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

Peter Brezany\textsuperscript{a} and Alok Choudhary\textsuperscript{b}

\textsuperscript{a}Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstr. 22, A-1092 Vienna
E-mail: brezany@par.univie.ac.at

\textsuperscript{b}ECE Department, Northwestern University, Evanston, IL
E-mail: choudhar@ece.nwu.edu

October 6, 1996

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 \textit{node} + \textit{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 those 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 ................................................................. 3  
  1.1 Background ............................................................ 3  
  1.2 Applications Perspective ............................................ 4  
  1.3 This Paper ............................................................. 6  

2 Models for Out-Of-Core Computations .................................... 6  
  2.1 Local Placement Model ................................................ 7  
  2.2 Global Placement Model .............................................. 7  
  2.3 Partitioned In-core Model ............................................ 8  

3 Irregular Problems ................................................................ 10  

4 Parallelization Methods and Runtime Support for OOC Irregular Problems 12  
  4.1 Problem Description and Assumptions ............................... 13  
  4.2 Data Arrays In-Core and Indirection/Interaction Arrays Out-of-Core 14  
  4.3 Parallelization Steps and Runtime Support .......................... 14  
  4.4 Implementation Design .................................................. 17  
    4.4.1 Data Partitioning ................................................... 17  
    4.4.2 Data Remapping ..................................................... 21  
    4.4.3 Partitioning and Redistribution of Loop Iterations ............ 21  
    4.4.4 Inspector and Executor .......................................... 23  
  4.5 Experiments .............................................................. 24  

5 Generalization of Irregular OOC problems .................................. 26  
  5.1 Parallelization Steps and Runtime Support .......................... 27  
  5.2 Implementation Outline ............................................... 32  

6 Summary and Future Work ................................................... 35  

References ........................................................................... 38
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, 29]. 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, 25, 21, 16, 30, 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 [24]. 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

1Such 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)

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

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 300KM² 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 150KM² 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.

2For 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 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.

The rest of the paper is organized as follows. In Section 2 we describe three models for OOC computations [2, 12, 32]. 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, 32, 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

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 straightforward 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 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
Figure 1: Local Placement Model.

Figure 2: Global Placement Model.
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, 31, 28, 27]. 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, 33, 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, 26, 27] 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 which are called from the compiler embedded code.

Large scale irregular applications involve large arrays and other data structures. Runtime

---

3CHAOS is a runtime library that subsumes PARTI.
4Most real world applications are large-scale, but their sizes have been limited by the memory of the available systems.
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.\footnote{This is also known as the scale-down problem.}

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 airfoil) 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 [28]. The primary 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 \( K \times N \) (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 amount of memory required to run an application with data size of \( N \) will be at least 50\( N \) 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.
Figure 5: Example Mesh with Seven Nodes and Twelve Edges.

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

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

This class of problems will be referred to as the DAI/IAO (Data Arrays are In-core / Indirection Arrays are Out-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. 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 5.

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

\footnote{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.}
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.](image)

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.