Parallelization of Irregular Out-of-Core Applications for Distributed-Memory Systems*

Peter Brezany\textsuperscript{a}, Alok Choudhary\textsuperscript{b} (presenting author), and Minh Dang\textsuperscript{a}
\textsuperscript{a}Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, E-mail: \{brezany,dang\}@par.univie.ac.at
\textsuperscript{b}ECE Department, Northwestern University, Evanston, IL
E-mail: choudhar@ece.nwu.edu, Phone: (847) 467-4129. Fax: (847) 491-4455

Abstract

Large scale irregular applications involve data arrays and other data structures that are too large to fit in main memory and hence reside on disks; such applications are called out-of-core applications. This paper presents techniques for implementing this kind of applications. In particular we present a design for a runtime system to efficiently support parallel execution of irregular out-of-core codes on distributed-memory systems. Furthermore, we describe the appropriate program transformations required to reduce the I/O overheads for staging data as well as for communication while maintaining load balance. The proposed techniques can be used by a parallelizing compiler or by users writing programs in node+message passing style. We have done a preliminary implementation of the techniques presented here. We introduce experimental results from a template CFD code to demonstrate the efficacy of the presented techniques.

1 Introduction

In this paper, we present design of various steps, runtime system and program 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. The runtime system is built up on top of the CHAOS library [4]. Furthermore, we

*The work described in this paper is being carried out as part of the research project "Language, Compiler and Advanced Data Structure Support for Parallel I/O Operations" supported by the Austrian Research Foundation (FWF Grant P11006-MAT).
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 rest of the paper is organized as follows. Section 2 describes the basic computational characteristics of irregular problems and presents the runtime system as well as transformations required for a class of irregular problems considered in this paper. Performance results are presented in Section 3. Finally, summary and conclusions are presented in Section 4.

2 Parallelization Methods for OOC Irregular Problems

One strategy used for parallelization of irregular codes, assuming that all the data structures fit into the memory (i.e., an in-core parallelization), constructs three code phases for each loop specifying irregular computations, called the work distributor, the inspector, and the executor [5, 9, 8, 7]. 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 from the execution set. Furthermore, in dynamic and adaptive irregular codes, the data arrays may be redistributed before executing the inspector and executor in order to minimize communication and balance load. Execution of irregular programs that are parallelized on this principle is supported by an appropriate runtime library [3].

The primary goal of data distribution, workload distribution, and inspector steps is to minimize communication while obtaining load balance so that 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 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.

2.1 Running Example

Figure 1 illustrates a typical irregular loop. The inner loop L2 in this code represents a sweep over the edges of an unstructured mesh of size: NNODE, NEDGE. Figure 2 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 nodes 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. 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.
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 1: An Example with an Irregular Loop.

2.2 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 \( KN \) (e.g., CFD) or \( O(N^2) \) (e.g., molecular dynamics). In this paper, we consider the parallelization strategy for this class of irregular problems only; paper [1] outlines a solution for OOC irregular problems in which all major data structures reside in files.

2.3 OOC Data Organization Model

The parallelization methods and runtime system designed in this paper are based on the the Local Placement Model proposed by Choudhary et al. [2]. The indirection 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). OCLA of each processor is stored in separate logical file called the Local Array File (LAF) of that processor. The node program explicitly reads from to the LAF when required.

2.4 Parallelization Steps

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. Therefore, our initial goal is to introduce additional steps as well as modify existing steps in the inspector-executcr model to satisfy the goals of minimizing I/O costs. In the following, we describe the proposed steps using the kernel illustrated in Figure 1. Each of these steps is described in more detail below.

A. Default Initial Data and Work Distribution. 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. Partition Data Arrays. 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.

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

D. Compute new iteration tiles. Once the new array distribution is obtained, loops iterations that access the distributed arrays must also be partitioned among processors to optimize load balance and communication volume.

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. For example, iterations 1, 2, 4, 10, and 12 in Figure 2 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. The non-local tiles include iterations that require off-processor accesses. For example, iterations 7, 8, 9, and 11 in Figure 2.

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

F. Out-of-Core Inspector. Since the inspector requires building communication schedules for only non-local tiles, it does not require a pass through the entire execution 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. Out-of-Core 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.

2.5 Implementation Design

Data Partitioning. 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. Below we illustrate a call to the recursive coordinate partitioner.

\[
\text{call partitioner.RCB(2, coord1, coord2, ndata, trat)}
\]
Here, the use of geometric information is shown. Arrays coord1 and coord2 carry the spatial coordinates for elements in \( x \) and \( y \). Variable \( trat \) denotes the translation table returned by the partitioner.

**Partitioning and Redistribution of Loop Iterations.** The newly specified data array distribution represented by the translation table is used to decide how loop iterations are to be distributed among processors. This distribution is done via procedure \( \text{iter} \cdot \text{partitioner} \) (a call to it is depicted below) which includes the generating the runtime data structure, called \text{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.

\[
\begin{align*}
\text{inp\_files(1)} &= \text{edge1\_file}; \quad \text{inp\_files(2)} = \text{edge2\_file}; \quad \text{number\_of\_inpfiles} = 2 \\
\text{out\_files(1)} &= \text{local\_edge1\_file}; \quad \text{out\_files(2)} = \text{local\_edge2\_file}; \quad \text{number\_of\_outfiles} = 2
\end{align*}
\]

\textbf{call iter\_partitioner} (\( \text{trat, inp\_files, number\_of\_inpfiles, out\_files, number\_of\_outfiles, } \\
\quad \text{number\_of\_all\_tiles, number\_of\_local\_tiles, lwbs, upbs})

A loop iteration is assigned to the processor that owns the maximum number of referenced data elements.

The next step is the splitting the iterations assigned to a processor into an ordered set of tiles, where \( \text{(number\_of\_local\_tiles)} \) local tiles are followed by \( \text{(number\_of\_all\_tiles - number\_of\_local\_tiles + 1)} \) non-local tiles.

Finally, two local files, \( \text{local\_edge1\_file} \) and \( \text{local\_edge2\_file} \), storing indirection arrays are created. The ordering of array elements in the files establishes alignment with the iterations in the tiles. Pointers to these local files are stored in the array \( \text{out\_files} \). The tile descriptions are returned in the arguments \( \text{lwbs} \) and \( \text{upbs} \), whose values point to the sections of edge1 and edge2 in the respective local files.

**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 3. For each tile, contiguous blocks of values \( \text{edge1} \) and \( \text{edge2} \) 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 \( \text{ind\_conv} \) performs the global to local index conversion for array references that don’t refer non-local elements. The local indices are used in the \textbf{Executor 1} phase.

In the phase that processes non-local tiles, the list \( \text{globref} \), its size and the translation table \( \text{(trat)} \) of the referenced array are used by the CHAOS procedure \( \text{localize} \) to determine the appropriate \textit{schedule}, the size \( \text{(nnloc)} \) of the communication buffers to be allocated, and the local reference list \( \text{locref} \) which contains results of the global to local index conversion. The declarati-
C–Processing local tiles
    do tiles = 1, number_of_local_tiles
        tile_size = upbs(tile) - lwbs(1) + 1
        call read_tile(files(1), edge1, lwbs(tile), tile_size)
        call read_tile(files(2), edge2, lwbs(tile), tile_size)
    end do
    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(trat, 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
C–Processing non-local tiles
    do tiles = number_of_local_tiles + 1, number_of_tiles
        call read_tile(files(1), edge1, lwbs(tile), tile_size)
        call read_tile(files(2), edge2, lwbs(tile), tile_size)
    end do
    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(trat, schedule, globref, locref, 2*tile_size, nnloc, local_size)
    call zero_out_buffer(y, y(loseg_y_size+1), sched)
    call gather(y, y(loseg_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(loseg_x_size+1), sched)
end do

Figure 3: The Inspector and Executor

7
ons of arrays $x$ and $y$ in the message passing code allocate memory for the local segments of size $localSize$ holding the local data and the communication buffers 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 $zeroOutBuffer$ 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 $scatterAdd$ which also accumulates the non-local values of $x$.

### 2.6 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 [6]. All the experiments were carried out on the Intel Paragon System. To be able to compare the performance 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 performance results are presented for four implementations of the kernel loop: $nonpart-inc$: the data arrays and indirect arrays are in-core and are distributed blockwise; $nonpart-ooc$: the data arrays are in-core (block distributed) and indirect arrays are OOC; $part-inc$: the data arrays and indirect 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 indirect 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.

<table>
<thead>
<tr>
<th></th>
<th>number of processors = 32</th>
<th>number of tiles = 16/32 (for out-of-core)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>250000</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>1747002</td>
</tr>
<tr>
<td></td>
<td>nonpart-inc</td>
<td>22.05</td>
</tr>
<tr>
<td></td>
<td>nonpart-ooc</td>
<td>38.00/41.36</td>
</tr>
<tr>
<td></td>
<td>part-inc</td>
<td>148.91 (p)</td>
</tr>
<tr>
<td></td>
<td>part-ooc</td>
<td>138.22/228.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>number of processors = 32</th>
<th>number of tiles = 16/32 (for out-of-core)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>250000</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>1747002</td>
</tr>
<tr>
<td></td>
<td>nonpart-inc</td>
<td>12.99</td>
</tr>
<tr>
<td></td>
<td>nonpart-ooc</td>
<td>34.81 / 38.06</td>
</tr>
<tr>
<td></td>
<td>part-inc</td>
<td>35.80(p)</td>
</tr>
<tr>
<td></td>
<td>part-ooc</td>
<td>10.47 / 14.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>number of processors = 64</th>
<th>number of tiles = 16/32 (for out-of-core)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>250000</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>1747002</td>
</tr>
<tr>
<td></td>
<td>nonpart-inc</td>
<td>22.09</td>
</tr>
<tr>
<td></td>
<td>nonpart-ooc</td>
<td>50.66/81.31</td>
</tr>
<tr>
<td></td>
<td>part-inc</td>
<td>201.79</td>
</tr>
<tr>
<td></td>
<td>part-ooc</td>
<td>230.81/311.28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>number of processors = 64</th>
<th>number of tiles = 16/32 (for out-of-core)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>250000</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>1747002</td>
</tr>
<tr>
<td></td>
<td>nonpart-inc</td>
<td>6.25</td>
</tr>
<tr>
<td></td>
<td>nonpart-ooc</td>
<td>45.91 / 76.84</td>
</tr>
<tr>
<td></td>
<td>part-inc</td>
<td>3.78</td>
</tr>
<tr>
<td></td>
<td>part-ooc</td>
<td>12.71 / 18.56</td>
</tr>
</tbody>
</table>
3 Conclusion

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. The appropriate transformations steps required to reduce the I/O overheads for staging data as well as for communication are also presented with several optimization possibilities. The experiment results show a clear performance improvement that can be achieved with our techniques. It also shows that virtual memory with paging mechanism cannot be a good alternative solution for OOC data because of significant degradation of performance.

References


