Exploiting Object-Based Parallelism on Multi-Core Multi-Processor Clusters

Xuli Liu
University of Nebraska at Kearney
Kearney, NE 68849, USA
liux1@unk.edu

Abstract

Programming using message passing or distributed shared memory are the two major parallel programming paradigms on clusters. However, these two models have high programming complexity, produce less maintainable parallel code, and are not suitable for multi-core multi-processor clusters. While object-oriented programming is dominant in serial programming, it has not been well exploited in parallel programming. In this paper, we propose an innovative automatic parallelization framework that employs past experience to parallelize serial programs and outputs the parallel code in the form of objects. Supported by a data-driven runtime environment, each parallel task is managed as a thread, exploiting the multiple processing cores on a cluster node. Based on this proposed framework, we have implemented a proof-of-concept parallelizer called PJava to parallelize Java code. The performance benefit of this framework is evaluated through case studies by comparing the execution time of the automatically generated PJava code to that of handcrafted JOPI (a Java dialect of MPI) code.

1. Introduction

Message-passing-based programming and Distributed-Shared-Memory-based (DSM) programming are the two major parallel programming paradigms on clusters. In a message-passing environment, parallel programmers uses a communication and synchronization library, e.g., Message Passing Interface (MPI) [19], to coordinate parallel processes on multiple cluster nodes. While message-passing offers parallel programmers a stable programming environment, it requires them to have certain level of concurrent programming knowledge and manage complicated and error-prone message passing. Furthermore, this message-passing-based programming model has long been criticized for mixing problem-solving algorithms with low-level inter-node communication code, making the parallel code hard to maintain.

DSM exposes a virtually shared memory to parallel programmers on a cluster, thus the programmers can focus on the problem-solving algorithm without explicitly handling message passing between parallel processes. However, to guarantee the semantic correctness of a parallel program, DSM still requires a programmer to carefully synchronize these parallel processes using semaphores or monitors, thus leaving open the opportunity for deadlocks. Furthermore, DSM has a high overhead due to its rigorous requirement for consistency control over replicated data on each cluster node [16]. In addition, a DSM may be infeasible or difficult to implement on a more loosely-coupled distributed environment (e.g., a computational Grid), where nodes may scatter across the boundaries between different administrative domains over a wide area network.

Furthermore, both message-passing and DSM are challenged by the advent of multi-core processor architecture, which favors multi-threaded applications [13]. A node in a typical modern cluster has multiple multi-core processors, thus making multi-threaded parallel code a necessity in order to fully exploit the multiple processing cores equipped on each cluster node. Unfortunately, the specification of MPI does not include explicit definition for multi-threading, and it is a common practice to execute multiple MPI processes on each cluster node or use hybrid programming, e.g., incorporating MPI and OpenMP [20] together [2]. However, these two solutions are either inefficient or too expensive to develop. DSM-based parallel programming is also complicated since processes or threads on the same cluster node prefer communication through physically shared memory, whereas DSM is still needed to support the communication among different cluster nodes.

The combination of the aforementioned limitations of the message-passing-based and DSM-based programming models challenges researchers to build a new programming paradigm. This new programming model should strive for (1) a more friendly programming interface for non-professional parallel programmers, (2) less error-prone and more maintainable parallel code, (3) thread-level paral-

0-7695-3049-4/07 $25.00 © 2007 IEEE
DOI 10.1109/.25

lelism, and (4) light-weight runtime environment.

Bearing these goals in mind, we have designed an object-based parallel programming model. Different from traditional procedural programming, to which current message-passing-based programming and DSM-based programming belong, this proposed object-based parallel programming model represents each parallel task as an autonomous object, encapsulating both data and its processing code. The synchronization among parallel tasks is coordinated through a directed acyclic graph called *dependency graph* that describes the inter-dependency among the tasks, eliminating the necessity of explicit synchronization code. In addition, the object-based parallel tasks are handled in a data-driven way – a task is triggered when all its prerequisites are met. Parallel tasks distributed to a cluster node are managed as threads in a multi-threaded application, thus being able to exploit thread-level parallelism. Furthermore, object-form parallel tasks are easier to be scheduled. For example, when more cluster nodes become available during the execution of a parallel application, some tasks can be simply ported to these newly available nodes since the cluster architecture is not reflected in the code. To further relieve programmers of parallel programming, we have also developed a framework to automatically parallelize a serial code into a parallel one.

The rest of this paper is organized as follows. In Section 2, we introduce the related work. After describing the parallelization framework in Section 3, we present the runtime environment in Section 4. In Section 5, we compare the performance of this proposed model to that of JOPI through two case studies. Finally, we conclude and discuss the future work.

2 Related Work

Practitioners in cluster computing by and large program with popular serial languages (e.g., C and Java) and handle inter-node communication with message-passing libraries, such as MPI [19], JOPI [6], and Linda [8]. The Message Passing Interface (MPI) is a *de facto* standard for communication among parallel processes on a cluster, and its implementations include FORTRAN, C, C++, and Ada. The Java Object Passing Interface (JOPI) is an implementation of MPI for Java, where an object is the very basic data exchanging unit. Linda provides a logical associative memory called a tuplespace and employs a few primitive instructions to add or retrieve data from this tuplespace. Implementations of Linda can be found in such languages as Prolog, C, and Java. Instead of requiring programmers to explicitly handle the communication and synchronization among parallel tasks as done in MPI, our proposed object-based programing model uses a dependency graph to synchronize parallel tasks, thus reducing the workload of parallel programmers and eliminating the possibility of deadlock.

Compared to message-passing, DSM provides parallel programmers a more user-friendly environment. Because processes on different cluster nodes virtually share a global memory, a programmer only needs to carefully synchronize the parallel processes without explicitly handling message passing. DSM has been one of the research hotspots in distributed systems, and many systems have been proposed and implemented, among which Treadmarks [15] is one of the most successful ones.

Due to the portability of Java as well as its parallel features such as Remote Method Invocation (RMI), monitor, and multi-threading, most of the current DSM studies on clusters concentrate on distributed Java Virtual Machine (JVM), and these studies can be categorized as three approaches. The first approach is to lay a distributed JVM on top of some existing DSM system, such as Java/DSM [26]. The second approach is based on a cluster-enabled implementation of JVM, of which cJVM [7] is a good example. The third approach, also the most widely used one by current research projects, is to put another layer on top of JVM, providing the necessary parallel and distributed features. Examples of the third approach include JavaParty [22], Hyperion [18], Jackal [24], etc. Instead of providing a virtually shared memory, our model still uses message-passing as the way of communication among cluster nodes. However, this communication is transparent to upper levels in that it is performed automatically according to a dependency graph, thus avoiding the heavy consistency control overhead of DSM.

The data-driven feature of our proposed model is similar to the dataflow idea of dataflow programming languages (e.g., SAC (Single Assignment C) [3], Prograph [4], and LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) [5]). The instructions coded with a dataflow language are grouped as ”black boxes”, with inputs and outputs, that are triggered to execute when all of their inputs become valid. This feature makes dataflow languages inherently suitable for task parallelism. Unfortunately, the feature of *single assignment of variables* (i.e., disallowing the reassignment of variables once their values have been assigned) in dataflow languages makes them hard to handle loops (i.e., data parallelism) [14]. In our proposed model, a parallel/distributed application is composed of medium-grained autonomous objects, and the synchronization of these objects is coordinated through a dependency graph instead of a new programming language. Thus, a parallel programmer can use his/her favorite language to perform parallel programming.

ProActive [1][10] is another ongoing project that studies object-based parallel programming. ProActive is an Open Source Java library with a reduced set of simple primitives, supporting parallel/distributed multi-threaded com-

puting on clusters, local area network, or even Internet Grids. Similar to our model, an application coded with ProActive consists of a number of medium-grained entities called *active* objects. Each of the active objects has its own thread of control, and their synchronization is handled by a mechanism called *wait-by-necessity*. Instead of exposing a parallel programming library to programmers as in ProActive, our model hides the parallelization details from programmers. Furthermore, the synchronization of objects in our model is controlled by a dependency graph, and the execution order of these objects is further optimized as described in Section 4.2.

3. Generating Object-based Parallel Code from Serial Code

In this section, we discuss the process of parallelizing a serial code into an object-based parallel code from a higher level. The procedure presented in Figure 1 is characterized by pattern-based parallelization and object-based parallel code.

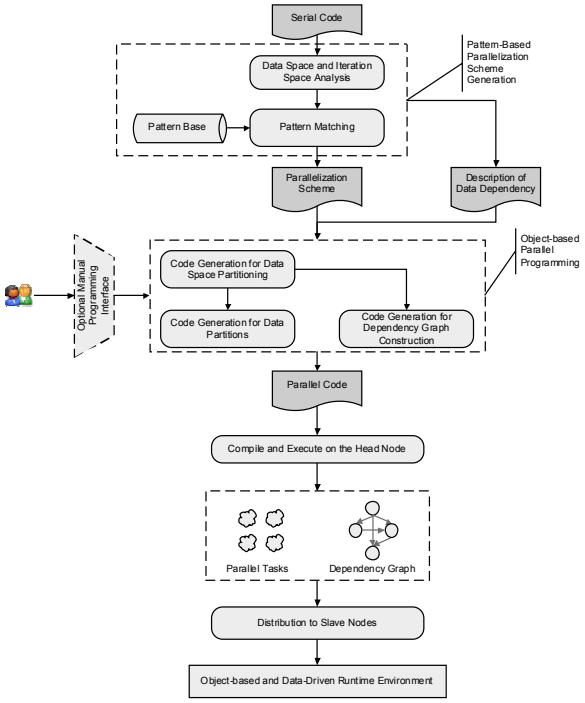

Figure 1. The procedure of generating object-based parallel code from a serial program

3.1 Pattern-based Parallelization

Although the framework presented in Figure 1 works for both data parallelism and task parallelism, our research focuses on data parallelism at its current phase. Having observed that loops sharing certain features can be parallelized with a similar, if not the same, scheme, we use past parallelization experience to help parallelize future programs. As shown in Figure 1, past successful parallelization schemes are abstracted as situation-solution pairs, which are stored in a pattern base. When a loop is to be parallelized, its features are extracted and then collectively labeled as the current situation. Then, the best-matching case in the pattern base is searched for [21] and used to guide the corresponding parallelization.

By using this pattern-based parallelization strategy, not only is the work of a parallel programmer eased, but also the stability and efficiency of the parallel code can hopefully be improved. In Figure 1, we can also see that a description of the data dependency relationship carried by the loop is produced in addition to the parallelization scheme. This description will help generate the code of constructing the dependency graph in the phase of parallel code generation.

3.2 Object-based Parallel Code

The features of object-oriented programming, such as encapsulation and inheritance, provide us a good opportunity to design an alternative parallel programming model. Following the parallelization scheme obtained from the pattern base, the parallel code is generated. As shown in Figure 1, the parallel code includes the code of partitioning the data space, the processing code of data partitions, and the code to construct the dependency graph that describes the inter-dependency among data partitions. This parallel code generation does not start from scratch, instead the fundamental functions, such as communication, are pre-defined into a set of *base* classes, from which the parallel code can inherit or extend.

In fact, the produced parallel code is still regular serial code, and it is compiled and executed on the head node of a cluster. By running the parallel code, the data space is partitioned with an optimal granularity [17], and each data partition is encapsulated with its processing code into autonomous objects. Also, the dependency graph is constructed. And then, these objects (i.e., parallel tasks) and the dependency graph are distributed to the slave nodes of the cluster for execution. The parallel tasks scheduled to a cluster node are executed as a multi-threaded application (refers to Section 4), which is able to effectively exploit the multiple processing cores of multi-core multi-processor clusters, as opposed to a single-threaded application. Furthermore, because all objects are self-contained, it is easy to migrate a task in the form of an object from one cluster node to another even during the execution of the parallel application.

4 The Runtime Environment

In this section, we study the runtime environment that supports the execution of the object-based parallel code discussed in Section 3.

4.1 The Architecture of the Runtime Environment

On each cluster node, a lightweight middleware, as shown in Figure 2, is provided to support the execution of parallel tasks. Figure 2 shows that each cluster node runs a service agent that accepts tasks (i.e., objects) from the head node and requests from other slave nodes. Each received parallel task will be executed as a thread, and parallel tasks scheduled to the same cluster node form a multi-threaded application.

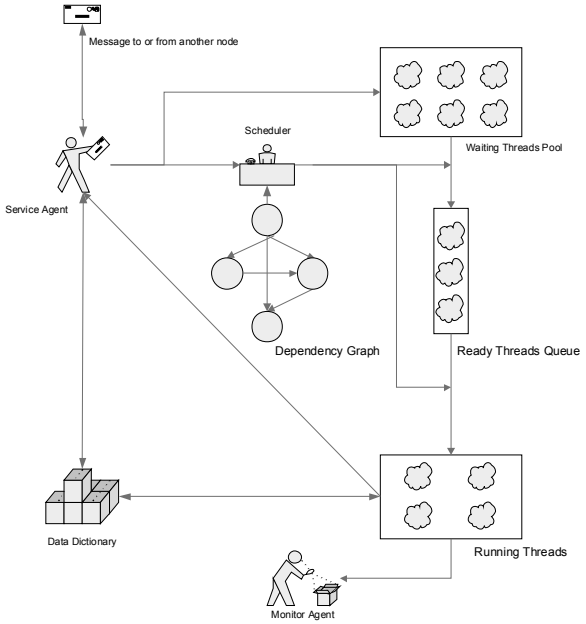

Figure 2. The workflow of a multi-threaded application running on a cluster node

In Figure 2, a dependency graph that defines the inter-dependency among all parallel tasks across cluster nodes sits at the heart of the runtime environment. All received parallel tasks are initially stored in the *waiting threads pool*, and a task is moved from the *waiting threads pool* to the *ready threads queue* when all of its prerequisites defined by the dependency graph are met. The tasks sitting in the *ready threads queue* will be properly scheduled to execute when a processing core becomes available.

The data dictionary shown in Figure 2 contains not only the data being processed by the local application but also replicas of the data owned by other applications that the local application refers to. This data dictionary is shared by all parallel tasks located on the same cluster node, and their communication goes through the physically shared memory.

Although the monitor agent in Figure 2 is not directly involved in the processing of a task, it plays an important role in this runtime environment. This monitor agent keeps track of the variance of the utilization of system resources (e.g., CPU usage), and this information can be used to perform load-balancing by migrating tasks from one node to another. In addition, this monitor agent tracks the access patterns (e.g., when and where the accesses are from) on an object, and this information can be used to better distribute the parallel tasks (e.g., distributing closely-related tasks to the same cluster node so as to reduce inter-node communication) in the future.

4.2 The Operational Mechanism of the Runtime Environment

In this subsection, we first introduce the data-driven execution model of parallel tasks, and then we discuss some technical issues related to the state transitions (i.e., among *waiting*, *ready*, and *running*) of a parallel task.

4.2.1 The Data-Driven Execution Model

The execution of parallel tasks is scheduled by a data-driven model, which can be explained with the sample dependency graph shown in Figure 3. In Figure 3, each vertex represents a task and an edge between two vertices symbolizes the existence of data dependency between those two corresponding tasks. For instance, there are two incoming edges to vertex `E`: $C \rightarrow E$ and $B \rightarrow E$. Thus, task `E` will not be started until tasks `B` and `C` are all finished. On the other hand, after a task is finished, it will trigger the execution of its dependent tasks. For example, the completion of task A will trigger the executions of tasks C and D. Without explicit synchronization as done in traditional parallel programming, a parallel task can be started immediately after all its required data is available, thus eliminating unnecessary synchronization time.

4.2.2 Switching a Parallel Task from *Waiting* State to *Ready* State

Notice that inappropriate scheduling may actually leave some processing cores of a cluster node idle. Consider a cluster node with only 2 processing cores that has a task dependency graph as shown in Figure 3. At some point, task B is taking over one processing core, and task A is just completed. Obviously, both tasks C and D can get scheduled. If task D is scheduled, even though tasks B and D

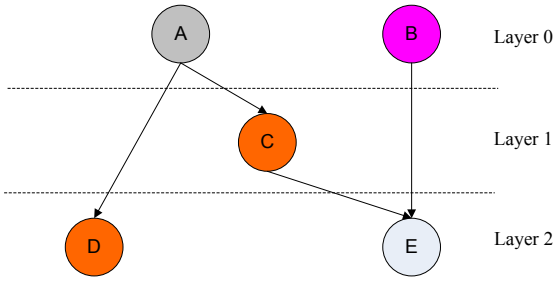

Figure 3. A sample task dependency graph

will be completed at the same time later on, only task C can be started then, leaving one processing core in an idle state. On the contrary, if task C gets scheduled, then both tasks D and E can be executed simultaneously assuming that tasks B and C can be completed around the same time. Clearly, a big difference can be made if the *ready* tasks can be properly ordered (refers to Figure 5 for the performance difference). We have designed and implemented an algorithm called PRIORITY_TOPOLOGICAL_SORT(G) to properly order *ready* tasks, as shown in Figure 4.

The correctness of the algorithm shown in Figure 4 can be proven as follows.

Proof:

Steps 1 to 3 carry out *topological sort*. Due to the characteristics of `DFS` (Depth-First Search), for any edge (u, v) explored by `DFS(G)`, we always have $f[v] < f[u]$, where $f[v]$ and $f[u]$ represent vertices v's and u's finish times in DFS, respectively. Thereby, a task is always triggered later than its predecessors (prerequisites), and the inter-dependency of the tasks held in the doubly linked list is met (Refers to [12] for more details).

Step 4 moves the vertices with higher weights (i.e., with more children) to the front of the list `L` as much as possible, so that more tasks can be made *ready* to start in order to fully exploit the processing cores on a cluster node. For each task in list `L`, we compare its weight with that of each task in the front of the list until reaching its parent in graph `G` or some other task that has a higher weight (See steps 4.5 to 4.7). Steps 4.8 to 4.13 move the task being processed to its correct position in the list.

Clearly, after the process described above, tasks with higher priority are put as close to the head of the list as possible, but never ahead of its parents. Thus the algorithm correctness is proved.

■

`PRIORITY_TOPOLOGICAL_SORT(G)`

`G = (V, E)` is a DAG that represents the inter-dependency relationship among tasks. `V` is the collection of vertices of `G`, and `E` is the set of the edges.

1. Call `DFS(G)` to compute finishing times `f[v]` and the weight (i.e., the number of children) `w[v]` for each vertex v.
2. Create an empty doubly linked list `L`.
3. As each vertex is finished, insert it to the front of `L`.
4. Reorder the vertices in the list `L`.

   In the following pseudo code, we use `prev[v]` and `next[v]` to represent the previous node and next node of vertex v in the list `L` respectively, and use `parent[v]` to denote the parent of vertex v in the graph `G`.

```
4.1.  curr = next[head]
4.2.  while(curr != NIL){
4.3.      p = prev[curr]
4.4.      q = next[curr]
4.5.      while(parent[curr] ≠ p and
      w[curr] > w[p]){
4.6.          p = prev[p];
4.7.      }
4.8.      next[prev[curr]] = q
4.9.      prev[q] = prev[curr]
4.10.     next[curr] = next[p]
4.11.     prev[next[p]] = curr
4.12.     next[p]] = curr
4.13.     prev[curr] = p
4.14.     curr = q;
4.15. }
```

Figure 4. PRIORITY_TOPOLOGICAL_SORT: An algorithm to create a task priority list

4.2.3 Switching a Parallel Task from *Ready* State to *Running* State

Since the round-robin (RR) scheduling algorithm, or its variant, is used to schedule threads on modern time-sharing

systems [23], putting a large number of threads in memory will result in frequent context-switching. Therefore, we need to carefully determine the number of running threads based on the characteristics of the particular parallel application and the number of available processing cores on a cluster node. For example, if all threads are compute-intensive with no I/O operations, the number of running threads may be set equal to the number of processing cores; if some threads need I/O accesses, then the number of running threads should be set to be slightly greater than the number of processing cores, so that there are sufficient number of threads to take over the CPU when one or more threads wait for I/O. Currently we set the number of running threads based on the characteristics of the particular parallel application and the cluster node's resource utilization.

5 Performance Evaluation of the Generated Object-based Parallel Code

Based on the framework presented in Section 3, we have implemented a proof-of-concept parallelizer called *PJava* on top of Jikes [11] to parallelize serial Java programs. We choose Java for our experiments to hide heterogeneity of runtime support while the performance penalty is acceptable compared to native code implementations [9]. In this section, we demonstrate PJava's performance benefits by comparing its execution time to that of the JOPI code [6] in two case studies – LU factorization and matrix multiplication.

5.1 The Experimental Setup

The experiments in this section were carried out on a Linux cluster with 10 nodes. Each node has 2 AMD dual core opteron 270 processors (2GHZ) and 4GB RAM, and all these cluster nodes are connected with a Gigabit Ethernet. We have chosen the Sun Java 2 Platform Standard Edition (J2SE 5.0) as the software platform.

5.2 Case Study 1: LU Factorization

LU factorization lies at the heart of Gaussian elimination, factoring a matrix into the product of a lower triangle and an upper triangle. LU carries complex data dependency, and it is an ideal application to evaluate the feasibility of this proposed framework to dependency-intensive applications.

In this case study, the size of the matrix to be factored was set to be 3600×3600 (double-precision floating numbers). The JOPI code and the PJava code used the same parallel algorithm provided by the Stanford SPLASH-2 benchmark suite [25]. In the automatically generated PJava code, the loop bounds of each parallel task are determined on the fly. Since all parallel tasks can be divided into three types and the loop bounds for each type are fixed, we manually optimized the generated PJava code. Both the non-optimized and optimized versions were tested in runtime environments with and without the scheduler discussed in Section 4.2.2. In the experiments on JOPI implementation, we have scheduled 4 processes onto each cluster node. To evaluate the scalability, the above experiments were done on 1 node, 2 nodes, 4 nodes, 6 nodes, and 8 nodes, respectively. We have also run the standard serial LU algorithm on one single node to see how parallel programming can help improve execution performance. For each scenario, we conducted the experiments for 5 times, and the average execution times were collected and are compared in Table 1 and Figure 5. In Table 1, *PJava1* and *PJava2* represent the automatically generated PJava code with and without the scheduler respectively, and *Optimized1* and *Optimized2* stand for the optimized PJava code with and without the scheduler respectively.

# of Nodes	1	2	4	6	8
Serial	909.4	N/A	N/A	N/A	N/A
JOPI	78.92	46.14	26.76	20.5	18.95
PJava1	78.11	51.36	33.94	25.01	18.23
PJava2	69.08	39.69	27.93	18.73	18.11
Optimized1	67.5	43.82	24.84	22.6	16.65
Optimized2	58.26	34.36	20.23	16.32	14.05

Table 1. Execution times of serial, JOPI, and PJava code for LU

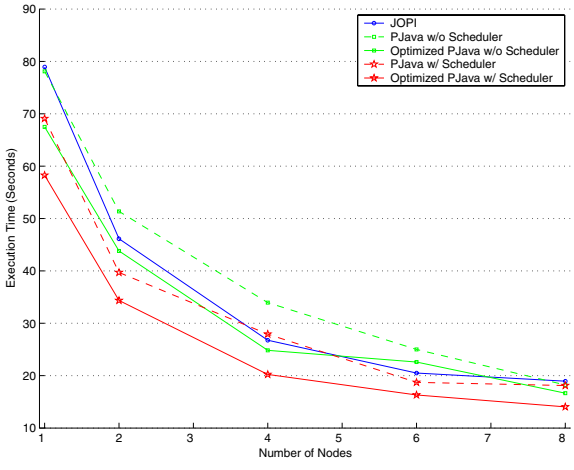

Figure 5. Performance comparison of JOPI and PJava code for LU

From Table 1, we can see that the parallel code (either JOPI or PJava) greatly outperformed the serial code even on 1 node since the parallel code can take advantage of

the multiple processing cores and has higher cache hit rate due to its partitioned data space. Figure 5 clearly shows that that the performance of the automatically generated PJava code was worse than that of the JOPI code if parallel tasks were not properly scheduled. With scheduling provided, the generated PJava code outperformed JOPI code. This performance advantage comes from the fact that PJava is thread-based and the communication between threads is through shared memory on a cluster node; on the contrary, the JOPI code used more expensive inter-process communication. Figure 5 also shows that the performance of the PJava code was greatly improved after being optimized thanks to the source-to-source parallelization employed by this framework.

Although the data-driven feature of PJava could bring potential performance benefit due to its minimized synchronization time, unfortunately this advantage was not demonstrated in this case study. By monitoring JOPI processes in our experiments, we did not see much synchronization time existed due to the characteristics of the parallel algorithm used by the JOPI code. In general, however, we expect PJava's data-driven model to bear more pronounced performance advantage when compared to explicitly-synchronized message-passing codes.

5.3 Case Study 2: Matrix Multiplication

Matrix multiplication ($C = A \times B$) is embarrassingly parallelizable. In this case study, the sizes of matrices of A and B are both set at 2500×2500 (double-precision floating numbers). In both the JOPI implementation and the PJava implementation, we simply assign a part of matrix C to each process, and the required data from matrix A and B is distributed accordingly. Because the parallelization of matrix multiplication was quite straightforward, there was no need to optimize the automatically produced PJava code. Furthermore, the scheduler equipped in the runtime environment had no impact on performance as no dependency is carried among parallel tasks. Thus, the automatically generated PJava code was simply executed in this case study. Similar to the case study of LU, the JOPI code and PJava code were both executed on 1 node, 2 nodes, 4 nodes, 6 nodes, and 8 nodes to evaluate their scalability. We have also run the standard serial matrix multiplication algorithm on one single node to see how parallel programming can help improve execution performance. For each scenario, we conducted the experiments for 5 times, and the average execution times were collected and are compared in Table 2 and Figure 6. Similar to the case study of LU, 4 processes were scheduled onto each cluster node in the experiments on the JOPI code.

Table 2 indicates that the parallel code (either JOPI or PJava) greatly outperformed the serial code even on 1 node,

# of Nodes	1	2	4	6	8
Serial	819.95	N/A	N/A	N/A	N/A
JOPI	149.54	80.59	51.49	49.69	45.89
PJava	91.67	68.28	49.53	34.12	29.07

Table 2. Execution times of serial, JOPI, and PJava code for matrix multiplication

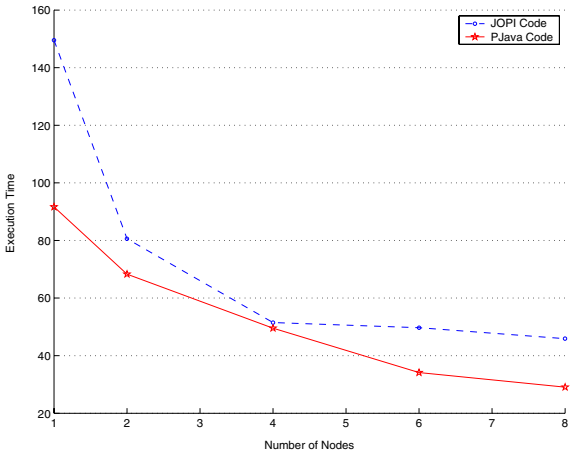

Figure 6. Performance comparison of JOPI and PJava code for matrix multiplication

and this performance gain comes from the use of multiple processing cores and the improved cache hit rate. For embarrassingly parallelizable applications, we cannot benefit from the data-driven execution and thread-level parallelism of PJava, because no dependency and communication exist at all among parallel tasks. However, it is interesting to see that the performance of the PJava code was constantly better than that of the JOPI code in Figure 6. In the JOPI implementation of matrix multiplication, the computation of a process will not start until all required data is received. In this matrix multiplication application, since more data is involved in the computation (think of $C = A \times B$), the cost of dispatching offset the benefit gained from parallel computation as the number of cluster nodes increased, leading to the weak scalability of JOPI. On the contrary, each parallel task in PJava is autonomous, so a task can be started whenever it is received by a node.

Conclusion and Future Work

This paper has proposed a novel pattern- and object-based framework for code parallelization on clusters. This framework utilizes an pattern-based approach to help find the parallelization solution based on past expertise and out-

puts the parallel code in the form of autonomous objects. The execution of these objects is guided by a dependency graph in a data-driven manner. The advantages of this proposed framework were evaluated through two case studies.

At the current phase, the number of parallelization cases in the pattern base is still limited and to be expanded. Using this framework to tackle task parallelism is another research goal in our next phase. Finally, we also plan to build parallelizers to parallelize FORTRAN code and C code since they are more widely used in scientific applications.

Acknowledgment

The author wants to thank Dr. Hong Jiang and Dr. Leen-Kiat Soh for their advise in both the design and the implementation of this framework. This work was supported in part by an NSF-SBIR Grant (DMI-0441249), an NSF-MRI Grant (0320889), and a Cyberinfrastructure Research Development Grant (UNL).

References

[1] *GRID COMPUTING: Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid. Springer,, 2006.

[2] http://www-unix.mcs.anl.gov/mpi/mpich/faq.htm, April 2007.

[3] http://www.sac-home.org/, April 2007.

[4] http://www.mactech.com/articles/mactech/Vol.10/10.11/PrographCPXTutorial/, April 2007.

[5] http://www.ni.com/labview/, April 2007.

[6] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. JOPI: A Java object-passing interface. *Concurrency and Computation: Practice and Experience*, 17(7-8):775–795, 2005.

[7] Y. Aridor, M. Factor, and A. Teperman. cJVM: a single system image of a JVM on a cluster. In *Proceedings of 1999 International Conference on Parallel Processing*, pages 4–11, Fukushima, Japan, September 1999.

[8] R. Bjornson and A. Sherman. Grid computing & the Linda programming model – an alternative to web-service interfaces. *Dr. Dobb's Journal*, September 2004.

[9] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, California, CA, USA, June 2001.

[10] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[11] P. Charles. *A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York University, May 1991.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press, 2001.

[13] J. Frueheis. Planning considerations for multicore processor technology. www.dell.com/downloads/global/power/ps2q05-20050103-Fruehe.pdf, January 2005.

[14] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[15] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of 1994 Winter Usenix Conference*, pages 115–131, San Francisco, California, USA, January 1994.

[16] X. Liu, H. Jiang, and L.-K. Soh. A distributed shared object model based on a hierarchical consistency protocol for heterogeneous clusters. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 515–522, Chicago, IL, USA, April 2004.

[17] X. Liu, H. Jiang, and L.-K. Soh. Exploiting the advantages of object-based dsm in a heterogeneous cluster environment. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Cardiff, UK, May 2005.

[18] M. W. MacBeth, K. A. McGuigan, and P. J. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *Proceedings of 8th Annual IBM Centers for Advanced Studies Conference*, pages 40–54, Missisauga, Ontario, Canada, December 1998.

[19] Message Passing Interface Forum (MPIF). *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2.5 edition, May 2005.

[21] S. Pal and S. Shiu. *Foundations of Soft Case-Based Reasoning*. Wiley-Interscience, 2004.

[22] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[23] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[24] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Distributed shared memory management for Java. In *Proceedings of 4th Annual Conference of the Advanced School for Computing and Imaging*, pages 256–264, Lommel, Belgium, June 2000.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd IEEE Annual International Symposium on Computer Architecture*, pages 24–36, Portofino, Italy, June 1995.

[26] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency – Practice and Experience*, 9(11):1213–1224, 1997.