Techniques and Optimizations for Developing Irregular Out-of-Core Applications on Distributed-Memory Systems

Peter Brezany[a] and Alok Choudhary[b]
[a]Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstr. 22, A-1092 Vienna
E-mail: brezany@par.univie.ac.at
[b]ECE Department, Northwestern University, Evanston, IL
E-mail: choudhar@ece.nwu.edu

Abstract

This paper presents techniques for implementing Out-Of-Core irregular problems. In particular we present a design for a runtime system to efficiently support out-of-core irregular problems. Furthermore, we describe the appropriate transformations required to reduce the I/O overheads for staging data as well as for communication. Several optimizations are described for each step of the parallelization process. The proposed techniques can be used by a compiler that compiles a global name space program (e.g., written in HPF and its extensions) or by users writing programs in $node + message\ passing\ style$. First we describe the runtime support and the transformation for a restricted version of the the problem in which it is assumed that only part of the data (large data structures) are out-of-core. Then we generalize these techniques in which all the major datasets of an application are out-of-core. The main objectives of the proposed techniques is to minimize I/O accesses in all the steps while maintaining load balance and minimal communication.

Contents

1 Introduction

1.1 Background

Input-Output (I/O) for parallel systems has drawn increasing attention in the last few years as it has become apparent that I/O performance much more than CPU or communication performance may be the limiting factor in future computing systems. Many software and hardware components play a role in I/O operations and any one of them can cause severe bottlenecks. Parallel computers combine a large number of increasingly more powerful processors with increasingly large amounts of memory. However, it is not clear how to scale I/O capabilities to keep pace with FLOPS; there are no scalable I/O systems that eliminate the I/O bottleneck for parallel computers. Disk Array (RAID) technology is one of the enabling hardware technologies for parallel I/O [11, 10, 30]. However, in most cases, RAID provides the traditional interface of a single logical disk where the parallelism is hidden at the lower (hardware level). Furthermore, disk arrays only address small-scale parallelism and do not directly address the important software issues including parallel file systems. Thus, large-scale parallelism in conjunction with supporting software must be used to overcome the I/O scalability problem, but doing so introduces new complexities. The use of hundreds or thousands of processors, each with a private memory, requires that data be partitioned into subsets and that these smaller pieces be transferred into or out of the local processor memories. Parallel file systems have been developed but their scalability has not been demonstrated for very large numbers of storage devices [5, 7, 17, 6, 4, 18, 20, 23]. Furthermore, with current languages and compilers, optimal use of scalable parallel file systems requires a substantial amount of tedious program restructuring.

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-1/2, nCUBE2/3, Meiko etc. provide some kind of hardware and software support for parallel I/O [14, 26, 21, 16, 31, 1]. Recently, a parallel file system has also been developed for network of workstations which incorporates the notion of transactions in I/O for reliability [25]. The file system software on these machines typically exploits parallelism by striping files across I/O nodes. In general, the interface presented to a user is very low level where the user is burdened with the task of manipulating a large number of file pointers and judiciously scheduling I/O accesses to obtain any reasonable performance. Recently, some file systems, e.g., VESTA [14], have begun to address the need for providing a higher-level interface.

From the system software perspective there are many I/O problems to be addressed in high-performance parallel and distributed computers. These include support for fast, parallel input-output of data from files, support for out-of-core algorithms, checkpointing and restart, prefetching, and other optimizations in parallel file systems. Support for runtime system and compiler support for performing I/O efficiently is important. In [18] an overview of some of these problems is presented.

1.2 Applications Perspective

Massively parallel computers, armed with mathematical models, are a key technological advancement which allow scientists to study, in intricate detail, complex physical, chemical and biological phenomena, as well as those of medicine and other sciences. They accelerate the investigative process way beyond the traditional trial and error approach. For instance, computational fluid dynamic simulations replace wind-tunnel testing and physical prototyping in the early stages of aircraft design; simulations of the growth of crystalline semiconductors are used by companies to determine how best to grow better and faster chips for future supercomputers. In addition, visualization techniques provide a more humanly palatable form of representation of the enormous amounts of experimental data that arise from simulation. However, along with more detailed or larger simulations comes an increase in the amount of data that must be dealt with, either as input to or as a product of the computation/simulation. This data must be collected, stored, viewed, or preprocessed.

Large-scale scientific computations as well as most information processing problems, in addition to requiring a great deal of computational power, also deal with large quantities of data. At present, a typical Grand Challenge application could require 1Gbyte to 4Tbytes of data per run [18]. Aside from being extremely complex and requiring significant amounts of processing time, these applications often deal with enormous quantities of data. Although the main memory regions of supercomputers are extremely large, applications that manipulate more data than can exist in memory, and therefore have very high I/O requirements, are appearing more frequently. For example, current archival sizes for a grand challenge group typically ranges from 500 MBytes to 500 GBytes of storage with a peak of 10 TBytes. It is anticipated that by the time Teraflops machines with their Terabytes of memory make their appearance, these I/O requirements will increase dramatically, in some cases over 100-fold (e.g., climate modeling) up to 10 PBytes per grand challenge group.

But space is not the only consideration. Supercomputers are commonly interfaced with a variety of peripheral devices - external disk storage systems, mass storage devices, visualization devices, video cameras, networks, other supercomputers, etc. - for pre or post-processing of data, or simply for additional working storage. In many cases, the speed with which data can be accessed by the supercomputer determines the rate at which it can complete the assigned job [1]. The need to access data via network connected remote devices introduces significant amounts of additional delay over access to the internal I/O subsystem. The rise in support for global computing paradigms amplifies the severity of this problem. Today, most high performance applications involve I/O rates of 1 MByte/s to 40 MBytes/s to secondary storage and 0.5 MBytes/s to 6 MBytes/s to archival

[1] Such jobs, for which I/O (not computation) is the bottleneck, are said to be "I/O bound".

Table 1: Examples illustrating current I/O requirements of some grand challenge applications (A: Archival Storage, T: Temporary Working Storage, S: Secondary Storage, B: I/O Bandwidth)

Application	I/O requirements
Environmental modeling	T: 10s of GB. S: 100s of MB - 1 GB per PE. A: Order of 1 TB.
Eulerian air-quality modeling	S: Current 1 GB/model, 100 GB/application; projected 1 TB/application. A: 10 TB at 100 model runs/application.
Earth system model	S: 108 MB/simulated day. 100 GB/decade-long simulation. B: 100 MB/s.
4-D data assimilation	S: 100 MB - 1 GB/run. A: 3 TB database. Expected to increase by orders of magnitude with the Earth Observing System (EOS) - 1 TB/day.
Ocean-Atmosphere Climate modeling	S: 100 GB/run (current). B: 100 MB/sec. A: 100s of TB.
Solar Activity and Heliospheric Dynamics	B: 200 MB/sec A: Up to 500 GB
Convective turbulence in Astrophysics	S: 5-10 GB/run B: 10-100 MB/s
Particle algorithms in Cosmology and Astrophysics	S: 1-10 GB/file; 10-100 files/run. B: 20-200 MB/s
Radio synthesis imaging	S: 1-10 GB A: 1 TB

store. Application developers indicate that in all probability about 1 GByte/s to secondary storage and 100 MBytes/s to archival store will be required in the near future [18].

Table 1 summarizes an estimate of the current requirements of a few large-scale computations [2]. These requirements are expected to increase by several orders of magnitude in most cases. Consider one such application, *Climate Prediction using General Circulation Model (GCM).* Current atmosphere/ocean models have the following requirements on an Intel Touchstone Delta [18]. For a 100-year atmosphere run with 300KM^2 resolution and 0.2 simulated years/machine hour, the simulation takes 3 weeks run time and generates 1144 GBytes of data at 38 MBytes per simulation minute. For a 1000-year coupled atmosphere-ocean run with a 150KM^2 resolution, the atmospheric simulation takes about 30 weeks, while ocean simulation takes 27 weeks; the process produces 40 MBytes of data per simulation minute, or a total of 20 TBytes of data for the entire simulation. Clearly, to enable application scientists to solve such large problems with reasonably scalable performance, system software support for I/O is crucial.

[2] For more details please refer to [18] and references provided there.

1.3 This Paper

In this paper, we present design of various steps, runtime system and compiler transformations to support irregular *out-of-core (OOC)* problems. The basic characteristics of these problems are that the problem size is too large to fit in the memory, and due to the irregular nature of the computations, the access and computation patterns, as well as data and computation partitioning can only be determined at runtime.

In particular we present a design for a runtime system to efficiently support OOC irregular problems. Furthermore, we describe the appropriate transformations required to reduce the I/O overheads for staging data. Several optimizations are presented for each step of the parallelization process. The proposed techniques can be used by a compiler that compiles a global name space program (e.g., written in HPF and its extensions) or by users writing programs in $node + message\ \ passingstyle$. First we describe the runtime support and the transformation for a restricted version of the the problem in which it is assumed that only part of the data (large data structures) are out-of-core. Then we generalize these techniques in which all the major datasets of an application are out-of-core. The main objectives of the proposed techniques is to minimize I/O accesses in all the steps while maintaining load balance and minimal communication.

The rest of the paper is organized as follows. In Section 2 we describe three models for OOC computations [2, 12, 33]. For the rest of the paper, we restrict ourselves to the Local Placement Model (described in Section 2). Section 3 describes the basic computational characteristics of irregular problems considered in this paper. Section 4 presents the runtime system as well as transformation required for a restricted version of the problem where it is assumed that enough memory is available so that the basic data elements (e.g., nodes of a mesh) are in-core, while the data structures describing the interactions among them are out-of-core. A rationale for this assumption is also presented. Section 5 removes the above restriction and presents the proposed runtime support and transformations for OOC irregular problems in which all major data structures reside in files. Finally, summary and conclusions are presented in Section 6.

2 Models for Out-Of-Core Computations

The models for OOC computations have been borrowed from Choudhary and his groups work on parallel I/O and specifically from the PASSION project [12, 13, 33, 3, 2]. The models provide and abstractions with which a runtime system and compiler can work so that they do not have to deal with machine specific features and low-level architectural organizations. Models help the user or compiler translate (maintain) locality between various working spaces in which an I/O intensive program operates (Bordawekar ([2]) considers Program Space, Processor Space, File Space, and Disk Space). Three execution models; namely, the Local Placement Model, the Global Placement Model and the Partitioned In-core Model [12] are normally used for designing runtime systems and compilation support.

2.1 Local Placement Model

The array stored in the file is termed as an *Out-of-Core Array* (OCA). The distribution concept of the High Performance Fortran is extended to OCAs. Each global OCA is associated with the corresponding set of *Out-of-Core Local Arrays* (OCLAs).

In the Local Placement Model, the out-of-core local array (OCLA) of each processor is stored in separate *logical* file called the Local Array File (LAF) of that processor. Each LAF can be stored separately as a different physical file or different LAFs can be appended to generate a single physical file. Every processor is said to *own* its LAF.

The portion of the array which is in main memory is called the In-Core Local Array (ICLA).

The node program explicitly reads from and writes to the LAF when required. A processor cannot explicitly access a file owned by some other processor. If a processor needs to access data from a file owned by a different processor, the required data first has to be read by the owner and then communicated to the requesting processor. Same principle applies to a write or store operation. Since each LAF contains the OCLA of the corresponding processor, the distributed (or user-specified) view of the OOC global array is preserved. In other words, locality (spatial and sequential) in processor space is translated into locality in the file space.

One way to view the Local Placement Model is to think of each processor as having another level of memory which is much slower than main memory. The extra memory is in form of a *virtual* disk which may consist of more than one physical disk. Each processor stores its LAF into its virtual disk. In other words, it is a straight-forward extension of the distributed memory model. Note that each processor may lack exclusive access to one or more physical disks that comprise its virtual disk. Disks may be shared by more than one processor. The mapping of physical to virtual disks is performed at runtime.

To store the data in the LAFs based on the distribution pattern specified in the program, redistribution of the data may be needed in the beginning when the data is staged. This is because the way data arrives (e.g., from archival storage or over the network) may not conform to the distribution specified in the program. Redistribution requires reading the data from the external storage, shuffling the data over the processors and writing the data to the local virtual disks. This increases the overall cost of data access. This cost can be amortized if an OOC array is used repeatedly.

Figure 1 presents an out-of-core array distributed in `BLOCK-BLOCK` fashion over 4 processors. Each processor stores its OCLA in its LAF (shown using different shades). Each local array file is stored into a virtual disk. During computation, each processor brings the current ICLA into its memory and operates on it and stores it back (if required by the computation).

2.2 Global Placement Model

In the Global Placement Model (GPM), an OOC global array is stored in a single file called the Global Array File (GAF). The global array is stored in the GAF and the file is stored in a single virtual disk that is shared by all the processors accessing this array. As in case of the Local Placement Model, the virtual disk consists of one or many physical disks and the mapping to physical to virtual disks is performed at runtime.

Each processor can read and write parts of its local array from the GAF. Since the local arrays of the processors are interleaved in the GAF, there is no concept of a local array file. A processor can *read* data owned by other processors, however, it can not *write* over the data owned by other processors. This is due to the fact that the underlying programming model is SPMD in which computations are performed in the local name space. This property of GPM eliminates any consistency problems arising from data sharing. Since any processor can read any other processor's data, the I/O can be performed in the global name space (using global coordinates). As the computation is still in the local name space, this model can be viewed as having shared memory on the files and distributed memory on the processors.

In the GPM, the global array view is preserved from a global name space HPF program. Consequently, localities from program space (temporal and spatial) are translated into localities in file space.

Figure 2 shows columns of local array of a processor interleaved with others in the global file (shown with different shades). The global file is distributed over four disks which are shared by all processors. Thus, parts of the global array, based on the array distribution, need to be brought into the memory for processing. For example, Figure 2 shows a sweep of the global array by columns (numbered 1 through 3), assuming that only one column per processor fits into memory. The corresponding file access pattern is also shown. In general, the accesses may be viewed as loosely synchronous. That is, the I/O can be performed in a collective manner (similar to collective communication) using a global index space. Subsequently, the data may need to be re-shuffled in memory by mapping the global indices (of current data) into local index space. Since there are no local files in the GPM, data redistribution into files is not required.

2.3 Partitioned In-core Model

Partitioned In-core Model (PIM) is a combination of the Global and Local Placement models. The array is stored in a single global file as in the Global Placement Model but there is a difference in the way data is accessed. In the PIM, the global array is logically divided into a number of *partitions*, each of which can fit in the combined main memory of the processors. Thus computation on each partition is primarily an in-core problem and no I/O is required during the computation of the partition.

Like LPM, PIM is also based on the data distribution concept proposed in HPF. The Local

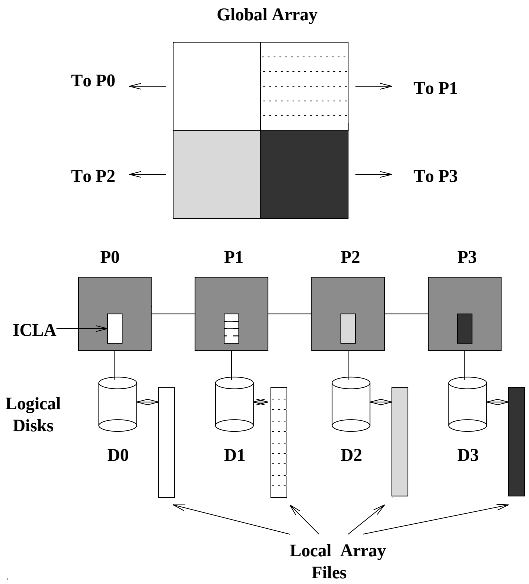


Figure 1: Local Placement Model.

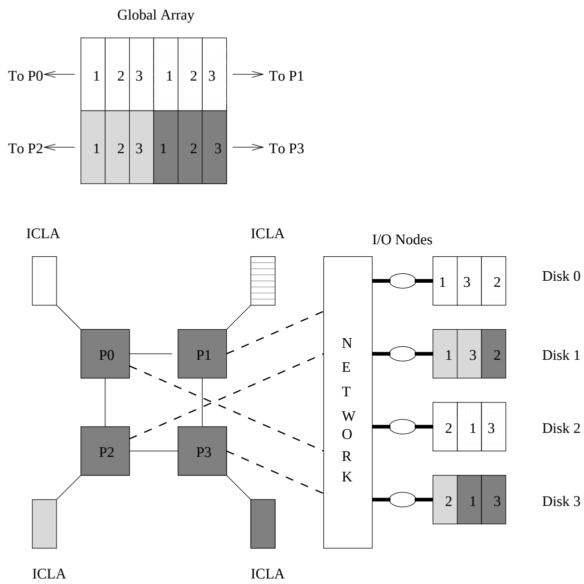


Figure 2: Global Placement Model.

Placement Model extends the distribution concept to the out-of-core data while the Partitioned In-core Model uses the in-core distribution pattern for assigning out-of-core data to processors. In case of an in-core program, the array to be distributed is assumed to be in-core (i.e. it can fit in combined memory of all processors). Similarly, the Partitioned In-core Model uses the combined memory to find an in-core data partition and then distributes the partition according to the user-provided distribution pattern.

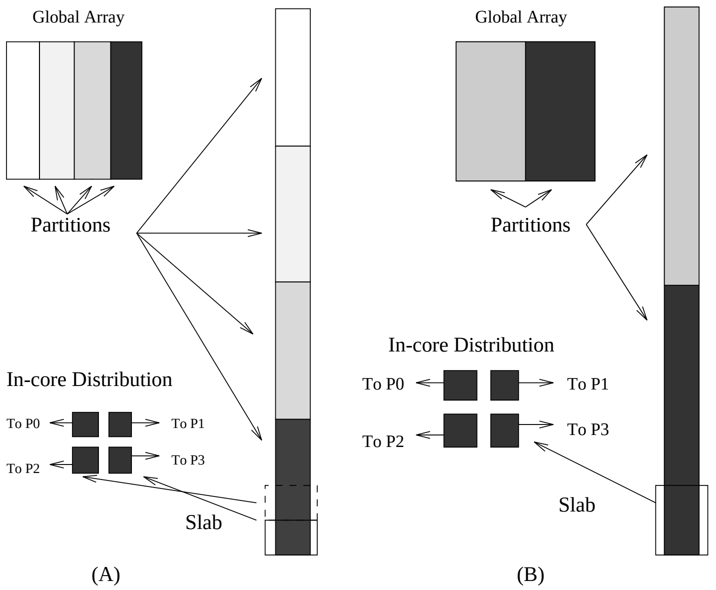


Figure 3: Partitioned In-core Model.

PIM creates some extra complications in data ownership computations. In LPM, the out-of-core global array is distributed as if it were an in-core array. Therefore, the data ownership of the out-of-core data can be easily computed using its distribution. In case of PIM, the data ownership depends on the cumulative available memory as well as distribution pattern. Figure 3 presents an out-of-core array to be distributed in a `BLOCK-BLOCK` fashion over 4 processors. Figures 3 A and B illustrate in-core data distribution for two different partition sizes. Note that data belonging to processor 2 in partition (A) may lie in processor 3 in partition (B). But in both cases, the in-core array (i.e., partition) is distributed in the `BLOCK-BLOCK` fashion.

Like GPM, PIM also performs the I/O in global name space and the computation in local name space. In both models, data read and write can be easily performed using the collective methods such as the two-phase access method [6, 17].

In this paper we only consider the LPM. A qualitative comparison of these models can be found in [2].

3 Irregular Problems

Parallelizing irregular problems is a challenging problem and is of growing importance. In such problems, access patterns to major data arrays are only known at runtime. Furthermore, major data structures are accessed through one or more levels of indirection, which requires preprocessing in order to determine the data access patterns. On the other hand, in regular problems, data distributions and access patterns may be described using an expression easily recognizable by a compiler (e.g., stencils). Application areas in which irregular codes are found include unstructured multigrid computation fluid dynamic solvers, molecular dynamics codes and diagonal or polynomial preconditioned iterative linear solvers etc.

In irregular codes, the array accesses cannot be analyzed at compile time to determine either independence of these or to find what data must be pre-fetched and where it is located. Therefore, the appropriate language support is needed, as well as compile time techniques relying on runtime mechanisms. High-Performance Fortran, HPF-1, provides several constructs to express data parallelism in irregular codes explicitly. These include the FORALL statement and construct, the DO loop prefixed by the INDEPENDENT directive, and the array statements. Further, extensions for *computation control* provided recently by HPF-2 allow the explicit *work distribution* specification in independent loops via an *ON clause*, and the control of reductions in such a loop.

When processing these constructs, the task of the compiler is to match the parallelism expressed by the construct to that of the target parallel system. One strategy used generates three code phases for each loop specifying irregular computations, called the *work distributor*, the *inspector*, and the *executor* [22, 32, 29, 28]. The work distributor determines how to spread the work (iterations) among the available processors; on each processor, it computes the *execution set*, i.e. the set iterations to be executed on this processor. The inspector analyzes the communication patterns of the loop, computes the description of the communication, and derives translation functions between global and local accesses, while the executor performs the actual communication and executes the loop iterations. All the phases are supported by an appropriate runtime library. For example, the PARTI library [15] is one system constructed to support the handling of irregular computations on massively parallel systems. During the last few years, the basic compilation techniques [8, 34, 19] relying on such a library have been established.

The language features mentioned above enable the user to assume full control of data and work distribution. However, the aim is also to provide a higher level language and interactive interface with which the user can direct the programming environment to derive data and work distributions automatically according to a selected strategy. Based on the specification provided by the user, the compiler generates a *partitioning* construct [9, 27, 28] before transforming the loop into the *inspector-executor*. This construct redistributes the data arrays and iterations in an irregular way to minimize communication and balance load. The essence of the approach is contained in CHAOS

[19] routines[3] which are called from the compiler embedded code.

Large scale irregular applications[4] involve large arrays and other data structures. Runtime preprocessing and automatic partitioning provided for these applications results in construction of large data structures which increase the memory usage of the program substantially. Consequently, a parallel program may quickly run out of memory. Therefore, the development of appropriate parallelization methods for out-of-core irregular programs is relevant and important research issue. Even if, the performance is somewhat worse than the case when the same size problem could fit in the memory, the capability to solve larger problems provides the appropriate flexibility to the users. Furthermore, not all users have access to large-scale systems with huge amounts of memory. Providing support that facilitates out-of-core solutions will enable such users to solve relatively larger size problems[5].

In the following sections, we describe compilation and runtime techniques to support out-of-core irregular problems.

4 Parallelization Methods and Runtime Support for OOC Irregular Problems

As described in the previous section, irregular problems are characterized by indirect data accesses where access patterns are only known at runtime. The input data (e.g., a mesh representing the discretization of an air-foil) represents a description of the problem domain which is irregular.

Figure 4 illustrates a typical irregular loop. The loop L2 in this code represents a sweep over the edges of an unstructured mesh of size: NNODE, NEDGE, where NNODE is the number of data elements and NEDGE represents the number of edges describing interdependencies among the nodes. Figure 5 depicts a structure of a simple unstructured mesh that will be used for illustration of our examples. Since the mesh is unstructured, indirection arrays have to be used to access the vertices during a loop sweep over the edges. The reference pattern is specified by the integer arrays *edge1* and *edge2*, where edge1(i) and edge2(i) are the node numbers at the two ends of the ith edge. The calculation on each node of the mesh requires data from its neighboring nodes. Arrays edge1 and edge2 are called *indirection arrays*. The arrays x and y represent the values at each of the NNODE nodes; these arrays are called *data arrays*. Such a computation forms the core of many applications in fluid dynamics, molecular dynamics etc.

A typical parallelization based on the inspector-executor model, assuming that all the data structures fit into the memory (i.e., an in-core parallelization) process involves several steps; namely, 1) Data Distribution, 2) Workload Distribution, 3) Inspector, and 4) Executor [29]. The primary

[3] CHAOS is a runtime library that subsumes PARTI.

[4] Most real world applications are large-scale, but their sizes have been limited by the memory of the available systems.

[5] This is also known as the scale-down problem.

goal of these steps and techniques is to minimize communication while obtaining load balance so that the a good parallelization is obtained. Even when the data is out-of-core, primary goals remain the same, that of minimizing various overheads. However, since I/O is very expensive, one of the most important (and probably more important than the others) goals is to minimize I/O accesses. Therefore, in order to extend the inspector-executor model, I/O access costs must be accounted for and incorporated into various steps.

As the models described in the previous sections indicate, a computation step in any OOC computation will require reading data from disks in slabs (or tiles), processing the data, and writing it back (if necessary). Therefore, it is important to minimize such steps. In other words, the steps described above must be modified so that the data is reorganized on disks at runtime so as to minimize I/O accesses. Furthermore, scheduling of slabs in the memory must be performed in such a way that not only the data present in the local memory is maximally used before being discarded (or written back onto disks), but the data, if required by other processors, should also be used during this phase.

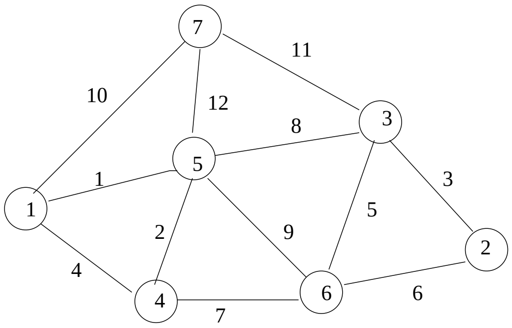
In the following, we propose several techniques which address these problems.

```
       integer, parameter :: NNODE = ..., NEDGE = ..., NSTEP = ...
       real x(NNODE), y(NNODE)                      ! data arrays
       integer edge1(NEDGE), edge2(NEDGE)           ! indirection arrays
L1:    do j = 1, NSTEP                              ! outer loop
L2:        do i = 1, NEDGE                          ! inner loop
               x(edge1(i)) = x(edge1(i)) + y(edge2(i))
               x(edge2(i)) = x(edge2(i)) + y(edge1(i))
           end do
       end do
```

Figure 4: An Example with an Irregular Loop.

4.1 Problem Description and Assumptions

We assume that the size of the data, data structures describing interactions among data and those describing computation patterns (e.g., indirection arrays, neighbor lists etc.) are very large. However, we assume that the data can fit in the system's memory, while the data structures describing interactions etc. are out-of-core. This assumption is realistic in many applications where the data size is N and there are few arrays storing the data, but there are a large number of structures describing the interactions with sizes of $K * N$ (e.g., CFD) or $O(N^2)$ (e.g., molecular dynamics). For example, if the value of K is 10 and there are 5 such data structures, then the



i	edge1(i)	edge2(i)
1	1	5
2	5	4
3	2	3
4	1	4
5	5	2
6	2	6
7	6	4
8	5	3
9	6	5
10	1	7
11	7	3
12	5	7

Figure 5: Example Mesh with Seven Nodes and Twelve Edges.

amount of memory required to run an application with data size of N will be at least $50N$ without considering temporary storage requirements. Therefore, even if the data structures containing the primary data of a problem of size N could fit into memory, data structures describing interactions are still out-of-core.

In this subsection, we consider the parallelization strategy for this restricted problem.

4.2 Data Arrays are In-Core and Indirection Interaction Arrays are Out-of-Core

This class of problems will be referred to as the *DAI/IAO* (*D*ata *A*rrays are *I*n-core / *I*ndirection *A*rrays are *O*ut-of-core) problems.

We describe our parallelization scheme using loop L2 shown in Figure 4 and the unstructured mesh depicted in Figure 5. We only consider the Local Placement Model.

The main goal of the proposed technique is to minimize I/O costs during the execution of an applications by data reorganization on disks and by using efficient schedules to stage data into memory. Specifically, we want to minimize I/O for both reading/writing slabs as well as for communication[6]. Therefore, our initial goal is to introduce additional steps as well as modify existing steps in the inspector-executor model to satisfy the goals of minimizing I/O costs. In the following, we describe the proposed steps using the kernel illustrated in Figure 4.

[6] Note that since data sets reside on disks, even the communication steps potentially require I/O if the data to be communicated is not present in the sender's memory at the time when communication takes place.

4.3 Parallelization Steps and Runtime Support

This subsection presents an overview of the principles and the functionality of the extended CHAOS runtime library, called *CHAOS/E*, that our parallelization scheme is based on. CHAOS/E can be used either by the application programmers to parallelize the *DAI/IAO* applications, or can be used by parallelizing compilers to automatically generate parallel OOC code for the *DAI/IAO* applications.

The *DAI/IAO* computation is performed in 7 steps that are outlined below.

A. Default Initial Data and Work Distribution

B. Partition Data Arrays

C. Redistribute Data Arrays

D. Compute new iteration tiles

E. Redistribute local files with indirection arrays

F. Out-of-Core Inspector

G. Out-of-Core Executor

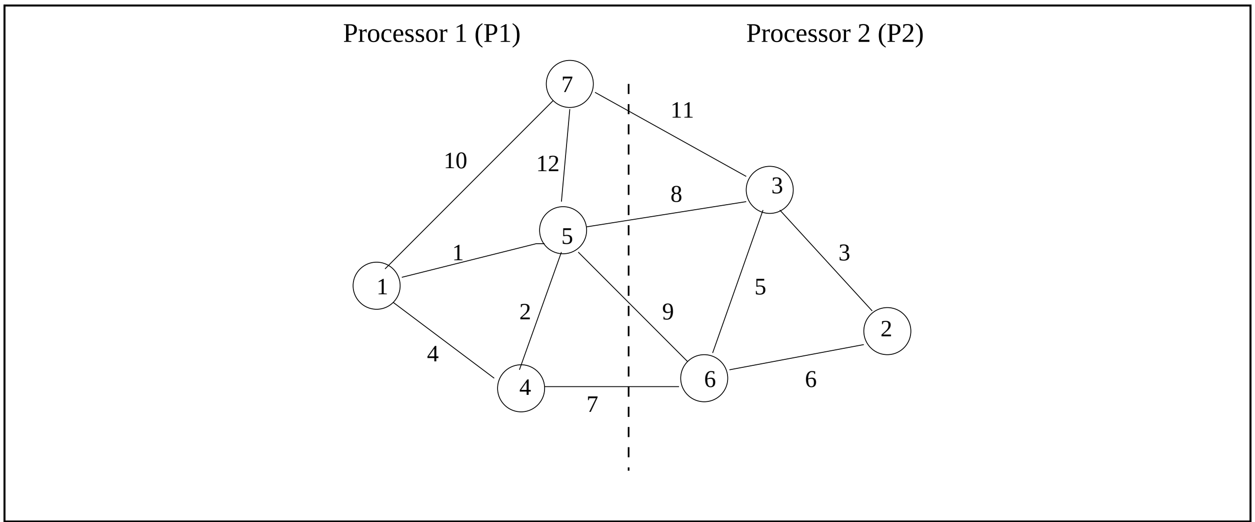


Figure 6: Example Mesh with Seven Nodes and Twelve Edges Distributed on 2 Processors.

Each of these steps is described in more detail below.

A. Clearly, in order to fully exploit parallelism, the step of data partitioning must also be done in parallel. Initially, it is assumed that the iterations and data arrays are distributed among processors in a known uniform manner. Thus, the global iteration set is partitioned blockwise among the processors and the iteration set assigned to a processor is further blockwise split into a number of tiles. Section of an indirect array associated with each tile can fit in the main memory of any one of processors.

B. This step essentially involves the use of a partitioner which uses out-of-core data to determine a new data distribution to minimize interprocessor communication and enhance locality and load balance. Note that any partitioner, traditionally used for in-core data, can be extended to be used in this situation. Partitioners can be available as library functions invoked by a user/compiler. A partitioner returns an irregular assignment of array elements to processors, which are stored as another data structure called *translation table*. A translation table lists the owner processor and the logical offset address of each data array element.

   There are many partitioning heuristics methods available based on physical phenomena and proximity. Partitioners (such as available in CHAOS or their appropriate extensions) are linked to programs by using data structures that store information on which data partitioning is to be based. They can utilize spatial, load, or connectivity information or combination of them. For example, a code segment can be inserted into the SPMD program to construct a graph representing the access patterns of the distributed arrays in the loops. The graph is constructed by executing a modified version of the loop which forms a list of edges. Note that the operations in the loop are not performed. This list is stored on disks. The generated graph is passed to a graph partitioner. The graph partitioner returns the new array distributions.

C. In step C, the data arrays that were initially distributed in block fashion are remapped to obtain the new distribution determined by the partitioner. For example, Figure 6 shows a possible data distribution of the nodes of our example mesh across 2 processors. This would be the the result of redistribution from an initial distribution in which processor one had nodes 1-4 and processor 2 had nodes 5-7.

D. Once the new array distribution is obtained, loops iterations that access the distributed arrays must also be partitioned among processors to balance computational load and maintain locality (to reduce communication costs). Furthermore, the indirection arrays must be reorganized (within each processor), to minimize I/O costs. To partition the loop iterations, for locality and load balance, the primitives used here generate a processor assignment graph for each iteration [29]. A loop iteration is assigned to a processor that owns the maximum number of distributed array elements referenced in that iteration.

   In the second part of this step, the data is reorganized within each local file to minimize I/O. The iteration set assigned to a processor is further split into a number of tiles/slabs, each large enough to enable all the relevant data to fit in the memory. The tiles are reorganized and are classified into two groups,; namely, *local tiles* and *non-local tiles*. The local tiles include only those iterations for which the corresponding data is available locally (i.e., on local files). For example, iterations 1, 2, 4, 10, and 12 in Figure 6 will be part of local tiles. This reorganization and classification enables creating large tiles because no buffers are required for communication thereby reducing the number of I/O operations. Secondly, this grouping allows prefetching and pipelining of local tiles because the access is independent of other processors since there is no communication. Therefore, computations can be overlapped with I/O. That is, not only the number of I/O operations may be reduced, but I/Os can be overlapped with computation. The non-local tiles include iterations that require off-processor accesses. For example, iterations 7, 8, 9, and 11 in Figure 6 require off-processor accesses. This grouping is also advantageous because communication may be coalesced for each tile, and I/O required for communication may be reduced by coalescing them together. The primitive iter_partitioner performs the iteration partitioning and returns a descriptor that specifies the tiles to be executed on each processor.

E. In this step, the local files containing indirection arrays are remapped to conform with the loop iteration partitioning in the previous step.

F. Step F represent the inspector. However, since the inspector requires building communication schedules for only non-local tiles, it does not require a pass through the entire local data set. Two options for building communication schedules are available. In the first option, a communication schedule is generated for each non-local tile, whereas in the second option, communication schedules for all non-local tiles may be combined. The former allows overlapping of computation with communication to remote processors, while the latter option reduces the number of communication steps.

G. Phase G represents the executor. In this phase, for each time step, the following steps are performed. 1) Fetch local tiles and update them, 2) Gather non-local data, 3) Update non-local tiles, and 4) Scatter data to update non-local tiles on remote processors. Note that steps 1 and 2 may be overlapped. Similarly, steps 3 and 4 may be overlapped if the second communication option in step F is chosen.

4.4 Implementation Design

All the processing phases discussed above are illustrated by means of the running example of Figure 4, which is shown in Figure 7. Code implementing the phases is shown in Figures 9, 10, 11, 12, and 13 and discussed in the following paragraphs.

4.4.1 Data Partitioning

We discuss the application of two partitioners

```
      integer, parameter :: NNODE = 8, NEDGE = 12, NSTEP = ...
      real x(NNODE), y(NNODE)                          ! data arrays
      integer edge1(NEDGE), edge2(NEDGE)               ! indirection arrays
L1:   do j = 1, NSTEP                                  ! outer loop
L2:       do i = 1, NEDGE                              ! inner loop
S1            x(edge1(i)) = x(edge1(i)) + y(edge2(i))
S2            x(edge2(i)) = x(edge2(i)) + y(edge1(i))
          end do
      end do
```

Figure 7: An Example with an Irregular Loop.

- *recursive spectral bisection partitioner* which is a graph partitioner that utilizes connectivity information
- *recursive coordinate partitioner* whose partitioning algorithm is based on spatial coordinate information.

Coupling a Graph Partitioner. Some data partitioners determine data distribution by breaking up a graph, where the nodes of the graph represent the data array elements and the edges of the graph depict the dependency pattern between the nodes. The graph partitioners divide the graph into equal subgraphs with as few edges as possible between them.

To interface with graph partitioners, a distributed data structure called Runtime Data Graph (RDG) must be constructed. Input data for the construction of an RDG are computed by the parallel execution of a modified version of the DO loop. Therefore, loop iterations of the loop are initially distributed among processors in some uniform manner. We use the block distribution in our implementation. Processing steps needed to derive the distribution of arrays are depicted in Figure 9.

RDG is generated in 4 steps:

1. *Distribute data and loop iterations to processors in uniform blocks (in-core processing).* First, the iterations are distributed blockwise and then each iteration block assigned to a processor is split into tiles. This functionality is provided by the primitive build_block_tiles. It accepts the lower and upper bounds of the loop iteration space and number of tiles the execution set of each processor owns as input parameters. It returns the description of tiles to be executed on the calling processor. An example of a possible initial loop and data partitioning for the loop from Figure 7 is shown in Figure 8. We assume two processors and

```
Processor 1                              Processor 2
lwbs = {1, 7}                            lwbs = {4, 10}
upbs = {3, 9}                            upbs = {6, 12}
This implicitly assumes:                 This implicitly assumes:
execution set = {1,2,3,7,8,9}            execution set = {4,5,6,10,11,12}
tile1 = {1,2,3}, tile2 = {7,8,9}         tile1 = {4,5,6}, tile2 = {10,11,12}
local_ind_list = {1,2,3,4}               local_ind_list = {5,6,7}
n_local = 4                              n_local = 3
```

Figure 8: An Example of Initial Iteration Partitioning.

two iteration tiles are generated for each processor. The values of *lwbs* and *upbs* point to the sections of arrays *edge1* and *edge2* that are stored in file *'mesh.dat'*.

2. *Generate local RDG on each processor (out-of-core processing).*
   The RDG is generated based on the access patterns of the arrays x and y in the DO loop. A slice of the original DO loop is executed in parallel to form out-of-core local lists of array access patterns for each processor according to the work distribution. The lists are incrementally constructed executing a sequence of tiles (two tile per processor in our example). Slabs of indirection arrays are read from the global file *'mesh.dat'* and slabs of computed lists are written to the appropriate local files. The lists *lhs_ind* and *rhs_ind* record the left hand side and right hand side array access patterns in statement S1. The RDG obtained using *S1* does not get altered by the statement *S2* as there are no new pairs of indices otherwise an additional pair of lists should be constructed for *S2*. All processors then call the primitive build_local_RDG with the files storing the left hand side and right hand side array indices as input arguments. The RDG is stored in the hash table which is identified by an integer (*hash_table*).

3. *Merge local graphs to produce a distributed graph (in-core processing).*
   In the next step, the local graphs are merged to form a distributed graph. Procedure generate_RDG is called on each processor to combine the local adjacency lists in the hash table into a complete graph (closely related to compressed sparse row format). The RDG is distributed so that each processor stores the adjacency lists for a subset of the array elements. Edges are grouped for each array element together and then stored in the list represented by the integer array csrc. A list of pointers, represented by the integer array csrp, identifies the beginning of the edge list for each array element in csrc.

4. *Data Partitioning.*
   The RDG representation of the distributed array access patterns is passed to the procedure

```
    open (mesh_file, file ='/pfs/mesh.dat')
    ... initialization of x and y ...
C-Constructing iteration tiles (block partitioning)
    call build_block_tiles(1, nedge, number_of_tiles, lwbs, upbs)
C-DATA PARTITIONING using a Graph Partitioner
C-Constructing local RDG
lhs_pntr = 1; rhs_pntr = 1
do tile = 1, number_of_tiles
    call read_slab (mesh_file, lwbs(tile),upbs(tile), edge1)
    call read_slab (mesh_file, NEDGE+lwbs(tile),NEDGE+upbs(tile), edge2)
    lcount = 1; rcount = 1
    do i=1, upbs(tile) - lwbs(tile) + 1
        lhs_ind(lcount) = edge1(i); rhs_ind(rcount) = edge2(i)
        lcount = lcount + 1; rcount = rcount + 1
    end do
    call write_slab(lhs_ind_file, lhs_ind, lhs_pntr, lcount-1)
    call write_slab(rhs_ind_file, rhs_ind, rhs_pntr, rcount-1)
    lhs_pntr = lhs_pntr + lcount-1; rhs_pntr = rhs_pntr + rcount-1
end do
C-Initialize hash table to store RDG
    hash_table = init_ht(init_argument)
    call build_local_RDG(hash_table, lhs_ind_file, rhs_ind_file)
C-Generating distributed RDG: stored in csrp and csrc
    call generate_RDG(hash_table, local_ind_list, n_local, csrp, csrc, ncols)
C-Parallel RSB partitioner: returns translation table dtrat
    call partitioner_RSB(local_ind_list, n_local, csrp, csrc, dtrat)
```

Figure 9: Data Partitioning

partitioner_RSB. This procedure uses a data mapping heuristic. Any graph partitioner can be used here as long as the partitioner recognizes the input data structure. The parallel partitioner returns a pointer to a translation table (*dtrat*) which describes the new array distribution. A translation table describes the distributed array layout amog processors.

Coupling a Coordinate Partitioner. Figure 10 illustrates a call to the recursive coordinate partitioner. Here, the use of geometric information is shown. Arrays coord1 a coord2 carry the spatial coordinates for elements in x and y.

```
C–DATA PARTITIONING USING RCB
C–Parallel RCB partitioner: it returns translation table dtrat
      coord1 a coord2 are coordinate arrays
      call partitioner_RCB(2, coord1, coord2, ndata, dtrat)
```

Figure 10: Partitioning that uses Two-Dimensional Geometry Information

```
C–REDISTRIBUTING DATA
C–Constructing schedules for redistribution
      call remap(dtrat, local_ind_list, dsched, newlindlist, newlindlist_size)
C–Redistributing data arrays
      call gather(x, x, dsched)
      call gather(y, y, dsched)
```

Figure 11: Redistribution of Data

4.4.2 Data Remapping

Once the new array distribution has been specified data arrays must be remapped and redistributed based on it. This is done in two steps which are shown in Figure 11.

1. Procedure **remap** is called which inputs the translation table (*dtrat*) describing the new array distribution and a list of initial array indices (*local_ind_list*). It returns a schedule pointer (*dsched*) and a new local index list (*newlindlist*),

2. the schedule pointer is passed to the routine **gather** to move data arrays x and y between the source and target distributions. The **gather** routine accepts the following arguments: addresses of the local data segment and communication buffer (they have to equal if arrays are to be remapped), and a pointer to the scheduler.

4.4.3 Partitioning and Redistribution of Loop Iterations

The newly specified data array distribution is used to decide how loop iterations are to be distributed among processors. This calculation takes into account the processor assignment of the distributed array elements accessed in each iteration. A loop iteration is assigned to the processor that owns the maximum number of distributed array elements referenced in that iteration.

Distributing loop iterations is done via procedure **iter_partitioner** which includes the generating the runtime data structure, called *Runtime Iteration Processor Assignment graph* (RIPA) and call to the iteration distribution procedure. The RIPA lists for each iteration, the number of distinct data references associated with each processor. It is constructed from the list of distinct distributed

```
      files(1) = lhs_ind_file; files(2) = rhs_ind_file; number_of_files = 2
C–ITERATION PARTITIONING
C–Global references for each local iteration are stored in the files. The pointers to these files are stored
C–   in the array files
C–Iterations assigned to a processor are partitioned into local and non-local tiles
C– Local files storing indirection arrays are generated; the ordering of array elements is aligned with tiles
C–Iteration partitioner returns the tile description and local files storing indirection arrays
C–The pointers to the local files are returned in the array files
      call iter_partitioner(dtrat, files, number_of_files, number_of_all_tiles, &
           number_of_local_tiles, lwbs, upbs)
```

Figure 12: Iteration Partitioning

array access patterns encountered in each loop iteration, which have already been recorded in files *lhs_ind_file* and *rhs_ind_file* and passed as arguments to the iter_partitioner procedure as can be seen in (Figure 12); the pointers to these files are stored in the array *files*. Using the RIPA the iteration distribution subroutine assigns iterations to processors.

In the next step the set of iterations assigned to a processor is split into an ordered set of of tiles; a number (*number_of_local_tiles*) of local tiles is followed by (*number_of_all_tiles* - *number_of_local_tiles* + 1) non-local tiles. The tile description is returned in the arguments *lwbs* and *upbs* in the same way like it was provided by the primitive build_block_tiles.

Finally, two local files storing indirection arrays are created. Ordering the array elements in the files establishes alignment with the iterations in the tiles. Pointers to these local files are stored in the array *files*.

4.4.4 Inspector and Executor

On each processor, the inspector and executor are executed for local tiles first, and then for non-local tiles, as shown in Figure 13. For each tile, contiguous blocks of values *edge1* and *edge2* can be read from the appropriate local files. The basic functionalities of the inspector and executor are described earlier.

In the phase that processes local tiles, the procedure ind_conv performs the global to local index conversion for array references that don't refer non-local elements. The local indices are used in the **Executor 1** phase.

In the phase that processes non-local tiles, the list *globref*, its size and the distribution descriptor of the referenced array are used by the CHAOS procedure localize to determine the appropriate *schedule*, the size of the communication buffers to be allocated, and the local reference list *locref* which contains results of the global to local index conversion. The declaration of the arrays x and y in the message passing code allocate memory for the local segments holding the local data and

```
C-Processing local tiles
      do tiles = 1, number_of_local_tiles
            tile_size = upbs(tile) - lwbs(1) + 1
            call read_slab(files(1), edge1, lwbs(tile), tile_size)
            call read_slab(files(2), edge2, lwbs(tile), tile_size)
C-Constructing the global reference list
            k = 1
            do i = 1, tile_size
                  globref(k) = edge1(i); globref(k+1) = edge2(i); k = k + 2
            end do
C-Index conversion
            call ind_conv(dtrat, globref, locref, tile_size*2)
C-Executor 1
            k = 1
            do i = 1, tile_size
                  x(locref(k)) = x(locref(k)) + y(locref(k+1))
                  x(locref(k+1)) = x(locref(k+1)) + y(locref(k)); k = k + 2
            end do
      end do
C-Processing non-local tiles
      do tiles = number_of_local_tiles + 1 , number_of_tiles
            call read_slab(files(1), edge1, lwbs(tile), tile_size)
            call read_slab(files(2), edge2, lwbs(tile), tile_size)
C-Constructing the global reference list
            k = 1
            do i = 1, tile_size
                  globref(k) = edge1(i); globref(k+1) = edge2(i); k = k + 2
            end do
            call localize(dtrat, sched, globref, locref, 2*tile_size, nnloc, local_size)
            call zero_out_buffer(y, y(locseg_y_size+1), sched)
            call gather(y, y(locseg_y_size+1), sched)
C-Executor 2
            k = 1
            do i = 1, tile_size
                  x(locref(k)) = x(locref(k)) + y(locref(k+1))
                  x(locref(k+1)) = x(locref(k+1)) + y(locref(k)); k = k + 2
            end do
            call scatter_add(x, x(locseg_x_size+1), sched)
      end do
```

Figure 13: The Inspector and Executor

the communication buffer storing copies of non-local data. The buffer is appended to the local segment. An element from *locref* refers either to the local segment of the array or to the buffer.

The **Executor 2** is the final phase in the execution of a non-local tile; it performs communication described by schedules, and executes the actual computations for all iterations in the tile. The schedules control communication in such a way that execution of the loop using the local reference list accesses the correct data in local segments and buffers. Non-local data needed for the computations on a processor are gathered from other processors by the runtime communication procedure *gather*. In the computation, the values stored in the array x are updated using the values stored in y. During computation, accumulations to non-local locations of array x are carried out in the buffer associated with array x. This makes it necessary to initialize the buffer corresponding to non-local references of x. To perform this action the function zero_out_buffer is called. After the tile's computation, data in the buffer location of array x is communicated to the owner processors of these data elements by the procedure scatter_add which also accumulates the non-local values of x.

4.5 Experiments

This subsection presents the performance of several parallel versions of a program which includes the data parallel loop (a sweep over edges) taken from the unstructured 3-D Euler solver [24]. All the experiments were carried out on the Intel Paragon System. To be able to compare the perfomance of OOC program versions with the in-core versions using the Paragon's virtual memory system, the programs operated on big unstructured meshes. So, the paging mechanism of Paragon was activated in some experiments. The perfomance results are presented for four implementations of the kernel loop: nonpart-inc: the data arrays and indirection arrays are in-core and are distributed blockwise; nonpart-ooc: the data arrays are in-core (block distributed) and indirection arrays are OOC; part-inc: the data arrays and indirection arrays are in-core and data arrays and loop iterations were redistributed at runtime using the RCB partitioner; part-ooc: the data arrays are in-core, the indirection arrays are OOC and the data arrays and loop iterations were redistributed using the RCB partitioner.

The performance results are given in seconds for different problem sizes and different number of processors and tiles (for OOC programs); 16 or 32 tiles were used on each processor. The symbol *(p)*, appended to some results, expresses that the Intel Paragon's paging mechanism was activated during that experiment. The tables demonstrate a big performance improvement when our OOC parallelization techniques were used in comparison with the unmodified computation using virtual memory. The significant benefit gained from partitioning is also illustrated on the performance improvement of the loop. Because of the very long execution time of some program versions, the loop was executed only once. In real CFD applications, where the loop is executed many times, a dramatical performance improvement can be achieved.

the whole program				itersteps=1
number of processors = 16		**number of tiles = 16/32 (for out-of-core)**		
nodes	250000	302500	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	21.76	380.05 (p)	401.09 (p)	419.35 (p)
nonpart-ooc	28.75 / 30.16	28.30 / 28.27	34.14 / 26.97	34.72 / 42.26
part-inc	543.68 (p)	593.97 (p)	797.08 (p)	1243.40 (p)
part-ooc	95.24 / 137.47	97.23 / 151.35	101.90 / 168.40	124.56 / 190.61

only the loop				
number of processors = 16		**number of tiles = 16/32 (for out-of-core)**		
nodes	250000	302500	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	13.21	367.31	386.58	406.91
nonpart-ooc	27.04 / 28.24	26.69 / 26.61	32.01 / 25.28	32.76 / 40.31
part-inc	339.36	376.87	512.79	746.02
part-ooc	18.17 / 16.52	17.58 / 16.70	16.87 / 16.10	17.23 / 8.75

the whole program				itersteps=1
number of processors = 32		**number of tiles = 16/32 (for out-of-core)**		
nodes	250000	302500	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	22.05	22.08	24.13	25.06
nonpart-ooc	38.00 / 41.36	39.06 / 40.65	39.43 / 41.24	43.51 / 56.80
part-inc	148.91 (p)	259.52 (p)	361.32 (p)	564.51 (p)
part-ooc	138.22 / 228.97	151.90 / 245.65	159.32 / 245.74	168.37 / 270.91

only the loop				
number of processors = 32		**number of tiles = 16/32 (for out-of-core)**		
nodes	250000	302550	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	12.99	13.01	12.96	11.90
nonpart-ooc	34.81 / 38.06	36.43 / 38.10	36.81 / 38.73	40.36 / 54.14
part-inc	35.80(p)	117.33(p)	160.14(p)	240.86(p)
part-ooc	10.47 / 14.05	10.66 / 15.35	14.19 / 15.00	120.78 / 162.97

the whole program				**itersteps=1**
number of processors = 64	**number of tiles = 16/32 (for out-of-core)**			
nodes	250000	302550	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	22.09	22.80	24.78	30.08
nonpart-ooc	50.66/81.31	57.79/94.26	58.87/100.41	58.92 / 92.52
part-inc	201.79	246.89	321.97 (p)	461.82 (p)
part-ooc	230.81/311.28	245.16/411.15	260.35/424.15	269.22 / 408.03

only the loop				
number of processors = 64	**number of tiles = 16/32 (for out-of-core)**			
nodes	250000	302550	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	6.25	9.05	10.98	13.09
nonpart-ooc	45.91 / 76.84	52.68 / 89.26	53.16 / 95.69	54.30 / 87.82
part-inc	3.78	4.61	37.28(p)	130.11(p)
part-ooc	12.71 / 18.56	10.92 / 22.49	12.74 / 26.03	12.14 / 19.29

the whole program				**itersteps=1**
number of processors = 128	**number of tiles = 16/32 (for out-of-core)**			
nodes	250000	302500	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	20.77	23.93	26.11	29.82
nonpart-ooc	86.77 / 157.00	93.33 / 160.72	97.95 / 177.96	102.91 / 171.70
part-inc	387.87	470.72	548.48	636.51
part-ooc	177.58 / 211.59	190.07 / 256.30	260.94 / 315.29	457.54 / 611.34

only the loop				
number of processors = 128	**number of tiles = 16/32 (for out-of-core)**			
nodes	250000	302500	360000	422500
edges	1747002	2114202	2516402	2953602
nonpart-inc	2.08	2.50	2.90	3.39
nonpart-ooc	77.22/147.99	84.32/151.70	88.56/168.67	91.55/162.21
part-inc	1.99	2.43	2.84	3.38
part-ooc	15.34 / 24.41	17.23 / 29.44	20.73 / 31.34	18.52 / 44.01

5 Generalization of Irregular OOC problems

In the previous section we described parallelization steps for OOC irregular problems in which it was assumed that the data arrays could fit in the memory but other data structures describing indirections and interactions could not (because they are an order of magnitude or more larger than the data arrays). Though the above assumption is quite realistic for many problems, in this section we remove this restriction. We present an overview of parallelization steps and the corresponding runtime support in which both the data arrays as well as interaction/indirection arrays are Out-of-Core.

The main problem that must be addressed in this case is that the data arrays are also out-of-core. Consequently, unlike in the previous case, update of a value of a node (data element) may require reading the value from disk, updating it and then storing it back. Therefore, it is possible that each update may result in two I/Os if the data element is not in memory when the iteration(s) updating it are executed. For example, in Figure 5, when executing iteration **10** (marked as edge **10** in the figure), if node **1** is not in memory, then it requires read-update-write operation, resulting in two I/Os. Also, note that, execution of an iteration in the original loop (Figure 4) requires updating both nodes connected to the edge representing the iteration (e.g., nodes **1** and **7** for edge **10** in Figure 5). It is important to note that iterations are organized according to the edges, while node values are updated.

We propose that in order to minimize I/O, it is important to update the values of a node (data element) for the maximum number of times before it is written back, thereby minimizing number of I/Os per data element. *Thus the loop iterations must be split and reorganized to achieve the above goal.* Minimizing I/O for updates does not necessarily minimize overall I/O (although it reduces it to a very large extent), because updates to nodes require reading values of the nodes connected on the other side of the edge. Therefore, scheduling tiles in such a way as to maximize reuse of the read only data, and choosing an appropriate replacement policy for the read-only tiles are also important. However, the total number of I/Os for update phase would still be much larger if the data is not reorganized and loop is not transformed.

In the following subsection, we describe steps for parallelization of OOC irregular problems when both the data elements as well as all indirection/interaction arrays are out–of-core. We will present these steps as extensions of the steps described in the previous section.

Our approach is based on the loop transformation called the *iteration splitting loop transformation* which is depicted in Figure 14. In the original loop, both nodes connected by an edge are updated by statements *S1* and *S2* in one loop iteration. In the transformed loop, in the first phase, a node with the lower index is updated only; the opposite node is updated in the second phase. The ordering of updates is achieved by an appropriate renumbering of *edge1* and *edge2* arrays, which is realized by functions *Renumber1*, ..., *Renumber4*.

If applied to the mesh depicted in Figure 5, the sequence of node accesses is illustrated in

The irregular data-parallel loop *L2* in the following code

```
        integer, parameter :: NNODE = 8, NEDGE = 12, NSTEP = ...
        real x(NNODE), y(NNODE)                        ! data arrays
        integer edge1(NEDGE), edge2(NEDGE)             ! indirection arrays
L1:     do j = 1, NSTEP                                ! outer loop
L2:         do i = 1, NEDGE                            ! inner loop
S1:             x(edge1(i)) = x(edge1(i)) + y(edge2(i))
S2:             x(edge2(i)) = x(edge2(i)) + y(edge1(i))
            end do
        end do
```

is transformed into two loops as follows

```
    temp = Renumber1(edge1);  edge1 = Renumber2(edge2);  edge2 = edge1
L1: do j = 1, NSTEP
      do iphase = 1,2
        do i=1, NEDGE
S:        x(edge1(i)) = x(edge1(i)) + y(edge2(i))
        end do
        if (iphase = 1) then
            temp = Renumber3(edge1);  edge1 = Renumber4(edge2);  edge2 = edge1
        end if
      end do
    end do
```

where *temp* is a temporary array

Figure 14: Iteration Splitting Loop Transformation.

Node Accesses in the Original Loop			PHASE 1 (iphase = 1)			PHASE 2 (iphase = 2)		
i	edge1(i)	edge2(i)	i	edge1(i)	edge2(i)	i	edge1(i)	edge2(i)
1	1	5	1	1	4	1	3	2
2	4	5	2	1	5	2	4	1
3	2	3	3	1	7	3	5	1
4	1	4	4	2	3	4	5	3
5	3	6	5	2	6	5	5	4
6	2	6	6	3	5	6	6	2
7	4	6	7	3	6	7	6	3
8	3	5	8	3	7	8	6	4
9	5	6	9	4	5	9	6	5
10	1	7	10	4	6	10	7	1
11	3	7	11	5	6	11	7	3
12	5	7	12	5	7	12	7	5

Figure 15: Node Accesses in the Original and Transformed Loops.

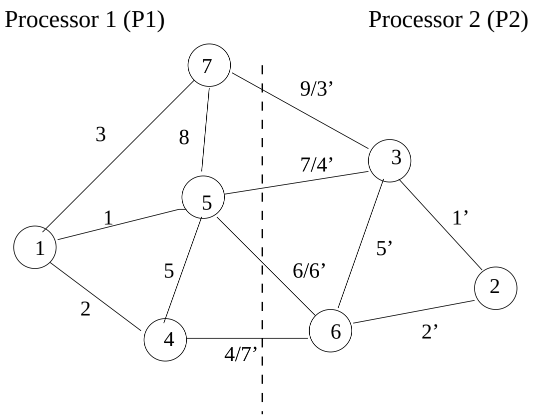
Figure 15.

5.1 Parallelization Steps and Runtime Support

For parallelization, steps up to step E are the same as in the previous problem described in Section 4. At the end of step E, each processor has local files that are created using the output of a partitioner. Furthermore, iterations (indirection arrays) are distributed to minimize interprocessor communication. The only difference is that the data elements also reside in local files. We describe the proposed support for the subsequent steps below.

F. In this step the loop iterations are first split. This essentially requires partitioning the indirection arrays (or edges) into two sets. Each of these sets is then sorted according to the node number. In other words, each iteration is enumerated and sorted (thereby renumbering the iterations) so that all iterations accessing the lowest number element is grouped first followed by the second lowest and so on. That is, they are organized in the ascending order. This is illustrated in Figure 16, where all iterations accessing node 1 are bunched together. Note that each processor performs this algorithm in parallel on its local iteration sets and data. The rationale behind this was alluded to earlier, where we stated that by this reorganization

THE FIRST PHASE

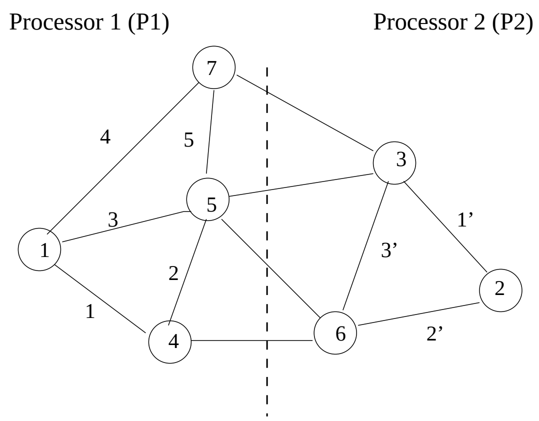



```
Processor 1                              Processor 2

1. x(1) = x(1) + y(5)              1'. x(2) = x(2) + y(3)
2. x(1) = x(1) + y(4)              2'. x(2) = x(2) + y(6)
3. x(1) = x(1) + y(7)              3'. x(3) = x(3) + y(6)
4. x(4) = x(4) + y(5)              4'. x(3) = x(3) + y(5)
5. x(4) = x(4) + y(5)              5'. x(3) = x(3) + y(7)
6. x(5) = x(5) + y(7)              6'. x(6) = x(6) + y(5)
7. x(5) = x(5) + y(3)              7'. x(6) = x(6) + y(4)
8. x(5) = x(5) + y(6)
9. x(7) = x(7) + y(3)
```

Figure 16: Iterations Executed in the 1st Phase.

THE SECOND PHASE




```
Processor 1                              Processor 2

1. x(4) = x(4) + y(1)             1'. x(3) = x(3) + y(2)
2. x(5) = x(5) + y(4)             2'. x(6) = x(6) + y(2)
3. x(5) = x(5) + y(1)             3'. x(6) = x(6) + y(3)
4. x(7) = x(7) + y(1)
5. x(7) = x(7) + y(5)
```

Figure 17: Iterations Executed in the 2nd Phase.

read-modify-write I/O operations will be minimized by performing a large number of updates before requiring more I/O operations. All nodes requiring interprocessor communication are also included in this set. This allows communication to be performed only in one phase. The second set minus the iterations requiring interprocessor communication are also reorganized by sorting in the ascending order to minimize I/O.

Figure 16 illustrates the iterations executed in the first phase. The edge numbers describe the iteration numbers. Iterations marked with a $'$ are executed on processor 2 whereas all others are executed on processor 1. For example, $7/4'$ means that update of node 5 is performed on processor 1 in this phase (in the executor step), where as the update of node 3 is performed in iteration 4 of processor 2. Figure 17 describes the iterations executed in the second phase. Note that no interprocessor communication is required since it is performed entirely in the first phase.

G. Once the above reorganization is done, the inspector can be executed. The inspector also requires additional steps than in the previous case. Inspector essentially develops a communication schedule for the first phase (again this can be performed either in its entirety or once for each slab/tile). Furthermore, inspector for each slab is used to determine the read-only slabs schedule for each tile being updated. This information can be effectively used for prefetching read-only slabs and determining the replacement policy. The number of read-only slabs depends on the amount of memory available.

   Once the schedules are developed for communication and I/O, executor may be executed.

H. Step H represents the executor. In this phase, for each time step, the execution in performed in two phases (once for each loop obtained after splitting).

5.2 Implementation Outline

For the class of applications considered, data and iteration partitioners must be out-of-core. They are based on so called *disk oriented partitioning algorithms*; the partitioner will have available only a limited main memory area.

Application of the parallelization strategy which was discussed in the previous paragraph is illustrated in Figures 18 and 20. Below, we outline the effect of individual processing steps, if they are applied to the mesh depicted in Figure 5.

1. *Initialization*

   OOC data arrays x and y are stored in the global files that are denoted by *x_file* and *y_file*, respectively. The mesh description is stored in the file denoted by *mesh_file*; in this file, the *edge1* elements precede the *edge2* elements, as expressed below

   mesh_file : {1,5,2,1,5,2,6,5,6,1,7,5, | 5,4,3,4,2,6,4,3,5,7,3,7}

1. *Initialization*
   open (y_file, **filename** = 'y.dat', **status** = 'old', , **action** = 'read')
   open (mesh_file, **filename** = 'edge1_edge2.dat', **status** = 'old', **action** = 'read')
   open (x_file, **filename** = 'x.dat', **status** = 'old', **action** = 'readwrite')
2. *Disk-Oriented Data Partitioning*
   Construct out-of-core data distribution descriptor (translation table)
   which is stored in the file trans_file.
3. *Creation of local data files*
   y_file_loc and x_file_loc are constructed for each processor from y_file and x_file according to
   the new data distribution computed in step 2.
4. *Basic Disk-Oriented Iteration Partitioning*
   Construct out-of-core iteration distribution descriptor
   (iteration table) which is stored in the file iter_file_loc at each processor.
5. *Construction of Iteration Tiles for Both Phases*
   Phase1: edge1_file_ph1loc, edge2_file_ph1loc; tiles that do not need communication
   precede other tiles
   Phase2: edge1_file_ph2loc, edge2_file_ph2loc

Figure 18: OOC Data and Iteration Partitioning and Reorganization of the Files.

2. *Data Partitioning*

   A disk-oriented partitioning algorithm (referred also to as the OOC partitioner) decomposes x and y data sets to a number of partitions and allocates each partition to a processor. The produced partitions need to have the minimum number of connections to other partitions (minimum communication and remote disk access requirements) and they should have the approximately equal number of data elements (good load balancing). The distribution produced by a partitioner results in a table called the *translation table* that stores a processor reference and the logical offset in that processor's memory for each array element. The translation table is blockwise distributed across the corresponding local files. If we assume that the partitioner produces the distribution depicted in Figure 6, and the translation table is distributed across two local files, each denoted by *transtab_file_loc*, the local files will have the following contents:

   transtab_file_loc at processor 1 (describes elements with indices 1-4) : {(1,0),(2,0),(2,1),(1,1)}
   transtab_file_loc at processor 2 (describes elements with indices 5-7) : {(1,2),(2,2),(1,3)}

   For example, information about the owner processor and offset for the array element *y(5)* is stored as the first record of the file *transtab_file_loc* at processor 2; *y(5)* has got offset 2 in the data segment of y assigned to processor 1.

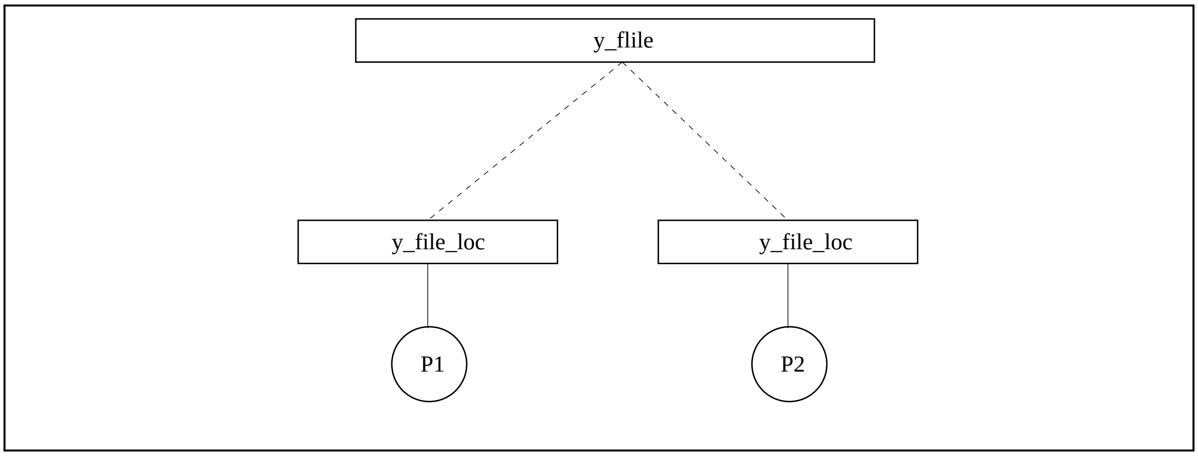


Figure 19: Redistribution of Datafiles.

3. *Creation of local data files*
To complete the data partitioning process, all files carrying data arrays are redistributed to the new data distribution described by the translation table. In our example, the appropriate local files are constructed from the files storing arrays x and y as illustrated in Figure 19.

The contents of the local files are as follows:

x_file_loc at processor 1 : {x(1),x(4),x(5),x(7)}   *x_file_loc* at processor 2: {x(2),x(3),x(6)}
y_file_loc at processor 1 : {y(1),y(4),y(5),y(7)}   *y_file_loc* at processor 2 : {y(2),y(3),y(6)}

4. *Basic Disk-Oriented Iteration Partitioning*
Loop iterations are redistributed according to the new data distribution. The table describing the iteration distribution is distributed blockwise across specific local files. In our case, two local files, *itertab_file_loc* at processor 1 and *itertab_file_loc* at processor 2, are created. Their contents are as follows (compare with Figure 6):

itertab_file_loc at processor 1 (iterations 1-6) : {1,1,2,1,2,2}
itertab_file_loc at processor 2 (iterations 7-12) : {2,2,**1**,1,2,1}

For example, the iteration 9 is currently assigned to processor **1** (this information is stored as the third record of the file *itertab_file_loc* at processor 2).

5. *Construction of Iteration Tiles for Both Phases*
Iterations assigned to individual processors by the basic partitioning algorithm in the previous step are reorganized and formed into tiles; on each processor one set of tiles is formed for Phase 1 and another one for Phase 2. Tiles that do not need communication (no communication

buffers have to be allocated) may be larger and they precede tiles that include inter-processor communication. The resulting sets of tiles determine the distribution of the edge data across local files. Below, local edge files are enumerated as they are needed in the corresponding phases, and their contents are described:

- Phase1: *edge1_file_ph1loc*, *edge2_file_ph1loc*
- Phase2: *edge1_file_ph2loc*, *edge2_file_ph2loc*

edge1_file_ph1loc at processor 1 : {1,1,1,4,4,5,5,5,7}
edge2_file_ph1loc at processor 1 : {5,4,7,5,6,7,3,6,3}
edge1_file_ph1loc at processor 2 : {2,2,3,3,3,6,6}
edge2_file_ph1loc at processor 2 : {1,1,1,4,4,5,5,5,7}
edge1_file_ph2loc at processor 1 : {4,5,5,7,7}
edge2_file_ph2loc at processor 1 : {1,4,1,1,5}
edge1_file_ph2loc at processor 2 : {3,6,6}
edge2_file_ph2loc at processor 2 : {2,2,3}

Note that accesses to the elements of y exhibit some sort of (irregular) temporal data locality.

With reorganization of the files, the data is prefetched as close to the processors that need them; moreover they can be read in blocks and the application of the data sieving technique [12] may be more efficient.

6. *Inspector/Executor - Phase 1*
Information describing the tiles which are going to be applied in Phase 1 and Phase 2 is stored in the descriptors *tile_ph1_desc* and *tile_ph2_desc*, respectively. The slab of *edge1* read at line *F2* is analyzed by procedure *analyze* (line *F3*) which determines the slab (tile) of x (described by the lower bound *x_lwb* and size *x_size*) to be read from file *x_file_x* at line *F4*. Moreover, procedure *analyze* precomputes the local indices and stores the results of this index conversion in array *locref_x*. The slab of *edge2* read at line *F5* is used by the procedure *disk_mem_localize* (line *F6*) to compute two data structures, *disk_sched* and *mem_sched*, that describe the required I/O and communication for y; the numbers of elements to be transferred are stored in variables *n_disk* and *n_mem* and the results of the index conversion in array *locref_y*. The elements of y are gathered from disk and other processors by procedures *disk_gather* and *mem_gather*. The incoming data elements are stored in the memory portion allocated for y; the first part of the portion is filled by the elements from disk and the second one by the elements from other processors. The executor is represented by the statements at lines *F9* - *F11*. The values of x computed by the inspector are stored in the local files *x_file_loc*.

6. Out-of-Core Inspector/Executor - Phase 1

```
F1:   DO tile = 1, number_of_tiles_phase1
F2:      CALL read_slab(edge1_file_ph1loc, edge1, tile_ph1_desc, tile)
F3:      CALL analyze(edge1, locref_x, x_lwb, x_size)
F4:      CALL read_slab(x_file_loc, x, x_lwb, x_size)
F5:      CALL read_slab(edge2_file_ph1loc, edge2, tile_ph1_desc, tile)
F6:      CALL disk_mem_localize(disk_sched, mem_sched, locref_y, edge2, n_disk, n_mem)
F7:      CALL disk_gather(disk_sched, y_file, y(1))
F8:      CALL mem_gather(mem_sched, y(n_disk+1))
F9:     DO i = 1, tile_ph1_desc%tile_size(tile)
F10:           x(locref_x(i)) = x(locref_x(i)) + y(locref_y(i))
F11:     END DO
F12:     CALL  write_slab(x_file_loc, x, x_lwb, x_size)
F13:  END DO
```

7. Out-of-Core Inspector/Executor - Phase 2

```
S1:   DO tile = 1, number_of_tiles_phase2
S2:      CALL read_slab(edge1_file_ph2loc, edge1, tile_ph2_desc, tile)
S3:      CALL analyze(edge1, locref_x, x_lwb, x_size)
S4:      CALL read_slab(x_file_loc, x, x_lwb, x_size)
S5:      CALL read_slab(edge2_file_ph2loc, edge2, tile_ph2_desc, tile)
S6:      CALL disk_localize(disk_sched, locref_y, edge2, n_disk)
S7:      CALL disk_gather(disk_sched, y_file, y(1))
S8:      DO i = 1, tile_ph2_desc%tile_size(tile)
S9:             x(locref_x(i)) = x(locref_x(i)) + y(locref_y(i))
S10:     END DO
S11:     CALL  write_slab(x_file_loc, x, x_lwb, x_size)
S12:  END DO
```

Figure 20: Inspector/Executor.

7. *Inspector/Executor - Phase 2*
   In Phase 2, no inter-processor communication is needed. Therefore, only the disk schedule is computed (line *S6*) that passed to the procedure *disk_gather* (line *S7*) that gathers the y elements from disk.

6 Summary and Future Work

In this paper we presented techniques for implementing Out-Of-Core irregular problems. In particular we presented a design for a runtime system to efficiently support OOC irregular problems. We also presented the appropriate transformations required to reduce the I/O overheads for staging data as well as for communication. For each step of the parallelization process, we described optimizations.

The proposed techniques can be used by a compiler that compiles a global name space program (e.g., written in HPF and its extensions) or by users writing programs in *node+message passing* style (e.g., *C+MPI*). The runtime system design, required transformations and implementation approach were presented in two parts. We first presented a restricted version of the problem in which it was assumed that only part of the data (large data structures) are out-of-core. We also described that this is a reasonable assumption since data describing interaction among the basic data elements of a problem may be an order of magnitude or more larger. Therefore, even if the problem size is such that the basic data sets could fit in memory, other data structures will still be out-of-core. Then we generalized these techniques in which all the major datasets of an application were assumed to be out-of-core. We illustrated through examples what are the various steps required in the parallelization process, and provided insight into the runtime system, transformations and implementation. Currently, we are implementing these techniques to evaluate their effectiveness and develop more optimizations.

Acknowledgment

The authors would like to thank Joel Saltz and his research group for providing the CHAOS library. Thanks also go to Minh Dang for implementing the out-of-core CHAOS extensions and Ka Heng Wong for carrying out all the performance measurements.

References

[1] Meenakshi Arunachalam, Alok Choudhary, and Brad Rullman. A prefetching prototype for the parallel file system on the Paragon. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 321–323, May 1995. Extended Abstract.

[2] Rajesh Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. PhD thesis, Department of Computer and Information Science, Syracuse University, Syracuse, NY, 1994.

[3] Rajesh Bordawekar and Alok Choudhary. Issues in compiling I/O intensive problems. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 3, pages 69–96. Kluwer Academic Publishers, 1996.

[4] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376, 1993.

[5] Rajesh Bordawekar, Alok Choudhary, and Rajeev Thakur. Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines. Technical Report SCCS-622, NPAC, Syracuse, NY 13244, September 1994.

[6] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[7] Rajesh Bordawekar, Rajeev Thakur, and Alok Choudhary. Efficient compilation of out-of-core data parallel programs. Technical Report SCCS-622, NPAC, April 1994.

[8] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proc. of the Scalable High Performance Computing Conference*, pages 314–321, Williamsburg, USA, April 1992.

[9] P. Brezany and V. Sipkova. Coupling parallel data and work partitioners to vfcs. In *Proceedings of the Conference EUROSIM – HPCN Challenges 1996*, pages 104–111. North Holland, Elsevier, June 1996.

[10] Peter Chen, Garth Gibson, Randy Katz, and David Patterson. An evaluation of redundant arrays of disks using an Amdahl 5890. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1990.

[11] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[12] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[13] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[14] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[15] R. Das and J. Saltz. *A manual for PARTI runtime primitives - Revision 2*, December 1990.

[16] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, April 1992.

[17] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[18] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[19] R. Ponnusamy et al. A manual for the chaos runtime library. Technical report, University of Maryland, May 1994.

[20] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[21] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, page 177, December 1991.

[22] C. Koelbel. Compiling programs for nonshared memory machines. Technical Report Ph.D. Dissertation, Purdue University, West Lafayette, November 1990.

[23] David Kotz and Nils Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Parallel and Distributed Technology*, pages 51–60, Spring 1995.

[24] D. J. Mavriplis. Three dimensional unstructured multigrid for the euler equations. In *AIAA 10th Computational Fluid Dynamics Conference*, pages paper 91–1549cp, June 1991.

[25] Steven A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. Technical Report CSTR-950601, Emory University, June 1995.

[26] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[27] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proc. of Supercomputing'93*, Portland, OR., November 1993.

[28] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified data distributions. *IEEE Transactions on Parallel and Distributed Systems*, August 1995.

[29] Ravi Ponnusamy. *Runtime Support and Compilation Methods for Irregular Computations on Distributed Memory Parallel Machines.* PhD thesis, Department of Computer and Information Science, Syracuse University, Syracuse, NY, 1994.

[30] A. Reddy and P. Banerjee. An evaluation of multiple-disk I/O systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:315–322, 1989.

[31] Brad Rullman and David Payne. An efficient file I/O interface for parallel applications. DRAFT presented at the Workshop on Scalable I/O, Frontiers '95, February 1995.

[32] J. Saltz, K. Crowley, R. Mirchandaney, , and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[33] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kudatipidi. Passion: Optimized i/o for parallel systems. *IEEE Computer*, June 1996.

[34] R. von Hanxleden. Handling irregular problems with fortran d. In *Proc. of the Forth Workshop on Compilers for Parallel Computers*, pages 353–364, Netherland, December 1993. Delft.