Communication Strategies for Out-of-core Programs on Distributed Memory Machines*

Rajesh Bordawekar **Alok Choudhary**
ECE Dept., and Northeast Parallel Architectures Center
Syracuse University, Syracuse, NY 13244
rajesh, choudhar@npac.syr.edu

Abstract

In this paper, we show that communication in the out-of-core distributed memory problems requires both inter-processor communication and file I/O. Given that primary data structures reside in files, even communication requires I/O. Thus, it is important to optimize 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, termed as the "out-of-core" communication method, follows a loosely synchronous model. Computation and Communication phases in this case are clearly separated, and communication requires permutation of data in files. The second method, termed as "demand-driven-in-core communication" considers only communication required of each in-core data slab individually. The third method, termed as "producer-driven-in-core communication" goes even one step further and tries to identify the potential (future) use of data while it is in memory. We describe these methods in detail and provide performance results for out-of-core applications; namely, two-dimensional FFT and two-dimensional elliptic solver. Finally, we discuss how "out-of-core" and "in-core" communication methods could be used in virtual memory environments on distributed memory machines.

*This work was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and IBM Corp., and in part by USRA CESDIS Contract # 5555-26. This work was performed in part using the Intel Touchstone Delta and Paragon Systems operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.

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 [Ini94, dRC94].

Since main memories may not be large enough to hold data of order of Gbytes, data needs to be stored on disks and fetched during execution of the program. 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, iPSC/860, Touchstone Delta, CM-5, SP-1, nCUBE2 etc. provide some kind of hardware and software support for parallel I/O [CFPB93, Pie89, DdR92]. del Rosario and Choudhary [dRC94] give an overview of the various issues in high performance I/O.

Data parallel languages like High Performance Fortran (HPF) [For93b] 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. We are developing compilation techniques to handle out-of-core applications [TBC94a, BCT94]. The compiler takes an HPF program as an input and produces the corresponding node program with calls to runtime routines for I/O and communication. The compiler *stripmines* the computation so that only the data which is in memory is operated on. Computation of in-memory data often requires data which is not present in processor's memory. 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 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 2-D elliptic solver using Jacobi relaxation is used to illustrate these strategies. Section 4 presents experimental performance results for two out-of-core applications, namely two-dimensional Jacobi Relaxation and two-dimensional FFT using these communication strategies. Section 5 describes how these communication strategies could be used in virtual memory environments. Conclusion are presented in section 6.

2 Computation Model

2.1 Out-of-core Computation Model

A computation is called an *out-of-core* (**OOC**) computation if the data which is 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 lies in 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 required). 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.

One may ask a natural question - why not use paging to handle large scale problems instead of explicitly performing data staging at the application level (using compiler and runtime support)? The following reasons collectively provide an answer. First, paging is not known to perform well when high-performance in an application is the issue even in sequential or vector computers. Second, not many parallel computers provide node level paging, especially those that incorporate a notion of a parallel program or collective operations. Third, some systems that do provide paging, the performance is shown to be poor [SS93]. Forth, the concept of locality in uniprocessors on which paging is based does not directly extend to parallel computers because of different interleaving of accesses by processors. Thus it is important to provide explicit support for data staging using a runtime system which has a notion of collective operation built into it.

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 distribution 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 [For93a, KLS+94]. 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`(m) or `CYCLIC`(m). In a `BLOCK`(m) distribution, contiguous blocks of size m are distributed among the processors. In a `CYCLIC`(m) distribution, blocks of size m 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 it. Our main assumption is that local arrays are stored in files from which the data is staged into main memory. When the global array is an out-of-core array, the corresponding local array will have to be also stored in files. 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, TBC94a]. 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 distribut-

ing a file into several logical file partitions, each belonging to a distinct processor [CFPB93, CF94]. In this paper we only consider the local placement model.

2.3 Local Placement Model

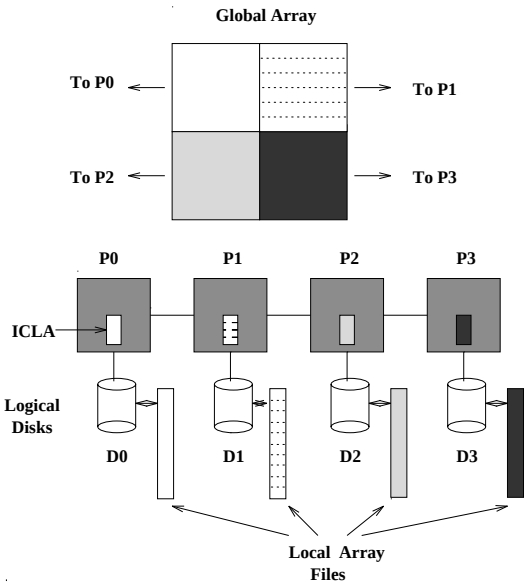

Figure 1: Local Placement Model

In the Local Placement Model, the local array of each processor is stored in a 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 file 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 is fetched and stored in main memory. The size of this portion depends on the amount of memory available. The portion of the local array which is in main memory is called the *In-Core Local Array* (**ICLA**). 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.

3 Communication Strategies in 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 in In-core Computations

Consider the HPF program fragment from Figure 2. The HPF program achieves parallelism using (1) Data Distribution and (2) 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 by the compiler during the compilation of parallel constructs like `FORALL` or array assignment statements (line 6, 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.

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

Figure 2: An HPF Program Fragment for Two-dimensional Jacobi Computations. The array A is distributed in `BLOCK-BLOCK` fashion over 16 processors.

In the example, it can be observed that for the array assignment (lines 6-7), 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 so called *collective* communication. After the com-

munication is performed, each processor begins computations on the local array. From this analysis, we can arrive to following conclusions

1. 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 it's memory. Both data distribution and work distribution strategies dictate the communication pattern.
2. 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 loosely synchronous constraint.

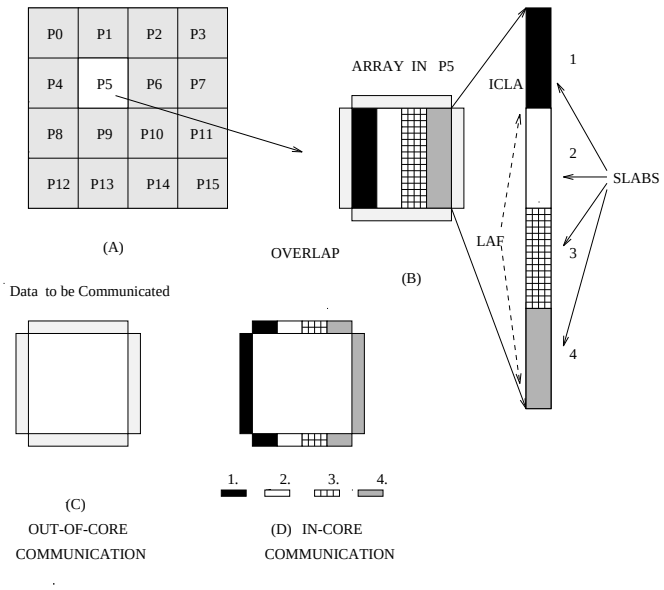

Figure 3: Figure illustrates compilation of out-of-core difference equation. The in-core slabs and the corresponding ghost areas are shown using distinct shades.

3.2 Communication Strategies in 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 (1) computation of in-core local array requires data which is not present in memory during the computation involving ICLA and, (2) ICLA contains data which is required by other processors for computation. The communication can be performed in two ways: (1) in a *collective* manner, using **Out-of-core Communication** and (2) in a demand basis, termed as "**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 it's local array into it's local array file.

3.2.1 Out-of-core Communication

In the out-of-core communication method, the communication is performed *collectively* considering the entire OOC local array. All processors compute *the elements which are required for the computation of the OCLA but are not present in the OCLA*. These elements are communicated either before or after the computation on the OCLA. The communication from node i to node j 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 not, node j first sends a request to read data from disk and then receives the requested data from disk. If the required data is in memory then the processor does not perform file I/O.
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 not, 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 it's 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 out-of-core method, all the processors communicate this data before the computation on the OCLA begins. To illustrate the point that out-of-core communication requires 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 out-of-core communication method.

In the out-of-core 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).

3.2.2 In-core Communication

For 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 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.

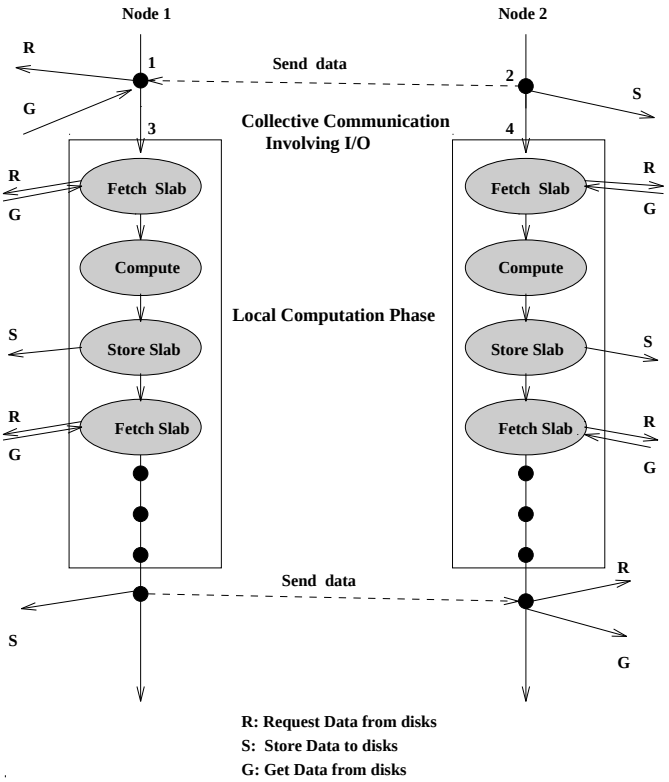

Figure 4: Out-of-core Communication. Collective communication performed either before or after the local computation phase.

In-core communication method differs from the out-of-core communication method in two aspects, (1) in the in-core communication method, communication is not performed *collectively*. The two phases, 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 in-core communication can be further divided into two types, (1) Demand-driven Communication and (2) Producer-driven Communication.

- *Demand-driven In-core Communication (Consumer decides when to fetch)*

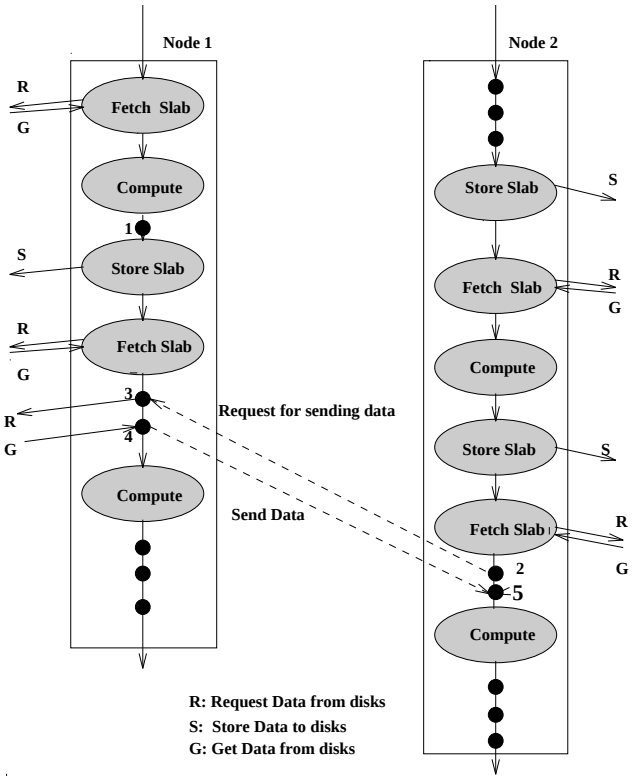

Figure 5: Demand-driven In-core Communication. Node 2 requests data from Node 1 (point 2). Node 1 reads data from disks and sends to node 2 (points 4-5).

In this strategy, the communication is performed when a processor requires off-processor data during the computation of the ICLA. Figure 5 illustrates the demand-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 (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'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 the first slab, the requested data lies in the 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 does not lie in the main memory. Therefor, processor 4 has to read the data (last column) from it's 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.

- *Producer-driven In-core Communication (Producer decides when to send)*

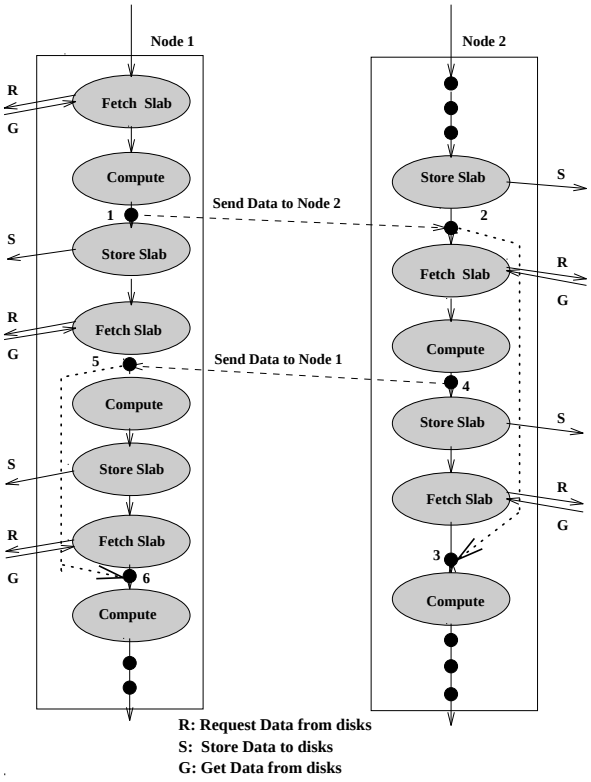

Figure 6: Producer-driven In-core Communication. Node 1 sends data to node 2 (points 1-2). Node 2 uses this data at point 3.

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 demand-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 it's 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 *Producer-driven communication* since in this method the producer (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 example of the elliptic 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 demand-driven communication method, processor 6 needs to fetch the column from it's local array file and send it to processor 5. In the producer-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.

3.2.3 Discussion

The main difference between the in-core and out-of-core communication methods is that in the latter, communication and computation phases are separated. Since the communication is performed before 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

1. Read(write) from my 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.

2. 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.

3. 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 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 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} depends on (1) Number of slabs to be fetched into memory and, (2) 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.

4 Experimental Performance Results

This section presents performance results of OOC applications implemented using the communication strategies presented in this paper. We demonstrate that under different circumstances, different strategies may be preferred, i.e., no one strategy is universally good. We also show performance by varying the amount of memory available on the node to store ICLAs.

The applications were implemented on the Intel Touchstone Delta machine at Caltech. The Touchstone Delta has 512 compute nodes arranged as a 16×32 mesh and 16 I/O nodes connected to 32 disks. It supports a parallel file system called the Concurrent File System (CFS).

4.1 Two-Dimensional Out-of-core Elliptic Solver

Figures 7 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 a 4*4 logical mapping). The size of the ICLA was varied from 1/2 of the OCLA to 1/16 of the OCLA.

Each graph shows three components of the total execution time; namely, Local I/O (LIO) time, Computation time (COMP) and the Out-of-core Communication time (COMM) for Out-of-core Communication Method, Demand-driven In-core Communication Method and Producer-driven In-core Communication Method. From these results we make the following observations

- COMP remains constant in all the three communication methods. This is expected as the amount of computation is the same for all cases.
- COMM is largest in the out-of-core 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 out-of-core 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.

- Producer-driven in-core communication, even though it performs the best, does not provide significant performance improvement over the Demand-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.

- In both Demand and Producer-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.

- 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.

Figure 8 illustrates the performance for the same problem with the same level of scaling for the problem size and the number of processors. This example solves a problem of 8K×8K on 64 processors. Clearly, for both problem sizes, 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 in the next application, when the communication volume is large 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 $O(Nlog(N))$ 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 some (*,cyclic(b)) 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. Figure 2 presents an

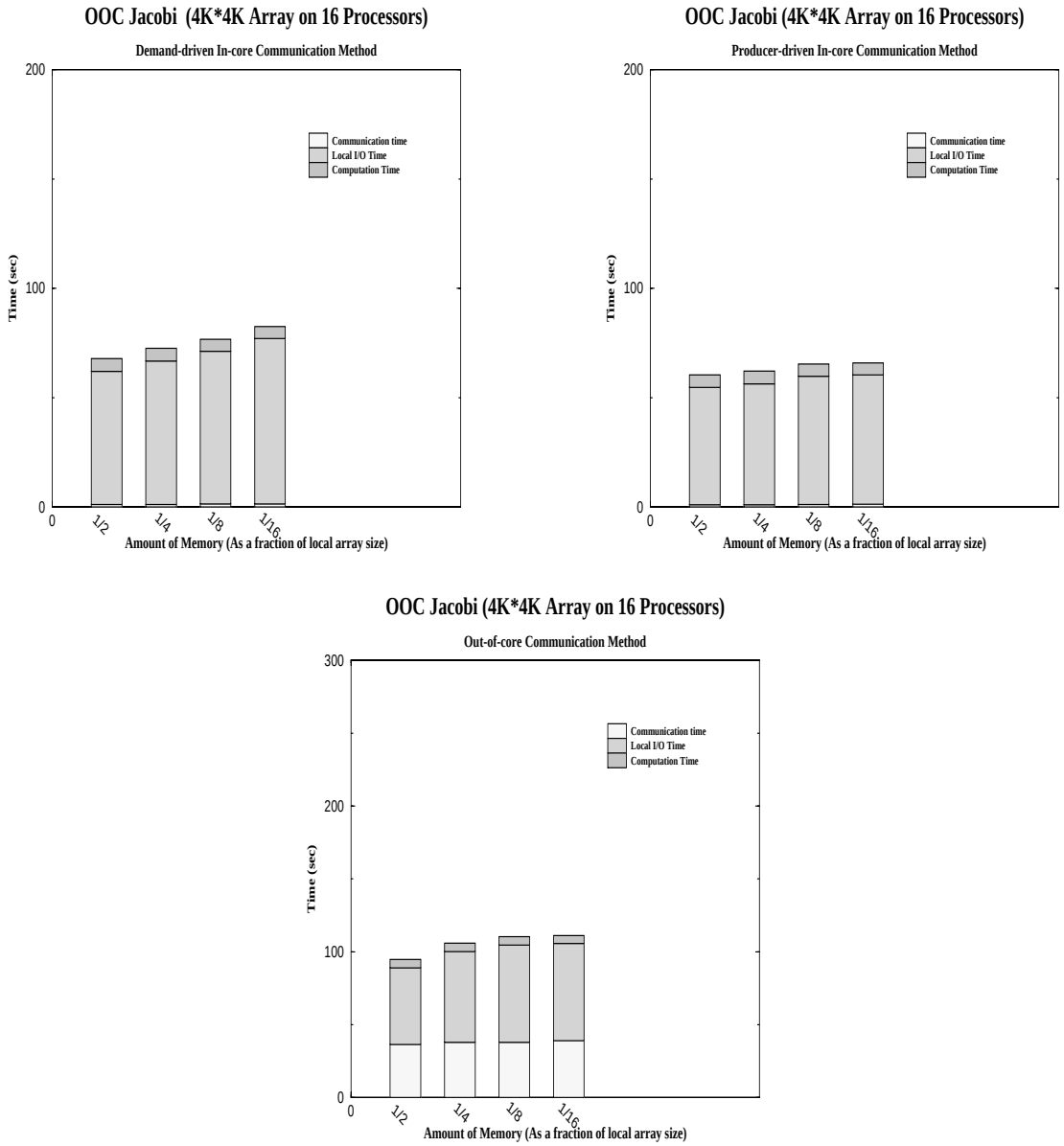

Figure 7: Out-Of-Core Jacobi Method, 4K*4K Array, 16 Processors. Communication time is greater for the out-of-core communication method. Both in-core communication methods require more local I/O than does the out-of-core communication method.

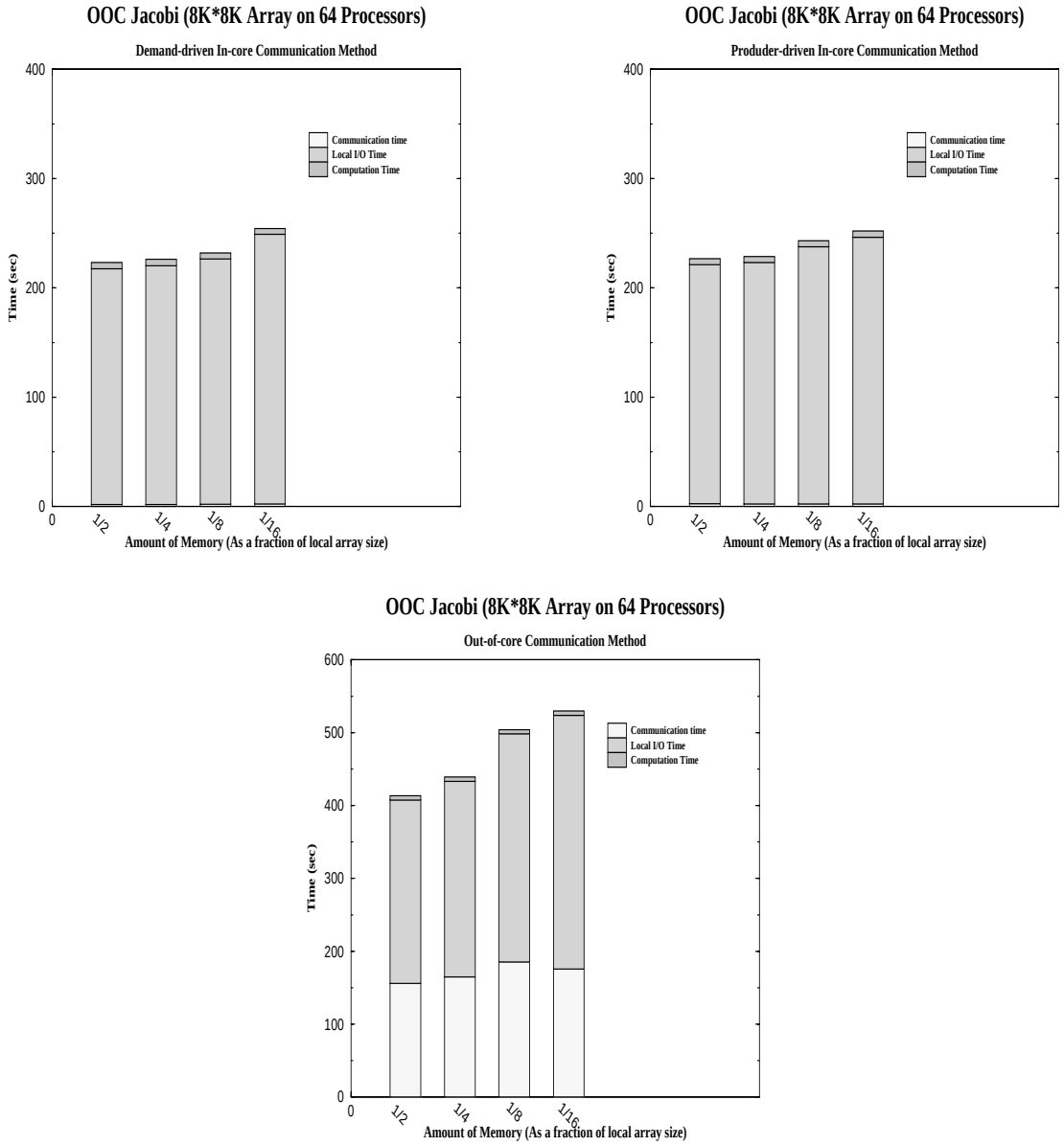

Figure 8: Out-Of-Core Jacobi Method, 8K*8K Array, 64 Processors. Both in-core communication methods give better performance than the out-of-core communication method.

HPF program to perform 2D FFT. The DO_1D_FFT routine performs 1-D FFT over the j^{th} column of the array.

```
          PROGRAM FFT
          REAL A(N,N)
!HPF$ TEMPLATE T(N,N)
!HPF$ DISTRIBUTE T(*,BLOCK)
!HPF$ ALIGN WITH T ::  A
          FORALL(J=1:N) &
          DO_1D_FFT(A(:,J) ! PERFORM 1-D FFT
          A=TRANSPOSE(A)
          FORALL(J=1:N) &
          DO_1D_FFT(A(:,J)) ! PERFORM 1-D FFT
          STOP
          END
```

Figure 9: An HPF Program for 2-D FFT. Sweep of the 1-D FFT in (X/Y) dimension is performed in parallel.

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 of the out-of-core transpose algorithm. The transpose can be performed using two ways, (1) Out-of-core Communication and (2) In-core Communication.

4.2.1 Out-of-core Communication

In the out-of-core communication method, the transposition is performed after the computation in the first phase as a collective operation.

Figure 10 (A) shows the communication pattern for the out-of-core transpose. Each processor fetches data blocks (ICLAs) consisting of several subcolumns from it's 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 1-D FFTs over the columns of the ICLA. The intermediate in-core data is then transposed. In the second phase, each processor fetches ICLAs from it's local array file and performs 1D FFTs over the columns in the ICLA.

Figure 10 (B) shows the in-core transpose operation. The figure assumes that the ICLA consists of one column. After the in-core transpose, the column is distributed across all the processors to obtain

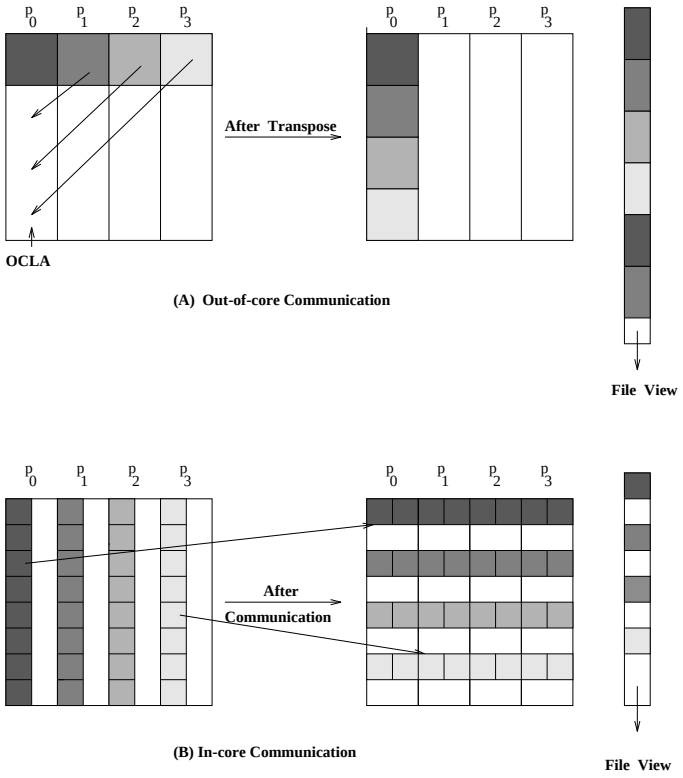

Figure 10: Out-of-core Transpose. The out-of-core communication method writes blocks of consecutive data using a small number of I/O accesses. In-core communication method requires a large number of I/O accesses. Each access writes a small block of consecutive data.

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.

Since in the transpose based FFT algorithm, the communication patterns for the demand-driven and producer-driver in-core communication are similar. We have implemented only the Producer-Driven communication.

4.2.3 Experimental Results

Figures 11 and 12 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 graph shows three components of the total execution time; namely, computation time (COMP), local I/O time (LIO) and communication time (COMM). From these results, we observe the following

- For the out-of-core FFT, COMM for the in-core communication method is larger than that for the out-of-core 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 out-of-core communication methods, COMM increases as the amount of available memory is decreased.
- The in-core communication method requires less LIO than the out-of-core 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.
- As the number of processors and grid size is increased, the out-of-core communication performs better but performance of the in-core communication method degrades.
- In both methods, the computation cost COMP remains the same.

5 Communication Strategies in Virtual Memory Environments

So far, we presented communication strategies for OOC computations, where data staging was done explicitly at the application level. This staging is performed by runtime routines (e.g. see [TBC+94b]). 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(s). For example, on 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, termed as "*local page faults*" and page faults caused by data required by remote processors due to

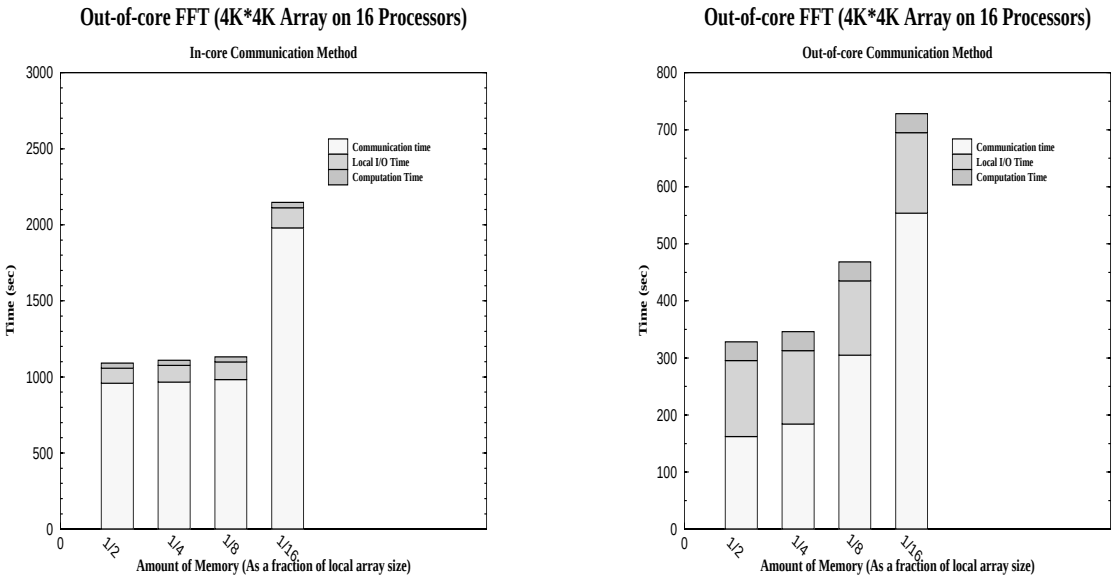

Figure 11: Out-of-core FFT of (4K*4K) array on 16 Processors. The overall execution time is dependent on the time required to do out-of-core communication. Communication time required for the out-of-core communication method is 1/4 of that required in the in-core communication method.

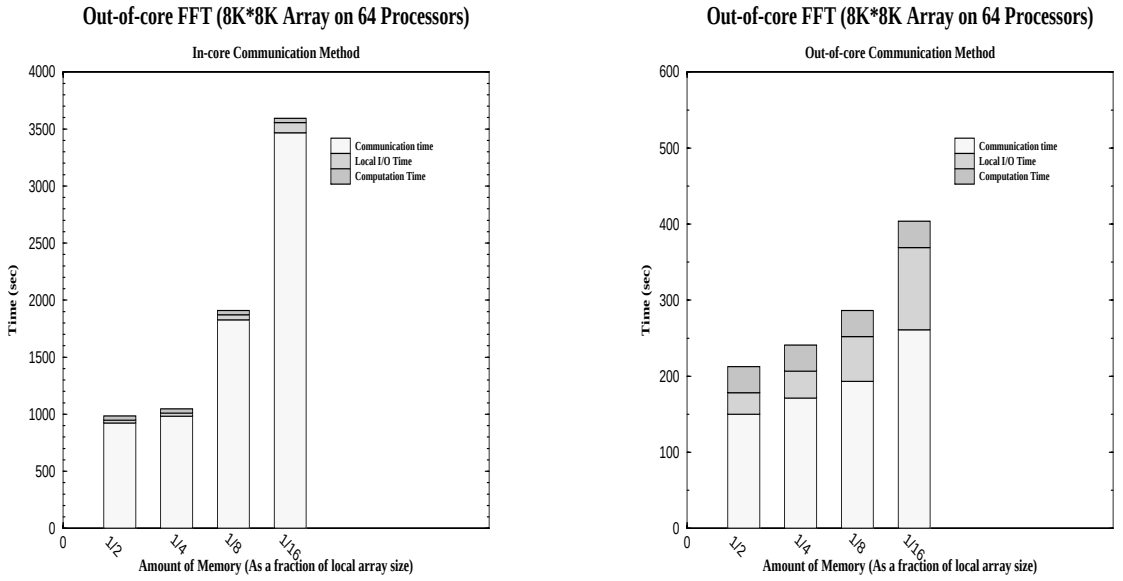

Figure 12: Out-of-core FFT of (8K*8K) array on 64 processors. The out-of-core communication method performs better than previous case. In-core communication methods perform worse than the previous case.

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 of the system level can be very expensive. 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. 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 (Demand-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 it's memory, a *local* page fault will occur else the owner processor can send the data (or page) without accessing it's own disk. This situation is very similar to the communication in the out-of-core scenario.

Since the Producer/Demand-driven communication strategies allow the nodes more control over how and when to communicate, these strategies are suitable for virtual memory environments. Consider the Producer-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.

6 Conclusions

We have shown that communication in the out-of-core problems requires both inter-processor communication and file I/O. Communication in the out-of-core problems can be performed at least in three different ways. The out-of-core communication method performs communication in a collective way while the in-core communication methods (Producer-driven/Consumer-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. Producer-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. We demonstrated, through experimental results, that different communication strategies

are suitable for different types of computations. We believe, these methods could be easily extended to support node virtual memories on distributed memory machines.

References

[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.

[BCT94] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, April 1994.

[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.

[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.

[CFPB93] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[DdR92] E. DeBenedictis and J. del Rosario. nCUBE parallel i/o software. In *Proceedings of* 11^{th} *International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[dRC94] J. del Rosario and A. Choudhary. High performance i/o for parallel computers: Problems and prospects. *IEEE Computer*, March 1994.

[For93a] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report CRPC-TR92225, Center for Research in Parallel Computing,Rice University, January 1993.

[For93b] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Version 1.0, May 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.

[Ini94] 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.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.

[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.

[SS93] S. Saini and H. Simon. Enhancing applications performance on intel paragon through dynamic memory allocation. In *Proceedings of the Scalable Parallel Libraries Conference, Mississippi State University*, October 1993.

[TBC94a] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the* 8^{th} *ACM International Conference on Supercomputing*, pages 382–391, July 1994.

[TBC+94b] 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.