Compilation and Communication Strategies for Out-of-core programs on Distributed Memory Machines

Rajesh Bordawekar **Alok Choudhary***
Electrical and Computer Engineering Department
121, Link Hall, Syracuse University, Syracuse, NY 13244
`rajesh, choudhar@cat.syr.edu`
URL: `http://www.cat.syr.edu/~{rajesh,choudhar}`

J. Ramanujam
ECE Dept., Louisiana State University, Baton Rouge, LA 70803
`jxr@gate.ee.lsu.edu`
URL: `http://www.ee.lsu.edu/jxr/jxr.html`

Abstract

It is widely acknowledged that improving parallel I/O performance is critical for widespread adoption of high performance computing. In this paper, we show that communication in out-of-core distributed memory problems may require both inter-processor communication and file I/O. Thus, in order to improve I/O performance, it is necessary to minimize the I/O costs associated with a communication step. We present three methods for performing communication in out-of-core distributed memory problems. The first method called the *generalized collective communication* method follows a loosely synchronous model; computation and communication phases are clearly separated, and communication requires permutation of data in files. The second method called the *receiver-driven in-core communication* considers only communication required of each in-core data slab individually. The third method called the *owner-driven in-core communication* goes even one step further and tries to identify the potential future use of data (by the recipients) while it is in the sender's memory. We describe these methods in detail and present a simple heuristic to choose a communication method from among the three methods. We then provide performance results for two out-of-core applications, the two-dimensional FFT code and the two-dimensional elliptic Jacobi solver. Finally, we discuss how the out-of-core and in-core communication methods can be used in virtual memory environments on distributed memory machines.

*Corresponding author

1 Introduction

The use of parallel computers to solve large scale computational problems has increased considerably in recent times. With these powerful machines at their disposal, scientists are able to solve larger problems than were possible before. As the size of the applications increase so do their data requirements. For example, large scientific applications like Grand Challenge applications require 100s of GBytes of data per run [otSII94].

Since main memories may not be large enough to hold data of the order of GBytes, it is necessary to store the data on disks and fetch it during program execution. Performance of these programs depends on how fast the processors can access data from the disks. A poor I/O capability can severely degrade the performance of the entire program. The need for high performance I/O is so significant that almost all the present generation parallel computers such as the Paragon, SP-2, nCUBE2 etc. provide some kind of hardware and software support for parallel I/O [Pie89, Ger95, dRC94].

Data parallel languages like High Performance Fortran (HPF) [Hig93] were designed for developing complex scientific applications on parallel machines. In order that these languages can be used for programming large applications, it is essential that these languages (and their compilers) provide support for applications requiring large data sets. As part of the ongoing PASSION[1] [CBH+94] project, we are currently modifying the Portland Group HPF [MMWY95] compiler to support out-of-core applications [BCK+95, Bor96]. The PASSION compiler takes an HPF program accessing out-of-core arrays as an input and produces the corresponding node program with calls to runtime routines for I/O and communication. The compiler *strip-mines* the computation so that only the data which is in memory is operated on, and handles the required buffering. Computation on in-memory data often requires data which is not present in a processor's memory requiring local I/O access as well as communication among processors for access to non-local data. Since the data is stored on disks, communication often results in disk accesses. In this paper we propose three strategies to perform communication when data is stored on disks. These strategies use different techniques to reduce I/O cost during communication. These techniques are illustrated using two scientific applications. Finally we show that these techniques could also be used in virtual memory

[1] For further information on PASSION Project, refer to `http://www.cat.syr.edu/passion.html`

environments.

The paper is organized as follows. Section 2 describes the out-of-core computation model. This section also introduces a data storage model called the *Local Placement Model.* Section 3 describes the three proposed strategies for performing communication in out-of-core data parallel problems. A running example of a 2-D elliptic solver using Jacobi relaxation is used to illustrate these strategies. A simple heuristic that can be used to decide which communication strategy based on communication patterns in the code is presented. Section 4 presents experimental performance results for the three communication strategies using two out-of-core applications, two-dimensional Jacobi Relaxation and two-dimensional Fast-Fourier-Transform (FFT). Section 5 describes how these communication strategies could be used in virtual memory environments. Section 6 presents related work and Section 7 concludes with a discussion and points to future work.

2 Computation Model

2.1 Out-of-core Computation Model

A computation is called an *out-of-core (OOC)* computation if the data used in the computation does not fit in the main memory. Thus, the primary data structures reside on disks and this data is called OOC data. Processing OOC data, therefore, requires staging data in smaller granules that can fit in the main memory of a system. That is, the computation is carried out in several phases, where in each phase part of the data is brought into memory, processed, and stored back onto secondary storage (if necessary).

The above phase may be viewed as application level demand paging in which data is explicitly fetched (or stored) at the application level. In virtual memory environments with demand paging, a page or a set of pages is fetched into the main memory from disk. The set of pages which lies in the main memory is called the *working set.* Computations are performed on the data which belong to the working set. After the computation is over, pages from the working set which are no longer required are written back on the disk, if necessary. When the computation requires data which is not in the working set, a page fault occurs and the page which contains the necessary data is fetched from disk. We can consider the out-of-core computation as a type of demand paging in which one or more pages form one slab. The slabs are fetched from disk when required and computation is performed on the in-core data slab. When the computation on the in-core slab is finished, the slab is written back to the disk.

2.2 Programming Model

In this work, we focus on out-of-core computations performed on distributed memory machines. In distributed memory computations, work distribution is often obtained by distributing data over processors. For example, High Performance Fortran (HPF) provides explicit compiler directives (`TEMPLATE`, `ALIGN` and `DISTRIBUTE`) which describe how the arrays should be partitioned over processors [Hig93]. Arrays are first aligned to a template (provided by the `TEMPLATE` directive). The `DISTRIBUTE` directive specifies how the template should be distributed among the processors. In HPF, an array can be distributed as either `BLOCK` or `CYCLIC`(k). In a `BLOCK` distribution, contiguous blocks of size $\frac{N}{p}$ (where N is the size of the array and p is the number of processors) are distributed among the p processors. In a `CYCLIC`(k) distribution, blocks of size k are distributed cyclically. The `DISTRIBUTE` directive specifies which elements of the global array should be mapped to each processor. This results in each processor having a *local array* associated with each array in the HPF program; the local array is typically a part of the entire array. Our main assumption is that local arrays are stored in files from which the data is staged into main memory.

An array that is too large to fit in main memory is referred to as an *Out-of-Core Array* or *OCA*[BCK+95]. The in-memory pieces of such an array are called *In-Core Arrays* or *ICAs*; they are analogous to the local arrays described above. On a parallel machine, each ICA may itself be partitioned among many processors. Thus, a second level of mapping is needed. When an ICA is distributed, we refer to the section on each processor as the *In-Core Local Array* or *ICLA*. It is sometimes convenient to refer to the portion of the OCA that is mapped to a single processor. We call this section of the array the *Out-of-Core Local Array* or *OCLA*. This is equivalent to the union of the ICLAs of that OCA mapped to that processor.

The out-of-core local array can be stored in files using two distinct data placement models. The first model called the *Global Placement Model* (GPM) maintains the global view of the array by storing the global array into a common file [CBH+94]. The second model called the *Local Placement Model* (LPM) distributes the global array into one or more files according to the distribution pattern. For example, the VESTA file system provides a way of distributing a file into several logical file partitions, each belonging to a distinct processor [CF94]. In this paper we only consider the local placement model.

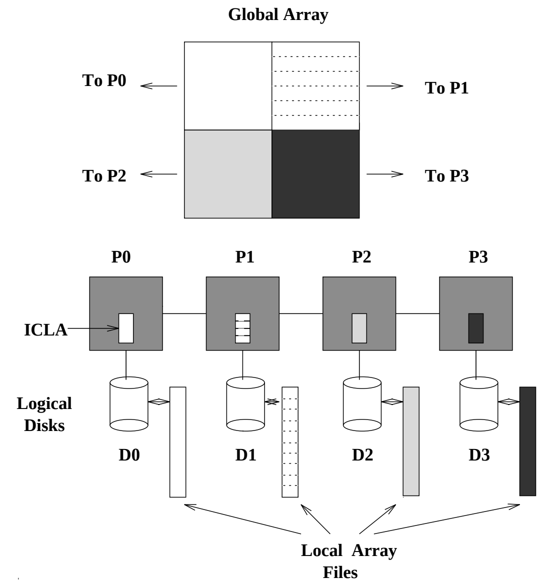

Figure 1: Local Placement Model

2.3 Local Placement Model

In the Local Placement Model, the local array of each processor is stored in a separate *logical* file called the *Local Array File* (LAF) of that processor as shown in Figure 1. The local array files can be stored as separate files or they may be stored as a single file (but are logically distributed). The node program explicitly reads from and writes into the LAF when required. The simplest way to view this model is to think of each processor as having another level of memory which is much slower than main memory. If the I/O architecture of the system is such that each processor has its own disk, the LAF of each processor can be stored on the disk attached to that processor. If there is a common set of disks for all processors, the LAF may be distributed across one or more of these disks. In other words, we assume that each processor has its own *logical disk* with the LAF stored on that disk. The mapping of the logical disk to the physical disks is system dependent. At any time, only a portion of the local array, the ICLA is fetched and stored in main memory. The size of this portion depends on the amount of memory available. All computations are performed on the data in the ICLA. Thus, during the course of the program, parts of the LAF are fetched into the ICLA, the new values are computed and the ICLA is stored back into appropriate locations in the LAF.

In this model, a processor cannot explicitly operate on a file owned by a different processor. If a processor needs to read data from a file owned by a different processor, the required data will be read by the owner and then communicated to the requesting processor. Since each local array file contains the OCLA of the corresponding processor, the distributed view of the out-of-core global array is preserved.

3 Communication Strategies for Compiling Out-of-core Computations

Given OOC computations, when the primary data sets reside in files on disks, any communication involving the OOC data would require disk accesses as well. In in-core computations on distributed memory machines, a communication step involves movement of data from one or more processor's memory to other processor's memories. For OOC computations, communication therefore would involve movement of data from one or more processor's files to other processor's files. Given that disk accesses are several orders of magnitude more expensive than memory accesses, and considerably more expensive than communication time itself, it is important to consider optimizations in the I/O part of a communication step. In this section, we propose several strategies to perform communication for OOC computations. We mainly focus on data parallel programs such as those written in HPF. We first describe how the communication is done for in-core programs and then describe three communication strategies for out-of-core programs. We explain both cases with the help of the HPF program fragment given in Figure 2. In this example, arrays A and B are distributed in (`BLOCK, BLOCK`) fashion on 16 processors logically arranged as a 4×4 two-dimensional grid.

3.1 Communication Strategies for Compiling In-core Computations

Consider the HPF program fragment from Figure 2. The HPF program achieves parallelism using data and work distribution. The data distribution may be specified by the user using compiler directives or may be automatically determined by the compiler. Work distribution is performed

```
1      REAL A(1024,1024), B(1024,1024)
         ..........
2      !HPF$ PROCESSORS P(4,4)
3      !HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P ::A,B
         ...........
4      FORALL (I=2:N-1, J=2:N-1)
5              A(I,J) = (B(I,J-1) + B(I,J+1) + B(I+1,J) + B(I-1,J))/4
6      END FORALL
```

Figure 2: An HPF Program Fragment for Two-dimensional Jacobi Computations. Arrays A and B are distributed in (BLOCK, BLOCK) fashion over 16 processors.

by the compiler during the compilation of parallel constructs like FORALL or array assignment statements (line 5, Figure 2). A commonly used paradigm for work distribution is the *owner-computes* rule [BCF+93, HKT92]. The owner-computes rule says that the processor that owns a datum will perform the computations which make an assignment to this datum.

In this example, it can be observed that for the array assignment (lines 4-6), each processor requires data from neighboring processors. Consider processor 5 from Figure 3. It requires the last row of processor 1, last column of processor 4, first row of processor 9 and first column of processor 6. This pattern can be considered as a logical *shift* of the data across processor boundaries. It should be noted that processor 5 needs to send data to processors 1, 4, 6 and 9 as well. Data communication can be carried before the local computation begins. Since computation is performed in a SPMD (loosely synchronous) style, all processors synchronize before communication. As all processors need off-processor data for their local computations, they simultaneously send and/or receive data. This is the so called *collective* communication. The HPF compiler inserts a call to a specific collective communication routine (*overlap shift*). After the communication is performed, each processor begins computations on the local array. From this analysis, we can arrive at following conclusions:

- Communication in an in-core HPF program is generated during the computation of (*in-core*) local array because the processor requires data which is not present in its memory. Both data and work distribution dictate the communication pattern.

- In an in-core SPMD (e.g. HPF) program, the communication can be performed *collectively* and is normally performed either before or (and) after the computation. This ensures that the computation does not violate the loosely synchronous constraint.

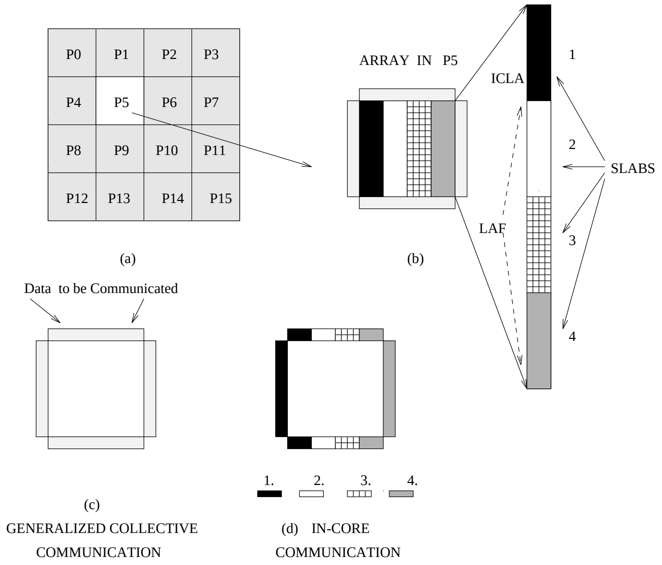

Figure 3: Figure illustrates compilation of out-of-core `FORALL`. The in-core slabs and the corresponding ghost areas are shown using distinct shades.

3.2 Communication Strategies for Compiling Out-of-core Computations

In an out-of-core application, computation is carried out in phases. Each phase reads a slab of data (or ICLA), performs computations using this slab and writes the slab back in the local array file. In this case processors may need to communicate because of the following reasons:

- computation of in-core local array requires data which is not present in memory during the computation involving ICLA; and
- ICLA contains data which is required by other processors for computation.

The required communication can be performed in two ways: (1) in a *generalized collective* manner, using out-of-core communication and (2) on a demand basis using in-core communication.

We will now illustrate the two communication approaches using the example of the 2-D elliptic solver (using Jacobi Relaxation) (Figure 2). We now assume that array A is an out-of-core array which is distributed over 16 processors. Each processor stores its local array into its local array file.

3.2.1 Generalized Collective Communication

In the collective communication method, the communication is performed *collectively* considering the entire global OOC local array. Every processor computes *the elements which are required for the computation of its OCLA but are not present in its OCLA*. These elements are communicated either before or after the computation on the OCLA. The communication from node j to node i involves following steps:

1. Synchronize (if necessary).

2. Node j checks if it needs to send data to other processors. If so, it checks if the required data is in memory. If the required data is in memory then the processor does not perform file I/O; if not, node j first sends a request to read data from disk and then receives the requested data from disk.

3. Node j sends data to node i.

4. Node i either stores the data back in local file or keeps it in memory. This would depend on whether the data required can be entirely used by the current slab in the memory; if this is not the case, then the received data must be stored in local files.

To illustrate these steps, consider processors 5 and 6 from Figure 3(a). Each processor performs operations on its OCLA in stages. Each OCLA computation involves repeated execution of three steps:

1. Fetching an ICLA

2. Computing on the ICLA

3. Storing the ICLA back in the local array file

Figure 3(b) shows the ICLA's using different shades. Figure 3(c) shows the data that needs to be fetched from other processors (called the *ghost area*). In the collective communication method, all the processors communicate the data in the ghost area before the computation on the OCLA begins. To illustrate the point that collective communication may require I/O, note that processor 5 needs to send the last column to processor 6. This column needs to be read from the local array file and communicated. Figure 4 shows the phases in the generalized collective communication method.

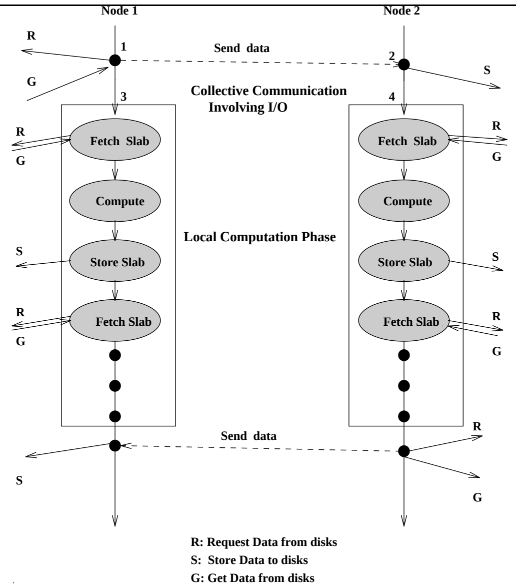

Figure 4: Generalized Collective Communication. Collective communication performed either before or after the local computation phase.

In the collective communication method, communication and computation are performed in two separate phases. As a result, the OCLA computation becomes *atomic*, i.e., once started it goes to completion without interruption. This method is attractive from the compiler point of view since it allows the compiler to easily identify and optimize collective communication patterns. Since the communication will be carried before the computation, this strategy is suitable for HPF `FORALL`-type of computations which have copy-in-copy-out semantics. In the above example, four shifts are required which result in disk accesses, data transfer and data storage (in that order).

While the *generalized collective* out-of-communication strategy is attractive for computations that proceed in phases, it is not applicable in situations where there are true dependences between in-core local arrays; in these situations, performing the I/O for the entire out-of-core array before the computation phase would violate dependence constraints.

3.2.2 In-core Communication Methods

In OOC computations, the communication may be performed in an entirely different way by just considering the communication requirements of the ICLA (or slab in memory) individually. In

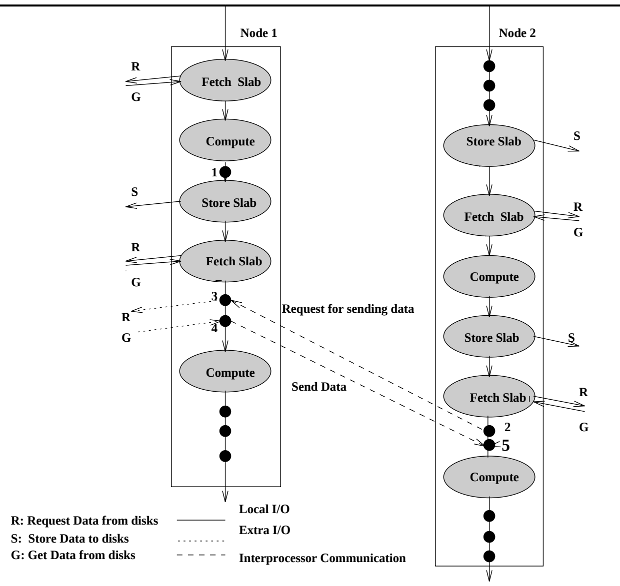

Figure 5: Receiver-driven In-core Communication. Node 2 requests data from Node 1 (point 2). Node 1 reads data from disks and sends to node 2 (points 4-5).

other words, communication set for each ICLA is generated individually. The basic premise behind this strategy is that if the data present in the memory can be used for communication while it is resident in memory, it may reduce the number of file I/O steps.

The in-core communication methods described next differ from the collective communication method in two aspects. In the two in-core communication methods:

1. communication is not performed *collectively*;

2. computation on the ICLA and communication are interleaved.

However the computation on the ICLAs is still carried out in an SPMD fashion. The data to be communicated is the data which is required for the computation of the ICLA but is not present in the memory (but it may be present in remote memory or another processor's file). The two types of in-core communication are the *receiver-driven in-core communication* and the *owner-driven in-core communication.*

Receiver-driven in-core communication (the receiver decides when to fetch) In this strategy, the communication is performed when a processor requires off-processor data during the computation of the ICLA. Figure 5 illustrates the receiver-driven communication method. Node 2 requires off-processor data at point 2 (Figure 5). Let us assume that the required data is computed by node 1 at point 1 and stored back on disk. When node 2 requires this data, it sends a request to node 1 to get this data. Node 1 checks if the data is in memory, else it reads the data from its local disk (point 3). After reading the data from disk, node 1 sends this data to node 2. Node 2 receives this data (point 5) and uses it during the computation of the ICLA.

This method can be illustrated using the example of the elliptic solver (Figure 3). Consider again processor 5. Figure 3(b) shows the different ICLAs for the processor 5. Let us consider slab 1 (shown by the darkest shade). The ghost area of this slab is shown in Figure 3(d). When this ICLA is in processor 5's memory, it requires data from processors 1, 4 and 9. Hence, processor 5 sends requests to processors 1, 4 and 9. After receiving the request, processors 1, 4 and 9 check whether the requested data is present in the ICLA or it has to be fetched from the local array file. Since processors 1 and 9 have also fetched their first slabs, the requested data lies in their main memory. Hence processors 1 and 9 can send the requested data without doing file I/O. However, since processor 4 has also fetched the first slab, the requested data (the last column which belongs to its fourth slab) does not lie in the main memory. Therefore, processor 4 has to read the data (last column) from its local array file and send it to processor 5. It is important to note that the shift collective communication pattern in the original OOC communication is broken into different patterns when in-core communication is considered.

Owner-driven in-core communication (The owner decides when to send) The basic premise of this communication strategy is that when a node computes on an ICLA and can determine that a part of this ICLA will be required by another node later on, this node sends that data while it is in its present memory. Note that in the receiver-driven communication, if the requested data is stored on disk (as shown in Figure 5), the data needs to be fetched from disk which requires extra I/O accesses. This extra I/O overhead can be reduced if the data can be *sent* to the processor either when it is computed or when it is fetched by its owner processor.

This approach is shown in Figure 6. Node 2 requires some data which is computed by node 1 at point 1. If node 1 knows that data computed at point 1 is required by node 2 later, then it can send this data to node 2 immediately. Node 2 can store the data in memory and use it when required (point 3). This method is called the *owner-driven communication* since in this method the

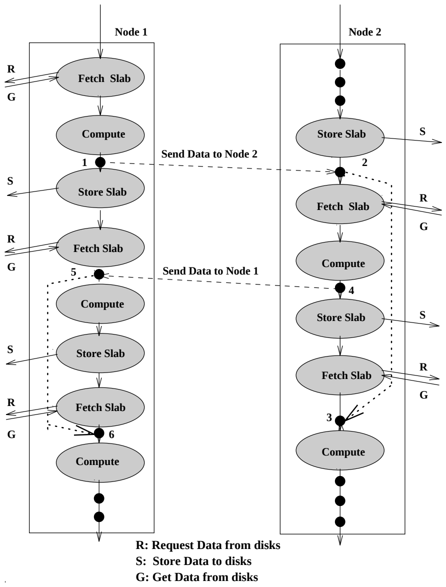

Figure 6: Owner-driven In-core Communication. Node 1 sends data to node 2 (points 1-2). Node 2 uses this data at point 3.

owner decides *when* to send the data. Communication in this method is performed before the data is used. This method requires knowledge of the data dependencies so that the processor can know beforehand what to send, where to send and when to send. It should be observed that this approach saves extra disk accesses at the sending node if the data used for communication is present in its memory.

In the elliptic solver solver, assume that processor 5 is operating on the last slab (slab 4 in Figure 3(d)). This slab requires the first column from processor 6. Since processor 6 is also operating on the last slab, the first column is not present in the main memory. Hence, in the receiver-driven communication method, processor 6 needs to fetch the column from its local array file and send it to processor 5. In the owner-driven communication method, processor 6 will send the first column to processor 1 during the computation of the first slab. Processor 5 will store the column in its local array file. This column will be then fetched along with the last slab thus reducing the I/O cost.

3.3 Choosing a communication method

The main difference between the two in-core communication methods and the collective communication method is that in the latter, communication and computation phases are separated. Since the communication is performed before (and/or after) the computation, an out-of-core computation consists of three main phases: local I/O, out-of-core communication, and computation. The local I/O phase reads and writes the data slabs from the local array files. The computation phase performs computations on in-core data slabs. The out-of-core communication phase performs communication of the out-of-core data. This phase redistributes the data among the local array files. The communication phase involves both inter-processor communication and file I/O. Since the required data may be present either on disk or in on-processor memory, three distinct access patterns are observed:

Read (write) from local logical disk: This access pattern is generated in the in-core communication method. Even though data resides in the logical disk owned by a processor, since the data is not present in the main memory it has to be fetched from the local array file.

Read from other processor's memory: In this case the required data lies in the memory of some other processor. In this case only memory-to-memory copy is required.

Read (write) from other processor's logical disk: When the required data lies in other processor's disk, communication has to be done in two stages. In case of data read, in the first stage the data has to be read from the logical disk and then communicated to the requesting processor. In case of data write, the first phase involves communicating data to the processor that owns the data and then writing it back to the disk. The processor reading and writing the data is said to perform *extra* file I/O.

The overall time required for an out-of-core program can be computed as a sum of times for local I/O T_{lio}, in-core computation T_{comp} and communication T_{comm}.

$$T = T_{lio} + T_{comm} + T_{comp}$$

T_{lio} is the time that a processor spends in transferring OCLA between memory and its LAF purely for the purpose of performing the computation on its ICLAs. T_{lio} depends on two factors, the number of slabs to be fetched into memory and the I/O access pattern. The number of slabs to be fetched is dependent on the size of the local array and the size of the available in-core memory. The I/O access pattern is determined by the computation and the data storage patterns. The I/O access pattern determines the number of disk accesses. T_{comm} can be computed as a sum of I/O time and inter-processor communication time. The I/O time depends on (1) whether the disk to be accessed is local (owned by the processor) or it is owned by some other processor, (2) the number of data slabs to be fetched into memory and, (3) the number of disk accesses which is determined by the I/O access patterns. The inter-processor communication time depends on the size of data to be communicated and the speed of the communication network. Finally the computation time depends on the size of the data slabs (or size of available memory). Hence, the overall time for an out-of-core program depends on the communication pattern, available memory and I/O access pattern.

3.3.1 Heuristic used

Here we present a simple heuristic that can be used to decide which communication method to use. Fox and Li and Chen [GMG$^+$88, LC91] showed that a compiler can take advantage of the highly regular communication patterns displayed by many computation and can match the pattern to collective communication routines such as shifts, broadcasts, all-to-all communication, transposes etc. We use the following technique to recognize communication patterns:

- compare the subscript expression of each distributed dimension of a read (RHS) reference with the corresponding dimension of the LHS reference, and the template dimension to to which these are aligned;
- identify programmer-specified communications such as transpose in Fortran 90 and High Performance Fortran;
- if the subscript difference is independent of loop index variables (or FORALL index subscripts) then the communication is a shift communication;
- if a subscript expression in a distributed dimension is loop-invariant, then the communication is a broadcast;
- if the two subscript expressions have differing axis alignment, then there may be a need for transpose or broadcast;

If the code requires only shift communication, in-core communication incurs much less overhead and is the method of choice. If a mix of patterns including transpose (or some form of all-to-all) exists, then the communication required involves many processors and large volume of data. In this case, frequent file I/O accesses to local data just for communication results in high overhead. The in-core methods incur a very high overhead due to frequent I/O accesses to relatively small pieces of data, compared to the generalized collective technique. Thus, the generalized collective method is chosen with such communication patterns. Work is in progress in deriving a heuristic that models costs accurately and chooses among the methods. As we illustrate in the next section, the simple heuristic works in practice.

4 Implementation and Experimental Performance Results

The local placement model is implemented using a runtime library as part of the PASSION compiler [TBC+94, Tha95, CBD+95]. The runtime routines can be classified into

- **mapping routines:** for mapping virtual files to physical files, for creating local files for each processor from the input data files, *etc.*
- **access routines:** for staging data in the memory once local files are created, reading and writing *etc.*
- **collective communication routines:** for implementing communication patterns such as *shifts* that provide file access for out-of-core data, *etc.*

In addition, PASSION includes routines that generate the local array files according to the user-specified data mapping.

This section presents performance results of OOC applications compiled using the communication strategies and the simple heuristic presented in this paper. We demonstrate that under different circumstances, different strategies are chosen by our system based on the communication patterns. We also show performance by varying the amount of memory available on the node to store ICLAs.

The applications were run on the Intel Touchstone Delta machine at Caltech. The Touchstone Delta has 512 compute nodes arranged as a 16×32 mesh and 32 I/O nodes connected to 64 disks.

Receiver-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.1	1.2	1.4	1.5
LIO	61.0	65.6	69.9	75.6
Owner-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.1	1.1	1.2	1.3
LIO	53.7	55.3	58.6	59.2
Collective Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	36.5	37.9	38.0	39.0
LIO	52.5	62.3	66.6	66.8

Table 1: Out-of-core 2D-Jacobi (4K×4K) on 16 Processors. Time in seconds.

It supports a parallel file system called the Concurrent File System (CFS).

4.1 Two-Dimensional Out-of-core Elliptic Solver

Our system recgonizes that the communication pattern here is a shift and chooses in-core communication; in our experiments, we measured the times for both the in-core methods and the out-of-core method. Table 1 presents performance of 2D out-of-core elliptic solver using the three communication strategies. The problem size is 4K×4K array of real numbers, representing 64 MBytes of data. The data distribution is (`BLOCK, BLOCK`) in two dimensions. The number of processors is 16 (with an 4×4 logical mapping). The size of the ICLA was varied from 1/2 of the OCLA to 1/16 of the OCLA. Tables 2 and 3 present the performance of the same sized problem for 64 and 256 processors respectively; while Tables 4 and 5 present the performance of the 2D out-of-core elliptic solver using 8K×8K real array for 64 and 256 processors respectively. In each case, the processors are logically arranged as a square mesh.

Each table shows the following components of the total execution time: Local I/O (LIO) time, and the Communication time (COMM) for the *generalized collective* out-of-core communication method, the *receiver-driven* in-core communication method and the *owner-driven* in-core communication method. The computation time was found to be the same for the three communication methods for a given problem size; hence, it is not shown in the tables. The experiment was performed for four values of the memory ratio (ICLA/OCLA). From these results we make the following

Receiver-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.1	1.1	1.1	1.1
LIO	57.7	61.3	62.5	72.7
Owner-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.1	1.1	1.1	1.0
LIO	54.4	61.2	67.0	73.5
Collective Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	250.8	261.7	289.0	279.4
LIO	51.9	63.9	128.3	200.4

Table 2: Out-of-core 2D-Jacobi (4K×4K) on 64 Processors. Time in seconds

Receiver-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	2.1	1.9	2.0	2.2
LIO	62.7	66.6	73.1	79.3
Owner-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	2.0	2.0	1.8	2.0
LIO	62.3	66.5	72.6	79.2
Collective Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	65.2	69.8	70.4	75.3
LIO	64.9	67.6	78.7	87.9

Table 3: Out-of-core 2D-Jacobi (4K×4K) on 256 Processors. Time in seconds.

Receiver-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.9	1.8	2.1	2.3
LIO	215.5	218.5	224.4	246.7
Owner-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	1.9	2.0	2.0	2.6
LIO	218.7	220.8	235.3	244.0
Collective Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	155.9	165.0	175.7	185.7
LIO	251.5	268.4	312.8	348.2

Table 4: Out-of-core 2D-Jacobi (8K$\times$8K) on 64 Processors. Time in seconds

Receiver-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	3.2	3.6	3.8	4.0
LIO	268.5	277.6	281.3	282.4
Owner-driven In-core Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	7.1	7.5	7.8	7.9
LIO	247.8	264.0	284.3	305.8
Collective Comm. Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	247.2	260.1	265.7	274.0
LIO	302.8	334.2	358.2	388.3

Table 5: Out-of-core 2D-Jacobi (8K$\times$8K) on 256 Processors. Time in seconds

observations:

1. COMM is largest in the collective communication method. This is because, each processor needs to read boundary data from a file and write the received boundary data into a file. Since the boundary data is not always consecutive, reading and writing of data results in many small I/O accesses. This results in an overall poor I/O performance. However, in this example, for the collective communication method, COMM does not vary significantly as the size of the available memory is varied. As the amount of data to be communicated is relatively small, it can fit in the on-processor memory. As a result, communication does not require stripmining (i.e. becomes independent of the available memory size). If the amount of data to be communicated is greater than the size of the available memory, then COMM will vary as the size of the available memory changes.

2. *Owner-driven* in-core communication, even though it performs the best, does not provide significant performance improvement over the *receiver-driven* in-core communication method. The main reason that is due to lack of on-processor memory, the receiver processor stores that received data on disk and reads it when needed. This results in extra I/O accesses.

3. In both the receiver and owner-driven communication methods, COMM does not vary significantly as the amount of available memory is changed. In the 2-D Jacobi method, the inter-processor communication forms a major part of in-core communication. Since the in-core communication requires small I/O, the in-core communication cost is almost independent of the available memory.

4. As the amount of memory is decreased, more I/O accesses are needed to read and store the data. This leads to an increase in the cost of LIO. It should be noted that the local I/O and the I/O during communication are the dominant factors in the overall performance.

5. In the two in-core communication methods, the structured communication pattern (*shift*) gets distributed into several unstructured patterns (for each in-core data slab). In order to optimize these communication patterns, we need to use the owner/receiver-driven communication methods.

The performance of 2D Jacobi solver illustrates similar patterns for a larger problem size (8K×8K) and for larger processor grids (64 and 256 processors). Clearly, in all cases, the out-of-core communication strategy performs the worst in terms of the communication time due to the fact that communication requires many small I/O accesses.

As we will observe with the 2-D FFT application next, when the communication is not a shift communication (transpose in this case) and the number of processors communicating is large, out-of-core communication provides better performance.

4.2 Two-Dimensional Fast Fourier Transform

This application performs two-dimensional Fast Fourier Transform (FFT). The FFT is an $O(N\log(N))$ algorithm to compute the discrete Fourier transform (DFT) of a $N \times N$ array. On a distributed memory machine, FFT is normally performed using transpose/redistribution based algorithms. One way to perform a transpose based FFT on a $N \times N$ array is as follows:

1. Distribute the array along one dimension according to (*,`BLOCK`) distribution.
2. Perform a sequence of 1D FFTs along the non-distributed dimension (column FFTs).
3. Transpose the intermediate array x.
4. Perform a sequence of 1D FFTs along the columns of the transposed intermediate array x^T.

Note that the above algorithm does not require any communication during 1D FFTs. However this algorithm requires a transpose (redistribution) which has an all-to-all communication pattern. Hence, the performance of the transpose based algorithm depends on the cost of the transpose. The heuristic that we use chooses the out-of-core method because of the transpose. Figure 2 presents an HPF program to perform 2D FFT. The `DO_1D_FFT` routine performs 1D FFT over the j^{th} column of the array.

The basic 2D-FFT algorithm can be easily extended for out-of-core arrays. The OOC 2D FFT algorithm also involves three phases. The first and third phase involves performing 1D FFT over the in-core data. The transposition phase involves communication for redistributing the intermediate array over the disks. Thus, the performance of the out-of-core FFT depends on the I/O complexity

```
          PROGRAM FFT
          REAL A(N,N)
!HPF$ PROCESSORS PR(P)
!HPF$ TEMPLATE T(N,N)
!HPF$ DISTRIBUTE T(*,BLOCK) ONTO PR
!HPF$ ALIGN WITH T ::  A
          FORALL(J=1:N)
              DO_1D_FFT(A(:,J))
          END FORALL
          A=TRANSPOSE(A)
          FORALL(J=1:N)
              DO_1D_FFT(A(:,J))
          END FORALL
          STOP
          END
```

Figure 7: An HPF Program for 2D FFT. Sweep of the 1D FFT in (X/Y) dimension is performed in parallel.

of the out-of-core transpose algorithm. The transpose can be performed using either collective communication or using the two in-core communication methods.

4.2.1 Collective Communication

In the collective communication method, the transposition is performed after the computation in the first phase as a collective operation. Figure 8(a) shows the communication pattern for the out-of-core transpose. Each processor fetches data blocks (ICLAs) consisting of several subcolumns from its local array file. Each processor then performs an in-core transpose of the ICLA. After the in-core transpose, the ICLAs are communicated to the appropriate processor which stores them back in the local array file.

4.2.2 In-core Communication

In this method, the out-of-core 2D FFT consists of two phases. In the first phase, each processor fetches a data slab (ICLA) from the local array file, performs 1D FFTs over the columns of the ICLA. The intermediate in-core data is then transposed. In the second phase, each processor fetches ICLAs from its local array file and performs 1D FFTs over the columns in the ICLA.

Figure 8(b) shows the in-core transpose operation. The figure assumes that the ICLA consists

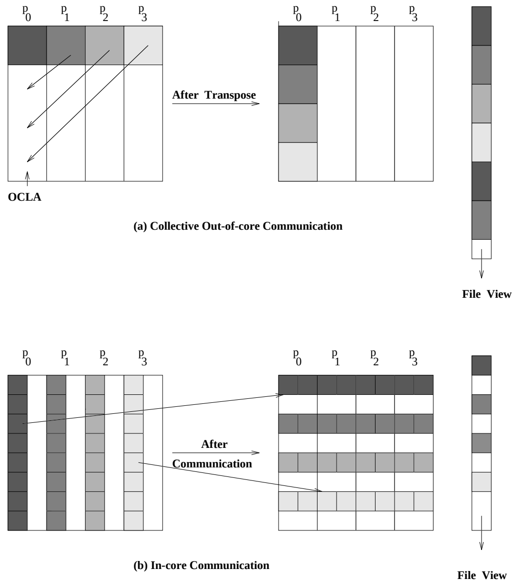

Figure 8: Out-of-core transpose algorithm. Communication can be performed in two ways: using (a) collective out-of-core method; or (b) in-core method

of one column. After the in-core transpose, the column is distributed across all the processors to obtain corresponding subrows. Since the data is always stored in the column major order, the subrows have to be stored using a certain stride. This requires a large number of small I/O accesses.

Note that in the transpose based FFT algorithm, the communication pattern do not change when in-core communication methods are used. As a result, two different in-core communication methods are not required for communication optimization. We present results only for the owner-driven in-core communication method.

4.2.3 Experimental Results

Tables 6, 7 and 8 present performance results for the out-of-core 2D FFT using the two communication strategies. The experiment was performed for two problem sizes, 4K×4K and 8K×8K array of real numbers, representing 64 MBytes and 256 MBytes respectively. The arrays were distributed in column-block form over 16 and 64 processors arranged in a logical square mesh. The amount of available memory was varied from 1/2 to 1/16 of the local array size. Each table shows the local I/O time (LIO) and communication time (COMM). As in the case of the elliptic solver, computation time has been omitted from the tables. Tables 6, 7 and 8 illustrate the variation of the total execution time with respect to the ratio (ICLA/OCLA).

In-core Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	933.5	1767.3	3382.7	6514.5
LIO	13.2	22.4	40.8	79.0
Collective Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	64.5	103.5	134.8	175.9
LIO	13.3	27.4	49.6	92.4

Table 6: Out-of-core FFT (4K×4K) on 16 Processors. Time in seconds

In-core Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	933.5	1767.3	3382.7	6514.5
LIO	13.2	22.4	40.8	79.0
Collective Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	64.5	103.5	134.8	175.9
LIO	13.3	27.4	49.6	92.4

Table 7: Out-of-core FFT (4K×4K) on 64 Processors. Time in seconds

In-core Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	923.8	982.1	1827.1	3466.8
LIO	23.3	26.9	44.7	90.1
Collective Communication Method				
Cost	Ratio=$\frac{1}{2}$	Ratio=$\frac{1}{4}$	Ratio=$\frac{1}{8}$	Ratio=$\frac{1}{16}$
COMM	149.8	171.1	193.2	260.9
LIO	28.3	35.5	58.8	108.2

Table 8: Out-of-core FFT (8K×8K) on 64 Processors. Time in seconds

From these results, we observe the following:

1. For the out-of-core FFT, COMM for the in-core communication method is larger than that for the collective communication method. COMM includes the cost of performing inter-processor communication and I/O. The in-core communication method requires a large number of small I/O accesses to store the data. In both in-core and collective communication methods, COMM increases as the amount of available memory is decreased.

2. The in-core communication method requires less LIO than the collective communication method. This is due to the fact that in the in-core communication method, part of the local I/O is performed as a part of the out-of-core transpose.

3. As the number of processors and grid size is increased, the collective communication performs better but performance of the in-core communication method degrades.

5 Communication Strategies in Virtual Memory Environments

So far, we presented communication strategies for OOC computations, where data staging was done explicitly by a compiler at the application level. This staging is performed by using runtime routines (e.g. see [TBC+94]). In this section, we briefly discuss how these strategies can be used when node virtual memory on nodes may be available.

Assume that node virtual memory is provided on an MPP, where the address space of each processor is mapped onto a disk or several disks. For example on an IBM SP2, each node has a disk associated with it for paging. Also assume that node has a TLB-like mechanism to convert virtual addresses to the corresponding physical accesses.

In such an environment, where demand paging is performed for accesses for data not present in the memory, sweep through a computation will involve page faults and accesses to pages from disks when needed. Two types of page faults are possible in this environment; namely, page faults caused by local accesses called *local page faults,* and page faults caused by data required by remote processors due to communication requirements termed as *remote page faults.* The former is equivalent to local I/O in the explicit method for data accesses in form of slabs using the compiler and runtime support. The latter is equivalent to the I/O performed during communication in the explicit method.

If no compiler and runtime support for stripmining computations, and no (explicit) access dependent support for I/O is provided, paging at the system level can be very expensive. Performance studies of the virtual memory provided by the OSF/1 operating system on the Intel Paragon have shown that the paging-in and paging-out of data from the nodes drastically degrade the user code performance. Also, most of the other massively parallel systems at present, such as the CM-5, iPSC/860, Touchstone Delta, and nCUBE-2, do not support virtual memory on the nodes. On the other hand, if explicit support by the compiler and the runtime system is provided to perform explicit I/O at the application level, all techniques discussed earlier in this paper can be applied in the systems that do provide virtual memory at the node level. In these cases, a compiler must translate a source program (for example in HPF) into a code which explicitly performs I/O. Even if the node virtual memory is supported, paging mechanisms are not known to handle different access patterns efficiently. The following briefly discusses the communication scenarios.

In the virtual memory environment, the computation can be stripmined so that a set of pages can be fetched in the memory. When the computation of data from these pages is over, either the entire or a part of the slab is stored back on disk. Suppose the local computation requires data which does not lie in the in-core slab (receiver-driven in-core communication). In this case, a *page fault* will occur. Since the required data will lie either on the local disk or on the disk owned by some other processor, both *local page faults* and *remote page faults* are possible. A *local* page fault fetches data from the local disk. A *remote* page fault fetches data from a distant processor. *Remote* page fault results in inter-processor communication. If the owner processor does not have the required data in its memory, a *local* page fault will occur else the owner processor can send the data (or page) without accessing its own disk. This situation is very similar to the communication in the out-of-core scenario.

Since the owner/receiver-driven communication strategies allow the nodes more control over how and when to communicate, these strategies are suitable for virtual memory environments. Consider the owner-driven in-core communication method. Suppose the processor A knows that a particular page will be required in the future by processor B. Then processor A can either send the page to processor B immediately or retain this page (this page will not be replaced) until processor B asks for it. Processor B also knowing that this page will be used later will not replace it. Further optimizations can be carried out by modifying basic page-replacement strategies. Standard LRU strategy can be changed to accommodate access patterns across processors, i.e. if a page owned by a processor A is recently used by a processor B, then this page will not be replaced in processor A. Work is in progress in implementing these techniques.

6 Related Work

Only in the recent past, compilation of out-of-core parallel problems has attracted the attention of research community. Other than the PASSION group, there are two other research groups working on developing compilation techniques to support out-of-core parallel problems. Paleczny et al. at Rice University are extending the Fortran D framework for compiling out-of-core problems [PKK95]. At Dartmouth, Cormen's group is working on an out-of-core C* compiler[CC94].

However, extensive research has gone into devising different compiler optimizations to improve memory access performance of sequential programs. Abu-Sufah in 1979 demonstrated the application of loop distribution and loop fusion for improving the memory access costs of numerical problems [AS79]; while Trivedi showed that opportunities for demand prefetching can be identified from a program's syntax [Tri77]. Another important area of investigation was use of iteration tiling for exploiting hierarchical memory [IT88, SD90, RS92]. Several researchers have analyzed communication patterns in parallel programs. Important work in this area includes [GMG+88, LC91, Gup92, HKT92].

There has been a lot of interest in developing runtime libraries for improving I/O performance of

I/O intensive (not just out-of-core) parallel applications. Chameleon was the first runtime system which provided extensive support for parallel I/O [GGL93]. del Rosario et al. proposed a two-phase access strategy for efficient access of distributed arrays [dRBC93, BdRC93]. This strategy was later extended by Kotz to optimize disk accesses [Kot94]. PASSION runtime system builds on [BdRC93] and provides runtime routines to access distributed multidimensional arrays in the Local as well as Global Placement Models [CBD+95]. Similar runtime projects include PANDA [SCJ+95] which uses array *chunking* to improve I/O performance, PIOUS [MS94] which is a parallel I/O runtime system based on PVM, and MPI-IO [PSCF94] which provides parallel I/O using the MPI message-passing interface. Further details about these projects and many others are available at the Parallel I/O archive (`http://www.cs.dartmouth.edu/pario.html`).

7 Conclusions

We have shown that communication in the out-of-core problems requires both inter-processor communication and file I/O. For out-of-core programs, the compilation strategy depends on how data is stored as well as how it is accessed. We use the Local Placement Model to organize, view, and access out-of-core data. Using this model, we described the impact of different communication strategies in the context of out-of-core compilation, and their impact on two commonly used example codes. We showed that a simple heuristic based on identifying communication patterns, can be useful in deciding which communication strategy to use for a given program. Although the techniques described in this paper are discussed with respect to HPF, they are applicable to data parallel languages in general.

We described three different ways of implementing communication that arises in out-of-core programs. The *generalized collective* out-of-core communication method performs communication in a collective way while the in-core communication methods (*owner-driven* and *receiver-driven*) communicate in a demand basis by only considering the communication requirements of the data slab which is present in memory. In-core communication methods are suitable for problems in which the communication volume and the number of processors performing communication is small. Owner-driven in-core communication method can be used to improve communication performance by optimizing file I/O. Out-of-core communication method is useful for problems having large communication volume. In both methods, the communication cost depends on the amount of file I/O. The heuristic we use chooses the in-core methods if the only communication pattern is shift; if a transposeor other communication such as all-to-all is required, it chooses th egeneralized collective out-of-core strategy. We demonstrated (using the PASSION implementation) through experimental results that different communication strategies are suitable for different types of computations and that the simple heuristic we use seems to work in practice. We believe, these methods could be easily extended to support node virtual memories on distributed memory machines. Work is in progress in this area. Currently, we are investigating different approaches for eliminating extra file I/O in in-core communication methods [BCR95]. We are also working on accurately modeling costs and more refined heuristics in choosing communication strategies. Finally, we plan to investigate the Global Placement Model.

Acknowledgments

The work of R. Bordawekar and A. Choudhary was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Advanced Research Projects Agency(ARPA) Administered by US Army at Fort Huachuca. R. Bordawekar is also supported by a Syracuse University Graduate Fellowship. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768, an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09. This work was performed in part using the Intel Touchstone Delta and Paragon Systems operated by Caltech on behalf of the Center for Advanced Computing Research (CACR). Access to this facility was provided by CRPC. Finally, the authors would like to acknowledge David Kotz for creating and maintaining the excellent Parallel I/O archives.

References

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois, 1979.

[BCF+93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[BCK+95] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Palecnzy. A Model and Compilation Strategy for Out-of-core Data Parallel Programs. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10, ACM Press, July 1995.

[BCR95] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Out-of-core Stencil Codes. Technical report, Scalable I/O Initiative, Center of Advanced Computing Research, California Insititute of Technology, November 1995. In preparation.

[BdRC93] R. Bordawekar, J. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing'93*, pages 452–461, November 1993. URL: `ftp://erc.cat.syr.edu/ece/choudhary/PASSION/sc93.ps.Z`.

[Bor96] Rajesh Bordawekar. *Compiling I/O Intensive Data Parallel Programs for Distributed Memory Machines*. PhD thesis, Electrical and Computer Engineering Dept., Syracuse University, 1996. In preparation.

[CBD+95] Alok Choudhary, Rajesh Bordawekar, Apurva Dalia, Sivaram Kuditipudi, and Sachin More. PASSION User Manual, Version 1.2, Syracuse University, October 1995. URL: `http://www.cat.syr.edu/passion.html`.

[CBH+94] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994. URL: `ftp://erc.cat.syr.edu/ece/choudhary/PASSION/passion_report.ps.Z`.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994. URL: `ftp://ftp.cs.dartmouth.edu/pub/CS-techreports/TR94-243.ps.Z`.

[CF94] P. Corbett and D. Feitelson. Overview of the Vesta Parallel File System. In *Proceedings of the Scalable High Performance Computing Conference*, pages 63–70, May 1994.

[dRBC93] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Runtime Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, April 1993.

[dRC94] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[Ger95] Jerry Gerner. Input/Output on the IBM SP2: An Overview, 1995. URL: `http://www.tc.cornell.edu/SmartNodes/Newsletters/IO.series/intro.html`. Online Web Article.

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[GMG+88] G.Fox, M.Johnson, G.Lyzenga, S.Otto, J. Salmon, and D.Walker. *Solving Problems on Concurrent Processors.* Prentice Hall,Eaglewood Cliffs,NJ, 1988.

[Gup92] Manish Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, College of Engineering, University of Illinois at Urbana-Champaign, September 1992.

[Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. In *Languages, Compilers and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[IT88] Francois Irigoin and Remi Triolet. Supernode Partitioning. *POPL'88, Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1988.

[Kot94] David Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. URL: `http://www.cs.dartmouth.edu/ dfk/papers/diskdir.html`. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[LC91] Jinke Li and Marina Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[MMWY95] Larry F. Meadows, Douglas Miles, Clifford Walinsky, and Mark Young. The Intel Paragon HPF Compiler. In *Proceedings of Intel Supercomputing User's Group Meeting 1995*, June 1995. URL: `http://www.pgroup.com/docs/papers/isug/ISUG_95.html`.

[MS94] Steven A. Moyer and V. S. Sunderam. A Parallel I/O System for High-Performance Distributed Computing. In *Proceedings of the IFIP WG10.3 Working Conference on*

Programming Environments for Massively Parallel Distributed Systems, 1994. URL: `ftp://ftp.mathcs.emory.edu/pub/vss/piousifip94.ps`.

[otSII94] Applications Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. Technical Report CCSF-38, Concurrent Supercomputing Consortium, Caltech, Pasadena, CA 91125, January 1994. Scalable I/O Initiative Working Paper No. 1.

[Pie89] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of 4^{th} Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160, Match 1989.

[PKK95] Michael Paleczny, Ken Kennedy, and Charles Koelbel. Compiler Support for Out-of-core Arrays on Parallel Machines. In *Fifth Symposium on Frontiers of Massively Parallel Computation*, pages 110–118, February 1995.

[PSCF94] Jean-Pierre Prost, Marc Snir, Peter Corbett, and Dror Feitelson. MPI-IO, A Message-Passing Interface for Concurrent I/O. Technical Report RC 19712 (87394), IBM T.J. Watson Research Center, August 1994.

[RS92] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992. URL: `ftp://gate.ee.lsu.edu/pub/jxr/papers/journal/jpdc_92_tiling.PS`.

[SCJ+95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[SD90] R. Schriber and J. Dongarra. Automatic Blocking of Nested Loops. Technical report, Research Institute for Advanced Computer Science, May 1990.

[TBC+94] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1994.
URL: `ftp://erc.cat.syr.edu/ece/choudhary/PASSION/splc94_passion_runtime.ps.Z`.

[Tha95] Rajeev Thakur. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs.* PhD thesis, Dept. of Electrical and Computer Engineering, Syracuse University, May 1995. URL: `http://www.mcs.anl.gov/home/thakur/phd_thesis.ps`.

[Tri77] Kishor S. Trivedi. On the Paging Performance of Array Algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.