A Parallel Programming Environment for a V-Bus based PC-cluster

Sang Seok Lim, Yunheung Paek and Kyu Ho Park
EECS Department
Korea Advanced Institute of Science and Technology
{sslim@core, ypaek@ee and kpark@ee}.kaist.ac.kr

Jay Hoeflinger
Kuck and Associates at Intel
jay.p.hoeflinger@intel.com

Abstract

Nowadays PC-cluster architectures are widely accepted for parallel computing. In a PC-cluster system, memories are physically distributed. To harness the computational power of a distributed-memory PC-cluster, a user must write efficient software for the machine by hand. The absence of global address space *makes the process laborious because the user must manually assign computations to processors, distribute data across processors and explicitly manage communication among processors. This paper introduces our recent three related studies on parallel computing. The first is* a V-Bus based PC-cluster *in which all PCs are interconnected through V-Bus network cards. The second is the implementation of an* one-sided communication library *on our PC-cluster system, which provides the user with a view of global address space on top of the distributed-memory PC-cluster. This encapsulation of global address allows the user to write shared memory code on our cluster system, which simplifies programming our cluster because the user does not need to explicitly cope with distributed memories. The third is* a parallelizing compiler *that automatically translates a sequential code to a shared-memory code for the PC-cluster. This compiler not only enables legacy code to be compiled for our cluster, but also further simplifies programming our cluster by allowing the user to continue using conventional sequential languages such as Fortran 77. In this work, the compiler was optimized particularly for our V-Bus based PC-cluster. The paper also reports our experimental results with the compiler on our PC-cluster system.*

1 Introduction

For several decades, multiprocessor systems that are able to utilize parallelism embedded in programs have been the most powerful machines for high-performance computing. Traditionally, supercomputers have been successfully used for parallel computing. But the costs of these supercomputers were generally too high. So, to achieve not only low cost but also high performance by parallel processing, a cluster computer with commodity hardwares has been widely adopted. The NOW and Beowulf are good example systems of cluster computers. Among the driving forces that have enabled this transition from a supercomputer to a cluster computer has been the rapid improvement in the availability of commodity high performance components for PCs, workstations and networks. This technology trend is making a network of computers(PCs or workstations) an appealing vehicle for parallel processing, and consequently leading to *low-cost commodity super-computing*.

For low cost and high performance computing, we have developed a V-Bus based PC-cluster. In the cluster, each 300MHz Pentium II PC is attached to a V-Bus network cards[11, 12, 13], and the V-Bus cards are connected to each other to form a mesh interconnection network. The V-Bus network cards are currently implemented on FPGA technology.

This paper reports our recent work on the development of a programming environment for our V-Bus based PC-cluster. The objective of our work is to provide the users with convenient and efficient software tools for programming the cluster system. To achieve this objective, we have implemented the MPI-2 library[7] and a parallelizing compiler on top of the cluster hardware.

MPI-2 is more difficult to implement than the original MPI, here we call MPI-1, but provides a more versatile programming environment by supporting not only traditional *message passing* programming but also *shared memory* programming. This is made possible by implementing both two-sided communications (SEND/RECEIVE) and one-sided communications (PUT/GET). MPI-2 is optimized to fully capitalize on the hardware features of the V-Bus network card, which will be detailed in Section 2.

Even with such strong support for parallel programming, writing parallel code for the cluster is still complex and error-prone because it requires in-depth knowledge on the underlying parallel computer system and parallelism in applications. To reduce this programming complexity, we

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

have implemented a parallelizing compiler, based on the *Polaris* infrastructure [1], originally developed by the authors and other colleagues at Illinois. Polaris automatically translates conventional Fortran 77 code to parallel code for a variety of shared memory multiprocessors. In this work, we have developed a *MPI-2 postpass* for Polaris, to retarget the compiler at MPI-2 running on our PC cluster. That is, Polaris takes sequential code as input and produces SPMD-style shared memory code with one-sided communication primitives, MPI_PUT/MPI_GET, provided by our MPI-2 library.

Although a V-Bus network card offers four times higher bandwidth and much lower latency than a *fast Ethernet* card, it still has high latency and low bandwidth as compared to high-end supercomputers such as CRAY T3E/T3D, SGI Origin[15, 16] and Myrinet[17]. Therefore, the success of Polaris targeting our PC-cluster heavily hinges on communication generation techniques that minimize the overall costs of communication to access arrays residing in memory of a remote PC in the cluster. In this work, we developed a new algorithm for communication generation specifically optimized for our V-Bus based PC-cluster. This algorithm is based on our novel *array access analysis* [4] which accurately identifies remote array accesses in a program.

This paper is organized as follows. Section 2 discusses several important features of the V-Bus network card and the implementation of MPI-2 on the V-Bus based PC-cluster. Sections 3 and 4 describe major components of Polaris. Section 5 shows how we have extended the compiler to target our V-bus based PC-cluster with our communication generation algorithm. Section 6 presents our experimental results, and Section 7 conclude our paper.

2 A V-Bus Based PC Cluster

This section presents a general overview of the hardware and software organization of our PC-cluster system. Our compiler which will be discussed in the subsequent sections translates a sequential code to a parallel code for this system.

2.1 The Interconnection Network Architecture

We have recently developed a high-speed and bandwidth-efficient network card based on two communication techniques, called *skew-tolerant wave-pipelining*(SKWP) and *Virtual-Bus* (V-Bus). SKWP [13] is an enhanced version of conventional *wave pipelining* that significantly increases the throughput of our network card. Wave pipelining considerably increases the throughput of the network card. However, it requires tremendous efforts to tune the skew between all signal lines. Furthermore, the end-to-end skew between signal lines can be magnified while passing through several wave-pipelined network cards, which can be neither predicted nor handled. So we integrated *an automatic skew sampling circuit* that eases the implementation of a wave-pipelined network cards on FPGA. The skew sampling circuit detects the delay differences between all signal lines, samples each signal, and merges the signals to have the same phase. With this technique, we achieve high bandwidth in point-to-point data transmission. Our experiment revealed that SKWP increases the bandwidth up to four times higher than conventional pipelining.

V-Bus [11, 12] is a new bus architecture that we have designed particularly to support efficient and low-cost broadcast communication in a parallel computer with a switched network (e.g., mesh, torus and hypercube) interconnecting its processing elements. Although parallel computers with switched networks are scalable as compared to those with shared networks (e.g, buses), they are less efficient for broadcasting operations. To support fast broadcasting, some of the computers have extra physical buses in addition to switched networks. But they are more expensive and suffer from low utilization of network bandwidth overall. In contrast, our V-Bus based PC-cluster is built on a mesh network that is specially designed to support efficient broadcast communications without extra physical buses. It dynamically constructs *virtual bus* for broadcasting on top of our mesh network when a broadcasting request is issued; that is, the network virtually establishes a bus only when a bus is required. If an urgent message occurs, it can intervene on-going point-to-point communication for faster broadcasting. The source and destination are connected directly through the virtual bus connection without intervening buffers and other on-going point-to-point messages are frozen in buffers.

As a result, a V-Bus network card integrated with SKWP supports faster routing in broadcasting and achieves more efficient bandwidth utilization. We showed through our recent studies [11, 12, 13] that a V-Bus network card provides about four times lower latency than the Fast Ethernet card.

2.2 Implementation of MPI-2 on the PC Cluster

In addition to a V-Bus network card, we also implemented Message Passing Interface (MPI-2) optimized for a V-Bus network card to make parallel programming simple on our PC-cluster. Our MPI-2 library reduces the communication overheads by sharing a message queue between device driver for a V-Bus network card and a MPI-2 daemon process, and by transferring data directly from a user buffer to a device drive buffer. Also, we optimize the collective communication of a MPI-2 library by making use of the collective facilities of a V-Bus network card.

The MPI-2 standard includes all the original func-

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

tions specified in MPI-1. In addition, it proposes new *one-sided communication* primitives, MPI_PUT/MPI_GET. With these new primitives, the MPI user is able to write shared memory code as well as conventional message passing code. The reason is that one-sided communication enables an MPI process to access, without the assistance of the remote process, the *logical* global memory space on top of physically distributed memories. This may significantly simplify the complexity of programming on our cluster. It may also help the compiler to simplify code generation for certain classes of computations, such as irregular computations and pointer chasing. This is because MPI_PUT/MPI_GET take place under the control of only a single processor, whereas two processors are needed for MPI_SEND/MPI_RECEIVE. In the implementation of MPI_PUT/MPI_GET, we optimized them specifically for a V-Bus network card.

According to the MPI-2 standard, one-sided communications are divided into two types: one for transferring contiguous memory regions, *contiguous* MPI_PUT/MPI_GET, and the other for transferring constant strided memory regions, *stride* MPI_PUT/MPI_GET. Contiguous MPI_PUT/MPI_GET use *DMA* so that data from the user buffer can be copied into the device driver buffer without interrupting the processor. But stride MPI_PUT/MPI_GET use *programmed I/O* where data in the user buffer is copied into the device driver buffer one-element by one-element. So, stride MPI_PUT/MPI_GET are generally less efficient than contiguous MPI_PUT/MPI_GET because they increase communication setup time significantly.

3 Overview of Polaris

Polaris is divided largely into two parts, the *front-end*(FE)[8] and *back-end*(BE), as shown in Figure 1. In the FE, *parallelism detection* is applied to a sequential program to identify parallel loops. The techniques implemented in Polaris to detect parallelism include: dependence analysis, inlining, induction variable substitution, reduction recognition and privatization[8]. The loops that are identified as parallel by these techniques are marked with parallel directive in the loops. Polaris will work only upon loops marked with parallel directive only. The original structure of the Polaris compiler is detailed in our earlier papers [2, 3].

The *linear memory access descriptor* (LMAD)[2, 3, 4] describes memory access pattern of the symbols of a program such as variables, arrays, functions and subroutines. The LMAD is used to detect parallelism in the FE and later to generate communication in the BE, where the program with a set of LMADs is transformed to a parallel program with MPI communication primitives. More details about the LMAD will be discussed in Section 4.

The target code generated by the BE is a shared memory program with one-sided communication primitives, MPI_PUT/MPI_GET, that are used to asynchronously transfer data between processing elements. It contains all the information on parallelism and communication generated by the transformation techniques implemented in Polaris. It is a machine independent intermediate representation with the following features:

- explicit synchronization through barriers, fences and locks,
- all data declared are intrinsically private, and
- explicit communication via MPI_PUT/MPI-GET primitives to keep data coherency between processors.

The target code is of a single program multiple data (SPMD) form using the *master/slave* model of execution, where one of the parallel processes (the master) executes all sequential sections and the other processes (the slaves) participate only in the computations of parallel sections. Barriers are used to explicitly control the flow of execution of masters and slaves. Slaves wait at barriers while the master is in a sequential section. When the master hits a barrier preceding a parallel section, the slaves are released to join the computation of the section. The master and slaves wait at a barrier after they complete the parallel section. Fences guarantee that all outstanding writes to remote memory have been completed. They are essential operations to maintain data consistency between distributed memories. Locks are useful for establishing critical sections where global operations using shared variables, such as reduction operations, are performed.

In the target program, the master initially holds all program data objects when the program begins. At the entrance of a parallel region, the master identifies the objects that each slave *may* access within the region and copies them

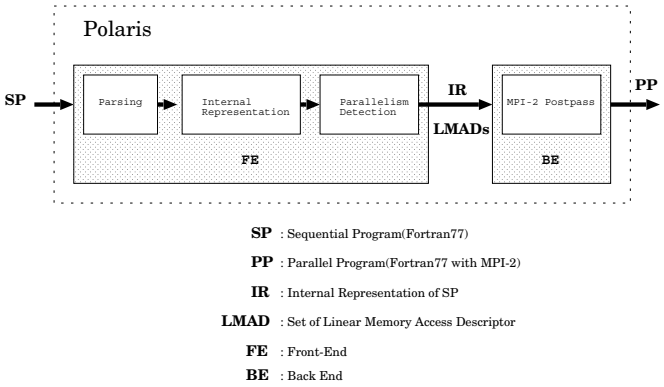

Figure 1. A system architecture for an parallel code generation

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

to the memory of each slave. We call this operation, *data scattering*. While executing the parallel region, slaves may modify some data objects duplicated to them, which makes the original master copies stale. Since all data in the program are intrinsically private, the master explicitly manages data consistency by collecting any modified data from slaves at the end of the parallel region and updating the copies in its memory with these up-to-date data. We call this operation, *data collecting*. We will explain these operations in Section 5.

4 LMADs and Summary Sets

Many compiler techniques depend heavily on the accuracy of *array access analysis*[4], which identifies the array elements accessed within the certain section of a code by a particular reference. For instance, to detect a parallel loop, the compiler first needs to identify memory access patterns of all variables used in the loop. Also, when the compiler generates communications, it needs the precise information about remote array accesses occurring in each local processor to optimize the overall communication cost. Compiler modules implementing such techniques must represent the array accesses in some standard representation. The LMAD is the array access representation that we used for this purpose. It was used to detect dependences on arrays in the *Access Region Test* [2], a dependence testing technique implemented in the Polaris FE. It was also used to generate MPI_PUT/MPI_GET communications in the Polaris BE, as will be discussed later in Section 4

4.1 The LMAD

The LMAD describes access movement through memory in terms of a series of *dimensions*. the dimension of an access is characterized as movement through memory with a consistent stride. For example, the stride is 2 for the accesses to memory locations shown in the Figure 2.

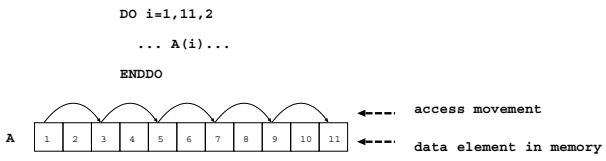

Figure 2. Example of Access movement

By a consistent stride, we mean that the expression representing the stride does not change. For instance, the following memory access in Figure 3 shows a stride which, although changing value, is consistent because it is represented by the expression 2I, where I represents the ordinal of the access in the sequence. An access dimension is characterized by three expressions. The data element access movement starts from *base offset*. The *stride* is the distance in the number of array elements between accesses generated by consecutive values of index. The *span* is the total element length that the access traverses when the index iterates its entire ranges. The span is defined as the difference between the offset of the last element in the access and the offset the first element in the access. The span is useful for doing certain operations and simplifications on the LMAD, however it is only accurate when the subscript expressions for the array access are *monotonic* [4].

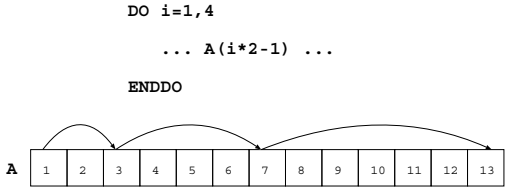

Figure 3. Example of Access movement with a variant stride

A nested loop enclosing an array reference causes as many access dimensions in the array reference as there are enclosing loops whose loop indices participate in the array subscripting expressions. The LMAD for the array access in Figure 4 is written as with a series of d comma-separated strides as superscripts to the variable name and a series of comma-separated spans as subscripts to the variable name, with a base offset written to the right of the descriptor, separated from the rest of the descriptor by a plus-sign. The dimension index is only included in the written form of the LMAD if it is needed for clarity. In that case, it is written as a subscript to the appropriate stride.

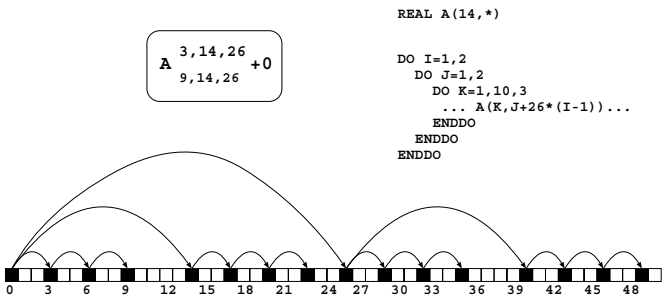

Figure 4. A memory access diagram for the array A in a nested loop and LMAD which represents it

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

4.2 Summary Sets

As can be seen from Figure 4, one common base offset exists for all dimensions of the LMAD. The LMAD gives us a perfectly accurate means for representing the memory access of a section of code, in the sense that precisely the memory locations referenced in that code section may be reproduced from the descriptor(s) alone. The LMAD is divided into three types from the standpoint of memory access operations:

1. ReadOnly : Regions are accessed by only read operations.
2. WriteFirst : Regions are accessed by a write operation first and then accessed by read or write operations.
3. ReadWrite : Regions are accessed by a read operation first and then accessed by read or write operations.

A *summary set* [2] is a symbolic description of a set of memory locations that are accessed in a certain program section (i.e., an individual statement, a loop or a subroutine). When memory locations are accessed in a program section, we group them according to their access types, classified above, and add each group to the appropriate summary set for the section When a summary set is associated with a statement, all variables accessed in the statement have their sets of LMADs which are stored in the summary set for the statement. To illustrate this, consider the statement (1) of Figure 5 where the memory location of the array A is accessed by a write operation. So the summary set for the statement (1) has a WriteFirst LMAD.

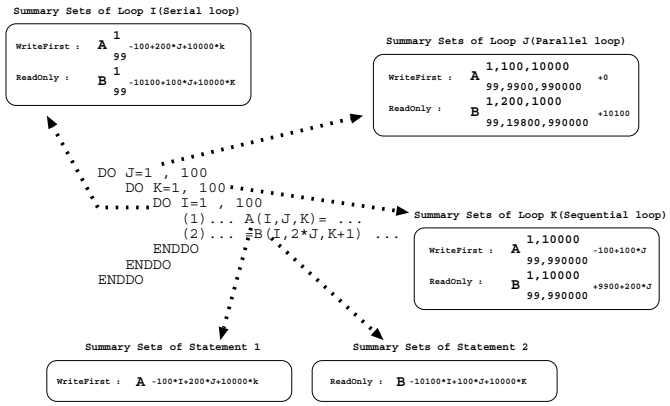

Figure 5. Example of the summary set of multiply nested loop

A summary set associated with a loop is a union of all summary sets for the statements nested inside the loop. For instance, the summary set for loop I of Figure 5 is obtained by first *expanding* [2] the summary sets for statements (1) and (2) with regard to the loop index I, and then by integrating them in a single set. To integrate summary sets, one LMAD is selected from the summary set for statement (1) or (2), and then the LMAD is added to the summary set for loop I. The procedure for generating a summary set for program statements is fully explained in our previous literature[2].

5 The MPI-2 Postpass for Polaris

The MPI-2 postpass is a Polaris backend (see Figure 1) that we have recently added to retarget Polaris at our V-Bus based PC-cluster. Figure 6 shows the overall structure of the MPI-2 postpass of Polaris. This section explains each compiler module of the postpass.

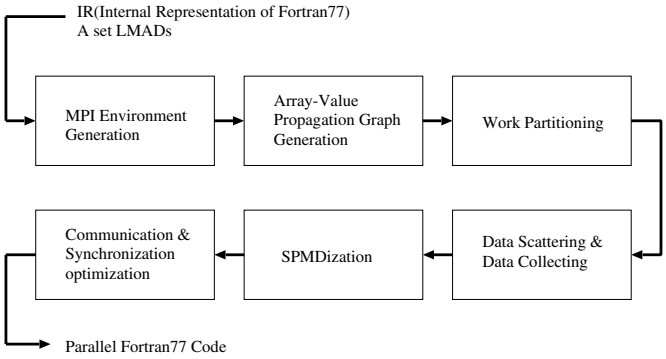

Figure 6. Compiler Modules of the MPI-2 Post-Pass

5.1 MPI Environment Generation

To run a MPI program, the program should interact with MPI processes which provide a parallel programming environment on top of the target machine. In the MPI-2 postpass, thus, we need to create a new MPI environment for the program. For this, all necessary symbols for functions and variables are first initialized and registered into a symbol table of the program.

For inter-processor communication, we create a *memory window*(using a MPI_WIN DOW call) which is a portion of the private memory of a local process that can be accessed by remote processes without intervention of the local process. So, the MPI-2 postpass must scan all the loops annotated with parallel directives to find scalar or array variables that need to be accessed by remote processes.

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

5.2 AVPG Generation

One problem with our data scattering and collecting scheme, mentioned in Section 3, is that it may create excessive communications to maintain data consistency between the master and the slaves. To alleviate this problem, we construct the *array-value-propagation graph* (AVPG), which captures the access patterns of arrays referenced in a sequence of consecutive loops. The AVPG is in fact a collection of directed, connected subgraphs, each of which represents the access patterns of an individual array, as shown in Figure 7. In the figure, the AVPG contains three subgraphs, respectively, for arrays A,B and C.

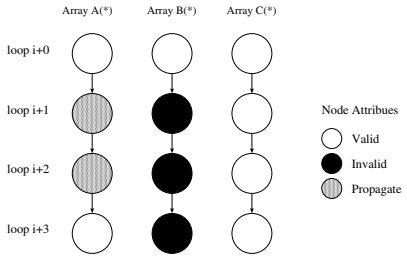

Figure 7. Example of the AVPG

Each node in a subgraph corresponds to the outermost loop in a loop nest. The nodes are connected according to the program control flow. We assign an attribute to each node according the types of array accesses in the loop. The attributes of a node are divided into three types.

- Valid : An array is used in the loop.
- Propagate : An array is not used in the loop. But somewhere in a subsequent loop in the control flow, the array will be used.
- Invalid : An array is not used in the loop and the array will not be used any more in a subsequent loop in the control flow.

In our data scattering and collecting scheme, MPI_PUT/MPI_GET communication and fence operations occur at every boundary of parallel regions, which corresponds to each edge between neighboring nodes in the AVPG. However, we have found that some of these communications can be eliminated because they are redundant. The boundary where such redundant communications occur is denoted by the edge from a valid node followed by an invalid node, like the one between the $loop_{i+0}$ node and the $loop_{i+1}$ for array B in Figure 7. Another case where we can reduce communication overhead is communications occuring at an edge between a valid node and its subsequent propagating nodes. In this case, communications are delayed until the next valid node, which reduces the overall communication overhead by eliminating all communication and fence operations among propagating nodes in-between. The example of such a case can been seen in the subgraph for array A in Figure 7. In the example, communications are required only at the edge preceding the $loop_{i+3}$

5.3 Work Partitioning

Internally, Polaris annotates loops with `parallel` directives. Work partitioning is a procedure that transforms the original program annotated with `parallel` directives into a statically scheduled SPMD form by assigning computations in each program section to processors according to the master/slave paradigm discussed earlier. In the current implementation, the iteration spaces of parallel loops are assigned using the conventional strategies: *cyclic* assignment for triangular loops, and *block* assignment for square loops. In practice, these static loop scheduling strategies lead to relatively good load balancing at low or modest costs [3].

5.4 Data Scattering and Collecting

The general description of the data scattering and collecting scheme is as follows. First, the summary sets for each node in the AVPG are computed, as described above. Then, depending on the type [1] of a set of LMADs in the summary set, data-scattering and data-collecting operations are applied to a node in the AVPG as follows;

- The ReadOnly LMAD: data-scattering
- The WriteFirst LMAD: data-collecting
- The ReadWrite LMAD: data-scattering and data-collecting

Since the access regions summarized by the **ReadOnly** LMAD are not written but only read during a parallel loop execution, they are simply copied to slaves at the beginning of the parallel loop and dropped at the end of the parallel loop without being copied back the master. Therefore, only data-scattering is necessary for the **ReadOnly** LMAD. In the case of the **WriteFirst**[9] LMAD, all access regions are first written before any following reads, which means that no values stored in the access regions summarized by **WriteFirst** LMAD are used in the parallel loop. So, data-scattering is not necessary for the regions. Instead, data-collecting must be performed to copy all values newly generated for the regions to the master. In the case of the **ReadWrite** LMAD, the access regions are both read and written. So, both data-scattering and data-collecting are required to maintain data consistency.

[1] Recall that there are three different types of LMAD sets in a summary set; that is, ReadOnly, ReadWrite, WriteFirst.

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

To explain the data-scattering algorithm, here, we define a couple of fundamental notations.

Definition 1 *Let I_j be the index of a j-th nested loop, and δ_j and α_j respectively be the span and stride[4] of the dimension of a j-th nested loop. Then, we define the* **array access in a dimension***(AAD) of the LMAD as follows:* $AAD(I_j) = (\delta_j, \alpha_j)$.

Definition 2 *Assume that a given d-dimension LMAD* **A** *is* $\mathbf{A}^{\delta_1,\delta_2,\delta_2,\ldots,\delta_d}_{\alpha_1,\alpha_2,\alpha_3,\ldots,\alpha_d}$ *and all the stride and span pairs are arranged in the order of loop nest. Suppose that I_p is the parallel loop index of which loop will be parallelized, and that α_p and δ_p are the stride and span of the dimension of a parallel loop index respectively. Let* $\mathbf{A}^{\delta_2,\delta_2,\ldots,\delta_{d-1},\delta_p}_{\alpha_2,\alpha_3,\ldots,\alpha_{d-1},\alpha_p}$ *and* $\mathbf{A}^{\delta_1}_{\alpha_1}$ *be two sub-LMADs splitted from the original LMAD* **A** *where the former is for memory offset calculation and the latter is for mapping data access into communication primitives. We call these two sub-LMADs the* **splitted LMADs** *of* **A** *which are respectively denoted by* $\mathbf{A}_{offset}$ *and* $\mathbf{A}_{mapping}$.

$\mathbf{A}_{offset}$ can be obtained by eliminating the lowest dimension of **A** and $\mathbf{A}_{mapping}$ is obtained by extracting only the lowest dimension from **A**. $\mathbf{A}_{offset}$ is used to calculate a set of data offsets from a base address in a sequence of communication primitive generation and $\mathbf{A}_{mapping}$ is used to map data access into communication primitives provided by the MPI-2 library.

Memory access patterns described by $\mathbf{A}_{mapping}$ appear repeatedly with different offsets from a base address. The set of the offsets that are calculated from $\mathbf{A}_{offset}$ is

$\{x_1 \times \alpha_2 + x_2 \times \alpha_3 + \ldots + x_{d-1} \times \alpha_{d-1} + x_p \times \alpha_p \mid 0 \leq x_i \leq \delta_i/\alpha_i$, where x_i is non-negative integer$\}$.

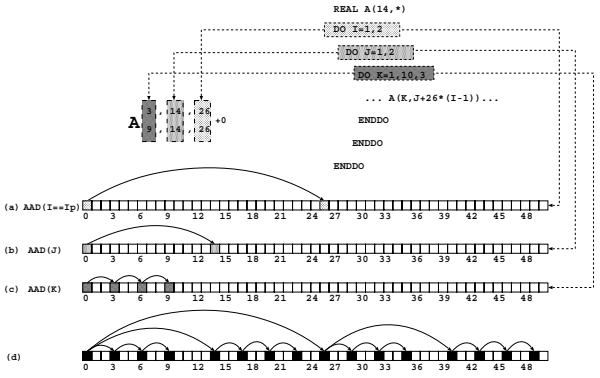

Figure 8. Association between a loop dimension and each dimension of the LMAD

For instance, let's see the LMAD in Figure 8. The (a), (b) and (c) show the diagrams of memory access pattern of each dimension respectively and (d) is the diagram of all dimensions together. The portion of black-colored memory regions of (d) need to be copied to the master or slaves, according to work partitioning schemes. Memory access pattern of (c) appears repeatedly in (d) with different offsets from 0(base address). In this example, memory offsets are 0*14+0*24, 1*14+0*24, 0*14+1*24 and 1*14+1*25. With memory offsets calculated with $\mathbf{A}_{offset}$, $\mathbf{A}_{mapping}$, AAD(K) in Figure8 is mapped to communication primitives. In our compiler, $A_{mapping}$ is mapped to MPI_PUT/MPI_GET as follows; If α_1 of $\mathbf{A}_{mapping}$ is constant, $\mathbf{A}_{mapping}$ can be mapped into a single MPI_PUT/MPI_GETd. But if α_1 is not a constant, it must be mapped into MPI_PUT/MPI_GET one memory element by one memory element. When α_1 is a constant, $\mathbf{A}_{mapping}$ is divided into two cases:

1. If α_1 is 1, contiguous MPI_PUT/MPI_GET will be used.
2. If α_1 is greater than 1, stride MPI_PUT/MPI_GET will be used.

Based on the splitted-LMAD described above, data-scattering and data-collecting are implemented. The detailed procedure of data-scattering is described in [21]. The data-collecting operation is virtual identical to the data-scattering operation except the direction of data transfer is the opposite; that is, slaves *put* their up-to-date copies to the master.

5.5 SPMDization

To generate a SPMD code, barriers and locks are inserted to the source code to explicitly control and synchronize the execution flow of the master and slaves. For this, we must first identify *synchronization points* in the code. Synchronization points are any node in a control flow graph that multiple out-going edges such as IF, FOR and GOTO statements. Then before every synchronization point, the MPI-2 primitive, MPI_BARRIER, is inserted. Another MPI-2 primitive, MPI_FENCE, is also inserted at the same place to guarantee that all out-standing writes to variables are complete to maintain data consistency among distributed memories.

5.6 Communication Optimization

Earlier in this section, we showed how we reduced overall communication costs in the data scattering and collection scheme by eliminating redundant copy operations with the AVPG. The communication costs were further optimized by reducing the number of data packets that are transfered to

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

carry data objects between the master and slaves. This optimization was based on our observation that in a typical PC-cluster system whose network latency is much higher than conventional high-performance parallel computers, it is often more efficient to aggregate small *exact* regions into a large *approximate* regions for communication in order to reduce the total communication time. Although approximate regions may contain redundant data objects, the overall communication costs to transfer them are usually lower than those to transfer a large number of small fragments of exact access regions. The reason is that communication setup costs to transfer a data packet in such a high-latency PC-cluster network are relatively very high.

As mentioned earlier, the MPI-2 library supports two types of MPI_PUT/MPI_GET operations: stride MPI_PUT/MPI_GET and contiguous MPI_PUT/MPI_GET. In this work, we have found that it is better to transfer a approximate regions with a single contiguous MPI_PUT/MPI_GET than transfer small exact regions with multiples stride MPI_PUT/MPI_GETs.

Considering the fact explained above, our compiler optimized communication at three different granularities: **fine**, **middle** and **coarse**. At the fine grain, exact regions are always transfered. In our compiler, exact regions are described by $\mathbf{A}_{mapping}$. Therefore, to transfer exact regions, we use

1. contiguous MPI_PUT/MPI_GET if the stride of $\mathbf{A}_{mapping}$ is equal to 1, and;

2. stride MPI_PUT/MPI_GET are used if α of $\mathbf{A}_{mapping}$ is greater than 1.

We can see in Figure 9 (b) that the stride of $\mathbf{A}_{mapping}$ is 3. Thus, black-colored elements in dashed boxes are transfered to the master or slaves by stride MPI_PUT/MPI_GET.

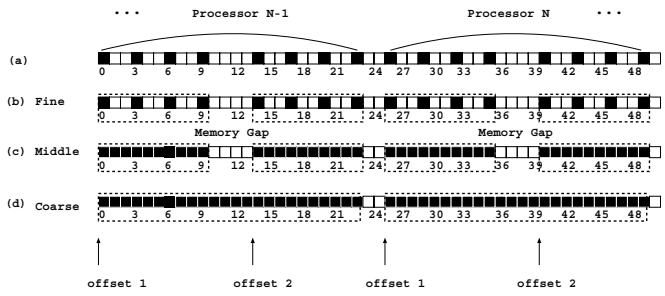

Figure 9. Example of communication optimization in V-Bus based PC-cluster

We introduce the notion of middle granularity in our compiler to avoid using stride MPI_PUT/MPI_GET in our communication generation. At the middle grain, exact regions are converted into approximate regions by setting the stride of $\mathbf{A}_{mapping}$ 1. Since the stride of $\mathbf{A}_{mapping}$ is 1, the approximate regions will be transfered by contiguous MPI_PUT/MPI_GET. In Figure 9 (c), black-colored memory regions, approximate regions, in dashed boxes are transfered to the master and slaves. The number of communications at both the fine and middle grains is calculated as follows:

$$(\delta_2/\alpha_2) \times (\delta_3/\alpha_3) \times ... \times (\delta_{d-1}/\alpha_{d-1}) \times (\delta_p/\alpha_p) + 1.$$

We introduce the notion of coarse granularity to reduce the total number of communications. At the coarse grain, exact regions are converted into approximate regions by setting $\mathbf{A}_{mapping}$ to $\mathbf{A}^{\delta_1,\delta_2,...,\delta_{d-1}}_{\alpha_1,\alpha_2,...,\alpha_{d-1}}$ and $\mathbf{A}_{offset}$ to $\mathbf{A}^{\delta_p}_{\alpha_p}$. So the number of communications is reduced to $\delta_p/\alpha_p + 1$. Since approximate regions are enlarged, the amount of redundant data will increase. Figure 9 (d) shows one big approximate regions in dashed boxes are transfered to each remote processor.

Communication optimization explained above always can be applied to data-scattering. But before applying to data-collecting, we need to check whether there exist overlapped data objects in approximate regions that cause program to execute abnormally by a *race condition*. To solve this problem, we implemented a routine to check the upper and lower bound of approximate regions. If overlapped data objects exist, we must generate data-collecting at the fine grain to prevent the master and slaves from overwriting remote data objects. For now, it is up to the user that selects the optimal granularity to minimize the communication time. The profiling tools [20] recently provided in Polaris would be useful to guide the user when such decision should be made.

6 Experiment

Our implementation of the MPI-2 postpass and the V-bus network system has not been completed. However, we needed to evaluate our system for future enhancement of our hardware and software designs for the PC-cluster. Thus, we experimented our system with a configuration of 4-node PCs, where the PCs are linked with wave-pipelined wormhole network cards implemented on FPGA[13, 11]. Each PC is equipped with a 300MHz CPU, a 64MB memory and runs Linux. The benchmark codes used in this experiment are MM for a matrix multiplication, a SWIM from the SPEC97 benchmark suite and CFFTZINIT, a major subroutine of TFFT for the NASA codes. These codes were translated to MPI-2 code by Polaris and tested on our PC-cluster.

We measured and analyzed the total execution and communication time of the benchmarks at several different levels of granularity. Table 1 shows the speedups(Execution time$_{sequential\ program}$/Execution time$_{parallel\ program}$) of

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

the code of a matrix multiplication. The sizes of a matrix are 256×256, 512×512 and 1024×1024. The speedup of parallel execution time ranges from 1.6 for 2 processors to 3.0 for 4 processors for a matrix of 1024×1024.

Table 1. Total execution time of the MM code

Speedups		Array Size		
		256*256	512*512	1024*1024
# of Nodes	1	0.96	0.96	0.96
	2	1.086	1.53	1.60
	4	1.75	2.74	3.033

Table 2 shows the communication time of the benchmarks at the three different levels of granularity classified in Section 5 (that is, *coarse*, *middle* and *fine*). In MM at the Coarse grain, we reduced communication time by roughly 30%. However, at the middle grain, communication cost increases about 17%. This is because the overhead caused by redundant communication outweighed the communication time saved by using efficient contiguous MPI_PUT/MPI_GET instead of stride MPI_PUT/MPI_GET. Likewise, the table shows that, in SWIM, we obtained poor results at the Middle grain. However, we achieved speedup in the communication time by a factor of 1.3 at the coarse grain. In CFFZINIT, we achieved relatively good performance in communication time at both the Middle and coarse grains. An the Middle grain, there exist several LMADs with the stride of 2 in the subroutine. Although 50% of communication was used to transfer redundant data, we were still able to reduce the overall communication time by 28%.

Table 2. Communication time for matrix multiplication, swim and CFFZINIT of TFFT

Total Communication Time (sec)	Granularity		
	fine	middle	coarse
MM(1024*1024)	0.72	0.89	0.01128
Swim(ITMAX=1)	0.20590	*	0.072166
CFFZINIT(M=11)	0.3584	0.0768	0.0068

The experimental results tell us the fact that communication optimization techniques are closely related to the communication patterns of application programs, as was already observed in our other experiments on parallel computers [3]. In other words, we learned from this work that any single technique does not work for all types of communication patterns in a program, and therefore, that an appropriate optimization technique must be selected only after precise analysis of data access pattern in the program.

7 Conclusion

In this paper, we presented our on-going work on the hardware and software development for a PC-cluster system based on V-Bus networking technology. We first discussed the architecture of the PC-cluster. Two major components of the architecture are V-Bus network cards integrated with the SKWP technique and the MPI-2 library implemented on the hardware. By supporting a shared memory programming paradigm through MPI_PUT/MPI_GET, our cluster system is easier to program than other cluster systems that support only message passing programming environments. The MPI-2 communication primitives were highly optimized for our V-Bus based PC-cluster. They perform *user-level communication* rather than system-level communication which incurs additional overhead for context switching between the user mode and the kernel mode to handle the communication request.

To make the programming even further easy, we are currently developing a parallelizing compiler specifically targeting our cluster. The compiler extended from Polaris by implementing the MPI-2 postpass automatically translates conventional FORTRAN 77 code to MPI-2 code ready to run on the cluster. To reduce the overall communication costs in our parallel code, the postpass has been implemented with the data scattering and compacting algorithm based on the AVPG.

The experimental work showed how by controlling a granularity level of communication, we were able to reduce the number of remote memory accesses, and consequently to minimize the overall communication costs. In particular, we found that our efficient implementation of MPI_PUT/MPI_GET based on DMA helped to achieve reasonable performance on the relatively slow V-Bus network line. However, the experiment results are still premature and need much more improvement. We currently plan to extend our experiment with a more variety of benchmarks after enhancing our MPI-2 postpass with additional optimization techniques based on the lessons from the experimental results reported in this paper.

References

[1] D.A.Padua et al , "Polaris : A new-generation parallelizing compiler for MPPs", *Technical Report CSRD-1306*, Center for Supercomputing Research and Development, Univ of Illinois at Urbana-Champaign, June, 1993

[2] J.Hoeflinger , "Interprocedural Parallelization Using Memory Classification Analysis", PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, August, 1998.

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

[3] Y.Paek , "Automatic Parallelization for Distirbuted Memory Machines Based on Access Regions Analysis", PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, August, 1997.

[4] Y. Paek, J.Hoeflinger, and D.Padua , "Simplication of Array Access Patterns for Compiler Optimizations", *Proceedings of he SIGPLAN Conference on Programming Language Design and Implementation*, June 1998

[5] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford , "The Polaris Internal Representation", *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[6] Message Passing Interface Forum. MPI , "A message-passing interface standard", http:www.mpi-forum.org,January 12,1996

[7] Message Passing Interface Forum. MPI-2 , "Extensions to the Message-Passing Interface", ,January 12,1996

[8] W.Blume, R. Doallo, R Eigenmann, J.Grout, J.Hoeflinger, T.Lawrence, J. Lee,D.Padua, Y.Paek,W.Pottenger,L.Rauchwerger, and P.tu , "Parallel Programming with Polaris", *IEEE Computer*, 19(12):78-82, December 1996.

[9] Pegh Tu and David Padua , "Automatic Array Privatization", *Sixth Workshop on Languages and Compilers for Parallel Computing*. Protland, Lecture Notes in Computer Science, volume 768, pages 500-521, August 12-14, 1993

[10] Gagan Agrawal , "Interprocedural Communication Optimizations for Message Passing Architectures", *The Seventh Symposium on Frontiers of Massively Parallel Computation* 1999 , Page(s): 174 -181

[11] J.H.Choi, B.W.Kim, K.H.Park and K.I. Park , "A bandwidth-efficient implementation of mesh with multiple broadcasting", *In Proceedings of International Conference on Parallel Processing*, 1999

[12] B.W.Kim, J.H Choi, K.H.Park and K.I. Park , "A Wormhole Router with Embeded Broadcasting Vritual Bus for Mesh Computers", *Parallel Processing Letter*, 2000

[13] Bong Wan Kim, Hyun Jin Choi, Kwang Il Park, Jong Hyuk Choi and Kyu Ho Park , "A Skew-Tolerant Wave-Pipelined Router on FPGA", *Hot Interconnects 7*, August 18-20, 1999

[14] Saman P.Amarasinghe and Monica S.Lam , "Comminication Optimization and Code Generation for Distributed Memory Machines", In the Proceedings of *The ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993

[15] R. Arpaci, D.Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick , "Empirical Evaluation of the CRAY-T3D : A Compiler Perspective", *International Symposium on Computer Architecture*, pages 320-331, June 1995.

[16] M. Kandemir, P. Banerjee, A.Choudhary, J. Ramanujam, and N. Shenoy , "A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D", *International Symposium on Computer Architecture*, June 1995.

[17] Myrinet Performance Measurements , http://www.myri.com/myrinet/performance/index.html

[18] T. von Eicken, A. Basu, V Buch, and W. Vogels , "U-Net:A User-Level Network Interface for Parallel and Distributed Computing", *In the Proceeding Of the 15th ACM Symposium on Operating Systems Principle*, Dec., 1995

[19] A.G Navarro, R. Asenjo, E.L Zapata, D. Padua, "Access descriptor based locality analysis for Distributed Shared Memory multprocessors", *International Conference on Parallel Processing*. pages 86-94 1999

[20] Gheoghe Calin Cascaval, "Compile-time Based Performance Prediction", Ph.D Thesis Univ. of Illinois at Urbana-Chanpaign, August 2000

[21] Sang Seok Lim, "Communication Generation for a V-Bus based PC-Cluster using LMADs", *KAIST CORE Technical Report 2000*

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE