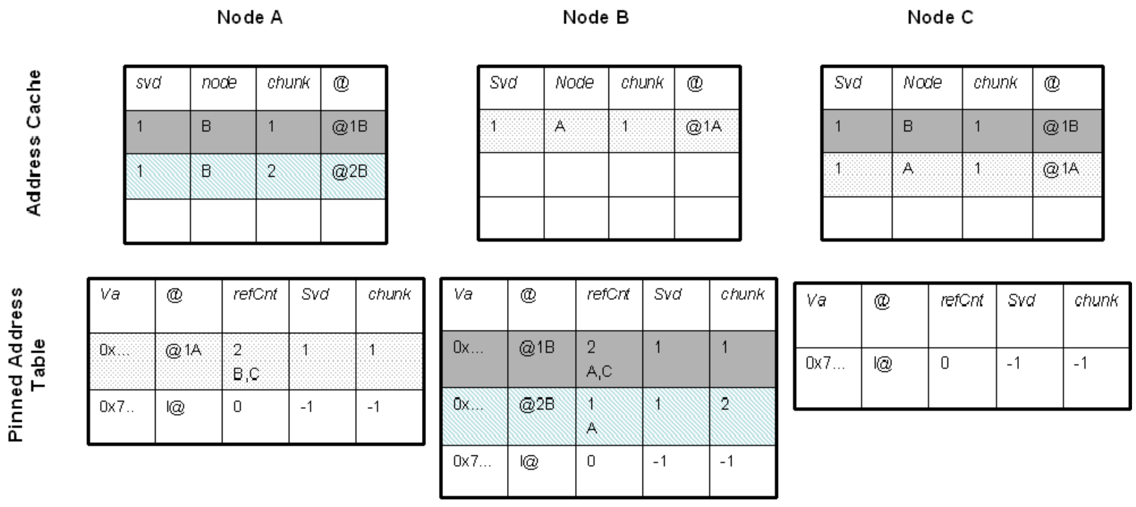
Address Cache	Hash table keyed on tuple of: **SVD handler**, **remote node** and **chunk identifier**. It is populated on an on-demand basis. The **physical address** on the remote node needed for the RDMA transfer A **usage counter** monitors usage of every entry to avoid invalidation during usage. Victim LRU to evict cache entries if a size limit is reached.
Pinned Address Table	Table keyed on **virtual address**, with one entry for each currently pinned memory area. It containes both local and shared addresses. The **physical address** and a **usage counter** to track usage. While the usage counter is greater than zero, the entry is not a candidate for unpinning because it is currently in-use by a locally-initiated operation. A **reference counter** counts the number of remote nodes that have cached this address, required to invalidate address cache entries. • 0 means the address is private local. • if 0 < reference count < N, we keep an **array of remote node** identifiers, with the nodes currently caching this address. • if reference count > N, then the invalidation message is broadcasted among all nodes. The corresponding **SVD handle** and **chunk identifier** are used in case of invalidation to be looked up in the Address Cache in O(1) Victim FIFO is maintained to select next candidate to be deregistered.

The default protocol has been modified to retrieve the chunk base address and piggyback it on the next message to the origin node.

If the upper bound for registable memory is reached, registered chunks are freed based on a FIFO policy and invalidation messages are sent to all nodes that have cached the base address of this chunk in its address cache. This invalidation messages need to be acknowledged to guarantee that the address is not being used.

The address cache data structures have been modified to accommodate the registration strategy. Table 6.1 summarizes the required information kept in both tables. A **Usage counter** has been added to both tables to avoid invalidation during usage. In addition, the *Pinned Address Table* maintains a **Remote Reference counter** to keep track for the remote nodes that have cached this address and need to be notified in case of invalidation. As an optimization, since most of the times, only a few nodes cache an address, we keep an array of up to N remote nodes identifiers. If reference counter is bigger than N the invalidation message is broadcasted. And finally, the corresponding **SVD handle** and **chunk identifier** are also kept on the table to be looked up in the address cache in $O(1)$ in case of invalidation.

Figure 6.9 portrays a typical runtime snapshot of how three nodes manage the address cache.



Node A – Address Cache

svd	node	chunk	@
1	B	1	@1B
1	B	2	@2B

Node B – Address Cache

Svd	Node	chunk	@
1	A	1	@1A

Node C – Address Cache

Svd	Node	chunk	@
1	B	1	@1B
1	A	1	@1A

Node A – Pinned Address Table

Va	@	refCnt	Svd	chunk
0x...	@1A	2 B,C	1	1
0x7..	l@	0	-1	-1

Node B – Pinned Address Table

Va	@	refCnt	Svd	chunk
0x...	@1B	2 A,C	1	1
0x...	@2B	1 A	1	2
0x7...	l@	0	-1	-1

Node C – Pinned Address Table

Va	@	refCnt	Svd	chunk
0x7...	l@	0	-1	-1

Figure 6.9 Runtime snapshot of three nodes (A, B and C). The Address cache of node A caches two remote addresses on node B, node B caches one address in A, and node C caches one address in A and another in B. Entries are shadowed to see correspondence between cached addresses and registered memory areas. The Pinned Address Table of node B stores two shared addresses referenced by A and C plus a local address. And node A stores one address referenced by B and C plus a local address. Node C only stores a local address.

Considerations:

- Our registration protocol supports lazy deregistration using both the usage and reference counters. While a chunk is pinned, subsequent first time remote requests to access this chunk merely increment the reference counter and incur no registration overhead. Also, by allowing a 0-usage count chunks to remain pinned in memory, much of the burden of unpinning and re-pinning is sidestepped. Although it may lead to increased physical memory consumption it has the potential to greatly reduce pinning overhead as a bucket lazily kept is likely to be the target for subsequent remote accesses or local operations.

- A Victim FIFO queue tracks pinned chunks in the Pinned Address Table. Pinned chunks are evicted when a request to access shared memory arrives for a chunk not currently pinned and the limit of registered physical memory is reached. Before releasing the entry we need to invalidate entries in the `address cache` to keep consistency between the two tables. Using the *reference counter* and the *array of nodes* the Pinned Address Table keeps track of which nodes have cached address of victim entry and invalidation messages are sent to this nodes. Remote nodes need to acknowledge the invalidation, to make sure that the remote address is not being used.

- Our algorithm deals with memory units of chunk size, which must be decided carefully. Reducing the chunk size will increase the size of the Pinned Address bookkeeping Table. Although it may be beneficial for some applications patterns accessing repeatedly the same few chunks because deregistrations may be avoided, it can also cause a performance degradation in other applications accessing the whole set of shared data.

Changes in the communication subsystem

In the communication subsystem, functionality to send and receive invalidation messages and its acknowledgments needs to be added, as well as an up call to the `address cache` when these messages are received in order to manage invalidations.

In addition, in GM, since memory registration is required for any data transfer and handled by the programmer, our original XLUPC Myrinet port implements a cache of registered memory regions with lazy de-registration to reduce registration costs. Our `address cache` tables maintain consistency with this one.

6.4 Evaluation

The scalability of the with the Shared Variable Directory has been demonstrated in the previous chapter 5.2. In this section we aim to demonstrate the effect of the `address cache` on messaging performance. Our experiments were performed on comparatively smaller machines: 512 nodes of the MareNostrum supercomputer and a 28-node Power5 cluster running AIX.

We defined a confidence coefficient of 95% and ran each experiment multiple times to reduce the standard error. We assumed experiments to be independent, therefore the formulas associated with a normal distribution apply [98].

6.4.1 Evaluation environment: MareNostrum

MareNostrum [32] is a cluster of 2560 JS21 blades, each with two dual core IBM PPC 970-MP processors sharing 8 GBytes of main memory.

The MareNostrum's interconnection network is Myrinet with a 3-level crossbar, resulting in 3 different route lengths (1 hop, when two nodes are connected to the same crossbar aka. *linecard*, and 3 hops or 5 hops depending on the number of intervening linecards). MareNostrum overall architecture is covered in appendix A. In order to reduce runtime variability in our experiments we arranged runs up to 256 nodes on nodes that share the same linecard, resulting in more uniform 1-hop latencies. As mentioned before, we implemented the transport on top of Myrinet/GM messaging library.

6.4.2 Evaluation environment: Power5 cluster

The second machine we performed our experiments on is a 28-node cluster of Power5 servers. Each node is with 16 GBytes of RAM and 8 2-way SMT Power5 cores running at 1.9 GHz. The nodes are connected with an IBM High-Performance Switch (HPS). We used the LAPI library provided as part of the Parallel Operating Environment on the system as the basis for the XLUPC transport implementation. More details about the machine architecture can be found in appendix A.

6.4.3 Microbenchmarks

Our first set of experiments sought to quantify the maximum benefit obtainable by the `address cache`. We wrote and executed microbenchmarks to compare GET round-trip latencies and PUT overheads of the with and without cache operation. Pin everything strategy was used in this experiments.

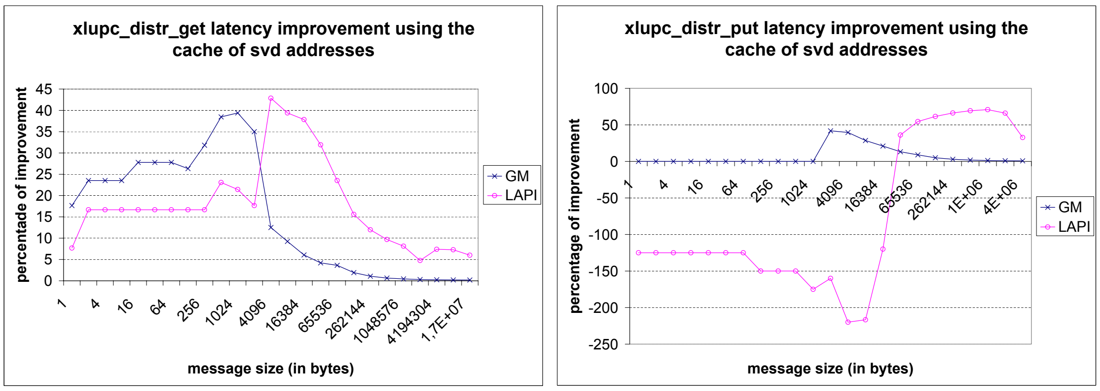


Figure 6.10 Latency improvement by using the `address cache` in both platforms: LAPI(`o`) and GM (`x`) considering different message sizes. The `Y` axis represents the performance benefit in terms of execution time reduction by using the `address cache`, expressed in percentage ($\frac{100(Z-W)}{Z}$, with `Z` being the average regular latency, and `W` being the average latency by making use of the `address cache`. `X` axis shows the message size.

Figure 6.10 shows the relative gains (i.e. execution time reduction) caused by deploying the `address cache` for PUT and GET on both LAPI and Myrinet/GM transports. We see three distinct modes on all platforms.

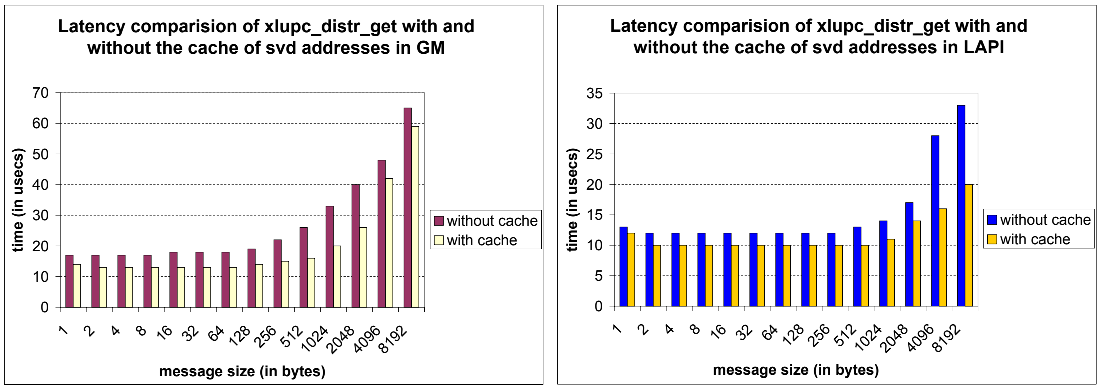


Figure 6.11 GET latency with and without the `address cache` in both platforms: GM and LAPI, considering short message sizes. The `Y` axis represents the latency. The `X` axis is the message size.

- For very small messages (up to 1 KByte in size) the gains in GET round-trip latency (left panel of Figure 6.10) are in 30% and 16% range respectively for GM and LAPI. This is the optimization that we had been targeting.
- For medium message size range messages (1 KByte to 16 KByte) there are even larger gains (around 40%). This is likely due to the rising cost of memory copies in the non-cached case.
- As expected, differences between cached and uncached behavior diminish as message size increases and communication becomes bandwidth dominated instead of overhead dominated.

With PUT messages (right panel of Figure 6.10), in GM we do not see any benefit of using the `address cache` for small message transfers, up to 2 KBytes. PUT execution time is not significantly affected by address translation overheads, and the load on the remote CPU is not measured on this microbenchmark.

The situation is different for LAPI, where we see a net decrease in performance of up to 200% by using the `address cache`. The cause of this performance decrease is the IBM switching hardware, which offers excellent throughput in RDMA mode, at the cost of higher latency. In a GET operation the higher latency is offset by the fact that the remote node's CPU time is not part of the roundtrip time; in the case of PUT the remote CPU's operation is overlapped with the next send. Following these results, we disabled the `address cache` for the PUT operations in LAPI.

Figure 6.11 shows another view of the GET roundtrip latency data: the absolute latencies in microseconds for small message GET operations with and without address caching on both our test platforms.

6.4.4 DIS Stressmark Suite

We have implemented a subset of the DIS Stressmark Suite [12] and ran it with and without address caching. The purpose of the benchmark suite is to recreate real *data-intensive* applications. Four of the benchmarks have been implemented and evaluated:

- The **Pointer Stressmark** is repeatedly following pointers (hops) to randomized locations in memory until a condition becomes true. The entire process is performed multiple times. Each UPC thread runs the test separately with different starting and ending positions on the same shared array.

- The **Update Stressmark** is a pointer-hopping benchmark similar to the *Pointer Stressmark*. The major difference is that in this code more than one remote memory location is read - and one remote location is updated - in each hop. All this is done by UPC thread 0, while the other threads idle in a barrier. This benchmark is designed to measure the overhead of remote accesses to multiple threads.

- The **Neighborhood Stressmark** is a stencil code prototype. It deals with data that is organized in multiple dimensions. The relationship of pairs of pixels within a randomly generated image are features that quantify the texture of the image, and require memory access to pairs of pixels with specific spatial relationships. Two statistical descriptors, gray-level co-occurrence matrix (GLCM) entropy and energy [153], are calculated using sum and difference histograms for multiple distances and directions.

  The histogram computation is performed in parallel based on the locality of the shared array. The two-dimensional pixel matrix is block-distributed in a row major fashion. This causes all pixel accesses in the same row to always be local. Accesses to pixels in other rows can be mostly local or remote depending on stencil distances and pixel positions.

- The **Field Stressmark** emphasizes regular access to large quantities of data. It consists of searching an array of random words for token strings, which are used as delimiters. All words between instances of the delimiter form a sample set, from which simple statistics are collected. The delimiters themselves are updated in memory. When all instances of a token are found and statistics computed, the process is repeated with a different token.

  The string array is blocked in memory (i.e. with a block size of $\lceil \frac{N}{THREADS} \rceil$, with `N` being the array size). Because the array is updated in every run, the outermost loop (which iterates over multiple tokens) cannot be parallelized. Parallelization is done instead in the inner loop, where each UPC thread searches the local portion of the data string for tokens. Because a token may span the boundary of two segments affine to different threads, the threads must overlap their search spaces by at least the width of a token to guarantee that all tokens are found. The algorithm ensures that only one of the two threads searching a boundary zone will actually report a token found on the boundary.

Since token lengths are typically much smaller than the string array's blocking factor, most accesses in the algorithm are local.

6.4.5 Cache size considerations

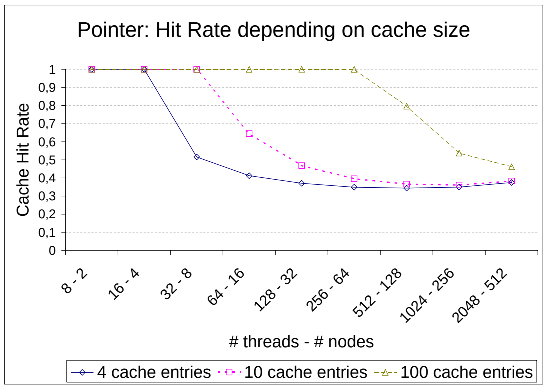
Cache size is an important metric that may affect overall application performance. The space needed by the `address cache` depends on both the number of shared variables declared by the UPC application and the communication pattern in the running application. Most UPC applications (with a few notable exceptions) declare a relatively small number of shared variables and have static and well defined communication patterns that result in insignificantly small caches even on large machines.

Our selected subset of benchmarks covers both types of applications. Field and Neigborhood follow the well defined communication patterns of typical UPC applications. As it is shown in figure 6.12 (c) and (d), it results in insignificantly small caches, only a few cache entries are used and the hit ratio keeps constant as we scale. Whereas Pointer and Update belong to the group of rare UPC applications that access remote unpredictable memory locations along the whole shared memory space, which results in address caches that grow with the number of nodes. Figure 6.12 (a) and (b) shows hit ratio degradation as we scale, with a prompt starting point as cache size is reduced. For this kind of applications we have a compromise between memory usage and speedup.

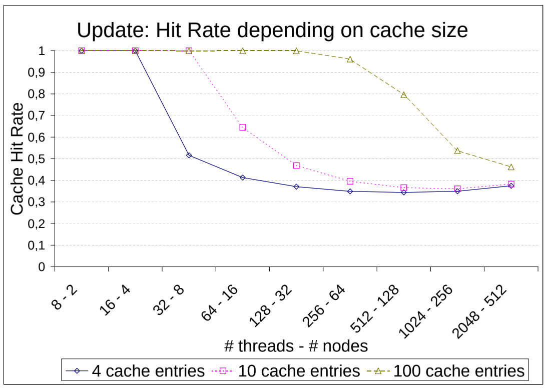
The Address Cache is currently implemented as a dynamic hash table. Its size is allowed to increase on demand to a fixed limit of 100 entries. Next section shows a considerably performance improvement, event for applications following an unpredictable communication pattern.

Concerning the *Pinned Address Table*, its appropriate size depends exclusively on the number of shared variables. Our experiments show that a table of 10 entries is more than enough for well defined UPC application.

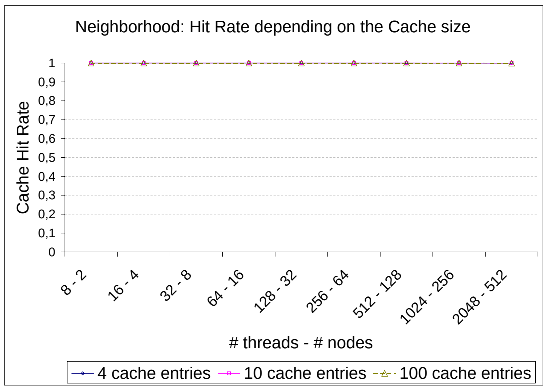
Finally, notice that chunk size affects the number of cache entries, in the figure, the pin everything strategy has been used. For applications showing regular access patterns, reducing the chunk size would reduce the registrations cost maintaining cache



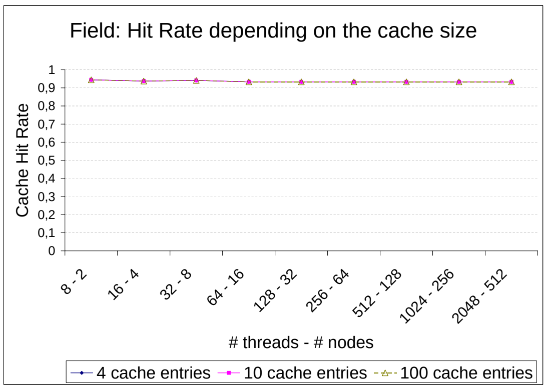
(a) Pointer



(b) Update



(c) Neighborhood



(d) Field

Figure 6.12 Address Cache Size Evaluation using DIS Stressmark Suite. The vertical axes show the percentage of hits on the Address Cache. The horizontal axes of the graphs show the threads and nodes used. Each line represents a different cache configuration in terms of cache size. Results are show considering a random thread except for Update where only thread 0 is relevant.

size and will be beneficial on the overall performance. Nevertheless, for applications with irregular access patterns small chunk sizes will increase cache size, replacements and registration/deregistration costs. Allocating a bigger cache will only soften the problem since for this kind of applications the real bottleneck is the registration/deregistration cost.

6.4.6 Stressmark evaluation on MareNostrum

We evaluated the 4 Stressmarks on a 512 blade subset of the MareNostrum computer, with 4 UPC threads running on each blade. First, representative subsets of our experiments are shown in Figure 6.13 to show the performance gain of our `address cache` , the pin-everything strategy has been used in these experiments. And finally, in Figure 6.14 the pin-everything and on-demand registration strategies are contrasted.

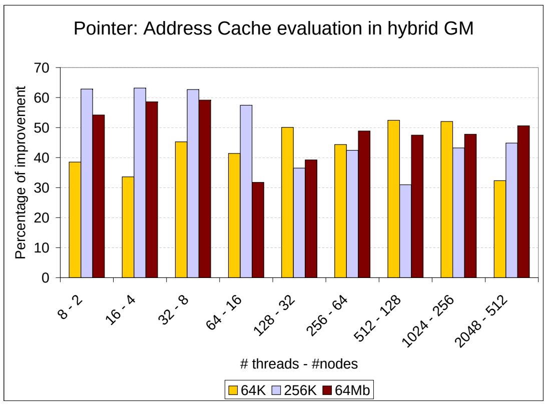
Since UPC threads running in the same blade share memory, remote communication between these threads does not involve the network hardware. This can affect measured performance improvements, since no benefit from the `address cache` can be expected when the network hardware is not in use.

We show performance improvement taking the default implementation (non-cache) as a baseline. Benchmarks's execution time varies significantly depending on the input parameters and overall improvement may also vary slightly. Input parameters are such as the total shared array size, the maximum number of hops (in Pointer and Update), the stencil (Neighborhood) or the number of tokens (Field). A subset of the most representative experiments is shown on the figures.

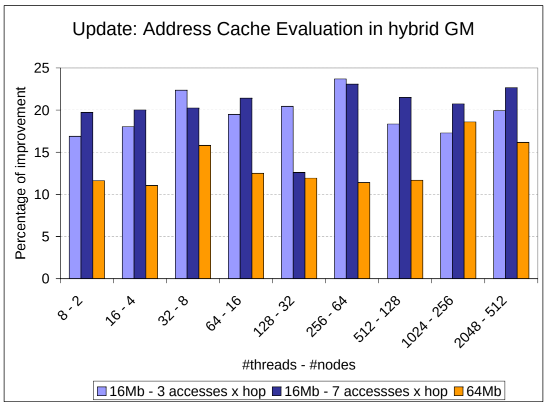
The *Pointer Stressmark* (Figure 6.13a) shows good performance, between 30% and 60% improvement depending on the total number of remote accesses in the benchmark. The performance of the Pointer stressmark is gated by network latencies and overheads; any reduction in these results in performance gains. In hybrid execution mode the network device is shared by all UPC threads running on a blade; the improvement caused by smaller CPU overheads was augmented by smaller network device overheads.

The *Update Stressmark*, (Figure 6.13b) shows a 10% to 25% performance improvement when the `address cache` is enabled, depending on benchmark parameters like total shared array size and number of shared memory accesses per hop. This corresponds to the performance measured by the PUT and GET microbenchmarks. We do not see performance improvement caused by two threads per node, because only thread 0 initiates communication.

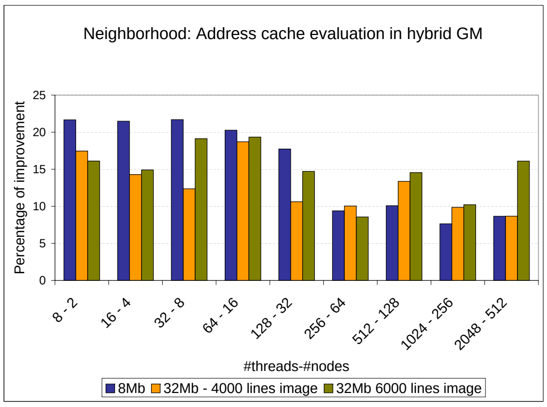
The *Neighborhood Stressmark*, (Figure 6.13c) shows 10% to 20% improvement. The stencil used in this experiment (with a stencil distance of 10) causes about $\frac{3}{16}$ of memory accesses to be remote, therefore the improvement is mostly along the lines we expected



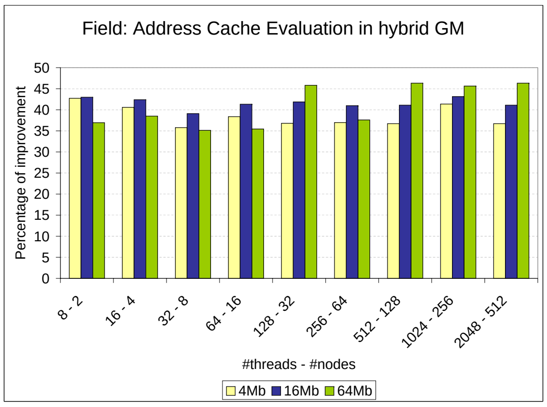
(a) Pointer



(b) Update



(c) Neighborhood



(d) Field

Figure 6.13 Address Cache Evaluation on GM using DIS Stressmark Suite with different configurations. The vertical axes show performance improvement when using the remote address cache: $\frac{100(Z-W)}{Z}$, where `Z` and `W` are the regular and address-cache-enabled runtimes respectively. The horizontal axes of the graphs show the threads and nodes used.

based on the microbenchmark.

The *Field Stressmark* (Figure 6.13d) shows a 35% to 45% performance improvement depending on array size and tokens' length and location. We analyzed the behavior of this benchmark using the Paraver performance analysis toolkit [136] [152]. The trace showed that the remote GET and PUT access times at the "overhangs" were abnormally large when `address cache` was not in use. As a reminder, each thread in the benchmark scans the local portion of a distributed array. The scan extends into "overhangs" that belong to the two neighboring threads. With normal operation the remote node's CPU is part of every remote access; but the Myrinet/GM transport does not overlap communication and computation. While a CPU is busy with the local portion of its array the network does not make progress, and other CPUs requesting data are forced into long waits.

By contrast, when the address cache is in use RDMA operations are used for remote accesses. These require no cooperation from the remote node's CPU; therefore remote access wait times decrease significantly, and performance improves.

Effect of the registration strategy

Figure 6.14 shows the effect of the registration strategy implemented, by contrasting the benefits of one of the experiments using both presented strategies: pin-everything 6.3.4 and Pin On-Demand 6.3.5 strategies.

Several chunk sizes were considered to test our Pin On-Demand registration strategy. Although we observed small differences between them, a summary of our observations is outlined as follows.

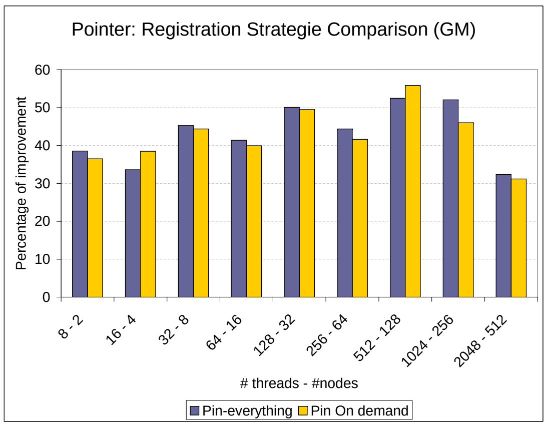
Smaller chunk sizes penalize Pointer and Update benchmarks because due to the chunk size the `address cache` is more populated and due to the poor locality there is more replacement. On the contrary, small chunk sizes benefit Neighborhood and Field benchmarks because they show locality in their access patterns and registration costs are reduced. A reasonable chunk size of 256Kb is shown in the figure.

As expected, the performance benefits of using both strategies are very similar, with the Pin On-Demand strategy showing a little bit more overhead in general.

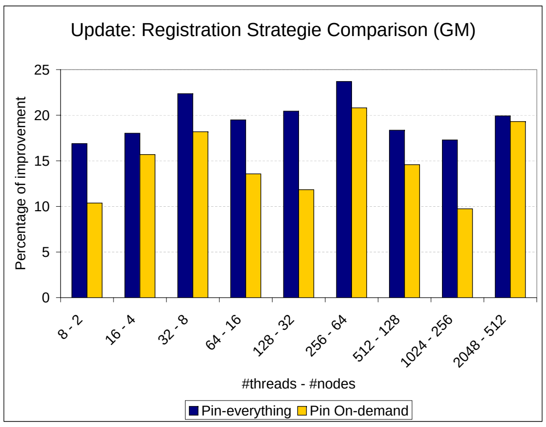
6.4.7 Stressmark evaluation on Power5 cluster

We have also run the four Stressmarks on the P5 cluster. We chose several thread/node configurations. Figure 6.15 shows the results with up to 448 UPC threads.

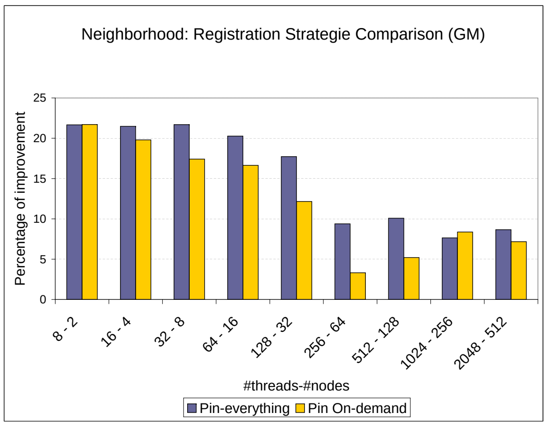
The *Pointer Stressmark* and *Update Stressmark* (Figures 6.15 (a) and (b)) show results



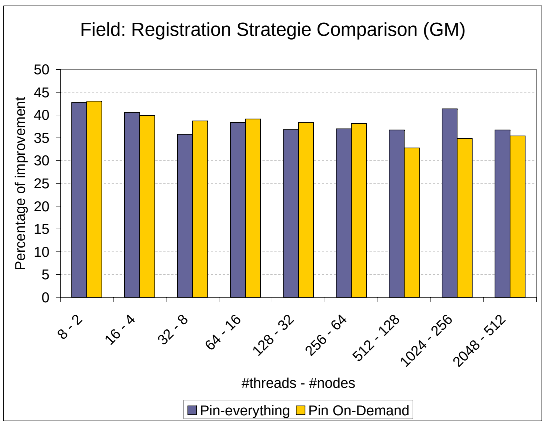
(a) Pointer



(b) Update

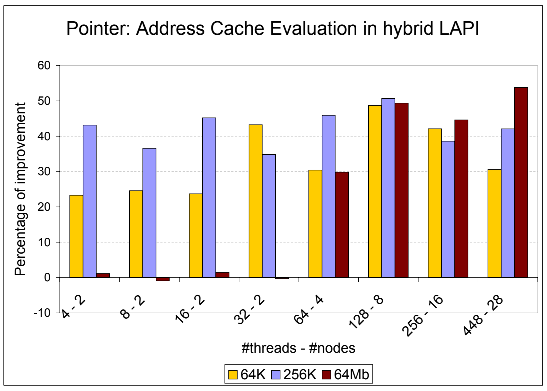


(c) Neighborhood

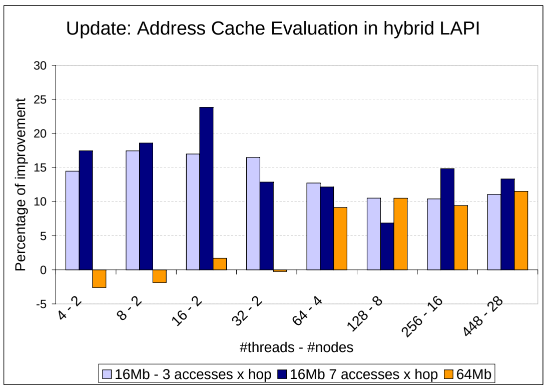


(d) Field

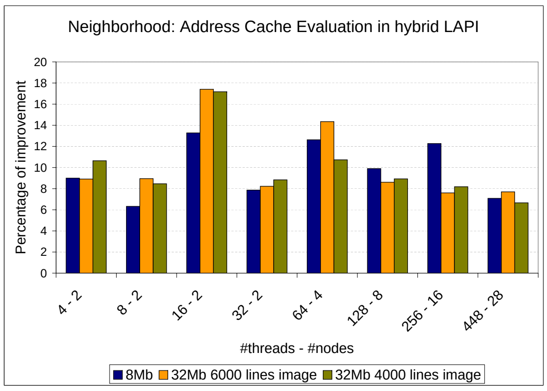
Figure 6.14 Registration Strategy Comparison on GM using DIS Stressmark Suite. First column represents the Pin everything strategy, while second column is the Pin on-demand. The vertical axes show performance improvement when using the remote address cache: $\frac{100(Z-W)}{Z}$, where Z and W are the regular and address-cache-enabled runtimes respectively. The horizontal axes of the graphs show the threads and nodes used.



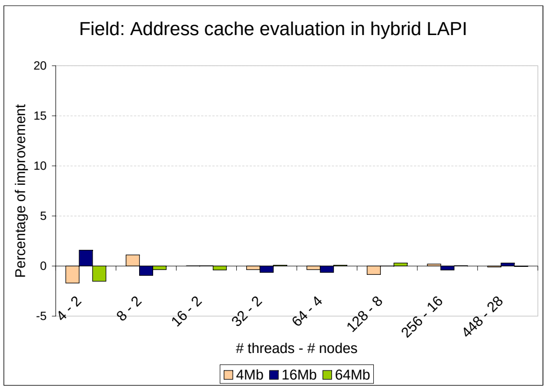
(a) Pointer



(b) Update



(c) Neighborhood



(d) Field

Figure 6.15 Address Cache Evaluation on Power5 machines with LAPI using DIS Stressmark Suite. The vertical axes show performance improvement when using the remote address cache: $\frac{100(Z-W)}{Z}$, where Z and W are the regular and address-cache-enabled runtimes respectively. The horizontal axes of the graphs show the threads and nodes used.

comparable to the measurements on MareNostrum, except for the cases where the local portion of the shared arrays exceeded 32 MBytes of size per node; which is the case for 2 node runs. As explained in Section 6.3.1, LAPI does not allow the registration of large contiguous memory chunks, disabling `address cache` operation.

The *Neighborhood Stressmark* (Figure 6.15c) behaves predictably and in a similar way to the GM measurements. The peak bars in the (16-2) and (64-4) threads-nodes combination are due to the fact that the number of remote node accesses is larger in this configurations. An access is remote if it is locate in a different node address space, an access from thread i to data with affinity to thread j ($i! = j$) can still be local if i is running on the same node as j. In this benchmark it depends on the data layout of the array (row-distribution), the stencil (10), and the threads per node allocation.

The behavior of the *Field Stressmark* (Figure 6.15d) on the LAPI transport is markedly different from the GM measurement. LAPI allows overlap of computation and communication, therefore wait times for PUT and GET operations on remote arrays are not excessive even without `address cache` operation. Since the ratio of remote and local operations is relatively low in this benchmark, the effects of the `address cache` are not measurable.

Effect of the registration strategy

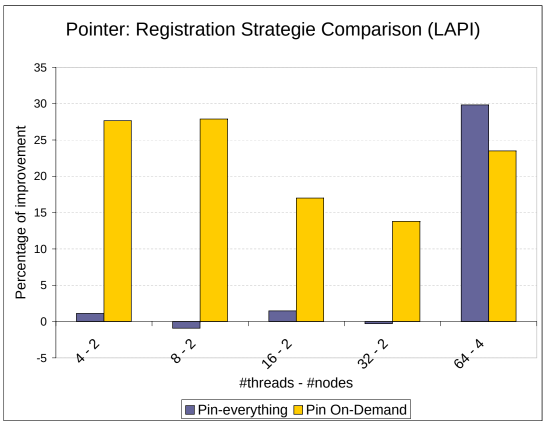
To evaluate the effect of the registration strategy, the experiment with the biggest data set was selected. If the local portion of the shared array exceeds 32 MBytes of size per node, the `address cache` could not be used with the Pin-Everything strategy and the Pin On-Demand strategy, based on chunks, is aimed to solve that problem.

Due to machine unavailability and some problems in our implementation, we were only able to collect information up to 4 nodes (64 threads). Figure 6.16 shows the evaluation.

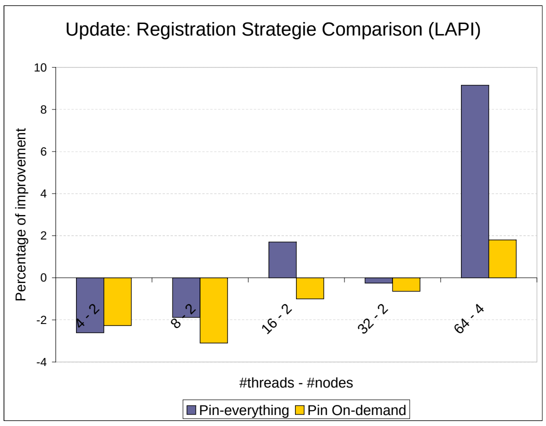
Several chunk sizes were also considered and the same chunk size as in MareNostrum configuration (256 KBytes) resulted to be reasonable and it is used for the presented results.

The *Pointer Stressmark* (Figure 6.16a) shows the expected behavior. In the first four columns (2 nodes) the `address cache` is functional in the Pin On-Demand strategy, showing between 15% and 25% improvement. The fifth column (4 nodes) shows the higher overhead of the Pin On-Demand strategic. In this case, performance comes mostly from the improvement caused by smaller network device overheads in the hybrid environment, and the better overlap.

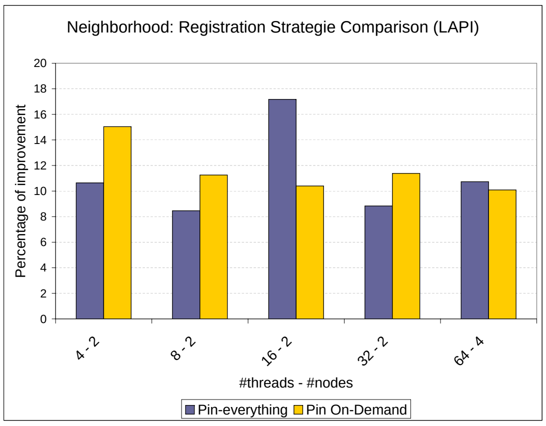
In the *Update Stressmark*, (Figure 6.16b) the Pin On-Demand strategy shows degra-



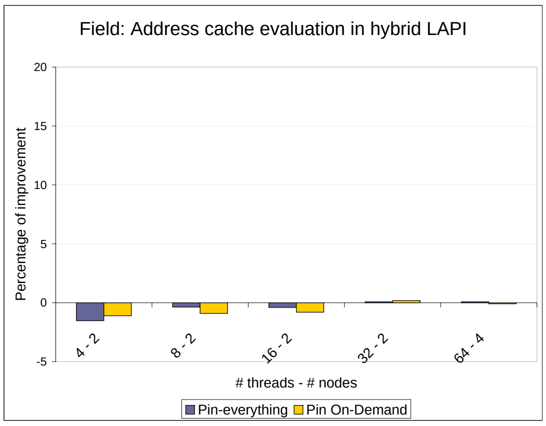
(a) Pointer



(b) Update



(c) Neighborhood



(d) Field

Figure 6.16 Registration Strategy Comparison on LAPI using DIS Stressmark Suite. First column represents the Pin everything strategy, while second column is the Pin on-demand. The vertical axes show performance improvement when using the remote address cache: $\frac{100(Z-W)}{Z}$, where `Z` and `W` are the regular and address-cache-enabled runtimes respectively. The horizontal axes of the graphs show the threads and nodes used.

dation or very poor performance improvement. There are a couple of reasons: (i) We do not see performance improvement caused by two threads per node, because only thread 0 initiates communication and (ii) we do not see improvement caused by the overlap because the other threads are constantly draining the network. Moreover PUT does not use the RDMA for short messages and the little benefit shown in the Pin-everything strategy disappears due to the overhead of the On-Demand strategy. For this kind of applications, a Pin-everything strategy is the best option despite the fact that the `address cache` will not be functional depending on the amount of shared memory required by the application.

The *Neighborhood Stressmark*, (Figure 6.16c) performs better with the Pin On-Demand strategy due to the chunk-based registration (15% improvement). The difference in benefit is due to the number of remote accesses that varies depending on the threads per node configuration: the 16 threads - 2 nodes run generates more remote accesses, increasing the benefit of cache usage. Chunk-based registration permits a better distribution of registered data and reduces the registration costs for these applications showing remote access locality.

The *Field Stressmark* (Figure 6.16d) performs as expected, due to the small quantity of remote accesses.

6.5 Conclusions and future work

In this chapter, we have shown how the IBM XLUPC compiler and runtime system provide a scalable design through the use of the Shared Variable Directory (SVD). We have addressed the potential performance problem encountered by short remote memory accesses that need to be dereferenced in the SVD. We have presented a mechanism to cache remote addresses that reduces the SVD access overhead and allows the exploitation of RDMA operations in the network hardware for very short messages, improving latency and scalability.

We have evaluated our proposed optimization on two different platforms: the MareNostrum supercomputer and a Power5 cluster of SMPs, using microbenchmarks and four benchmarks of the DIS Stressmark Suite. And the proposed optimization has been implemented using two different memory registration strategies: the *pin-everything*, which gives as a close idea of the maximum expected benefits if all memory can be registered at once. And a *chunk-based on-demand registration* strategy that deals with network transports that limit the amount of contiguous memory registered in a single call, by using chunks, and overcomes the limitation on the total amount of memory pinned by allowing lazy deregistration.

Our results demonstrate an average reduction in execution time of up to 40% and 30% respectively on the two architectures. The overhead of unsuccessful attempts to cache remote addresses is relatively small, typically 1.5% and never worse than 2%.

6.5.1 MPI

We have used UPC as a case study for our optimization, although MPI-2 specifies one-sided communication and the same solution could apply, once more MPI shifters the responsibility to the programmer that is required to exchange the remote address explicitly using collective communication. MPI specifies RMA operations such as MPI_Get, MPI_Put and MPI_Accumulate plus calls for synchronization. Each MPI process defines a memory area, *window*, thorough a collective call where all processes exchange their respective window base address. Every RMA operation specifies the *window*, length, datatype and displacement at target process. RMA communication occurs during an *access epoch* defined between synchronization calls, and if *active communication* is performed remote data must be accessed during an *exposure epoch* [127].

Our address cache could be applied to MPI one-sided communication. Our pin-everything version is equivalent to the implementation proposed by Jian et al. [109] and it suffers from the same limitations: they ignore limits in the total amount of pinned contiguous memory at a single call and may incur into registratio/deregistration costs. The *chunk-based on-demand registration* strategy could be implemented and it will certainly be useful when the *exposed window* does not fit in the registrable memory space. The *exposed window* directly maps to the UPC shared data, the `address cache` will be simplified since one exposure window is effective at any given moment, the cache would be tagged by *chunk identifier* and *remote process*. Instead of the SVD in the UPC Runtime, MPI maintains internally a list of the physical remote addresses exchanged in the collective call where every process initializes its window. The value of the `address cache` in MPI, would be, more than the remote address itself which can be calculated at origin, the guarantee that a cached remote address is registered at the target.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we have presented three complementary solutions toward the optimization of programing models for massively parallel computers. Two different programming models have been selected as our case studies and contributions have been analyzed on both models.

First, we have analyzed the problem of data and control-data reception and we have presented and evaluated two strategies to deal with data message reception and a memory management algorithm that guarantees control-data reception without any extra overhead. All together they improve the memory management of the communication system and makes the system scalable to potentially any number of processes.

Secondly, we have proposed a language extension that allows for a better control of data layout and data locality can be easily exploited. Thus, it reduces the amount of communication resulting in more scalable applications and the overall performance is improved.

Finally, we have presented a cache of remote addresses that reduces control-data communication and allows one-sided primitives to exploit RDMA native calls provided by some modern networks. As a result latency is substantially improved and true overlap between computation and communication is achieved. The overall performance is improved while the scalability is maintained.

Next we summarize in more detail the work performed in this thesis.

7.1.1 Improving memory management in the communication system

The first contribution of this thesis improves the memory management in the communication system to guarantee message reception (for both data and control-data) regardless of the number of processes and the available memory resources.

Data message reception

In the direction of guaranteeing data message reception, two solutions have been proposed, one of them is based on prediction. The main contributions are: (i) first the characterization of the temporal communication patterns of MPI applications. We have investigated the predictability of MPI communication streams at both the physical and logical levels. We have concluded that communication streams are very predictable at the logical level, with an accuracy of over 90% (even in application with many collective operations). We have also seen that prediction at the physical level is not as accurate in some cases, due to the randomness of the environment.

Second, we have proposed two prediction schemes, one is based on periodicity detection and another one based on a graph. Both achieve high prediction rates.

Third, we have proposed a memory management mechanism based on prediction that solved the scalability problems by taking advantage of the high-predictability observed. The mechanism has been implemented and evaluated resulting in a more reliable implementation of the library with a little overhead, but not good enough due to the cost of memory copies (we envisioned the problem and found a solution to alleviate the overhead of these copies, but it needs kernel support to be implemented).

And finally, we have compared the proposed mechanism based on prediction against a control flow solution based on the `rendezvous` protocol, also interesting due to its simplicity. And we have also shown how in architectures with large number of nodes and high-performance networks, like MareNostrum supercomputer or BG/L, this simpler solution is totally feasible since it guarantees control flow and user's data message reception with a reasonable overhead.

The added value of our prediction mechanism is that it eliminates the handshake between the two processes involved in the communication and reduces the amount of control-data which may be very beneficial in high-latency networks.

The trend followed by current supercomputers goes for thousands of ordinary CPU connected through increasingly faster high-performance networks which makes the rendezvous protocol a good choice to guarantee control flow, however the situation would be different in architectures with more CPU power and lower network performance. Moreover, as we have proven there are applications where this protocol is not suitable and our proposal would benefit.

This study has been applied to the MPI messaging system, improving its robustness and scalability. The UPC language messaging system does not show the same problem because it is based on one-sided communication. Message matching is not necessary in one-sided communication since the target process does not need to be actively involved in the message transfer. Thus, a message can not arrive *unexpectedly* since the final destination address of the message is already known at the origin process and temporary space to receive data is not required.

Control-data reception

In the direction of guaranteeing control-data reception, the main contributions are: First, we have shown the necessity of controlling the correct arrival of any message, containing either data or control-data, for current trend MPP machines like BG/L, that may have short memory available.

Second, we have proposed a memory management protocol, it has been implemented and evaluated over BG/L, overcoming the limitation of short-memory MPPs in order to gain scalability. The idea behind our solution is to extend the MPI queues (used to manage message reception) from the receiver to the sender side. In that way, it allows MPI to support as many outstanding unexpected messages as many memory is available not only in the receiver node but also in all senders.

And finally, we have shown that this mechanism works, without any overhead under normal conditions (no memory problems) and within less than twice the execution time, for an application that would crash without the proposed memory management protocol.

Concerning control-data reception, both languages, MPI and UPC (and potentially any other communication system) if running on short memory machines (such as BG/L) show the same problem. Control-data needs to be received in order to manage message reception, regardless of the type of communication (one-sided or two-sided), and if the target process is short in memory, handling control-data message reception may be a problem. Although our solution has been implemented in MPI with very good results, the same solution applies to UPC and may be implemented into UPC messaging system to guarantee control-data reception.

7.1.2 Extending the language to reduce communication

The second contribution of this thesis is a language extension that allows a better exploitation of data locality and reduces inter-node communication. Thus, it improves overall application performance and scalability to large scale machines.

The work performed in this area has contributed to the following points: First, we have shown that shared memory programming for large scale distributed memory machines is not a myth. Scaling non-trivial shared-memory programs to hundreds of thousands of threads is possible with the right support from the compiler and from the run-time system. We have presented our XL UPC compiler infrastructure and the UPC Run-Time System and the features that contributed to high performance. And we have illustrated our work with three benchmarks, two of which we scaled to more than a hundred thousand processors on the BlueGene/L machine.

Second, we have presented a language extension for UPC shared arrays that provides fine control over data array layout. This extension allows the programmer to obtain better performance while simplifying the expression of computations, in particular matrix computations. An added benefit is the ability to integrate existing libraries written in C and Fortran, which require specific memory layouts. Shared memory accesses have been optimized by exploiting data locality and the overheads of translating every shared memory access to a runtime call have been reduced.

This work has contributed to improve the scalability of applications, reducing inter-node communication by allowing a better data distribution to exploit data locality. The contribution was crucial to win the HPC Challenge Productivity Award on the Supercomputing conference in 2005 and 2006, receiving the community endorsement for this work.

This work has been applied to the UPC language, because MPI handles the data distribution explicitly. A typical MPI program wanting to use the total available memory will organize processes such that every process allocates its piece of the array and some additional information is required to logically reconstruct the whole array. This practice results in a source code by far more complex than in UPC. On the other hand, the closest somehow equivalent to tiles in MPI are the *MPI Datatypes*. MPI datatypes are very flexible allowing the programmer to exploit data locality and minimize communication. The datatype is a new type that allows message coalescing, it increases average message size to hide the overheads of two-sided communication.

7.1.3 Caching the final address to speed up communication

The third contribution of this thesis is a remote address cache that reduces control-data communication and allows the use of native RDMA calls, maintaining scalability. As a result, applications reduce their communication latencies and improve the overlap between communication and computation.

In this work, we have shown how the IBM XLUPC compiler and runtime system provide a scalable design through the use of the Shared Variable Directory (SVD). We have addressed the potential performance problem encountered by short remote memory accesses that need to be dereferenced in the SVD. We have presented a mechanism to cache remote addresses that reduces the SVD access overhead and allows the exploitation of RDMA operations in the network hardware for very short messages, improving latency and scalability.

We have evaluated our proposed optimization on two different platforms: the MareNostrum supercomputer and a Power5 cluster of SMPs, using microbenchmarks and four benchmarks of the DIS Stressmark Suite. The proposed optimization has been implemented using two different memory registration strategies: the *pin-everything*, which gives as a close idea of the maximum expected benefits if all memory can be registered at once. The other one is a *chunk-based on-demand registration* strategy that deals with network transports that limit the amount of contiguous memory can be registered by a single call, by using chunks, and overcomes the limitation on the total amount of memory pinned by allowing lazy deregistration.

Our results demonstrate an average reduction in execution time of up to 40% and 30% respectively on the two architectures. The overhead of unsuccessful attempts to cache remote addresses is relatively small, typically 1.5% and never worse than 2%.

We have used UPC as a case study for this optimization. Although MPI-2 specifies one-sided communication and the same solution could be implemented, once more MPI shifts the responsibility to the programmer that is required to exchange the remote address explicitly using collective communication. Jian et al. [109], propose an implementation on top of Infiniband native RDMA operations where memory registration is done in the initialization period when the window is created. This implementation, similarly to our pin-everything strategy, is limited by the total amount of registrable contiguous memory at a single call, and may incur high registration and deregistration costs. Our address cache using the *chunk-based on-demand registration* strategy could be implemented and it will certainly be useful. However, in this particular case, the value of the `address cache` in MPI will not be the remote address itself, which can be calculated at origin, but the guarantee that a cached remote address is registered at the target.

Moreover, all the work done to reduce communication, which includes the prediction mechanism (chapter 3) the multi-dimensional blocking language extension (chapter 5) and the cache of remote addresses (chapter 6), improves performance and scalability, which is the main goal of our thesis, but also they use the network efficiently which contributes to a power reduction. Power efficiency will be a challenge for next generation of supercomputers [171] and the software stack will be a key point.

7.1.4 MPI and UPC

In this thesis we have pointed to the software communication system as a key factor for future trend machines scalability. We have proposed three optimizations targeting to reduce the overall application time spend in communications.

We have taken two different parallel models for our case studies: MPI and UPC. Despite of its similarities we have found the communication system requirements to be quite different.

In general terms, from the communication system point of view, we have observed two main differences that are inherent to the language:

- **Two-sided versus one-sided:** MPI offers a two-sided communication model, a process sends a message to another process that has to explicitly receive it (`send, receive`), and both processes are involved in the communication (and synchronized). In contrast, UPC is based on one-sided communication and only the process that initiates the communication needs to be involved (`get, put`). Two-sided communication implies synchronization, a point to point communication synchronizes the two processes involved in the data transfer (the *sender* and the *receiver*), whereas one sided communication does not imply synchronization and only the *origin* process is required to be involved on the transfer allowing a better overlap between communication and computation.

- **MPI message matching:** This is related to the two-sided paradigm. In MPI any arriving message needs to match a posted receiver. Queues need to be maintained and temporary storage may need to be provided in order to perform the matching. On the contrary, in UPC, the *target* process is not aware of incoming message arrival and it is required to explicitly make progress from time to time by means of calling an expensive global synchronization operation such as a *barrier*. Moreover, depending on the implementation, communication may not actually take place until the synchronization point is reached causing undesirable overheads and wasting the potential overlap.

  To sum up, synchronization on every communication together with the *message matching* constitute the limiting factors for performance and scalability in MPI. Whereas, UPC is required to force expensive global synchronization to inquire for message arrival, being its limiting factor.

- **Communication pattern** MPI applications tend to be bandwidth-bound, they rely on a “few" big messages. As opposed to UPC where small message sizes dominate the communication pattern, specially in unoptimized codes that tend to be latency-bounded.

Referring to our contributions: Both models need a better memory management to deal with control-data reception in case of memory shortage, although only MPI, since it is based on two-sided communication, needs to deal with unexpected message reception. Also, both languages can speed up data transfer by reducing the control-data exchanged and exploiting the RDMA network capabilities.

Two out of three of our contributions can be applied (or have already been applied) to both languages. But the second contribution aims to reduce communication by exploiting data locality and since it is highly dependent on the language it only applies to UPC.

Reflections about productivity

Finally, we outline a few thoughts about productivity. During the development of this thesis, HPC specialists have turned they attention to programmability. It is said that 70% of parallel application programmers are not computer science experts and the assumption that the programmer is the best to tell how parallelism can be exploited for an application is no longer valid.

MPI forum has been dormant for decades, to wake up recently to clarify some concepts in the MPI-2 specification and to discuss a proposal for MPI-3. Some of the hot topics under discussion are: fault tolerance, rma operations and a new family of data-centric non blocking collective communication. Whereas, PGAS languages such as UPC, X10 or Titanium are becoming more popular.

In this thesis we have proven the scalability and performance of our implementation of UPC language, outperforming MPI in some cases. We have also shown pieces of our simple UPC codes, as a prove of programmability. There is a lot of work that needs to be done, nevertheless in terms of productivity, UPC is a very promising language.

MPI is still the dominant language for HPC, and it will probably last since an MPI application may take several years to be programmed efficiently. However, PGAS languages are very promising and MPI will need to make some efforts to improve its productivity.

As a conclusion, as far as productivity is concerned, we state that coexistence of both languages will be highly desirable considering future trend machines. It would allow parallel programmers to recycle their parallel applications and scale them up to a million nodes. We further discuss this issue in next section 7.2.

7.2 Future Work

We divide this section in two parts: (i) future work addressing the shortcomings or unpolished details of the work presented in this thesis and (ii) future research lines derived from the work done in this thesis.

7.2.1 Shortcomings

A number of issues still remain to be resolved and could be addressed for each one of our contributions. We have defined four directions that we will address in the near future.

- **Prediction mechanism**. Considering our work analyzing the communication patterns, several ideas remained to be implemented: one of them is the use of prediction to predict the final buffer address where the message should be stored and use this information to provide a 0-copy mechanism. Another one is to predict the best place within the library to make progress and receive messages.

- **Multiblocked arrays**. We believe that adding processor tiling will increase the programmer's ability to write codes that scale to large numbers of processors. Defining a set of collectives that are optimized for the UPC programming model will also address several scalability issues, such as the ones occurring in the LU Factorization and the High Performance Linpack kernel [94]. This will be included in one of the new research lines presented below.
- **Address Cache**. We plan to extend the range of our scalability experiments to confirm that the performance benefits we measured on relatively small machine configurations continue into the range of tens of thousands of processors. We also plan to perform a deeper analysis enlarging our test bed, we will measure the benefits of the address cache on applications as opposed to benchmarks.
- **Benchmarks**. Another problem which we will continue to address is the lack of benchmarks and algorithms written in UPC that can scale to the size of a massively parallel computer such as the BlueGene/L computer.

7.2.2 Future research directions

In addition to the above mentioned shortcomings, we have defined three main research lines following the work presented in this thesis.

Data centric parallel programming model

Motivation: Today's high performance codes (mostly MPI) will simply not run on the next generation (> 10PF) of HPC machines. Combined solutions such as OpenMP plus MPI are limited in applicability and impact productivity and portability. Collective communication would provide the efficiency but there are shortcomings in the way current languages express such patterns.

Goal: We are looking for a programming model/library/language that enables performance portability across a range of mixed-model (distributed and shared memory) capability machines, taking into account the issues of network latency, bandwidth, on-chip memory limitations and cache sizes.

By far the most scalable component of MPI is its system of collective communication primitives. By data-centric we mean that the notion of collective communication is expressed as an operation on a data structure rather than on data movement among all the threads. The programmer benefits from high productivity due to the data container approach, and the implementation of communication primitives can be done on a per-architecture basis.

Details: This effort would involve:

- Study the application characteristics and come up with the appropriate syntax for data containers and operations on them.
- Compiler optimizations to find and express parallel data operations.
- Runtime optimizations (implement machine-specific optimized collective primitives). This part links to our pending issues in HPL: When implementing High Performance Linpack [94] in the UPC language several weaknesses were identified 5.3.4. Defining a set of collectives that are optimized for the UPC programming model will address several scalability issues, such as the ones occurring in the LU Factorization in the HPL Kernel. A lot of work has been done in collectives, but the fresh angle is the non-blocking collectives working with *teams* (aka. low overhead communicators) or data-centric collectives.
- More about collectives. Also, we could explore collectives optimization by leveraging the address cache (third contribution) in optimized collective implementations. For instance, for very short broadcasts it may make more sense programming RDMA transfers than building a whole tree of processes.

We will start looking at dense linear algebra (HPL) and a subset of the NAS benchmarks (NAS CG at first). We will use the UPC (soon to be called PGAS) runtime on as many network architectures as we can (Myrinet, IBM HPS, Infiniband and BlueGene transports for MareNostrum, Power5 clusters and Blue Gene/W).

Related Work: Although this is a new topic, preliminary research has been done: The MPI Forum is discussing about data-centric collectives for the upcoming MPI-3 [128, 129]. Published research papers include a collective communication extension for the UPC language [145], that improved the kernels of Cholesky and matrix multiply benchmarks. Also, the work performed by C von Praun et al. in [192] that models optimistic concurrency using quantitative dependence analysis.

Language designs with data centric approaches include ZPL [43], X10 [198] and HPF [81].

MPI and UPC Coexistence

Motivation: MPI (by itself) is not scalable to a million nodes due to inherent limitations imposed my memory sizes of current large scale machines. Current MPPs tend to be clusters of SMPs. MPI plus OpenMP is the current approach for these hybrid architectures, but it has its limitations. Parallel applications show a pattern with a small set of loosely coupled parallel tasks that may not fit into an SMP. Currently OpenMP has not reported very good results when running over distributed memory on top of a DSM system, and here is where PGAS languages (like UPC, CAF or X10) step in.

However, most current parallel applications and parallel libraries are written in MPI. Coexistence of newer programming paradigms with MPI would allow programmers to scale up their applications without re-programming them from scratch.

Goal: We seek MPI-UPC coexistence in both directions: using MPI inside UPC programs will allow the use of parallel libraries such as PETSc [15], SCALAPACK [48] or PESSL. Running UPC programs inside MPI applications would allow scaling up current MPI applications and writing libraries in UPC to support future scientific applications.

Details: This will involve work like defining the semantics of the interaction, retargeting the runtimes and exploiting it in the applications.

The starting point is adapting the Runtime to coexist both ways (MPI in or on top of UPC). We are in the process of identifying an scenario to work with.

Related work: The most common mixed approach targeting hybrid architectures is MPI plus OpenMP. While this approach may provide significant benefit in some situations, like: if the MPI code suffers from poor scaling due to a load imbalance or to fine grain problem size, or if the memory is a limitation due to the use of replicated data strategy [176, 54]. However, this mixed programming model is limited in scalability. MPI is not scalable to a million nodes and OpenMP does not show good performance in distributed environments, we believe that an MPI plus UPC combination could solve these problems.

Mark Snir and Bill Gropp are involved in this MPI plus UPC initiative, as part of PERCS DARPA HPCS project: High Productivity Computer Systems [156].

Balance of dynamic data structures

Motivation: Parallel applications making use of non-traditional HPC data structures, such as graphs, trees or lists tend to result in a highly unbalanced parallelism. Moreover, parallel languages do not provide the appropriate syntax to express this kind of shared data structures that are usually expressed in terms of arrays resulting in cumbersome codes and making them difficult to re-balance.

Goal: Study a convenient syntax to express this kind of shared data and provide a way to balance applications by re-distributing dynamic data if it grows in a way that moves away of a uniform/balanced distribution.

Details: It is related to the data-centric approach and independent of the language underneath. Most of the work will be in the runtime, and parallel applications. The compiler will also be involved to recognize the syntax.

The starting point is to look at applications and find a good scenario. Graph applications [53] and irregular dynamic programs [161] will be considered.

Bibliography

[1] A. Afsahi and N. Dimopoulos. Hiding communication latency in reconfigurable message -passing environments. In *Proceedings of the of IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 55–60, April 1999. 2.2.1

[2] A. Afsahi and N. J. Dimopoulos. Communications latency hiding techniques for a reconfigurable optical interconnect: Benchmark studies. In *PARA*, pages 1–6, 1998. 2.2.1

[3] A. Afsahi and N. J. Dimopoulos. Efficient communication using message prediction for cluster multiprocessors. In *CANPC '00: Proceedings of the 4th International Workshop on Network-Based Parallel Computing*, pages 162–178, London, UK, 2000. Springer-Verlag. 2.2.1

[4] A. Afsahi and N. J. Dimopoulos. Efficient communication using message prediction for cluster multiprocessors. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 162–178, 2000. Its a predictability study to reduce memory copies inside the library. 2.4.1, 3.3.1, 3.3.1

[5] S. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and J. S. Tobin-Hochstadt. *The Fortress Language Specification*, April 2005. `http://research.sun.com/projects/plrg/fortress0618.pdf`. 2.1

[6] G. Almasi, C. Archer, J. G. Castanos, M. Gupta, X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. Mpi on blue gene/l: Designing an efficient general purpose messaging solution for a large cellular system. In *Euro PVM/MPI workshop.*, 2003. 2.1.1

[7] G. Almasi et al. Design and implementation of message-passing service for the BlueGene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, 2005. 3.4.2, 5.2.2, 5.2.2, 5.3.2, 6.2

[8] G. Almási, L. D. Rose, B. B. Fraguela, J. Moreira, and D. A. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2958 of *Lecture Notes in Computer Science*, pages 162–176, College Station, TX, October 2003. Springer. 5.2.2

[9] A.Petitet, R.C. Whaley, J.Dongarra, A.Cleary. Hpl- a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`. 5.3.4

[10] ASC Program. Asci red home page. `http://www.sandia.gov/ASCI/Red/`. 2.2.2, 4.1

[11] ASC Program. ASCI White Home Page. `http://www.llnl.gov/asci/platforms/white/`. 2.2.2

[12] Atlantic Aerospace Division, Titan Systems Corporation. Dis stressmark suite: Specifications for the stressmarks of the dis benchmark project. version 1.0. *MDA972-97-C-0025*, 1:10, 2000. 6.4.4

[13] E. Ayguade, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In *Languages and Compilers for Parallel Computing*, pages 61–75, 1994. 2.3.2

[14] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 3.2.1

[15] Barry, Satish, Matt, Hong, Victor, Dmitry, Lisandro. PETSc: Portable, Extensible Toolkit for Scientific Computation. `http://www-unix.mcs.anl.gov/petsc/petsc-as/publications/index.html`. 2.1, 7.2.2

[16] C. Barton, C. Cascaval, G. Almasi, R. Garg, J. N. Amaral, and M. Farreras. Multidimensional Blocking Factors in UPC. *LCPC2007: International Workshop on Languages and Compilers for Parallel Computing*, 2007. 1.3.2, 5.3.3, 6.2.1

[17] C. Barton, C. Cascaval, G. Almasi, Y. Zheng, M. Farreras, and J. N. Amaral. Shared memory programming for large scale machines. *PLDI 2006: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006. 1.3.2, 5.1, 5.2.3, 5.2.4, 6.1

[18] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, pages 198–198, Los Alamitos, CA, Apr. 22–26 2003. IEEE Computer Society. 2.4.2, 6.3.3

[19] C. Bell, D. Bonachea, R. Nishtala, and K. A. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)*, Rhodes Island, Greece, Apr. 2006. IEEE Computer Society. 2.3.1

[20] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the cray x1. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, New York, NY, USA, 2004. ACM Press. 2.3.1

[21] C. Bell and R. Nishtala. Firehose: An algorithm for distributed page registration on clusters of smps, 2004. CS262B Final Project. 2.4.2

[22] Berkeley. UPC Project Home Page, 2005. `http://upc.lbl.gov/`. 2.4.2

[23] G. Bhanot, D. Chen, A. Gara, and P. Vranas. The BlueGene/L Supercomputer. *ArXiv High Energy Physics - Lattice e-prints*, Dec. 2002. 4.1

[24] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B. B. Fraguela, M. J. Garzarán, D. A. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPOPP*, pages 48–57, 2006. 2.3.2, 5.3.1

[25] Basic linear algebra subprograms: Blas library. `http://www.netlib.org/blas/`. 5.3.4

[26] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995. 2.2.2

[27] D. Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, U.C. Berkeley, November 2002. 5.2.2, 6.2

[28] R. Brightwell, R. Riesen, B. Lawry, and A. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *16th International Parallel and Distributed Processing Symposium (IPDPS '02 (IPPS and SPDP))*, page 164, Washington - Brussels - Tokyo, Apr. 2002. IEEE. 2.2.2

[29] R. Brightwell, R. Riesen, and A. B. Maccabe. Design, implementation, and performance of MPI on Portals 3.0. *The International Journal of High Performance Computing Applications*, 17(1):7–20, Spring 2003. 2.2.2, 4.2.2

[30] R. Brightwell, R. Riesen, K. D. Underwood, T. Hudson, P. G. Bridges, and A. B. Maccabe. A performance comparison of linux and a lightweight kernel. In *CLUSTER*, pages 251–258. IEEE Computer Society, 2003. 4.1

[31] R. Brightwell and K. D. Underwood. An analysis of nic resource usage for offloading mpi. In *IPDPS '04: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004. 2.4.1

[32] BSC: Barcelona Supercomputing Center. Marenostrum system architecture. http://www.bsc.es/plantillaA.php, 2004. `http://www.bsc.es/plantillaA.php?cat_id=200`. 3.4.3, 6.1, 6.3.2, 6.4.1, A.2.2

[33] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 2.1, 5.2.2

[34] C. Cascaval, C.Barton, G.Almási, Y.Zheng, M.Farreras, P.Luk and R.Mak. The UPC/BlueGene Class II Submission to the HPC Challenge Award Competition, 2005. 1.3.2, 5.4

[35] C. Cascaval, G. Almási, C. Barton, E. Tiotto, G. Dózsa, M. Farreras, P. Luk and T. Spelce. HPC Challenge 2006 Awards Competition: xlUPC on BlueGene/L, 2006. 1.3.2

[36] D. Callahan, B. L. Chamberlain, and H. P. Zima. CHAPEL: The Cascade High Productivity Language. *hips*, 00:52–60, 2004. 2.1

[37] R. Canonico, R. Cristaldi, and G. Iannello. A scalable flow control algorithm for the fast messages communication library. In *CANPC '99: Proceedings of the Third International Workshop on Network-Based Parallel Computing*, pages 77–90, London, UK, 1999. Springer-Verlag. 2.2.2, 4.2.2

[38] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber. Fast address translation techniques for distributed shared memory compilers. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2005. 2.4.2

[39] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999. 5.2

[40] 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, George Washington University, 1999. 5.1

[41] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 1996. ACM Press. It performs a compiler analysis to reduce communication. 2.3.2

[42] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007. 2.1

[43] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000. 2.1, 2.3.2, 7.2.2

[44] W. Chen. Building a source-to-source upc-to-c translator. In *Masters Report*, 2005. 2.3.1

[45] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society. 2.3.2

[46] W.-Y. Chen, C. Iancu, and K. A. Yelick. Communication optimizations for fine-grained UPC applications. In *IEEE PACT*, pages 267–278. IEEE Computer Society, 2005. 2.3.1

[47] CHIMP/MOI Project. CHIMP. `http://www.epcc.ed.ac.uk/epcc-projects/CHIMP`. 2.1.1, 3.1

[48] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical report, Knoxville, Knoxville, TN 37996, USA, 1995. 2.1, 5.3.4, 7.2.2

[49] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, and D. Walker. A proposal for a set of parallel basic linear algebra subprograms. Technical report, University of Tennessee, Knoxville, TN, USA, 1995. 2.1

[50] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, Fall 1996. 5.3.4, 5.3.4

[51] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study. citeseer.ist.psu.edu/693364.html. 2.1

[52] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, and Y. Yao. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM Press. 2.3.1

[53] G. Cong and D. A. Bader. Techniques for designing efficient parallel graph algorithms for smps and multicore processors. In *ISPA*, pages 137–147, 2007. 7.2.2

[54] J. Corbalán, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 195–202, 2004. 7.2.2

[55] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running openmp applications efficiently on an everything-shared sdsm. *J. Parallel Distrib. Comput.*, 66(5):647–658, 2006. 2.4.2

[56] Cray Home Page. Cray. `http://www.cray.com/`. 3.1

[57] Cray. Red Storm. Catamount lightweight kernel. `http://www.sandia.gov/asc/redstorm.html`. 4.1

[58] Cray UPC home page. `http://docs.cray.com/books`. 2.1.2

[59] Unified parallel c (upc) lecture. `http://upc.lbl.gov`, 2008. 5.2.4

[60] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. *SIGMOD Rec.*, 22(2):257–266, 1993. 3.3.1

[61] D. Greenberg, R. Brightwell et al. A system software architecture for hight-end computing. In *Proceedings of Supercomputing 1997*, 1997. 4.1

[62] D. H. Bailey, E. Barszcz, et al. The NAS Parallel Benchmarks. In *The International Journal of Supercomputer Applications*, Fall 1991. 3.3.2, 4.4.2, 5.2.4

[63] D. Womble and D. Greenberg and S. Wheat and R. Riesen. LU Factorization and the LINPACK Benchmark on the Intel Paragon. Technical report, Sandia National Laboratories Tech Report SAND94–0425, 1994. 5.3.4

[64] W. Y. Darius. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2. In *International Conference on Parallel Processing, (ICPP 2003)*, 2003. 2.4.1

[65] Department of Computer Science and Engineering The Ohio State University. High Performance MPI on IBA MPI over InfiniBand Project. `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/`. 2.2.2, 2.4.2, 4.2.2

[66] J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, 1988. Springer-Verlag. 5.3.4

[67] J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, 2003. 5.3.4

[68] J. Dongarra, H. Meuer, H. Simon, and E. Strohmaier. High performance computing today. In P. T. Cummings, P. R. Westmoreland, and B. Carnahan, editors, *Foundations of molecular modeling and simulation: proceedings of the First International Conference on Molecular Modeling and Simulation, Keystone, Colorado, July 23–28, 2000*, volume 97(325), pages ??–??, New York, NY, 2000. American Institute of Chemical Engineers. 5.3.4

[69] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988. 5.3.1, 5.3.4

[70] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003. 5.3.4

[71] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. Stewart. *LINPACK Users' Guide*. SIAM Pub, Philadelphia, 1979. 5.3.4

[72] J. J. Dongarra, R. A. Van de Geijn, and D. W. Walker. Scalability issues affecting the design of a dense linear algebra library. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994. 5.3.4

[73] J. Duell. Memory management in the upc runtime (version 1.1). `http://upc.lbl.gov/docs/system/runtime_notes/memory_mgmt.html`. 2.3.1

[74] Earth Simulator Center/JAMSTEC. Earth simulator home page. `http://www.es.jamstec.go.jp/`. 2.2.2

[75] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 1.1.2, 2.3.1, 5.1

[76] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 5.2.4

[77] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specifications*, v1.1.1 edition, October 2003. 5.1, 5.2

[78] ESSL User Guide. `http://www-03.ibm.com/systems/p/software/essl.html`. 5.3.4, 5.3.4

[79] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes. Scalable RDMA performance in PGAS languages. In submission process. 1.3.3

[80] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling mpi to short-memory mpps such as bg/l. *ICS06: Proceedings of the 2006 International Conference on Supercomputing*, 2006. 1.3.1

[81] H. P. F. Forum. Hpf: High performance fortran language specification v1.0. Technical Report CRPCTR92225, Huston. Texas, 1993. 2.1, 7.2.2

[82] F. Freitag, J. Caubet, M. Farreras, T. Cortes, and J. Labarta. Exploring the predictability of mpi messages. *IPDPS '03: Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003. 1.3.1

[83] F. Freitag, J. Corbalan, and J. Labarta. A Dynamic Periodicity Detector: Application to Speedup Computation. In *IPDPS '01: Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 2–2, 2001. 3.2.2, 3.2.2, 3.3.1

[84] F. Freitag, M. Farreras, T. Cortes, and J. Labarta. Predicting mpi buffer addresses. *ICCS2004: The International Conference on Computer Sciences 2004*, 2004. 1.3.1

[85] G. Almasi, R. Bellofatto et al. An Overview of the Blue Gene/L System Software Organization. In *Proceedings of Euro-Par*, 2003. 4.2.1

[86] A. Gara et al. Overview of the Bluegene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195, 2005. 3.4.3, 5.2, A.2.1

[87] GCC UPC home page. `http://www.intrepid.com/upc/`. 2.1.2

[88] George Almasi, C Archer, J Castaños et al. Implementing mpi on the bluegene/l supercomputer. In *Proceedings of Euro-Par 2004*. 4.2.2

[89] GM: A message-passing system for Myrinet networks. GM 2.1.24. `http://www.myri.com/scs/GM-2/doc/html/`. 3.4.3, 6.2, 6.3.2

[90] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An hpf compiler for the ibm sp2. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 71, New York, NY, USA, 1995. ACM Press. Compilation optimizations to reduce communication, aggregation, collectives, exploite locality. 2.3.2

[91] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996. 2.3.2

[92] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences using OpenMP based on compiler directed software DSM on a PC cluster. In *WOMPAT2002.*, 2002. 2.4.2

[93] Hpc challenge results: Optimized version. `http://icl.cs.utk.edu/hpcc/`. 5.2.4

[94] HPL Algorithm description. `http://www.netlib.org/benchmark/hpl/algorithm.html`. 5.4, 7.2.1, 7.2.2

[95] HP/Compaq UPC. http://h30097.www3.hp.com/upc/index.htm. 2.1.2

[96] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM system based on a new cache coherence protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463–472, 1999. 2.4.2

[97] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000. 2.4.2

[98] Hunter W.G., Hunter J.S., Hunter W.G. *Statistics for Experimenters: Design, Innovation, and Discovery*. Box,G. E, Wiley, 2nd Edition, 2005. 6.4

[99] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM Press. 2.3.1

[100] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991. 4.2.2

[101] C. Iancu, P. Husbands, and P. Hargrove. Hunting the overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society. 2.4.1

[102] IBM. RSCT LAPI Programming Guide, 1990. http://publib.boulder.ibm.com/epubs/pdf/bl5lpg04.pdf. 6.2.2

[103] IBM MPI Parallel Environment for AIX. IBM MPI Programming Guide, Version 2 Release 3. IBM 1997. http://www.nersc.gov/vendordocs/ibm/pe/am106mst02.html. 2.2.2, 3.1

[104] IBM research. IBM Research Blue Gene Project Page. http://www.research.ibm.com/bluegene/. 2.2.2

[105] Y. Iwamoto, K. Suga, K. Ootsu, T. Yokota, and T. Baba. Receiving message prediction method. *Parallel Computing*, 29(11-12):1509–1538, December 2003. 2.2.1

[106] J. White III and S. Bova. Portability Issues Associated with Overlapping MPI Communication and Computation. In *Poster in Supercomputing conference SC98*, 1998. 2.4.1

[107] J. White III and S. Bova. Where's the Overlap? An Analysis of Popular MPI Implementations. In *MPIDC '99*, 1999. 2.4.1

[108] P. L. Jack Dongarra. Hpc challenge awards: Class 2 specification. `http://www.hpcchallenge.org`. 5.2, 5.2.4, 5.2.4, 5.3.4

[109] W. Jiang, J. Liu, H. Jin, D. Panda, W. Gropp, and R. Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *In Proc. of 4th IEEE/ACM Cluster Computing and the Grid, April 2004.*, 2004. 2.4.1, 6.5.1, 7.1.3

[110] D. P. Jiuxing Liu. Implementing efficient and scalable flow control schemes in mpi over infiniband. In *IPDPS '04: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004. 2.2.2

[111] Y.-S. Kee, J.-S. Kim, and S. Ha. Parade: An openmp programming environment for smp cluster systems. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2003. IEEE Computer Society. 2.4.2

[112] 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. 2.4.2

[113] J. Kim and D. J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *CANPC '98: Proceedings of the Second International Workshop on Network-Based Parallel Computing*, pages 202–216, London, UK, 1998. Springer-Verlag. 2.2.1, 3.3.2

[114] S. Kini, J. Liu, J. Wu, P. Wyckoff, and D. Panda. Fast and scalable barrier using rdma and multicast mechanisms for infiniband-based clusters. In *Euro PVM/MPI Conference*, Sept 2003. 2.4.1

[115] U. Kremer. Automatic data layout for distributed memory machines. Technical Report TR96-261, Rice University. Houston, TX., 14, 1996. 2.3.2

[116] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for atm networks: credit update protocol, adaptive credit allocation and

statistical multiplexing. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 101–114, New York, NY, USA, 1994. ACM Press. 2.2.2, 4.2.2

[117] L. S. Blackford et al. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, page 15 (electronic), Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics. 2.1, 2.3.2, 5.3.4

[118] LAM/MPI home page. LAM. `http://www.lam-mpi.org`. 1.1.2, 2.1.1, 3.1

[119] Lawrence Livermore, Los Alamos, Sandia National Labs. Sweep3D ASCI benchmarks. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/`. 3.2.1

[120] Lawrence Livermore National Laboratory. Advanced simulation and computing: The fastest supercomputer in the world: Bluegene/l, November 2007. `https://asc.llnl.gov/computing_resources/bluegenel/`. 4.1

[121] J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Parallel and Distributed Processing Symposium (IPDPS 04)*, April 2004. 2.4.1

[122] J. Liu, J. Wu, S. Kini, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. Panda. Mpi over infiniband: Early experiences, 2003. 2.4.1

[123] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance rdma-based mpi implementation over infiniband. In *ICS*, page 10, 2003. 2.4.1

[124] A. B. Maccabe, W. Shu, J. Otto, and R. Riesen". Experience in offloading protocol processing to a programmable nic. In *ICCS'02: International Conference on Cluster Computing*, September 2002. 2.4.1

[125] A. Moody, J. Fernandez, F. Petrini, and D. Panda. Scalable NIC-based Reduction on Large-scale Clusters. In *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_reduce.pdf`. 2.4.1

[126] MP Lite A Lightweight message passing library. MP Lite HP. `http://www.scl.ameslab.gov/Projects/MP_Lite/`. 2.1.1

[127] MPI-2: Extension to the Message Passing Interface July 1997. Message passing interface forum. `http://www.mpi-forum.org/docs/docs.html`. 6.5.1

[128] MPI-3: Forum. Collective Communications Working Group Wiki page. `http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/CollectivesWikiPage`. 7.2.2

[129] MPI-3 Forum. MPI Plans: an alternative for all other collectives proposals? Collective communications working group wiki page. `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPIplans`. 7.2.2

[130] J. MPI: A message Passing Interface Standard. Message passing interface forum. 1.1.2, 4.1, 5.3.4

[131] MPI Forum. MPI-Forum: A Message Passing Interface Starndard. `http://www.mpi-forum.org`. 3.1, 4.3.4

[132] MPICH home page. MPICH. `http://www.unix.mcs.anl.gov/mpi/mpich`. 2.1.1, 3.1, 3.2.1, 3.2.1

[133] MPICH on CPlant. Mpich 1.2.0 over portals 3.1. `http://www.cs.sandia.gov/cplant/doc/mpich-portals3.html`. 2.1.1, 2.2.2

[134] MPICH-V MPI implementation for volatile resources. MPICH-V HP. `http://www.lri.fr/~{}bouteill/MPICH-V/`. 2.1.1

[135] MPI/ES for the Earth Simulator Home Page. MPI/ES Home Page. `http://www.ccrl-nece.de/~{}ritzdorf/mpies.shtml`. 2.2.2, 4.2.2

[136] MPItrace Users Guide: Instrumentation of Generic MPI applications. BSC: Barcelona Supercomputing Center. `http://www.cepba.upc.es/paraver/docs/MPItrace.pdf`. 6.4.6

[137] MTU. Michigan technological university, July 2003. `http://www.mtu.edu/`. 2.4.2

[138] MVICH home page MPI for Virtual Interface Architecture. MVICH. `http://old-www.nersc.gov/research/FTG/mvich/`. 2.1.1

[139] Myricom. Myrinet Express (MX): A hight performance, low-level,message-passing interface for Myrinet, July 2003. `http://www.myri.com/scs/MX/doc/mx.pdf`. 6.3.2

[140] Myrinet Software and Documentation Home Page. Myricom. Myricom: GM, MX, MPICH-GM, MPICH-MX and Sockets-GM. `http://www.myri.com/`. 2.1.1, 2.2.2, 2.4.2, 3.1, 3.4.3, 6.2.2

[141] N. R. Adiga et al. An Overview of the Blue Gene/L Supercomputer. In *High Performance Networking and Computing: Proceedings of Supercomputing 2002*, November 2002. 4.2.1, A.2.1

[142] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586:533–??, 1999. 2.4.2

[143] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, ???? 1996. 5.2.2

[144] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. 2002. 2.4.2, 6.3.2, 6.3.3

[145] R. Nishtala, G. Almasi, and C. Cascaval. Performance without pain = productivity: data layout and collective communication in upc. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 99–110, New York, NY, USA, 2008. ACM. 5.3.4, 7.2.2

[146] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998. 2.1, 5.1

[147] Object Management Group (OMG). Common object request broker architecture (corba), July 2003. `http://www.corba.org/`. 2.1

[148] OpenMP Specifications. Openmp application programing interface. v3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`. 2.1

[149] OpenMPI. Open mpi: Open source high performance computing. `http://www.open-mpi.org/`. 2.1.1, 2.4.2, 3.1

[150] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 55.1, 1995. 2.2.2, 4.2.2, 4.2.2

[151] Parallel Programming Laboratory. University of Illinois at Urbana-Champaign. *Charm ++ Programming Language Manual. Version 6.0 (release 1)*, April 1996. 2.1

[152] Paraver: Parallel Program Visualitzation and Analysis tool. BSC: Barcelona Supercomputing Center. `http://www.cepba.upc.es/paraver/`. 6.4.6

[153] J. R. Parker and J. R. Parker. *Algorithms for Image Processing and Computer Vision*. John Wiley & Sons, Inc., New York, NY, USA, 1996. 6.4.4

[154] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. ISBN 1558604286, second edition, 1998. 1.1

[155] K. Pedretti and R. Brightwell. A NIC-Offload Implementation of Portals for Quadrics QsNet. In *Proceedings of the Fifth LCI International Conference on Linux Clusters*, May 2004. 2.4.1

[156] DARPA High Productivity Computing Systems project. `http://www.highproductivity.org/`. 7.2.2

[157] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, / 2002. 6.3.3

[158] F. Petrini, J. Fernández, A. Moody, E. Frachtenberg, and D. K. Panda. NIC-based Reduction Algorithms for Large-scale Clusters. *International Journal of High Performance Computing and Networking (IJHPCN)*, 2005. Accepted for publication. 2.4.1

[159] Piotr Luszczek. Notes on LINPACK NxN Benchmark on Hewlett-Packard Systems. *HiPer*, 1:10, 2001. 5.3.4

[160] Piotr Luszczek and Jack J. Dongarra and David Koester and Rolf Rabenseifner and Bob Lucas and Jeremy Kepner and John McCalpin and David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. *SuperComputing SC*, 2005. 5.3.4

[161] O. G. Plata, R. Asenjo, E. Gutiérrez, F. Corbera, A. G. Navarro, and E. L. Zapata. On the parallelization of irregular and dynamic programs. *Parallel Computing*, 31(6):544–562, 2005. 7.2.2

[162] R. Ponnusamy, J. H. Saltz, A. N. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, 1995. 2.3.2

[163] R. Preissl, M. Schulz, D. Kranzlmueller, B. R. de Supinski, and D. J. Quinlan. Using mpi communication patterns to guide source code transformations. In *SC2007 Reno, NVPoster*, November 2007. 2.2.1

[164] R. Brightwell. Sandia National Laboratory. A lightweight kernel project. `http://www.cs.unm.edu/~{}fastos/03workshop/brightwell.pdf`. 4.1

[165] Reid Atlas Centre. The LINPACK Benchmark in Co-Array Fortran. citeseer.ist.psu.edu/332083.html. 5.3.4

[166] A. R. K. U. Ron Brightwell, Sue Goudy. Implications of application usage characteristics for collective communication offload. In *Internation Journal of High-Performance Computing and Networking*, 2005. 2.4.1

[167] R. R. Ron Brightwell and K. D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.*, 19(2):103–117, 2005. 2.4.1

[168] J. Savant and S. Seidel. Mupc: A run time system for unified parallel c. Technical Report CS-TR-02-03, Department of Computer Science, Michigan Technological University, 2002. 2.1.2

[169] SGI Message Passing Toolkit (MPT). Sgi. `http://www.sgi.com/products/software/mpt/`. 3.1

[170] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with lapi - a new high-performance communication library for the ibm rs/6000 sp. In *Proceedings of IPPS '98*, 1998. 5.2.2, 5.3.2, 6.2

[171] J. Shalf, S. Kamil, E. Strohmaier, and D. Bailey. Power Efficiency and the Top500, Nov 2007. `http://www.nersc.gov/projects/SDSA/reports/uploaded/Top500PowerNov14SC07.pdf`. 7.1.3

[172] S. Shao, A. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. *Parallel and Distributed Processing Symposium, International*, 0:65, 2006. 2.2.1

[173] S. Shao, Y. Zhang, A. Jones, and R. Melhem. Symbolic expression analysis for compiled communication. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. 2.2.1

[174] P. Shivam, P. Wyckoff, and D. Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 57–57, New York, NY, USA, 2001. ACM Press. 2.4.1

[175] P. Shivam, P. Wyckoff, and D. K. Panda. Can user-level protocols take advantage of multi-cpu nics? In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 88, Washington, DC, USA, 2002. IEEE Computer Society. 2.4.1

[176] L. Smith and M. Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000. 7.2.2

[177] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, I. Eugene W. Hodges, and P. Banerjee. Advanced compilation techniques in the paradigm compiler for distributed-memory multicomputers. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 424–433, New York, NY, USA, 1995. ACM Press. cyclic and block-cyclic arrays. 2.3.2

[178] Sun Microsystems. Simple Object Access Protocol (SOAP), May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`. 2.1

[179] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990. 1.1.2

[180] Supercomputing Technologies Group. MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, April 2005. `http://supertech.lcs.mit.edu/cilk`. 2.1

[181] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39, New York, NY, USA, 2006. ACM. 2.4.1, 6.3, 6.3.3

[182] R. B. T. Hudson. Network performance impact of a lightweight linux for cray xt3 compute nodes. 4.1

[183] A. B. e. a. T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *1995: Proceedings of the 15th ACM SIGPLAN symposium of Operating Systems Principles*, 1995. 4.2.2

[184] H. Tezuka, A. Hori, and Y. Ishikawa. Pm: a highperformance communication library for multi-user parallel environments. In *Usenix'97, 1996.*, 1996. 2.1, 2.4.2

[185] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *IPPS/SPDP*, pages 308–314, 1998. 2.4.2, 6.3.2, 6.3.3

[186] The Ohio State University. MVAPICH: MPI over InfiniBand and iWARP. `http://mvapich.cse.ohio-state.edu/overview/mvapich/`. 2.1.1, 2.4.2

[187] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host-assisted zero-copy remote memory access communication on infiniband. In *IPDPS*. IEEE Computer Society, 2004. 2.4.1, 6.3, 6.3.3

[188] Top 500 list. top500. `http://www.top500.org/`. 1.1.2, 3.1, 4.1, 5.3.4, A.2.1

[189] K. D. Underwood, S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. "a hardware acceleration unit for mpi queue processing". In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005. 2.4.1

[190] R. A. van de Geijn. Massively parallel linpack benchmark on the intel touchstone delta andipsc/860 systems (progress report). Technical report, University of Texas at Austin, Austin, TX, USA, 1991. 5.3.4

[191] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM Press. 4.2.2, 4.2.2

[192] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 185–196, New York, NY, USA, 2008. ACM. 7.2.2

[193] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *ACM Transactions on Computer Systems*, 19(4), Nov. 2001. 2.4.1

[194] J. S. Wei Chen, Jason Duell. A software caching system for upc. Technical Report CS265 Project, Department of Computer Science, U.C. Berckley, 2003. 2.4.2

[195] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical report, Cornell University, Ithaca, NY, USA, 1997. 4.2.2

[196] G. V. Wilson. *The History of the Development of Parallel Computing*. 2008. 1.1

[197] P. Wyckoff, D. K. P, O. Infiniband, J. Wu, and J. Wu. High performance implementation of mpi derived datatype communication over infiniband. In *in Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, 2004. 2.3.2

[198] The X10 programming language. `http://x10.sourceforge.net`, 2004. 2.1, 2.3.2, 7.2.2

[199] K. Yelick. Partitioned Global Address Space Languages: Titanium and UPCÂăexperience. Presentation at IBM TJ Watson Research Center, Nov. 2005. 5.2.2

[200] 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, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press. 2.1, 5.1

[201] Z. Zhang, J. Savant, and S. Seidel. A upc runtime system based on mpi and posix threads. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society. 2.4.2

[202] Y. Zhu and L. J. Hendren. Communication optimizations for parallel c programs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 199–211, New York, NY, USA, 1998. ACM Press. 2.3.2

Appendix A

Appendix 1: Architectures

In the development of this thesis, several machine architectures have been used, the main reason is because machines evolve fast and our work, as a research work, intends to be based on current modern machine architectures. We also seek scalability and as soon as a newer/bigger machine is made available to us it is included in our testing bed.

This section gives some details about the overall architecture of all machines used in this thesis. Table A.1 summarizes the most relevant characteristics of our concerns. The first column gives the name of the machine, one machine can be split in two rows if it has been upgraded during the development of this thesis. Second column shows the number of nodes, third column gives the number of processors per node. Fourth column the type of processor and its speed. Fifth column is the amount of memory per node. Sixth column is the type of network connection used in the architecture and finally, column number seven tells in which chapter the machine has been used to evaluate our research work.

Table A.1 Summary of used machine platforms

Machine	Nodes	PxN	Processor	Mem/node	Network	Chapt
IBM Clusters	8	16	Power3 (375Mhz)	64 Gb	SP Switch2	3
	4	8	Power5 (1.9GHz)	16 Gb	HP-Switch	5
	28	16	Power5 (1.9GHz)	16 Gb	HP-Switch	6
BG/L	65,536	2	PPC440 (700MHz)	512 Mb	3D Torus	3, 4, 5
	106,496	2	PPC440 (700MHz)	512 Mb (1Gb)	3D Torus	
BG/X	1,024	2	PPC440 (700MHz)	512 Mb	3D Torus	3, 4, 5
BG/W	20,480	2	PPC440 (700MHz)	512 Mb	3D Torus	3, 4, 5
MareNostrum	2048	2	PPC970-MP (2.2GHz)	8 Gb	2Gb Myrinet	
	2560	4	PPC970-MP (2.3GHz)	8 Gb	2Gb Myrinet	3, 6

A.1 Clusters

A.1.1 IBM RS-6000 SP2

The first experiments were done in an IBM RS-6000 SP machine with 8 nodes * 16 Nighthawk Power3 375Mhz (192 Gflops/s) with 64 Gb RAM per node, all nodes were connected through a SP Switch2 operating at 500MB/sec. This machine is currently out of operation and belongs to the BSC-DAC Museum.

A.1.2 IBM Squadron cluster

It is built of 4 nodes of an IBM Squadron^{TM}cluster. Each node has 8 SMP Power5 processors running at 1.9 GHz and 16 GBytes of memory. The nodes are connected with an IBM High-Performance Switch (HPS). It is located at Toronto IBM Laboratories.

A.1.3 28-node IBM cluster

The third cluster is a 28-node cluster of Power5 servers. Each node is with 16 GBytes of RAM and 8 2-way SMT Power5 cores running at 1.9 GHz. The nodes are connected with an IBM High-Performance Switch (HPS). Located at IBM Poughkeepsie Benchmarking Center.

A.2 Massively Parallel Computers

A.2.1 BlueGene/L

In November 2001 IBM announced a partnership with Lawrence Livermore National Laboratory to build the Blue Gene/L supercomputer [141, 86], a 65,536 dual-processor node machine designed around embedded Power-PC processors. Later, the machine was scaled up from 65,536 to 106,496 nodes in five rows of racks; Through the use of system-on-a-chip integration coupled with a highly scalable cellular architecture, the machine was designed for 360 Teraflops peak performance, it sustains 280 Teraflops when running the (optimized) HPL Linpack performance application [188]. Blue Gene/L represents a new level of scalability for parallel systems. BlueGene/L has occupied the No. 1 position in the Top 500 lists From November 2004 to June 2008.

The Blue Gene/L supercomputer is a 106,496-node machine designed around embedded Power-PC 440 processors (700 MHz of clock frequency each). Two nodes share a

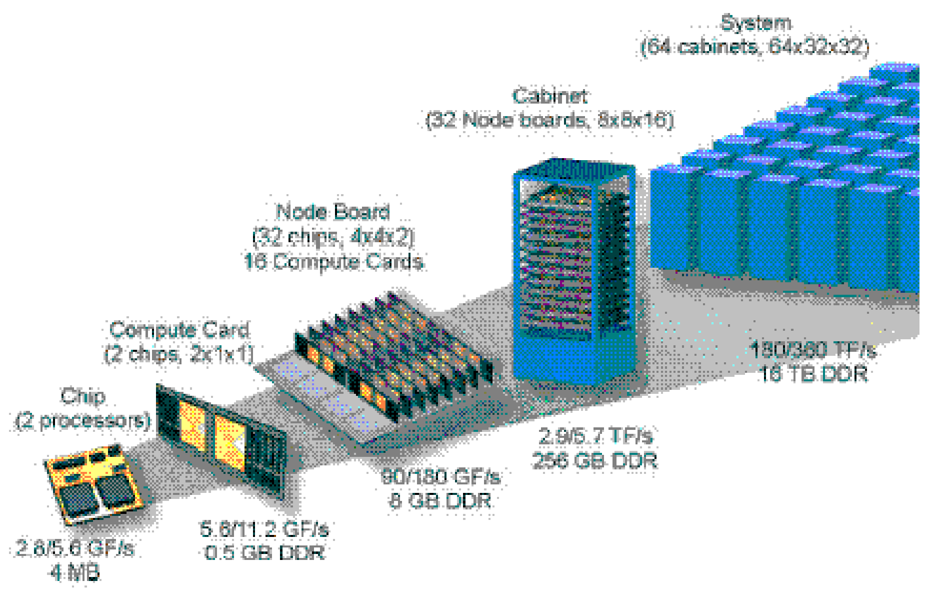


Figure A.1 High-level organization of the Blue Gene/L supercomputer

node card and each of the original nodes carries 512 MBytes of DDR memory and the 40,960 new nodes have double the memory of those installed in the original machine.

The low power characteristics of BG/L permit a very dense packaging as shown in Figure A.1. Sixteen compute cards can be plugged in a node board. A cabinet with two midplanes contains 32 node boards for a total of 2048 CPUs. The complete system has 64 cabinets and 16 TB of memory. In addition to the 104K compute nodes, BG/L contains a number of I/O nodes. Compute nodes are organized into processing sets (psets). In the first machine, The 65,536 compute nodes are organized into 1,024 processing sets (psets). A processing set is a cluster of compute nodes under control of a special node, called an I/O node (1,024 I/O nodes in the first design, one I/O node for every 64 compute nodes)

Although compute nodes and I/O nodes are physically identical, concerning the software, the I/O nodes execute a version of the Linux kernel for embedded processors and are the primary offload engine for most system services. No user code directly executes on the I/O nodes. And Compute nodes execute a single user, single process minimalist custom kernel, and are dedicated to efficiently run user applications.

The compute nodes of BlueGene/L are organized into a partitionable 64 x 32 x 32 three-dimensional torus network, the strong point of the machine. Each compute node contains six bi-directional torus links for direct connection with nearest neighbors. The network hardware guarantees reliable and deadlock-free, but unordered, delivery of variable length (up to 512 bytes) packets, using a minimal adaptive routing algorithm. The I/O nodes are not connected to the torus network.

In addition to the original BlueGene/L installation at Lawrence Livermore National Labs (LLNL), there are now a number of smaller installations scattered across the globe. In our experiments we use:

- free-standing “node cards” (64 processors) for most of the development work.
- `Blue Gene/X`: composed of a single BlueGene/Lrack (2048 processors).
- `Blue Gene/W`: machine at IBM TJ Watson (20 racks, 40960 processors).
- `Blue Gene/L`: the big machine before the upgrade at the LLNL installation (64 racks, 131072 processors).

A.2.2 MareNostrum

In March 2004 the Spanish government and IBM signed an agreement to build one of the fastest computer in Europe. In July 2006, its capacity has been increased due to be the large demand of scientific projects.

MareNostrum [32] is a cluster of 2560 JS21 blades, each with two dual core IBM PPC 970-MP processors (at 2.3GHz) sharing 8 GBytes of main memory. The processors are equipped with a 64 KByte instruction/32 KByte data L1 cache and a 1024 KBytes of L2 cache and run the SLES9 (Linux) operating system.

The MareNostrum’s interconnection network is 2Gb Myrinet with a 3-level crossbar, resulting in 3 different route lengths (1 hop, when two nodes are connected to the same crossbar aka. *linecard*, and 3 hops or 5 hops depending on the number of intervening linecards).

MareNostrum was the most powerful supercomputer in Europe and the fifth most powerful as of November 2006, according to the LINPACK benchmark–it delivers 94.21 Teraflops peak performance. Currently it occupies the 26th position (June 2008). It is located at Barcelona Supercomputing Center.