SANDIA REPORT

SAND2009-2287
Unlimited Release
PrintedApril 2009

Parallel Phase Model: A Programming Model for High-end Parallel Machines with Manycores

Ron Brightwell, Mike Heroux
Sandia National Labs
Albuquerque, NM 87185
Email:{rbbrigh, maherou}@sandia.gov

Zhaofang Wen (supported by NSF grant 0833147)
Sandia National Labs
Albuquerque, NM 87185
Email:zwen@sandia.gov

Junfeng Wu (supported by NSF grant 0833152)
Syracuse University
Syracuse, NY 13244-1150
Email:juwu@syr.edu

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

DEPARTMENT OF ENERGY • UNITED STATES OF AMERICA

SAND2009-2287
Unlimited Release
Printed April 2009

Parallel Phase Model: A Programming Model for High-end Parallel Machines with Manycores

Ron Brightwell [*]
Mike Heroux [†]
Zhaofang Wen [‡]
Junfeng Wu [§]

Abstract

This paper presents a parallel programming model, Parallel Phase Model (PPM), for next-generation high-end parallel machines based on a distributed memory architecture consisting of a networked cluster of nodes with a large number of cores on each node. PPM has a unified high-level programming abstraction that facilitates the design and implementation of parallel algorithms to exploit both the parallelism of the many cores and the parallelism at the cluster level. The programming abstraction will be suitable for expressing both fine-grained and coarse-grained parallelism. It includes a few high-level parallel programming language constructs that can be added as an extension to an existing (sequential or parallel) programming language such as C; and the implementation of PPM also includes a light-weight runtime library that runs on top of an existing network communication software layer (e.g. MPI). Design philosophy of PPM and details of the programming abstraction are also presented. Several unstructured applications that inherently require high-volume random fine-grained data accesses have been implemented in PPM with very promising results.

[*] Ron Brightwell, Sandia National Labs, Albuquerque, NM 87185. Email:rbbrigh@sandia.gov

[†] Mike Heroux, Sandia National Labs, Albuquerque, NM 87185. Email:maherou@sandia.gov

[‡] Zhaofang Wen, supported by NSF grant 0833147. Sandia National Labs, Albuquerque, NM 87185. Email:zwen@sandia.gov

[§] Junfeng Wu, supported by NSF grant 0833152. Syracuse University, Syracuse, NY 13244-1150. Email:juwu@syr.edu

Acknowledgement

Zhaofang Wen's research was partially supported by NSF Grant 0833147, and also partially sponsored by the Disruptive Technology program of Computer Science Research Institute at Sandia National Labs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under Contract DE-AC04-94-AL85000. The work of Junfeng Wu was fully funded by NSF Grant 0833152.

This research used resources of the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

This research would not have been possible without the encouragements and supports of Danny M. Rintoul, Neil Pundit, and Almadena Y. Chtchelkanova. In addition, Junfeng Wu would like to express appreciation to Professor Yuesheng Xu for his advice.

Contents

Figures

Tables

Parallel Phase Model: A Programming Model for High-end Parallel Machines with Manycores

1 Introduction

Large scale high performance computing (HPC) applications often need the computing power of high-end parallel machines with tens of thousands of the processors [1] For the past two decades, most of those high-end parallel machines have been based on a distributed memory cluster architecture consisting of a networked cluster of commodity processors, each with its own memory.

For distributed memory parallel machines, parallel programming has mostly been done using the message-passing programming model, such as MPI [21], which has been very successful in providing good application performance for structured (or regular) scientific applications. An important reason for this success is that the message-passing model closely matches the characteristics of the underlying distributed memory architecture, and allows the application programmers to exploit the parallel capabilities of the hardware; traditionally, programmers usually need to handle certain low-level tasks including explicit managements of data distribution, data locality management, communication preparation and scheduling, synchronization, and load-balancing in order to achieve good application performance. To many application developers who are trained physical scientists, these low-level tasks are not related to their domain expertise, and represent additional programming difficulties, which we refer to as **parallel programming difficulties**. For structured applications, domain decomposition methods can be applied, and the low-level programming tasks generally do not pose a real problem for achieving good performance for structured (or regular) scientific applications, such as dense matrix algorithms. However, for unstructured (irregular) applications that inherently require high-volume random fine-grained communication, those low-level tasks can be challenging to application developers. Unstructured applications represent a very significant percentage of scientific applications, including graph algorithms, handling large-meshes, sparse-matrix algorithms, iterative solution methods, multi-grid, and material physics simulations. Some of the unstructured applications are so difficult to implement efficiently that they are considered unsuitable for MPI parallel programming, for example, [20].

With the recent introduction of multicore technology, commodity processors in cluster architecture are being replaced by multicore processors. This trend indicates that the next-generation high-end parallel machine architecture will include a networked cluster of nodes, each having a large number (hundreds) of processor cores. Developing applications for these machines will be much more difficult than it is today, because programmers will need to simultaneously exploit node-level parallelism (many cores and shared memory) and cluster-level parallelism (distributed memory) in order to achieve good application performance; and the above-listed low-level parallel programming tasks on the new architecture can become overwhelmingly difficult for most application developers. Therefore, a high-level programming model is needed in order to support programmer productivity and more importantly good application performance.

In this paper, we present a programming model, Parallel Phase Model (PPM), for the next-generation high-end parallel machines, to address the parallel programming difficulties listed above in order to help programmers achieve good application performance and programming productivity. PPM has a high-level and unified programming abstraction for both node-level (for the cores) and cluster level parallelisms, to make it easy for parallel application algorithms to exploit and express such layered parallelisms that can be efficiently mapped to the parallel capabilities of the next-generation architecture. The programming abstraction will be suitable for expressing both fine-grained and coarse-grained parallelism. It includes a few high-level parallel programming language constructs that can be added as an extension to an existing (sequential or parallel) programming language such as C; and the implementation of PPM also includes a light-weight runtime library that runs on top of an existing network communication software layer (e.g. MPI). Consequently, PPM will be compatible with existing programming environments such as MPI + C (or Fortran). This compatibility can help the transitions of legacy code base and their programmers; and smooth transitions are vitally important to the success of a new parallel programming environment, because legacy code base represents huge past investments and mission-critical applications that can not be abandoned; and developers of those applications represent the dominant majority of expert programmers in HPC today.

2 Parallel Programming Models

A parallel programming model provides an abstraction for programmers to express the parallelism in their applications while simultaneously exploiting the capabilities of the underlying hardware architecture. A programming model is typically implemented in a programming language, or a runtime library, or both; and the implementation is also referred to as a programming environment.

2.1 Technical Criteria

A practical programming model must balance the often conflicting technical factors including application performance, ease of use, performance scalability to increasing number of processors, and portability to a wide range of platforms. Application performance is arguably the most important factor, and therefore has received the most attention in the past. Ease of use, without sacrificing application performance, is also an important factor in order to get good programming productivity. With the architectural complexities of the next-generation high-end parallel machines, ease of use of the programming environment is becoming essential in order to achieve good application performance.

2.2 Related Work

Over past few decades, many parallel computation and programming models (language extensions and libraries) have been explored. Proposed in the 1970s, PRAM was a shared-memory SIMD model on which a huge volume of parallel algorithms were generated for two decades by the theoretical computation science community [15]. PRAM is a good tool for studying and expressing the inherent parallelism in the mathematical problems because of it simplistic assumptions, which on the other hand have made most (if not all) PRAM algorithms impractical on real-world distributed memory machine platforms where programming is mostly done using a message-passing model (e.g. MPI). In 1989 Valiant [26] proposed the Bulk-Synchronous Parallel (BSP) style for writing efficient parallel programs on distributed memory machines and message-passing environments such as MPI and BSP-specific environments ([13]). So far, MPI has been the most successful. PVM [10] and SHMEM [19] have made an impact on a subset of platforms and applications. On physical shared memory machines, models such as POSIX [25] Threads and OpenMP [24] are also very useful; but in reality, high-end parallel machines tend to be distributed memory machines.

In recent years, a lot of research efforts have been placed on improving parallel programming productivity, including languages and libraries of the global address space models (e.g. [7, 22, 8, 27, 18]) for distributed memory (commodity processor) machines as well as languages supported by the DARPA HPCS program, such as IBMs X10 ([14]) and Crays Chapel ([23]). There are many active research efforts to develop ways to address multi-core programming challenge. These include developing new programming languages and extensions, compilers, runtime libraries, domain-specific application libraries (See [3], [2], [16], [17] and [5] for a partial list of such research efforts). For example, there are attempts to combine MPI with OpenMP or multi-threads as the programming model for the next-generation high-end parallel machines. Since this is still in an early stage, so far, we are not aware of any satisfactory solutions yet.

One thing seems to be clear: the hardware architecture, a cluster of many-cores with both distributed and shared memory, is much more complex than before. Any programming model that require users to deal with the traditional parallel programming difficulties directly at the low-level will make it not only harder to write applications, but also harder to get good application performance. In other words, a high-level programming model is needed not only for better usability, but also for good application performance.

3 Parallel Phase Model

In this section, we formally present a high-level and unified programming abstraction for cluster-level parallelism (distributed memory) and core-level parallelism (shared memory). The programming abstraction enables those low-level parallel programming tasks (as stated in the Introduction Section) to be handled by the compiler and the runtime systems; therefore application developers are relieved from most of those traditional parallel coding difficulties and can focus more of their attentions on the parallel algorithms according to mathematical formulations of their domain problems rather than low-level machine architecture details. Specifically, the PPM abstraction includes the following principles.

- **Virtualization of processors**: Programs are written with unbounded number of virtual processors rather than a fixed number of physical processors. (This concept can be found in the theoretical PRAM model [15] and in programming models such as Charm++ [16] and PRAM C [4].) Virtualization of processors allows for maximal expression of inherent parallelism that exists in the application algorithms, and therefore provides opportunities for the compiler and runtime system to do optimizations such as load balancing.
- **Virtualization of memory**: In this model, virtual processors "communicate" through shared variables. There are two types of shared variables: globally-shared (through virtual shared memory) at the cluster level, and physically shared memory at the node level. Shared memory provides a global view of data structures that make it easier to exploit and express both fine-grained and coarse-grained parallelisms in applications.
- **Implicit communication**: shared variables make communication between processors implicit rather than explicit (as in message-passing model), and allow programs to be high-level and less cryptic, because array syntax can be used in computation code as they are in the mathematical algorithms.
- **Implicit synchronization**: the programming language constructs have build-in, well-defined, and implicit synchronization in the semantics. This avoids the need for explicit barriers as in some other parallel programming models. The benefit is a simple memory model and data synchronization mechanism.

- **Automatic data distribution and locality management**: They are automatically managed by the runtime system.

- **Layered parallelisms**: The PPM programming abstraction allows node level parallelism and global level parallelism to be separately expressed, with a unified syntax. This facilitates design and expression of algorithms to exploit the parallel capabilities of the cluster of many cores. (**Note**: PPM supports parallelism in two levels, but it does not mean that both levels must be used in one program. In fact, programmers can just use only one of the levels, say the global level, when they see fit. When an algorithm step fits naturally, using the node-level can save overhead in global communication and synchronization; this is because the node-level involves fewer processor cores than the global level, and also because data exchange at the node level is done through physical shared memory rather than the network for communication.)

3.1 PPM Language Constructs

The programming abstraction of PPM has several new language constructs, which can be added to an existing programming language such as C. We assume that SPMD model (Single Program Multiple Data) is used. The new constructs are as follows.

1. **Declaration**: PPM supports two kinds of the shared variables, those shared globally across the networked cluster and those shared at the node-level. They can be declared by putting keywords *PPM_global_shared* and *PPM_node_shared* in front of the variable declarations (as in the C language).

   Note: When keyword *PPM_global_shared* is used to declare a variable, only one globally shared variable is declared; however, When keyword *PPM_node_shared* is used to declare a variable, multiple variables of the same name are declared, one for each physical node on the networked cluster (because PPM is a SPMD model). Shared variables provide a convenient way for virtual processors to work on the same data.

   In addition to shared variables, PPM supports variables that are local to a node in the networked cluster (similar to variables in MPI programs). Both PPM local variables and node-level shared variables are stored in the physical shared memory of the node; but they are used differently. Shared variables can be directly accessed by PPM virtual processors (in PPM functions to be described later), but PPM local variables cannot be. Virtual processors can also have their own *private* variables (as declared inside the PPM functions).

2. **Control Construct to Start Virtual Processors**:

   ```
   PPM_do(K) func(arguments);
   ```

This parallel control construct creates K instances of the function "func", on the current machine node, to be executed in parallel, each by a virtual parallel processor with a unique rank in the range between 0 and $K-1$. This unique rank is algorithmically useful in the function instance to determine which portion of a shared data structure to operate on, among other things. Here *PPM_do* is a key word. K is an expression; and its runtime value will be used. It may also be helpful to think that K represents the degree of parallelism that needs to be expressed in an algorithmic step. The function invoked in this control construct is referred to as the PPM function.

3. **PPM Functions**: A PPM function is different from a regular C function in that the PPM function declaration must be preceded with the *PPM_function* keyword. Variables declared inside a PPM function are private to the function; and they become private variables of the virtual processors (started by the PPM_do construct). A PPM function can also include PPM phase constructs (to be presented next).

4. **Parallel Phase Construct**:

   ```
   PPM_global_phase compound_statement;
   PPM_node_phase compound_statement;
   ```

   The parallel phase construct is used inside a PPM function, to provide a mechanism for implicit synchronizations of parallel execution and shared variable updates, across multiple instances of the PPM function (as created by the *PPM_do*). The construct has an implicit barrier at the end of the compound statement, meaning that the parallel executions of the compound statement are not synchronized until the end. Also, any read access to a shared variable will get the value of the variable at the beginning of the phase; and any write access to a shared variable will only take effect after the end of the phase. There are two versions of the parallel phase construct, one for synchronization at the node level, the other for synchronization at the global (cluster) level.

5. **System variables**: The PPM programming environment has several system level variables, *PPM_node_count*, *PPM_cores_per_node*, *PPM_node_id*. Their meanings are self-explanatory, and therefore not discussed here.

6. **Utility functions**: The PPM programming environment also has some utility functions, some of which are common in other programming environments, such as reduction, parallel prefix etc. There is also a function to dynamically allocate space for shared variables. There are utility functions related to the relative ranks of the virutal processors started by a *PPM_do* construct. They are *PPM_VP_node_rank* and *PPM_VP_ global_rank*. There are a few utility functions for mapping (casting) between node-level physical space and globally shared variables.

3.2 Execution Model

PPM is a SPMD model, which means that there is one copy of the same program on each physical node of the cluster. All these copies run in parallel. In addition, the PPM_do construct can be used to start many virtual processors (image them as threads if it helps) for parallel execution of the PPM function. These virtual processors are synchronized according to the phases contained in PPM function. Within every phase, any read access to a shared variable always gets the value as it was the beginning of the current execution of the phase; and updates made to a shared variable become effective only after the end of the current execution of the phase.

3.3 PPM Design Focus

Besides providing the general capabilities of a parallel programming model, PPM focuses on the following areas.

Algorithm parallelism expressiveness: PPM encourages parallel algorithm design based on the inherent parallelism of the application problems rather than low-level hardware architecture details. The *PPM_do* construct allows for maximal expression of parallelism and computation is decomposed for unbounded number of virtual processors rather than a fixed number of physical processors (or cores). Maximal expression of parallelism using virtual processors leaves more room for the compiler and runtime library to do optimizations (such as load balancing).

Layered parallelisms: The cluster of many-core architecture features high-degree of parallelism both at the cluster level (thousands of nodes) and at the node level (hundreds of cores). This is reflected in PPM through the concepts and constructs of global-level and node-level shared variables and synchronization phases. Such high-level programming concept and constructs facilitate the design and expression of parallel application algorithms that can exploit the parallel capabilities of the underlying hardware, without the needs for the application developers to directly handle the low-level machine architectural details.

Guidance for good programming style: It is well-known in the message-passing programming community that, on distributed memory machines, programs written in the Bulk-Synchronous Parallel ([26]) style tend to run more efficiently than programs that require frequent fine-grained communication. This is also true for virtual shared memory programming on the distributed memory machines, as shown in the BEC programming model that seamlessly combines the convenience of the share-memory programming with the efficiency of the BSP style ([27, 12]). The key concept of the BSP style is the super-step. This concept along with the experiences from the BEC virtual shared memory programming have led to the (high-level) phase construct in PPM, in order to capture the essence of efficient high-level programming style involving virtual shared memory on cluster of many-core architecture with a hybrid of physical distributed memory and shared memory.

Simple and implicit synchronization: PPM programs need synchronization just like parallel programs in other programming models. But PPM programs synchronize due to logical needs of the application algorithms rather than low-level hardware constraints in the program implementations. Specifically, synchronizations of both program execution and shared data updates always happen at the end of a PPM phase; there is no need for explicit barrier and locking (as in many other programming models).

Simple memory model and data coherence: Within a PPM phase, reading of a shared variable always gets the value of the variable at the beginning of the phase execution; and writing to a shared variable only takes effect at the end of the phase execution. Such a data coherence scheme avoids the potential unexpected updates to shared variables (race condition) in many other parallel programming models.

No need for explicit communication: In PPM, data exchange is done by accessing shared variables; and the same array syntax is used for both off-node and on-node shared array accesses.

Shared data coherence does not reply on hardware cache coherence capability: In PPM, coherence of shared data is guided by PPM phase and managed by the runtime library. It does not rely on hardware data coherence capabilities.

Supporting both synchronous and asynchronous modes on different nodes: At a $PPM\_do(K) func()$ construct, the PPM function that is invoked can be different on different nodes. (This can easily been done by using function pointers.) Also, expression K can evaluate to different values on different nodes. Furthermore, it is possible that only node-level phase is used in the PPM function for each node. Therefore, a PPM program can make different nodes work on completely different tasks asynchronously using different number of virtual processors. On the other hand, PPM can also have all the nodes work on a common task synchronously using the same number of virtual processors.

Automatic scheduling of computation and communication needs, cores, and network resources: PPM enables the runtime system to dynamically schedule the computation needs (of the virtual processors) and the communication needs based on the available processor cores and network resources. For example, the PPM runtime library is capable of **bundling up fine-grained remote shared data accesses into coarse-grained packages** in order to reduce overall communication over head; the runtime library is also capable of scheduling communication needs and computation tasks to enable (automatic) **overlap of computation and communication**; and the runtime is able to schedule network communication needs to **reduce contention of multiple cores competing for network resources**.

3.4 Implementation of PPM

There can be various ways to implementing PPM, depending on the technical and non-technical constraints. We have a preliminary implementation based on a combination of a source-to-source compiler and a light-weight runtime library. A PPM program is converted into C source code with function calls to the PPM runtime library, which does most of the optimizations in computation work scheduling, remote communication management, and shared data management, among many other things. The virtual processors in PPM can potentially be thought of as threads and also implemented as such. In the absence of hardware support for heavy threading, overheads of context switching and thread scheduling can be a serious performance issue. In our implementation, the PPM compiler converts the work of multiple virtual processors into loops (for a general approach, see [11]), so that there will be fewer threads (each doing the work of multiple virtual processors by looping), which can then assigned to the processors cores.

4 Applications

In our experiments, we intentionally select applications that inherently require high-volume random fine-grained data accesses (communication), which are the most common sources of programming difficulties in writing efficient parallel application programs using existing programming models on distributed memory machines.

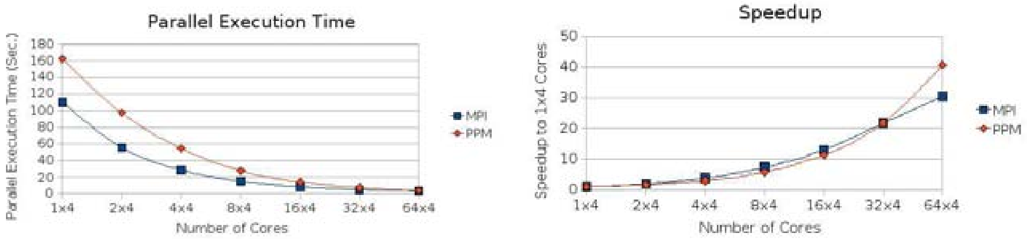
4.1 Machine Platform

The applications presented here were run on a multicore cluster supercomputer named as Franklin (URL: franklin.nersc.gov). It is a Cray XT4 machine with a total of 9660 compute nodes, each node having 4 cores (AMD Opteron 2.3GHz Quad Core) and 8G of physical shared memory.

4.2 Application 1: Conjugate Gradient Solver of Linear Systems

The application is a parallel linear system solver for $Ax = b$ using the Conjugate Gradient (CG) method. The linear system solved in this program is from the diffusion problem on 3D chimney domain by a 27 point implicit finite difference scheme with unstructured data formats and communication patterns. In the test, A is a sparse matrix of size 16777216 X 16777216 with about 400 million nonzero entries.

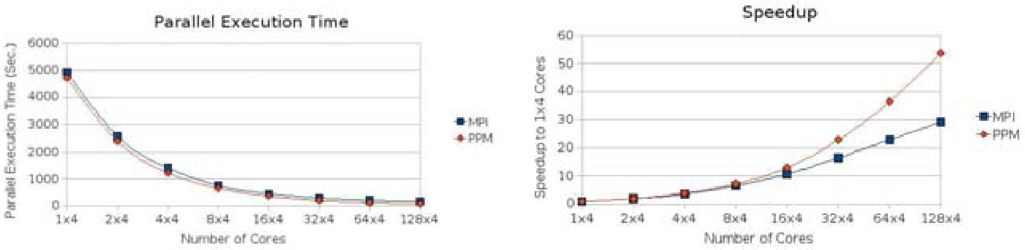
Figure 1. Application Performance of the CG Solver



4.3 Application 2: Sparse Matrix Generation for Multi-scale Collocation Method

This application is a sparse matrix generator in a multi-scale collocation method for integral equations [6]. Every non-zero entry of the generated matrix is a linear combination of multiple functions' values at multiple collocation points. The evaluation of these function values involves numerical integrations of very high computational complexity. To reduce the computational cost, the algorithm iterates through multiple levels of computation, on each of which the intermediate results of the numerical integrations are stored as global data, and then very randomly accessed in the patterns determined by the linear combinations as well as the non-zero pattern of the sparse matrix. In this test, the generated sparse matrix has 1 million rows, 1 million columns and over 200 million nonzero entries.

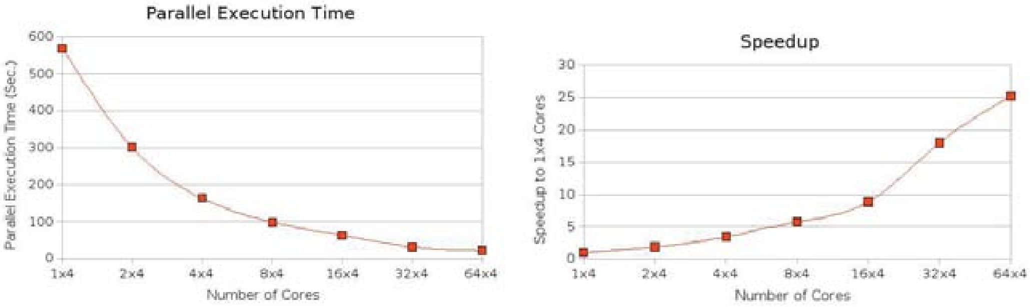
Figure 2. Application Performance of the Matrix Generation



4.4 Application 3: Barnes-Hut Simulation

This application is a simple hierarchical algorithm for N-body simulation. In every time step, the algorithm creates a tree from the particles according to the distribution of their coordinates, then updates the coordinates by computing the particles' forces using the tree. The advantage is the reduced $O(n \log n)$ computation complexity (originally $O(n^2)$), but the drawback is the totally data-driven random access to the tree and the particles. In this test, the number of the particles simulated is 2 million.

Figure 3. Application Performance of Barnes-Hut Simulation



4.5 Performance Discussions of the Applications

Before discussing the performance, it should be pointed out that in the test machine platform, the MPI processes running on the cores of the same node still try to communicate by message-passing; even though such message-passing does not need to go through the network, it can still incur much overhead. [1]

On PPM model, virtual processors on the same node exchange data by accessing the physical shared memory (without any communication overhead). But unlike accesses to variables in standard C language, accesses to the PPM shared variables go through the PPM runtime library, which will bring in some overhead. Such runtime library overhead can weigh on the overall performance using node-level shared variables, but this overhead will weigh much less when the number of nodes and the total network communication cost increase.

Now lets look at the runtime of performances of the PPM version and the MPI version of the Conjugate Gradient Solver. The MPI version is a highly-tuned implementation by a top MPI programmer. PPM version started out much slower than the MPI version when there is only one node (4 cores) but catches up quickly as the number of nodes increases. There can be multiple reasons. We are still trying to conduct in-depth analysis to understand the exact reasons. When there is only one node, the performance results suggest that the MPI intra-node communication overhead is not as significant as the overhead of in PPM shared variable accesses. As the number of nodes increases, the amount of computation per node decreases while inter-node

[1] This intra-node communication overhead can potentially be reduced if the SmartMap mechanism [3] is added to the multicore implementation of MPI runtime library, as demonstrated by the enhanced version of MPI on Catamount at Sandia National Labs. But SmartMap is currently not supported on Linux.

communication increases; in other words, the overhead of the PPM shared variable accesses becomes less of an issue. On the other hand, the PPM runtime library is capable of communication optimizations for reducing network resource contention by multiple cores, for better scheduling communication and computation tasks to allow overlap. These further help the PPM version to catch up in performance relative to the MPI version.

In the sparse matrix generation in multi-scale collocation method for integral equations, the amount of data is not significant, but the computation algorithm is rather complex. The PPM program consistently performs better than the MPI implementation, because the PPM runtime library overhead associated with shared variable accesses is not a significant factor in the total execution time. The PPM program scales better as the number of nodes increases, because the PPM runtime library has many built-in optimizations (as discussed above) that do not exist in the MPI program.

The Barnes-Hut simulation inherently involves high-volume, random, and fine-grained data accesses. Furthermore, the data access locations are data-driven; that is, they cannot be anticipated and prepared in advance; and therefore, it is virtually impossible to prepare and bundle up such data access requests before computation. Such applications are generally unsuitable for MPI implementation (extremely difficult to code to get good application performance). For example, there is an MPI implementation method [9] for the Barnes-Hut simulation; in that method, a hierarchical representation of the force field data is implemented a tree data structure on each MPI node, then in every round of computation, each node needs to receive copies of the trees from all other nodes. This requires extremely high-volume of data exchange and therefore communication costs. On the other hand, PPM is very suitable this type of applications because there is built-in message bundling capability that efficiently handle fine-grained remote shared data accesses; and this capability avoids the need to copy the entire tree structures from other nodes. The PPM program scales well as the number of nodes increases.

In summary, the PPM model provides good performance for unstructured applications on current multicore clusters. Some of the features optimizations have yet to be fully utilized. These features include separation of global and node phases and shared variables; the optimizations include intelligent communication scheduling to allow overlap of communication and computation and reducing network resource bottleneck caused by contention of many cores. **We believe that when the number of cores per node increases, the benefits of the PPM mode by exposing multiple levels of parallelism will be more obvious and significant.**

Table 1. Code Size (Number of Lines)

Application	PPM Program	MPI Program
Conjugate Gradient	161	733
Matrix Generation	424	744
Barnes Hut	499	N/A

4.6 Application Code Size

The PPM implementations are much smaller (and simpler) than the MPI implementations of the same applications. The MPI programs and the PPM programs have similar sizes in their computation codes because they are based on the same mathematical formulas and algorithms. However, the MPI programs include very significant codes in bundling and unbundling fine-grained communication messages in order to achieve good performance; and there is also a fair amount synchronization related code in each of the MPI programs. Such communication and synchronization codes are what make parallel programming difficult. On the other hand, both communication and synchronization are implicit in PPM, so there is no need to write any communication and synchronization code; and PPM has built-in communication bundling/unbundling capabilities and other optimizations; therefore the PPM programs are much smaller and simpler, while achieving comparable (and better) performances.

5 Code Example

Given a sorted array A and another array B, the problem is to find the location in A for each and every element of B. The following is a piece of PPM code to solve this problem. In this code, we assume that both arrays A and B are already initialized. We only show the part of the code to do the parallel binary search of each element of B inside array A. **Note**: This is not an optimal parallel algorithm for the problem. It is used just for its simplicity for a PPM code example.

```
/* In the following code, the binary search of B elements
in array A are carried out in parallel; specifically, the
search of each element is performed by a virtual processor.
Here assume B is a node-level shared array and A is a global
shared array. */
PPM_function binary_search(int n,
    PPM_global_shared double A[],
    PPM_node_shared double B[],
    PPM_node_shared int rank_in_A[])
{
    PPM_global_phase {
        int left, middle, right;
        left = 0;
        right = n;
        while (left + 1 < right) {
            middle = (left + right) / 2;
            if (A[middle] < B[PPM_VP_node_rank()])
```

```
                left = middle;
            else
                right = middle;
        }
        rank_in_A[PPM_VP_node_rank()] = right;
    }
}

int main(int argc, char** argv)
{
    /* Other code, including initialization of arrays
       A[0..N-1] and B[0..K-1]*/
    ...

    PPM_do(K) binary_search(N, A, B, rank_in_A);

    ...
}
```

6 Conclusion

We have presented a parallel programming model, Parallel Phase Model (PPM), for the next-generation high-end parallel machines, which has a cluster of nodes with a large number of cores on each node. We have implemented several unstructured applications that inherently require high-volume random fine-grained data accesses. Such applications are generally very difficult to implement in existing models such as MPI in order to get good application performance. Using MPI implementations as references, the PPM implementations of these applications show favorable results (good performance and scaling) and easy programmability in term code simplicity and sizes. **Although the performance advantage is rather limited on current machines, we believe the benefits of the PPM model will be more significant when the number of cores per node increases (far beyond the current 4 cores per node).**

References

[1] TOP500 Supercomputing Sites. URL http://www.top500.org/.

[2] Ayon Basumallik, Seung jai Min, and Rudolf Eigenmann. Towards openmp execution on software distributed shared memory systems. In *Proc. WOMPEI02, LNCS 2327*, pages 457–468. Springer Verlag, 2002.

[3] Ron Brightwell, Trammell Hudso, and Kevin Pedretti. Smartmap: Operating system support for efficient data sharing among processes on a multi-core processor. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08)*, Austin, TX, 2008.

[4] Jonathan Brown and Zhaofang Wen. PRAM C: A new parallel programming environment for fine-grained and coarse-grained parallelism. Technical Report SAND2004-6171, Sandia National Laboratories, 2004.

[5] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In *the Proceedings of workshop on state-of-the-art in scientific and parallel computing (Para06). Springer's Lecture Notes in Computer Science 4699*, pages 1–10, Umeå, Sweden, 2007.

[6] Zhongying Chen, Bin Wu, and Yuesheng Xu. Fast collocation methods for high-dimensional weakly singular integral equations. *Integral Equations Appl.*, 2007.

[7] UPC Consortium. UPC language specification (v 1.2). http://www.gwu.edu/ upc/documentation.html.

[8] K. Yelick et. al. Titanium, a high-performance Java dialect. *Concurrency: Practice and Experience*, 10:825–836, 1998.

[9] David Garmire and Emil Ong. Object-oriented parallel barnes-hut. URL http://www.cs.berkeley.edu/ emilong/research/oopbh.pdf.

[10] A. Geist et al. PVM home page. www.csm.ornl.gov/pvm/pvm_home.html, 2005.

[11] Sue Goudy, Shan Shan Huang, and Zhaofang Wen. Translating a high level PGAS program into the intermediate language BEC. Technical Report SAND2006-0422, Sandia National Laboratories, 2006.

[12] Mike Heroux, Zhaofang Wen, and Junfeng Wu. Initial experiences with the BEC parallel programming environment. In *the 7th International Symposium on Parallel and Distributed Computing*, 2008.

[13] Jonathan Hill. The Oxford BSP Toolset (url). www.bsp-worldwide.org/implmnts/oxtool/.

[14] IBM. The X10 Programming Language. http://x10-lang.org/.

[15] J. JaJa. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[16] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c. In *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108. ACM Press, 1993.

[17] Milind Vidyadhar Kulkarni. The Galois System: Optimistic Parallelization of Irregular Programs. http://hdl.handle.net/1813/11139.

[18] J. Neplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[19] NPACI. SHMEM tutorial page. www.npaci.edu/T3E/shmem.html, 2005.

[20] E. Rothberg and A. Gupta. Parallel ICCG on a hierarchical memory multiprocessor addressing the triangular solve bottleneck. *Parallel Computing*, 18(7):719–741, July 1992.

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1, The MPI core.* The MIT Press, 1998.

[22] Co-Array FORTRAN Working Group (url). Co-Array FORTRAN home page. www.co-array.org, 2005.

[23] Cray (url). Chapel — The Cascade High-Productivity Language. http://chapel.cs.washington.edu/.

[24] OpenMP Architecture Review Board. (url). OpenMP fortran application interface version 1.1. www.openmp.org.

[25] The Open Group (url). POSIX home page. www.opengroup.org/certification/posix-home.html, 2005.

[26] Leslie G. Valiant. A bridging model for parallel computation. *Comm. ACM*, August 1990.

[27] Junfeng Wu Zhaofang Wen. BEC specification and programming reference. Technical Report SAND2007-7617, Sandia National Laboratories, Albuquerque, NM, 2007.

DISTRIBUTION:

1 MS 0817
James Ang, 1422

1 MS 0817
Bob Benner, 1422

1 MS 0376
Ted Blacker, 1421

1 MS 1319
Ron Brightwell, 1423

1 MS 0382
Kevin Brown, 1543

1 MS 1319
S. Scott Collis, 1414

1 MS 0318
George Davidson, 1412

1 MS 1319
Erik DeBenedictis, 1423

1 MS 0817
Doug Doerfler, 1422

1 MS 0316
Sudip Dosanjh, 1420

1 MS 0382
Mike Glass, 1541

1 MS 8960
James Handrock, 9151

1 MS 1319
William Hart, 1415

1 MS 0822
Rena Haynes, 1424

1 MS 1319
Bruce Hendrickson, 1414

1 MS 1319
Michael Heroux, 1414

1 MS 0316
Scott Hutchinson, 1437

1 MS 0817
Sue Kelly, 1422

1 MS 0378
Marlin Kipp, 1431

1 MS 1111
Patrick Knupp, 1411

1 MS 0801
Rob Leland, 4300

1 MS 0370
Scott Mitchell, 1411

1 MS 1319
Steve Plimpton, 1412

1 MS 0807
Mahesh Rajan, 4328

1 MS 1319
Rolf Riesen, 1423

1 MS 0318
Elebeorba May, 1412

1 MS 0321
Jennifer Nelson, 1430

1 MS 0321
James Peery, 1400

1 MS 1319
Cynthia Phillips, 1415

1 MS 1319
Neil Pundit, 1423

1 MS 1316
Mark D. Rintoul, 1412

1 MS 1319
Suzanne Rountree, 1415

1 MS 1111
Andrew Salinger, 1416

1 MS 0378
Stewart Silling, 1431

1 MS 0378
James Strickland, 1433

1 MS 0378
Randall Summers, 1431

1 MS 0370
Tim Trucano, 1411

1 MS 0817
John VanDyke, 1423

1 MS 0817
Courtenay Vaughan, 1422

1 MS 1319
Zhaofang Wen, 1423

1 MS 0822
David White, 1424

1 MS 1319
David Womble, 1410

1 MS 0823
John Zepper, 4320

2 MS 9018
Central Technical Files, 8945-1

1 MS 0899
Technical Library, 9536