2005

DCDP: A Novel Data-Centric and Design-Pattern Based Approach to Automatic Loop Transformation and Parallelization for A Shared-Object Environment in Clusters

Xuli Liu
University of Nebraska-Lincoln, xuliu@cse.unl.edu

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Leen-Kiat Soh
University of Nebraska, lsoh2@unl.edu

University of Nebraska-Lincoln, Computer Science and Engineering
Technical Report # TR-UNL-CSE-2005-0009

DCDP: A Novel Data-Centric and Design-Pattern Based Approach to Automatic Loop Transformation and Parallelization for A Shared-Object Environment in Clusters

Xuli Liu, Hong Jiang, Leen-Kiat Soh
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{xuliu, jiang, lksoh@cse.unl.edu}

Abstract

Most of the parallelism associated with scientific/numeric applications exists in the form of loops, and thus transforming loops has been extensively studied in the past, especially in the areas of programming languages and compiler designs. Almost all the existing transformation approaches are control-centric, in which the transformation process starts from partitioning the iteration space, followed by the decomposition of the data space only as a side-effect. Originally designed for shared-memory multi-processors, these control-centric approaches might not be suitable under some circumstances for current loosely-coupled clusters and the Grid with physically distributed memories. In this paper, we introduce a novel data-centric and design-pattern based approach called DCDP to transform loops and automatically generate parallel code to execute on clusters. DCDP partitions the data space first, and then generates the processing code for each data partition, so as to minimize data communication and synchronization among cluster nodes. For the sake of generating more efficient parallel code, DCDP incorporates the notion of design pattern popularly used in software engineering into the field of parallel compiler. Instead of generating MPI-like code, DCDP outputs self-contained objects to distribute and execute on cluster nodes, thus exploiting the advantages of object-oriented programming. In addition, a feedback mechanism is employed by DCDP to gather and analyze dynamic runtime environment information to further optimize the parallelization process. To evaluate the feasibility and advantages of DCDP, we have designed and implemented a proof-of-concept compiler called PJava, and compared the performance of the code generated automatically by PJava to that of the handcrafted JOPI (Java Object-Passing Interface, a Java dialect of MPI) code on the benchmark application – matrix multiplication. Experiments show that the PJava-generated code achieves comparable performance to the JOPI code.

Keywords: Parallel Compiler, Shared Object, Data Centric, Design Pattern, Object Oriented, Loop Transformation,

Adaptive

Technical Areas: Cluster Computing, Compiler

Corresponding Author's E-Mail Address: xuliu@cse.unl.edu

1. Introduction

Parallel computing is widely used by scientists and engineers to solve problems in areas as diverse as galactic evolution, climate modeling, aircraft design, and molecular dynamics [32]. On clusters, programmers by and large handcraft parallel code using existing programming languages, e.g., C, C++, Fortran, and Java, and the inter-node communication and synchronization are being tackled with such subroutine libraries as MPI [29] [30], Linda [6], and JOPI [1]. With the access to low-level functions, programmers are able to produce highly efficient parallel programs, and these parallel programs are portable to a variety of distributed systems. However, these programmers have to deal with everything involved, including the design of the parallel algorithm, data/computation decomposition and mapping, communication and synchronization among processes, and data reduction.

To relieve MPI programmers of the complex parallel algorithm design and the tedious message passing management, parallel compilers, e.g., HPF (High Performance Fortran) [14], SUIF (Stanford University Intermediate Format) [20], Fx [34], Polaris [7], Jasmine [35], and zJava [9], have been developed during the past decade or so to facilitate parallel programming. The bulk of the parallelism inherent in numeric/scientific applications takes the form of loops. Thus, most parallel compilers aim at transforming the loops to obtain maximum parallelism and data locality. In the control-centric transformations that most of the existing parallel compilers employ, the transformation of a loop starts from partitioning the iteration space, which is followed by the decomposition of the data space only as a side-effect of the control flow manipulation. It is well known that data communication among cluster nodes is very expensive, compared to that on shared-memory multi-processor computers, and therefore it seems advantageous to directly orchestrate the data movement rather than as a side-effect of the control flow manipulations [16]. That is, in a data-centric approach, the data space is decomposed first, and then the processing code for each data partition is generated accordingly. Consequently, if the data space is optimally partitioned, the data communication among cluster nodes can be minimized, thus achieving improved data locality and performance.

However, data-centric transformations are nontrivial, and a few challenging issues must be addressed, including (1) effectively investigating the data dependence carried by loops (e.g., an array element is updated in one iteration and accessed in another), (2) properly partitioning the data space based on the data dependence, thus reducing the possible data communication among cluster nodes, (3) transforming the original loop construct and generating efficient processing code for each data partition, and (4) designing a proper middleware in support of running the code generated by a parallel compiler. Although the data dependence problem has been extensively studied (e.g., GCD Test [37],

Banerjee Test [37], I-Test [18], Range Test [8], Omega Test [31], NLVI-Test [19]), the other three issues have not been systematically addressed so far since the research on data-centric parallel compilers is still relatively new.

In this paper, we introduce a novel data-centric and design-pattern based approach called DCDP to perform loop transformations, and present the design and implementation of a proof-of-concept parallel compiler called PJava. In DCDP, the data dependence is categorized as *free*, *partially constrained*, and *constrained*, based on which the data space is partitioned accordingly. Design patterns have long been used in the area of software engineering to speed up the development process by providing tested and proven programming paradigms. Considering the complexity of the loop construct and the data dependence, DCDP borrows the notion of design pattern to support the generation of efficient code for each data partition. Rather than generating MPI-like code, DCDP encapsulates the data partitions and their corresponding processing code into objects (hereafter referred to as shared-objects for short) that are to be distributed to and executed on cluster nodes. To support the execution of these automatically generated shared-objects on cluster nodes, instead of providing a global address space as in traditional high-performance computers, we designed and developed an agent-powered middleware system called DSO-SP (Distributed Shared-Object Support Package) [23]. Sitting on top of JVM (Java Virtual Machine), DSO-SP deploys a daemon process on each cluster node to receive shared-objects and then to execute appropriate encapsulated functions by starting threads. Furthermore, software agents are used by DSO-SP to ensure the consistency of data replicas and to collect some of the runtime parameters, e.g., the CPU usage and the access patterns upon shared-objects, which are fed back to PJava, the proof-of-concept implementation of DCDP, for the purpose of parallel code optimization. PJava and DSO-SP can be easily applied onto clusters and potentially the Grid because of the simple yet effective agent-powered framework of DSO-SP and the object-form of parallel code generated by PJava.

The rest of this paper is organized as follows. Section 2 presents the related work in parallel compiler designs in general and loop transformations in particular. We overview the framework of DCDP in Section 3, and then discuss its PJava implementation and its runtime support environment – DSO-SP in Section 4. The details of data space decomposition and design-pattern based parallel code generation are introduced respectively in Sections 5 and 6. To evaluate the feasibility and advantages of DCDP, the PJava-generated code (hereafter referred to as PJava code) is compared in performance to the corresponding JOPI code on the benchmark application – matrix multiplication in Section 7. Finally, we conclude the paper and discuss possible future work in Section 8. Throughout this paper, the terms of *distributing* and *mapping* are interchangeable, and so are *partitioning* and *decomposing*.

2. Related Work

The bulk of the parallelism inherent in numeric/scientific applications is found in loops, and thus the research on parallel compilers has focused mainly on loop transformations. The data dependence carried by loops makes loop

transformation a non-trivial task. Therefore, a parallel compiler must first analyze the loops and the array references that they contain, proving or disproving the existence of data dependence. The GCD and Banerjee tests [37] are the earliest standard array subscript dependence tests to determine whether loops may be parallelized or vectorized. I-Test [18] extends both the range of the applicability and the accuracy of the GCD and Banerjee tests by refining a combination of them. All the above data dependence tests can only be applied to the cases where the array subscripts and the loop bounds are linear functions of the loop indices. For the other cases, the Range test [8], Omega test [31], and NLVI-Test [19] are available options. The Range test proves independence by determining whether certain symbolic inequalities hold for a permutation of the loop nest, and the Omega test determines data dependence in terms of systems of constraints. Based on the theory of variable intervals for nonlinear functions, the NLVI-Test is able to prove or disprove dependences in the presence of non-linear expressions, complex loop bounds, arrays with coupled subscripts, and if-statement constraints. With accurate results from the data dependence analysis, a parallel compiler should be able to carry out the loop transformation correctly and optimally.

In the past ten years, computer scientists and engineers have successfully developed a number of parallel compilers that transform loops in sequential programs to subsequently generate parallel code accordingly. As the de facto standard of data parallelism (i.e., the data set is partitioned and mapped onto multiple processors), HPF [14] provides a group of statements and constructs (e.g., *FORALL*) to indicate parallelism and uses directives (e.g., *!HPF$ DISTRIBUTE*) to handle data mapping and synchronization. HPF does take some burden off the shoulders of programmers; however, it still needs the active involvement of the programmers, e.g., requiring the programmers to determine the data decomposition scheme. In addition, HPF was primarily designed to express fine-grained data parallelism, and it is not convenient to deal with coarse-grained data parallelism. Noticing that the size of the data is sometimes not big enough to be partitioned, Fx [34] extends HPF with a couple of new directives to support task parallelism (i.e., different processors run different code or task). To fully relieve programmers of the parallelization process, the Polaris compiler [7] takes a sequential Fortran program as input and transforms it into one of the several possible parallel Fortran dialects, e.g., HPF. Using Polaris as the infrastructure, the Jasmine compiler [35] adds new loop transformations, aiming at enhancing cache and memory locality while preserving existing parallelism in programs. SUIF [13] is another successful compiler that automatically transforms a sequential scientific program into parallel code for scalable parallel machines. SUIF also provides an interactive interface to get programmers involved in the process of parallelization so as to maximize parallelism while minimizing communication [20]. The parallelization approach used by SUIF is called affine partitioning and handles only affine array accesses [22]. Since Java is starting to play a more and more important role in the parallel programming world due to its portability, researchers have started to develop parallel Java compilers. For example, zJava [9] investigated the automatic parallelization technology for Java programs that use pointer-based data structures (e.g., linked list and trees) and recursion, and proposed a novel

combined compile-time/run-time approach. Instead of targeting high-performance parallel computers with a global address space or traditional shared-memory multi-processors as most of the current parallel compilers do, our PJava compiler generates parallel code for a shared-object runtime environment (to be elaborated in Section 4.4) that is compatible with clusters and potentially the Grid. Moreover, PJava uses a novel data-centric and design-pattern based approach to help generate more efficient parallel code considering the complexity of the loop construct in sequential code and the high communication cost on clusters.

Presently, control-centric approaches are used to transform loops in most parallel compilers. In these approaches, the parallelization process starts from partitioning the iteration space, and the data space is decomposed as a side-effect. In cluster computing, the cost of inter-node communication is very high, compared to tightly-coupled multi-processors, and thus massive data communication can be detrimental to the performance of these parallel programs. Without the presence of physical shared-memory in clusters, the control-centric approaches are either difficult to apply or simply inefficient under some circumstances. In recent years, data-centric loop transformations have been used to improve data locality for several classes of applications. For example, a parallel compiler proposed by Sahoo et al. performs data-centric transformations on the XQuery source code for applications with non-integer iteration spaces [33]. In these data-centric transformations, the data space is split into chunks. After a chunk is loaded into memory, the program fragments or iterations of a loop that need this data chunk is executed. The efficiency of this approach has been demonstrated via experiments conducted on parallel computers. As another example, the research of Low et al. [26] showed that loop-based dense linear algebra algorithms could be systematically derived from the mathematical specification of the operation. Although aiming at generating parallel code for SMPs, this research used a data-centric approach to split matrices and to generate corresponding tasks. Furthermore, these data-centric strategies are also employed by sequential compilers. For example, the SGI MIPSPro compiler successfully employs a technique called data shackling [17] to improve the performance of sequential programs. In data shackling, the order of traversal through the data structures of a program is fixed, and the computations to be performed are determined when a data item is touched, thus achieving an improved performance thanks to the enhanced data locality. Although our PJava compiler also uses a data-centric approach, it is distinguished from the other data-centric approaches by having the following three features. First, design patterns are used to help generate more efficient code for data partitions. Second, the output of PJava consists of self-contained objects, which are appropriate for execution on the shared-object runtime environment. Finally, an adaptive framework is employed by PJava to facilitate the parallelization process.

In control-centric parallel compilers, effectively maximizing parallelism through loop transformations has been one of the most active research areas. Loop permutation [2][37], a process of switching inner and outer loops, is a well-know approach to obtain coarser parallelism granularity. In the case where effective parallelism cannot be obtained by solely applying loop permutation, loop reversal [37], loop skewing [37], loop fission [2][28] (e.g., dividing the state-

ments in one single loop into separate loops), and statement reordering [4][5] are used to facilitate the parallelization process. Wolfe et al. proposed an approach called unimodular transformation [39] to arbitrarily combine loop permutation, reversal, and skewing together. Unimodular transformation transforms the original loop nest into a canonical form, i.e., a fully permutable loop, and thereafter appropriate coarse- or fine-grain parallelism can be exploited. In the unimodular transformation, a loop transformation can be modeled as elementary matrix transformations, and combinations of these transformations can be simply represented as products of the elementary transformation matrices. In fact, some of these loop transformation techniques, e.g., loop fission, can be appropriately applied to data-centric parallel compilers.

To further improve the performance of a parallel program, traditional control-centric parallel compilers take into consideration a number of factors, including data locality, synchronization, and data communication. Due to the hierarchical memory architecture of modern computers, effectively reusing data can dramatically improve the performance of applications. Therefore, blocking/tiling [15][36][38] is widely used in the parallelization of nested loops. Blocking/Tiling breaks the iteration space defined by the loop structure into blocks or tiles, which leads to a high degree of data reuse across several loops and better performance for register, cache, or memory hierarchy. Furthermore, block/tiling can also reduce the communication cost since less data is passed during communication. It is important to notice that block/tiling is different from the technology used in data-centric approaches as the former's data blocking is only a side-effect of the control flow manipulation. By merging the loop bodies of multiple loops into one single loop, loop fusion [2][28][37] eliminates unnecessary barrier synchronization and reduces the communication of shared data among loops. Fusion can also enhance locality by reducing the time between uses of the same data, thereby increasing the likelihood of the data being retained in the cache [27]. Loop alignment [2] is an approach to change the index of an array in a statement so as to align it with other statements, by which a synchronization-free parallel loop could probably be obtained. Affine partitioning [21] is a program transformation framework proposed to minimize communication by favoring near-neighbor communication over data restructuring and by using pipelined parallelism if necessary. Further, all possible combinations of the above proposed loop transformations can be expressed as affine transformations. Realizing that decompositions of data space and of iteration space are inherently tied together and should be solved at the same time, Anderson et al. studied the problem of assigning an iteration and the data that it accesses onto the same processor to avoid data communication after the separate decompositions of the data space and the iteration space [3].

3. The Framework of DCDP

The framework of DCDP, as shown in Figure 1, is characterized by features of *data-centric task partitioning*, *design-pattern based parallel code generation*, *object-form of compiler output*, and *adaptivity through feedback*. In

what follows, these four features are briefly introduced respectively.

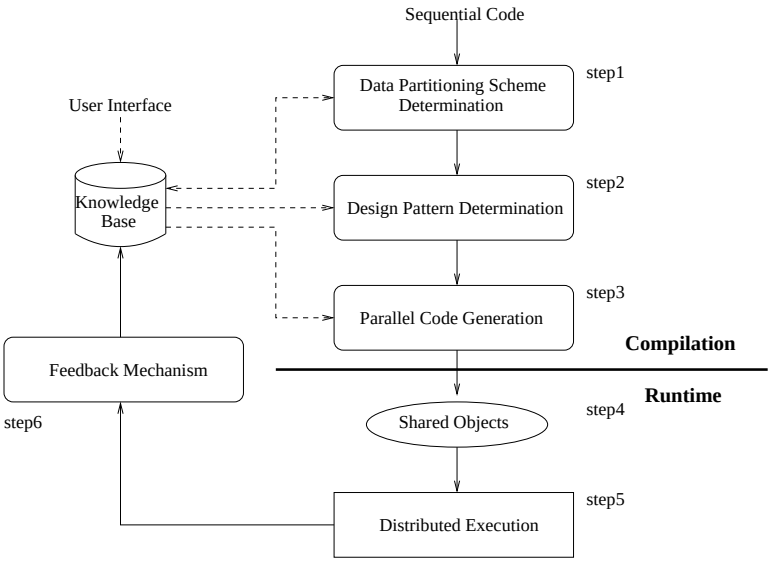

Figure 1. The framework of DCDP

3.1. Data-Centric Task Partitioning

As shown in Figure 1, DCDP employs a data-centric approach to produce parallel tasks for a cluster – Taking a sequential program as input, DCDP first determines the appropriate data space partitioning scheme, conforming to which the processing code for each data partition is generated, thus better reducing the data communication and synchronization among cluster nodes as compared to traditional control-centric approaches. The details of data space partitioning are illustrated in Section 5.

3.2. Design-Pattern Based Parallel Code Generation

Design patterns are used by DCDP to facilitate the generation of parallel code – As shown in Figure 1, DCDP chooses an appropriate design pattern in step 2 before starting the parallel code generation process in step 3. Design patterns were originally used in the field of software engineering to speed up the development process and to help prevent subtle issues that may cause major problems by providing tested and proven programming paradigms. Considering that loops in a sequential program share some common features, it is possible to determine an appropriate parallelization scheme, i.e., design pattern, by abstracting and analyzing these features, thus easing the work of parallel code generation and improving the stability and hopefully the efficiency of the generated code. In DCDP, the loop construct is abstracted, and the abstraction, combined with the data space partitioning scheme, is used to retrieve

an appropriate design pattern from the knowledge base. As more design patterns are being accumulated and consolidated in a growing knowledge base, more and more types of sequential programs are likely to become parallelizable automatically. More details about the parallel code generation can be found in Section 4.2 and Section 6.

3.3. Object-Form of Compiler Output

Although MPI is currently the dominating parallel programming paradigm on cluster computers, the output of the DCDP compiler consists of self-contained objects called shared-objects (as shown in step 4 of Figure 1) rather than an MPI-like program. This object-form of output brings in a number of advantages. First, only related data is encapsulated into the same shared-object, thus false-sharing is reduced. Furthermore, the size of the shared-object, which can be viewed as granularity of parallelism, can be adapted to the variance of runtime environment factors, e.g., CPU and network usage, and this issue has been investigated in one of our previous studies [24]. Third, encapsulating corresponding code together with the data in a shared-object greatly simplifies the runtime environment: a virtual global address space is no longer needed, and only a few software agents reside on cluster nodes to receive shared-objects and then to execute the corresponding encapsulated functions by threads (refer to Section 4.4 for details). Finally, in terms of implementation, common functions can be pre-implemented in base classes and inherited when generating these shared-objects, thus reducing the workload of parallel code generation. An inside look of a shared-object is presented in Section 4.3.

3.4. Adaptivity through Feedback

During the execution of shared-objects in step 5 of Figure 1, the runtime environment information is gathered and analyzed, and then fed back to the knowledge base through the feedback mechanism shown in step 6 of Figure 1. By feedback, an optimal shared-object size can be chosen for future runs, adapting to the variance of runtime environment. Furthermore, a more appropriate protocol can be set for each shared-object, based on the access pattern occurred on each, to maintain the consistency among data replicas. The feedback mechanism will be further discussed in Section 4.4.

4. The Implementation of DCDP: An Overview of PJava and DSO-SP

To evaluate the framework of DCDP, we implemented a proof-of-concept parallel compiler called PJava, and built a runtime support environment called DSO-SP to execute the parallel code automatically generated by PJava. PJava aims at parallelizing loops of a sequential Java program and partitioning the task of each loop into multiple smaller sub-tasks in the form of shared-objects. PJava is implemented using C++ on top of Jikes [11], an OSI certified open

source Java compiler. In this section, we first overview how a sequential Java program is transformed and executed in parallel, and then illustrate the work flow of PJava in details. After examining the internal structure of a shared-object produced by PJava in runtime, we introduce its runtime support environment – DSO-SP.

4.1. From Sequential to Parallel

Figure 2 shows how a sequential Java program named *Foo.java*, in which there exists at least one loop, is transformed by PJava and executed in parallel with the support of DSO-SP.

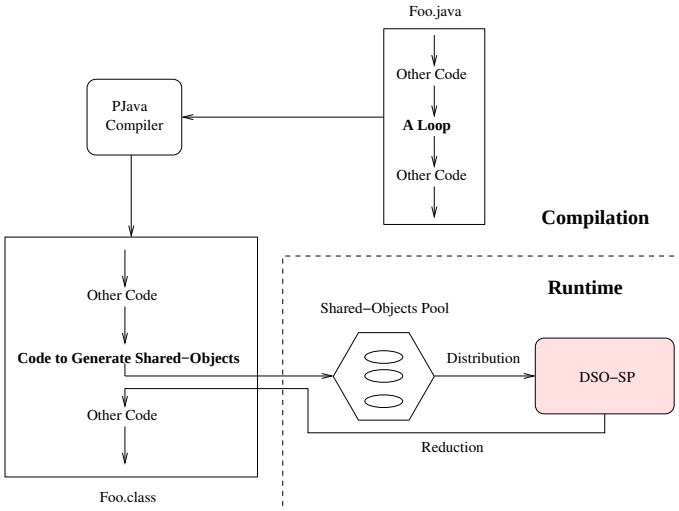

Figure 2. Transforming and executing a Java program

The PJava compiler takes *Foo.java* as input, transforms all its loops, compiles the transformed code (i.e., PJava Code), and outputs the class file – *Foo.class*. As mentioned in Section 3, PJava generates shared-objects rather than MPI-like code. However, since the data may still be unknown during the time of compilation, PJava actually produces the code that dynamically generates shared-objects during runtime. *Foo.class* is scheduled and executed as the master thread on a single cluster node. When arriving at the shared-object generation phase, *Foo.class* creates these shared-objects, and then distribute and execute them on multiple cluster nodes. In the meantime, the master thread simply waits until all subtasks are finished, and then continues to execute the code that follows.

4.2. The Work Flow of PJava

The transformation from *Foo.java* to *Foo.class* is a nontrivial task. PJava uses a novel data-centric and design-pattern based approach to transform and compile regular sequential Java programs, and the work flow of this process is depicted in Figure 3.

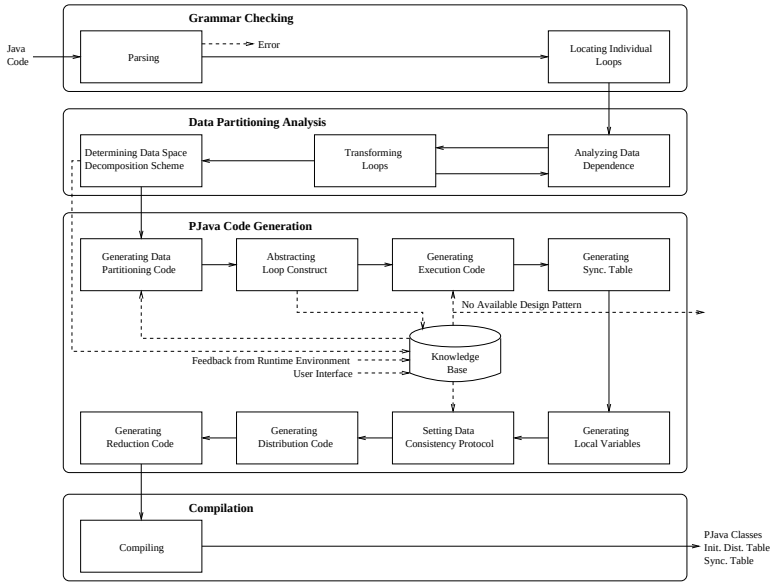

Figure 3. Overview of the work flow of PJava

As shown in Figure 3, the work flow of PJava can be divided into four phases, which are *grammar checking*, *data partitioning analysis*, *PJava code generation*, and *compilation*. In the phase of *grammar checking*, the sequential Java code is first parsed by PJava as in normal Java compilers. If there exists any grammar error, PJava stops the compilation process and returns the control to the programmer. Otherwise, PJava locates all individual loops, parsing and storing them into a data structure for further processing. For the non-looping sequential code, PJava simply replicates it to the PJava code.

After checking the grammar of the Java code, the work flow of PJava goes to the phase of *data partitioning analysis*, which corresponds to step 1 of the DCDP framework shown in Figure 1. In this phase, the data dependence carried by each loop is first proved or disproved. In the case of data dependence, the type of the data dependence is identified in order to obtain an optimal data decomposition scheme. Notice that there is a functional module called *transforming loops*, which borrows some loop transformation techniques (e.g., loop fission and loop fusion) in traditional control-centric transformations to enhance parallelism. This module is not yet implemented in PJava.

The third phase is about *PJava code generation*, which corresponds steps 2 and 3 of the DCDP framework in Figure 1 and tackles the most challenging problem – parallel code generation. In this phase, the data partitioning code is first generated based on the decomposition scheme obtained from the previous phase. And then, the loop construct is analyzed, and the analysis result, combined with the data space decomposition scheme, is passed to the knowledge base to retrieve a proper design pattern. If a proper design pattern is located, it is used by PJava to generate the processing code for each data partition; otherwise, an error message is returned to the programmer, and the automatic

parallelization fails. To ensure the correctness of the PJava code, the synchronization problem needs to be taken care of. That is, before a shared-object can be processed, it needs to wait until all the data that it depends on is ready. As a by-product of the generation of the processing code, a synchronization table is built. Instead of being encapsulated within the shared-objects to be produced, this synchronization table is passed to DSO-SP for the purpose of synchronization control.

Considering that the code inside a loop may refer to some variables defined outside the loop, corresponding variables need to be defined in the PJava code and to be initialized during runtime. To improve data locality, PJava supports data replication. Data consistency is ensured by DSO-SP, and the appropriate consistency protocol to be used is retrieved from the knowledge base and specified in the PJava code. Notice that the knowledge base exposes two interfaces to the user and the runtime environment respectively. Through the user interface, design patterns can be supplemented; and by analyzing the feedback (e.g., access patterns upon the shared-objects) from the runtime support system, an optimal shared-object granularity and an appropriate consistency protocol can be determined (refer to Section 4.4 for details). PJava now needs to generate the distribution code that maps the sub-tasks, i.e., shared-objects, onto cluster nodes for execution, and to build the reduction code that collects the results from these cluster nodes. The guideline for distributing shared-objects is to maximize the utilization of each cluster node and to minimize the inter-node communication. Finding an optimal solution to the distribution problem is NP-hard [12], and is beyond the scope of this paper. In the implementation of PJava, these shared-objects are simply distributed based on the power of each cluster node. Owing to the inheritance feature of object-oriented programming, a lot of common functions have been pre-implemented in base classes, and PJava simply inherits these functions when generating the PJava code, thus greatly simplifying the work of PJava.

Finally, the PJava code is compiled into the bytecode format, and is to be executed on cluster nodes with the support of DSO-SP, as shown in Figure 2. As by-products, the intitial distribution table and synchronization table are also produced. Among all the functional modules shown in Figure 3, determination of the data space partitioning scheme and generation of the processing code for each data partition are the keys to the success of PJava, and will be further described in Sections 5 and 6 respectively.

4.3. The Internal Structure of A Shared-Object

To exploit the power of object-oriented programming and simplify the runtime support environment, PJava outputs self-contained objects, i.e., shared-objects, rather than MPI-like code, as shown in step 4 of the DCDP framework in Figure 1. In this sub-section, we take a look into the inside of a shared-object, as shown in Figure 4.

Figure 4 shows that the data and its corresponding functions are encapsulated together. The data items include the data partition (data block) to be processed, the variables defined outside of the loop in the sequential Java code

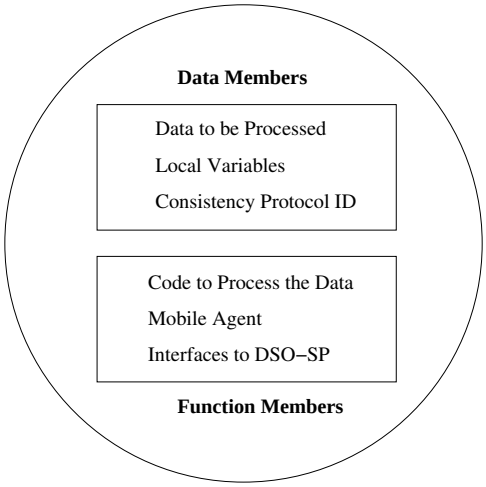

Figure 4. The internal structure of a shared-object

(refer back to Section 4.2 for details), and the identifier of the consistency protocol to be applied. A set of consistency protocols is implemented by DSO-SP (refer to Section 4.4 for details), and this consistency protocol identifier simply specifies which protocol to use, thus enabling different objects to apply different consistency protocols or even different consistency levels [10] under certain circumstances in specific applications. In addition to the data processing code, the functions encapsulated in the shared-object include a mobile agent and interfaces to DSO-SP. The mobile agent keeps track of the data access patterns that can be fed back to and used by PJava to choose a more appropriate consistency protocol for this shared-object for future runs. When PJava generates the data processing code for each shared-object, it does not need to consider the data replication and consistency as long as it operates on these shared-objects through the interfaces exposed by DSO-SP, which handles the data replication and consistency behind-the-scene.

4.4. The Runtime Support Environment: DSO-SP

To support processing the shared-objects generated by PJava, a runtime support environment called DSO-SP has been designed and implemented, corresponding to steps 5 and 6 of the DCDP framework shown in Figure 1. DSO-SP employs an agent-powered framework, as shown in Figure 5, to handle the communication and synchronization among shared-objects, to manage data replication and its consistency behind-the-scene, and to help PJava make better compilation plans through the feedback mechanism.

As illustrated in Figure 5, a software agent resides on each cluster node, receiving tasks (shared-objects), the initial distribution table, and the synchronization table from the PJava parallel compiler. After receiving a shared-object, the agent spawns a work thread to execute the encapsulated data processing code. At the same time, a couple of service threads are started to respond to the requests from other agents. A shared-object distribution table is maintained by

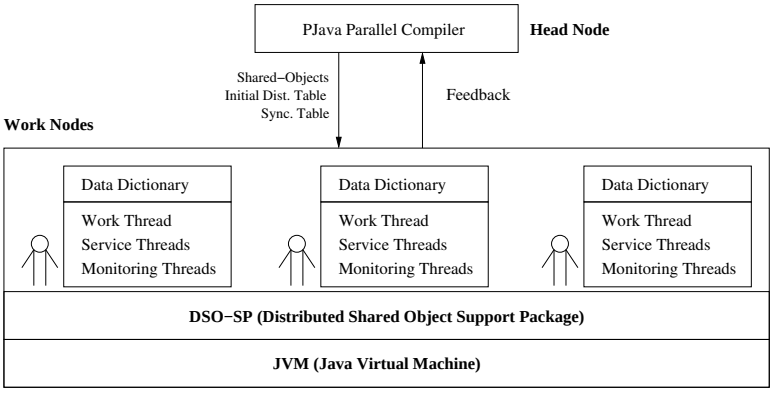

Figure 5. DSO-SP – A middleware in support of executing shared-objects

DSO-SP for the purpose of data replication and consistency control. To coordinate the execution of these shared-objects, the synchronization table that contains a set of barriers plays the role of responding to synchronization calls in the parallel code. Furthermore, a monitoring thread is running to keep track of the variance of the runtime environment, e.g., CPU usage, and this information is fed back to and used by PJava to determine the optimal shared-object size when partitioning the data space [24] for future runs. In addition, the mobile agent integrated within the shared-object is triggered to track the access pattern whenever this shared-object is read/written, and this access pattern can be used by PJava to choose a more appropriate consistency protocol for this object's future runs. A data directory is also maintained by the agent, and it contains not only the data being processed by this agent but also replicas of the data owned by other agents. The consistency among data replicas is ensured behind-the-scene by DSO-SP that sits on top of JVM in favor of program portability. In PJava's current form, the mobile agent module has not yet been fully implemented.

5. Data Space Partitioning

The goal of the data-centric transformation is to better reduce possible data communication than control-centric approaches do, so studying data space partitioning schemes has been one of the research foci of DCDP. In this section, we first introduce the data dependence analysis in DCDP, and then describe the guidelines for data space partitioning.

5.1. Data Dependence Categorization

Since data dependence has been well studied [37] [18] [8] [31] [19], DCDP simply applies existing approaches to identify the data dependence carried by loops, and then categorizes the dependence relationships into three types – *free*, *partially constrained*, and *constrained*, which are used to determine a proper data partitioning scheme. In what follows, the definitions of these three types of data dependence are explained through the examples shown in Figures

6 and 7.

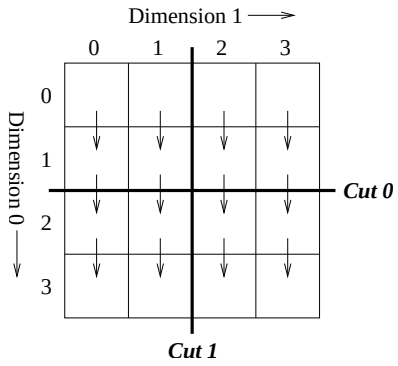

Figure 6. The data dependence on a two-dimensional array of A with size of 4×4

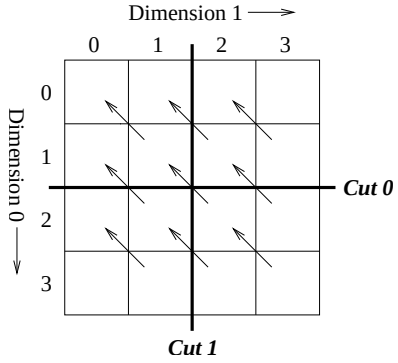

Figure 7. The data dependence on a two-dimensional array of B with size of 4×4

Figures 6 and 7 illustrate the data dependences, which are represented by arrows, on two two-dimensional arrays of A and B [1] respectively. For example, the arrow from $A[0][0]$ to $A[1][0]$ in Figure 6 means that the computation of $A[1][0]$ refers to (or depends on) the value of $A[0][0]$. In the following discussion, we assume that the iteration space walks through the array items in Figures 6 and 7 from lower index to upper index on each dimension. The three data dependence types are defined as

- ***free.*** As shown in Figure 6, the computation of any array item $A[i_1][j_1]$ does not depend on any other items $A[i_2][j_2]$, where i_1 can be either equal to or not equal to i_2 and $j_1 \neq j_2$. That is, splitting A on dimension 1, i.e., *Cut 1*, will not break any data dependence relationship, thus causing no inter-partition data communication and synchronization. Therefore, we say that A is *free* on dimension 1 and denote it as $\mathcal{F}_1(A)$.

- ***constrained.*** In Figure 6, some array items (e.g., $A[1][0]$) depend on other items (e.g., $A[0][0]$) on dimension 0. Consequently, the data dependences will be broken if a data partitioning is conducted on dimension 0.

[1] Note that two-dimensional arrays are used only for the sake of simplicity as the following definitions also apply to arrays of higher or lower dimensionality.

For example, Figure 6 shows that some data dependences cross the partitioning of *Cut 0*. Obviously, data communication and synchronization will be induced by *Cut 0* among partitions in this case, and we say that A is *constrained* on dimension 0 and denote it as $\mathcal{C}_0(A)$.

- ***partially constrained.*** In the scenario illustrated in Figure 7, data dependences exist on both dimensions 0 and 1. For example, the computation of $B[1][1]$ refers to $B[2][2]$, and this dependence will be broken by either the partitioning of *Cut 0* or *Cut 1*, thus causing inter-partition data communication and synchronization. Fortunately, this dependence can be eliminated by replicating the data partition that includes $B[2][2]$ and putting it together with the partition that holds $B[1][1]$, since the computation of $B[1][1]$ happens before that of $B[2][2]$ and $B[1][1]$ only needs the old value of $B[2][2]$ rather than the new one. Furthermore, the cost of this data replication can be hidden by overlapping with computation. Here, we say that B is *partially constrained* on dimensions 0 and 1 and denote them as $\mathcal{PC}_0(B)$ and $\mathcal{PC}_1(B)$ respectively.

5.2. Guidelines for Data Space Partitioning

After the data dependence type upon each dimension of an array is determined, the array to be modified can be partitioned properly following the rules presented in Figure 8. The partitioning of the arrays that are only referenced (or dependent upon) by an array that is to be modified is determined by how the former are referenced by the latter and how the latter is partitioned. Assume, for example, that array A is to be modified and refers to another array B. If the decomposition of A is on dimension 0, which corresponds to dimension 1 of B, e.g., $A[i][j] = f(B[k][i])$ where f represents a function, then we partition array B on dimension 1, thus avoiding data communication by putting inter-related partitions together. Note that not only is the data itself to be encapsulated into a shared-object, but also the features of a data partition, e.g., its position in the original array, will be included in support of future reduction.

---

1. If there is at least one *free* dimension, then partition this array on one or more of these free dimensions and go to step 4. Otherwise, go to step 2.
2. If at least one *partially constrained* dimension exists, then partition this array on one or more of these *partially constrained* dimensions and go to step 4. Otherwise, go to step 3.
3. Partition the data on each dimension, and go to step 4.
4. Exit.

Figure 8. Guidelines for the data space partitioning

In order to minimize the communication among data partitions, we first look for *free* dimensions. If the data is

partitioned on *free* dimensions, no communication cost will occur. In the case that no *free* dimension is available, *partially constrained* dimensions are to be used, since data communication and synchronization can be eliminated through data replication. If every dimension of the array is *constrained*, no matter how partitioning is performed, data communication and synchronization are inevitable, and thus straightforward and explicit parallelism is not available. In this case, we partition the array on every dimension, and then try to exploit other forms of parallelism such as pipelined parallelism. More details about the data dependence analysis and the data decomposition schemes can be found in [25]. When partitioning the data space, it is desirable to use as many dimensions as possible since finer partitions not only mean more parallelism but also conform to the hierarchical memory structure of modern computers. However, finer partitions also imply more synchronization points. Therefore, the granularity of data partitions needs to be carefully determined – an optimal data partition granularity is able to greatly improve the performance of parallel programs. Readers are referred to one of our earlier studies [24], where the issue of determining the optimal granularity of shared-objects is studied.

6. The Design-Pattern Based Parallel Code Generation

After a data space partitioning scheme is determined and the data partitioning code is generated, the processing code for each data partition, hereafter referred to as parallel program/code for short, is to be generated. In what follows, we first summarize the main issues that must be addressed when generating the parallel code, and then introduce the forms of parallelism currently supported in PJava.

6.1. Main Issues in Parallel Code Generation

Parallel code generation is a non-trivial process, and in what follows, we address the main issues involved.

(1) Loop bounds

As the parallel code in the DCDP framework is to deal with a data partition rather than the whole data space, loop bounds of the iteration space obviously need to be adjusted accordingly on the partitioned dimensions. The calculation of the loop bounds for each partition is an extremely difficult task when the original loop bounds are not linear functions of the loop indices. However, it is by no means an impossible task since extensive data dependence analysis has been done for the case of nonlinear loop bounds, and we believe that such analysis can be extended for this purpose. In the current implementation of PJava, the calculation of the loop bounds has been simplified, focusing only on linear loop bounds.

(2) Structure of the iteration space

Depending on the kind of parallelism to be employed, the original structure of the iteration space may need to be reconstructed, e.g., in the case of pipelined parallelism. The new loop structure is constructed based on the design pattern being used, which will be further explained in Section 6.2.

(3) **Access frequency**

If the array items are accessed more than once in the original iteration space, a different design pattern may need to be used for the parallelism. For example, synchronization statements may need to be properly set in the parallel code (see Section 6.2 for details).

(4) **Synchronization**

In order to ensure its correctness, the parallel code needs to properly set synchronization calls. DSO-SP maintains a synchronization table that contains the prerequisites (i.e., the shared-objects to be referenced) for each shared-object. In fact, as each shared-object may be updated multiple times in its life-cycle, each version of this shared-object has a corresponding entry in this synchronization table – the process on the same version of the shared-object is blocked until all prerequisites are met.

(5) **Data consistency**

To improve data locality, the data replication and consistency control mechanisms are provided by DSO-SP. DSO-SP exposes a few primitives, e.g., *dso_read* and *dso_write*, which can be used in PJava code to guarantee that correct versions of shared-objects are processed.

6.2. Design Patterns Currently Supported

The parallel code is generated based on the framework provided by the design pattern being used, and the choice of the proper design pattern is determined by the data space partitioning scheme, the number of times that each array item is accessed, the correspondence between the iteration space and the array subscripts, and the data dependence distance[2]. In this subsection, we introduce the design patterns currently implemented in PJava from a high level, and the implementation details, e.g., their storage and retrieval, are not addressed due to the page limit and will be discussed in a future paper.

6.2.1 Communication-Free Parallelism

Communication-free parallelism is the simplest design pattern in parallel code generation and exists in many numeric/scientific applications. As described in Section 5, if *free* dimensions are available, partitioning on these *free*

[2]Dependence distance describes the difference in the loop variables between the source and sink of the dependence.

dimensions will not impose any data dependence between data partitions; in other words, it is communication- and synchronization-free. From the view of parallel code generation, the original sequential loop code can be directly used as the parallel code except that the loop bounds for each data partition needs to be properly adjusted.

6.2.2 Communication-Free Parallelism with Data Replication

In the case where no *free* dimensions are available, parallelism without communication and synchronization is still achievable if there are *partially constrained* dimensions. We can partition the data space on one or more of the *partially constrained* dimensions, distribute the data partitions to be modified and the replicas of the corresponding dependent partitions to the same cluster node. Recall that replicas are updated by DSO-SP behind-the-scene properly through the chosen consistency protocol. Of course, the loop bounds of the original sequential code need to be changed accordingly in the parallel code.

6.2.3 Pipelined Parallelism

If the array inside a nested loop has only *constrained* dimensions, this nested loop may or may not be able to be parallelized. In the current form of PJava, we only consider the case where the distances of all data dependences are equal. If the array has more than one dimension or has only one dimension but each array element is accessed multiple times, then pipelined parallelism can be applied; otherwise, no parallelism is available.

First, we demonstrate how pipelined parallelism works with a two-dimensional array A that has only *constrained* dimensions. As shown in Figure 9, the array A is decomposed into 4×4 partitions, and the data dependence between two partitions is expressed with an arrow. Note that the number of partitions on each dimension does not have to be the same.

Data partitions shown in Figure 9 can not be processed simultaneously due to dependences among them. Nevertheless, the processing of the data partitions can be pipelined along the **Time Axis** – first the partition in the row marked by step 1 is dealt with, and then the two partitions in the second row are processed, and so on. This pipelined process continues step by step until all data partitions are finished.

Obviously, the processing of these partitions needs to be carefully coordinated. As a result, in addition to transforming the original loop in the same way as in the case of *communication-free parallelism*, we set a group of *barrier_wait* statements at the beginning in order to obtain the correct versions of the shared-objects being referenced, as shown in Figure 10. After the processing code, a *barrier_reach* statement is called, informing DSO-SP that this round of processing on this shared-object is done, so as to trigger the execution of other related shared-objects.

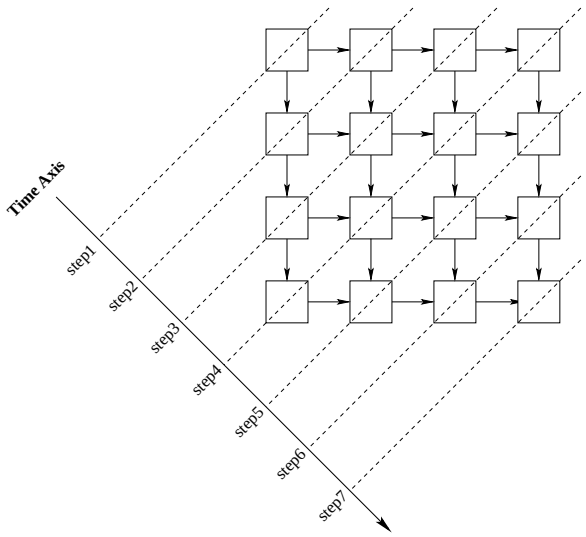

Figure 9. The pipelined parallelism on a two-dimensional array A with only *constrained* dimensions

1. barrier_wait statements;
2. parallel code corresponding to the original loop;
3. barrier_reach statement;

Figure 10. The synchronization in the parallel code of pipelined parallelism

6.2.4 Variations Caused by Multiple Accesses

In the description of the above three types of design patterns, we assume that only one access occurs on each data partition. In numeric/scientific applications, an array element is often processed multiple times inside a nested loop, and this needs to be taken into account when constructing the parallel code. For *communication-free parallelism*, no matter how many times a data partition is accessed, the parallel code is not affected. However, for *communication-free parallelism with replication* and *pipelined parallelism*, synchronization statements are needed to ensure the correctness of the parallel code, thus their design patterns need to be adapted as shown in Figure 11. Prior to and after each round of processing a shared-object, *barrier_wait* and *barrier_reach* statements are used to synchronize with other shared-objects.

With each data partition being accessed multiple times, pipelined parallelism can also be applied to a one-dimensional array with only *constrained* dimensions. In Figure 12, this one-dimensional array is decomposed into 3 data partitions. The pipelined parallelism works as follows – In step 1, only data partition 1 can be dealt with; however, in step 2, both partitions 1 and 2 can be handled as the dependence of data partition 2 on partition 1 has been met, and similar rules apply to future steps. In this case, obviously, a design pattern similar to the one shown in Figure 10 can be applied.

begin loop

1. barrier_wait statements;
2. parallel code in single-access mode;
3. barrier_reach statement;

end loop

Figure 11. The adaptation of the design patterns in multiple-access mode

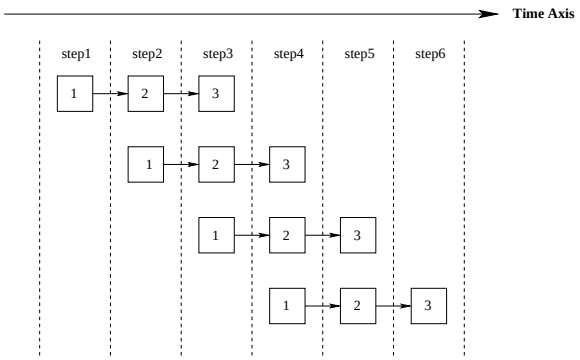

Figure 12. The pipelined parallelism on a one-dimensional array with only *constrained* dimensions

7. Performance Analysis

The design goal of our proposed data-centric design-pattern-based approach – DCDP is to reduce the data communication and synchronization cost and to best customize the parallel code to the characteristics of sequential programs. In this section, we demonstrate the feasibility of this new approach by comparing the performance of the automatically generated parallel code by PJava, a proof-of-concept implementation of DCDP, to that of the handcrafted JOPI [1] code on a benchmark application: matrix multiplication.

7.1. Experimental Setup

This performance evaluation experiment was conducted using the benchmark program of matrix multiplication on a heterogeneous cluster called *Sandhills*. *Sandhills* has 10 nodes connected by a Gigabit Ethernet, and each node has 2 AMD Athlon MP 1600+ CPUs (1.2GHz to 1.4GHz) and 1GB RAM.

7.2. Performance Evaluation of DCDP

In this subsection, the efficiency of the automatically generated PJava code is demonstrated by comparing its performance to that of the handcrafted JOPI code on the benchmark program of matrix multiplication. JOPI is a Java dialect of MPI, and the JOPI implementation of an application marks the best parallel performance that this application is able to achieve using Java, assuming that the design of the parallel algorithm is optimal.

In this implementation of the matrix multiplication, we set the size of the two matrices (filled with doubleprecision floating-point numbers) to be 2000×2000, and fixed the size of each data partition at 500×500. The sequential implementation of matrix multiplication had about 70 lines of code, the handcrafted JOPI code was 200 lines or so, and the automatically generated PJava code used around 400 lines of code. To evaluate scalability, we carried out the experiments on 1 node, 2 nodes, 4 nodes, and 8 nodes. Each experiment was conducted 5 times, and the average values are used when the PJava code and the JOPI code are compared. Figure 13 shows the comparison of PJava and JOPI in terms of execution times, and the speedup comparison is shown in Figure 14.

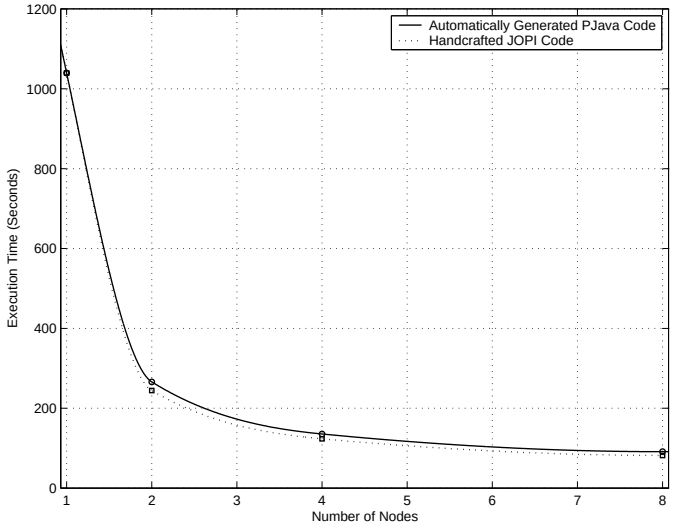

Figure 13. The comparison of PJava and JOPI in terms of execution time

Both Figure 13 and Figure 14 clearly show that the PJava code achieves very comparable, though slightly inferior, performance to the handcrafted JOPI program. The performance difference is caused by the overhead of dynamic data partitioning, data distribution, and data reduction in the PJava code. Another possible reason for the slightly inferior performance of the PJava code is that the statements generated by PJava are not as succinct as the manual JOPI code, considering that these statements execute millions of times. From Figures 13 and 14, we can see the performance is improved super-linearly from the case of 1 node to that of 2 nodes. This is because the data space is partitioned, and smaller partitions better fit into caches. Figure 14 also shows that the scalability of the PJava code is slightly worse

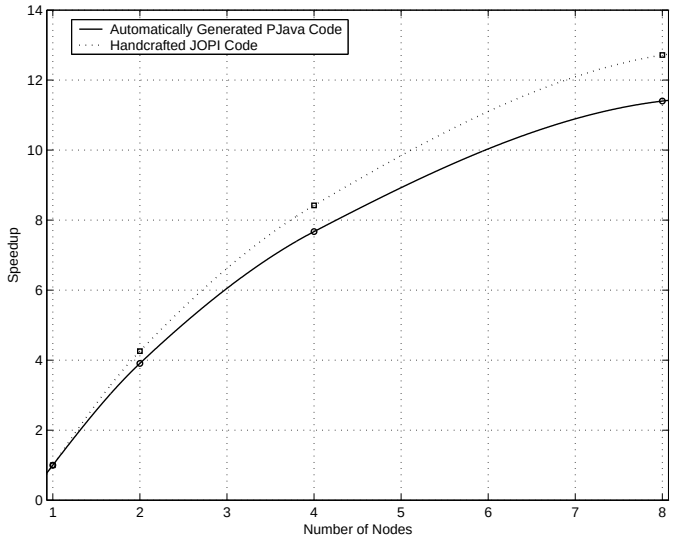

Figure 14. The comparison of PJava and JOPI in terms of speedup

than the JOPI code. Nevertheless, the experimental result is still promising considering how much burden the parallel compiler has removed from the shoulders of parallel programmers by making the parallelization and compiling process *completely automatic*.

8. Conclusion and Future Work

This paper has proposed a novel data-centric and design-pattern based framework (DCDP) for automatic parallel code generation and implemented a proof-of-concept parallel compiler called PJava and an agent-powered runtime support middleware called DSO-SP for this new framework. PJava transforms loops in a Java program and then generates small sub-tasks in the form of self-contained shared-objects to distribute and execute on a cluster. The data-centric approach used in PJava helps reduce the data communication and synchronization cost. Due to the complexity of the loop structures in applications, design patterns are used to provide a framework for the parallel code to be generated. Further, DCDP proposed a feedback mechanism to adapt the parallelization process to the runtime environment. An experiment of comparing the automatically generated PJava code to the handcrafted JOPI code was conducted using the benchmark application of matrix multiplication, validating the design of DCDP and its effectiveness.

So far, the framework of DCDP has been successfully constructed in the PJava and DSO-SP prototypes with basic but critical features. However, the implementation of some functional modules was simplified and is yet to be enhanced. For example, PJava only supports four design patterns at present, and is by no means capable of transforming all kinds of loops found in a sequential program. By applying PJava to more applications, we expect more design patterns are likely to be abstracted and then added into the knowledge base. Moreover, the calculation of the nonlinear

loop bounds needs to be studied in order to apply PJava to more complex applications. In addition, the mobile agent integrated within the shared-object needs to be researched for the sake of choosing a more appropriate data consistency protocol. As a long-term goal, we plan to extend PJava to support recursive programs and loops that use pointer-based data structures rather than arrays.

9. Acknowledgment

This work was supported in part by a NSF SBIR Grant (DMI-0441249). Dr. David Swanson and Cesar Delgado kindly provided the experimental environment.

References

[1] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. Middleware infrastructure for parallel and distributed programming models on heterogeneous systems. *IEEE Transactions on Parallel and Distributed Systems - Special Issue on Middleware Infrastructures*, 14(11):1100–1111, November 2003.

[2] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, Germany, January 1987.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM/SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, New Mexico, USA, June 1993.

[4] W. F. Appelbe and K. Smith. Determining transformation sequences for loop parallelization. In *The 5th Workshop on Languages and Compilers for Parallel Computing*, pages 208–222, New Haven, Connecticut, USA, August 1992. Berlin: Springer-Verlag.

[5] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, December 1994.

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

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[8] W. Blume and R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12), December 1998.

[9] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of java programs. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, pages 113–120, Anaheim, CA, USA, August 2001.

[10] T. Chang, G. Popescu, and C. Codella. Scalable and efficient update dissemination for distributed interactive applications. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 143–150, Vienna, Austria, July 2002.

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

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, CA, USA, 1979.

[13] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specifi cation*, January 1997.

[15] R. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM/SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California, USA, January 1988.

[16] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM/SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, Nevada, USA, June 1997.

[17] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy. In *Proceedings of the 13th International Conference on Supercomputing*, pages 482–491, Rhodes, Greece, June 1999.

[18] X. Kong, D. Klappholz, and K. Psarris. The i-test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[19] K. Kyriakopoulos and K. Psarris. Efficient techniques for advanced data dependence analysis. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 143–153, St. Louis, Missouri, USA, September 2005.

[20] S.-W. Liao, A. Diwan, R. Bosch, A. Ghuloum, and M. Lam. Suif explorer: An interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of parallel Programming*, pages 37–48, Atlanta, Georgia, USA, May 1999.

[21] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing*, pages 228–237, Rhodes, Greece, May 1999.

[22] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, Snowbird, Utah, June 2001.

[23] 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.

[24] 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.

[25] X. Liu, H. Jiang, and L.-K. Soh. Parallelizing a sequential program in a cluster environment: A novel data-centric approach. Technical Report TR05-07-03, Department of Computer Science and Engineering, University of Nebraska - Lincoln, July 2005.

[26] T. M. Low, R. A. van de Geijn, and F. G. V. Zee. Extracting smp parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 153–163, Chicago, Illinois, USA, June 2005.

[27] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[28] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 54–63, Manchester, UK, July 1994.

[29] Message Passing Interface Forum (MPIF). *MPI: A Message-Passing Interface Standard. Version 1.1*, June 1995.

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

[31] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.

[32] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, USA, June 2003.

[33] S. K. Sahoo and G. Agrawal. Data-centric transformations on non-integer iteration spaces. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 133–142, St. Louis, Missouri, USA, September 2005.

[34] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, Las Vegas, Nevada, USA, June 1997.

[35] S. Tandri and T. S. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Proceedings of the 25th International Conference on Parallel Processing*, pages 64–73, Bloomington, IL, USA, August 1997.

[36] M. E. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, Reno, Nevada, USA, November 1989.

[37] M. E. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Massachusetts, USA, March 1989.

[38] M. E. Wolfe and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM/SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[39] M. E. Wolfe and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.