Language and Compiler Support for Out-of-Core Irregular Applications on Distributed-Memory Multiprocessors*

Peter Brezany[1], Alok Choudhary[2], and Minh Dang[1]

[1] Institute for Software Technology and Parallel Systems, University of Vienna Liechtensteinstrasse 22, A-1090 Vienna, EM: {brezany,dang}@par.univie.ac.at

[2] ECE Department, Northwestern University, Evanston, EM: choudhar@ece.nwu.edu

Abstract. Current virtual memory systems provided for scalable computer systems typically offer poor performance for scientific applications when an application's working data set does not fit in main memory. As a result, programmers who wish to solve "out-of-core" problems efficiently typically write a separate version of the parallel program with explicit I/O operations. This task is onerous and extremely difficult if the application includes indirect data references. A promising approach is to develop a language support and a compiler system on top of an advanced runtime system which can automatically transform an appropriate in-core program to efficiently operate on out-of-core data. This approach is presented in this paper. Our proposals are discussed in the context of HPF and its compilation environment.

1 Introduction

A wide class of scientific and engineering applications, called *irregular applications*, greatly benefit from the advent of powerful parallel computers. However, the efficient parallelization of irregular applications for distributed-memory multiprocessors (*DMMP*s) is still a challenging problem. In such applications, access patterns to major data arrays are only known at runtime, which requires runtime preprocessing and analysis in order to determine the data access patterns and consequently, to find what data must be communicated and where it is located.

The standard strategy for processing parallel loops with irregular accesses, developed by Saltz, Mehrotra, and Koelbel [9], generates three code phases, called the *work distributor, the inspector,* and *the executor*. Optionally, a dynamic *partitioner* can be applied to the loop [2,12].

* The work described in this paper is being carried out as part of the research project "Aurora" supported by the Austrian Research Foundation.

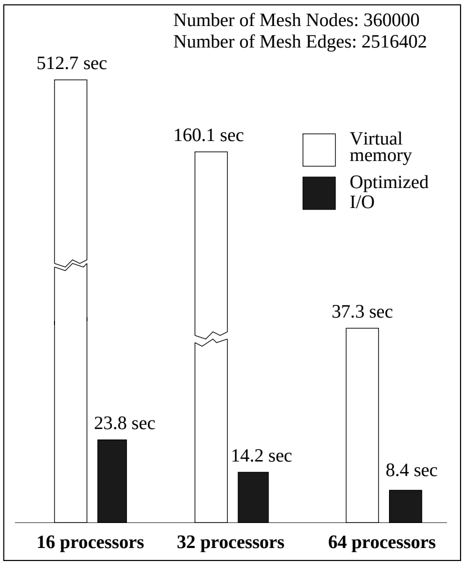

Fig. 1. Optimized file I/O vs. VM

Large scale irregular applications involve large data structures. Runtime preprocessing provided for these applications results in construction of additional large data structures which increase the memory usage of the program substantially. Consequently, a parallel program may quickly run out of memory. Therefore, some data structures must be stored on disks and fetched during the execution of the program. Such applications and data structures are called *out-of-core (OOC) applications* and *OOC data structures*, respectively. The performance of an OOC program strongly depends on how fast the processors, the program runs on, can access data from disks. Traditionally, in scientific computations, OOC problems are handled in two different ways: (1) virtual memory (VM) which allows the in-core program version to be run on larger data sets, and (2) specific OOC techniques which explicitly interface file I/O and focus on its optimization. Although VM is an elegant solution (it provides the programming comfort and ensures the program correctness), it has been observed that the performance of scientific applications that rely on virtual memory is generally poor due to frequent paging in and out of data.

In our recent papers [3,4], we describe the *runtime support* CHAOS+ we have developed for parallelization of irregular OOC applications on DMMPs. Fig. 1 shows the performance improvement obtained by the OOC version[1] of the Euler 3-D solver which was built on top of CHAOS+ against VM on the Intel Paragon. Although the interface to CHAOS+ is a level higher than the interface to a parallel file system and the interface to a communication library, it is still difficult for the application programmer to develop efficient OOC irregular programs. A promising approach is to develop a language support and a compiler system on top of an advanced runtime system which can automatically transform an appropriate in-core program to efficiently operate on OOC data. This approach is discussed in this paper.

[1] Only indirection arrays were out-of-core in this program version.

In Section 2, we describe the language directives available to the programmer which provide useful information to the compiler. These directives are proposed as a part of the language HPF$^+$ [5]. Section 3 presents basic and advanced compiler methods to transform the OOC program and insert communication and I/O. Experimental results are discussed in Section 4. We review related work and conclude in Sections 5 and 6.

2 Language Support

The DISTRIBUTE directive in HPF partitions an array among processors by specifying which elements of the array are mapped to each processor. This results in each processor storing a local array which is called the *distribution segment.*

In order to allocate memory and handle accesses to OOC arrays, the HPF compiler needs information about which arrays are out-of-core and also the maximum amount of in-core memory that is allowed to be allocated for each array. The user may provide this information by a directive of the following form:

$$\text{!HPF+\$ } [\textit{dist\_spec},]\ \mathsf{OUT\_OF\_CORE}\ [,\mathsf{IN\_MEM}\ (\textit{ic\_portion\_spec})] :: ar_1, .., ar_k$$

where ar_i specify array identifiers, and the optional part *dist_spec* represents an HPF *distribution-specification* annotation. The keyword OUT_OF_CORE indicates that all ar_i are out-of-core arrays.

In the second optional part, the keyword IN_MEM indicates that only the array portions of the global shape that corresponds to *ic_portion_spec* are allowed to be kept in main memory. During computation, each processor brings a section of ar_i into its memory part, called *in-core local array (ICLA)*, and operates on it and stores it back, if necessary. The shape of ICLA is computed from *ic_portion_spec.*

If the data for an OOC array comes from an input file or is to be written to an output file then the file name must be specified. For this the ASSOCIATE directive ([7]) is used.

The OUT_OF_CORE directives can be introduced in the specification part of the program unit or can immediately precede the parallel loop specification. For example, in Fig. 2, arrays X, Y, EDGE1 and EDGE2 will be handled as OOC arrays in the loop L which represents a sweep over the edges of an unstructured mesh. The data for X comes from the file 'X_file.dat'. ICLAs of shape ($\lceil K/M \rceil$) are allocated for EDGE1 and EDGE2 on each processor. The shape of ICLAs allocated for X and Y is determined by the programming environment.

```
!HPF$ PROCESSORS P(M)
REAL X(NNODE), Y(NNODE); INTEGER EDGE1(NEDGE), EDGE2(NEDGE)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: X, Y, EDGE1, EDGE2
...
... X and Y are data arrays, EDGE1 and EDGE2 are indirection arrays ...
!HPF+$ ASSOCIATE ('X_file.dat', X)
... ASSOCIATE directives for Y, EDGE1, and EDGE2 ...
!HPF+$ OUT_OF_CORE :: X, Y
!HPF+$ OUT_OF_CORE, IN_MEM (K) :: EDGE1, EDGE2
!HPF+$ INDEPENDENT, REDUCTION (Y), USE (SPECTRAL_PART(X,Y))
L:  DO I = 1, NEDGE
      ...
          Y(EDGE1(I)) = Y(EDGE1(I)) + F(X(EDGE1(I)),X(EDGE2(I)))
          Y(EDGE2(I)) = Y(EDGE2(I)) - F(X(EDGE1(I)),X(EDGE2(I)))
    END DO
```

Fig. 2. Code for Out-of-Core Unstructured Mesh in HPF Extension

HPF+ enables the user to specify a partitioning strategy either for all or a selected set of arrays. We illustrate this approach by the example in Fig. 2. The USE clause of the INDEPENDENT loop enables the programmer to select a partitioner from those provided in the environment (in the example, this is SPECTRAL_PART) and the arrays (in the example, X and Y) to which it is applied.

3 Compilation Strategies

The compilation process consists of four steps: (1) processing the input HPF+ program by the front end, program normalization, and initial program analysis, (2) basic restructuring, (3) optimizations, and (4) target code generation. In the first part of this section, we describe the basic restructuring strategy and then discuss several optimizing techniques.

3.1 Basic Parallelization Strategy

The out-of-core parallelization strategy is a natural extension of the inspector-executor approach. The set of iterations assigned to each processor by the work distributor is split into a number of *iteration tiles*. Sections of arrays referenced in each tile, called *data tiles*, are small enough to fit into local memory of that processor. The inspector-executor strategy is then applied to each iteration tile.

Generally, the code generated for an OOC data-parallel loop consists of four phases which are connected by control flow depicted in Fig. 3. In the following we describe briefly each phase[2].

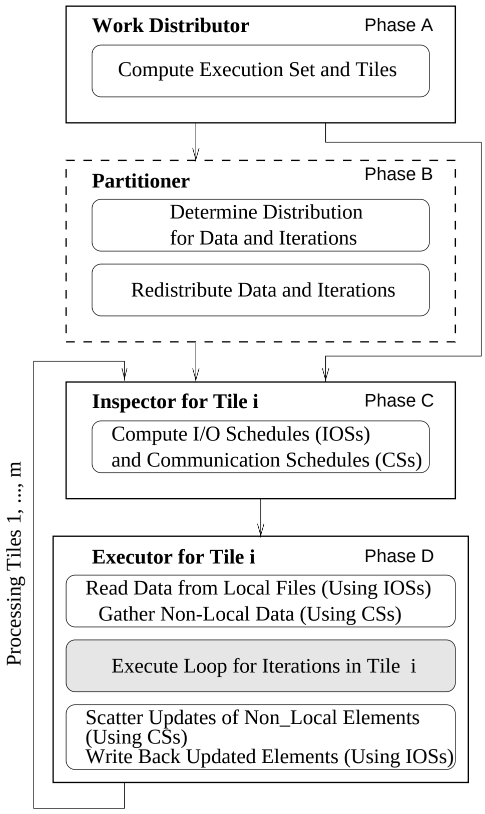
Work Distributor
Phase A
Compute Execution Set and Tiles
Partitioner
Phase B
Determine Distribution for Data and Iterations
Redistribute Data and Iterations
Inspector for Tile i
Phase C
Compute I/O Schedules (IOSs) and Communication Schedules (CSs)
Executor for Tile i
Phase D
Read Data from Local Files (Using IOSs) Gather Non-Local Data (Using CSs)
Execute Loop for Iterations in Tile i
Scatter Updates of Non_Local Elements (Using CSs) Write Back Updated Elements (Using IOSs)
Processing Tiles 1, ..., m

Fig. 3. Computing Phases

A. Default initial data and work distribution Initially, the iterations and data and indirection arrays are distributed among processors in block fashion. The distribution segment of each OOC array belonging to a processor is stored in a separate *local file* of this processor[3]. Next, an ICLA is allocated for each OOC indirection array. Moreover, the iteration set assigned to a processor, called the *execution set* of this processor, is blockwise split into a number of iteration tiles. The result of this operation depends on the shape of ICLAs allocated for the indirection arrays.

B. OOC Partitioning This optional phase involves the use of a disk-oriented partitioner which uses OOC data (for example, the node coordinates of the mesh) to determine a new data and work distribution. The data and indirection arrays are then redistributed. The redistribution results into the modification of the local files storing the corresponding distribution segments.

C. OOC Inspector This preprocessing phase results in computation of *I/O schedules* which describe the required I/O operations and *communication schedules* which describe the required communication. The inspector phase also computes the shape of ICLAs for the data arrays.

[2] This compilation strategy is based on the CHAOS+ runtime support.
[3] We consider the Local Placement Execution Model [6].

D. OOC Executor Once preprocessing is completed, we are in a position to carry out the necessary memory allocation, I/O, communication, and computation, following the plan established by the inspector.

A simplified compilation scheme can be applied to the OOC applications in which data arrays can fit in main memory, while indirection arrays are OOC.

3.2 Optimizations

We have proposed a set of optimizations which minimize the file access costs and other runtime overheads. Some of them are discussed below.

Redundancy elimination. Execution of the loop iteration tiles performed in the execution phase corresponds to the computation specified in the HPF program. On the other hand, other phases introduce additional runtime and memory overhead associated with the parallelization method. Reduction of this overhead is a crucial issue in the compiler development. The optimization techniques we apply to OOC problems are based upon a data flow framework called Partial Redundancy Elimination.

Hiding I/O latency. To overlap time expensive I/O accesses, the compiler can generate a code for two types of processes: *application processes* and *I/O processes.* The application processes implement the computation specified by the original program. The I/O processes serve the data requests of the application processes.

Eliminating extra file I/O by reordering computation. In order to minimize I/O and communication, it is important to update the value of a mesh node (see Fig. 2) for the maximum number of times before it is written back to the file. To achieve this goal the loop iterations and indirection arrays must be reorganized. Our approach [4] is based on a loop transformation called the *loop renumbering.*

4 Performance Results

This section presents the performance of the unstructured mesh linear equation solver GCCG which is a part of the program package FIRE [1]. The application was hand-coded using the methods described in Section 3. All the experiments were carried out on the Intel Paragon. 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 of this solver for different number of processors and tiles is given in Table 1. We compare the performance of the implementation which includes overlapping I/O with communication and computation with the solution in which the application process is blocked while waiting on I/O. Each I/O process was running on the same node as the corresponding application process. The experimental results show a big performance improvement obtained by the OOC versions against virtual memory. The performance was significantly improved by overlapping I/O with computation and communication.

360000 mesh elements						
number of processors	4	4	8	8	16	16
number of tiles	8	16	4	8	2	4
non-overlapping	207.5	257.8	136.9	196.5	109.7	184.3
overlapping	122.9	167.8	90.1	171.3	82.3	149.0
in-core with paging	627.3		328.5		290.3	

Table 1. Performance of the OOC GCCG Solver (time in seconds).

5 Related Work

Compiling OOC data-parallel programs is a relatively new topic and there has been little research in this area. Bordawekar, Choudhary, and Thakur [11] have worked on compiler methods for out-of-core HPF regular programs. Cormen and Colvin have worked on a compiler for out-of-core C*, called ViC* [8]. Paleczny, Kennedy, and Koelbel [10] propose a compiler support and programmer I/O directives which provide information to the compiler about data tiles for OOC regular programs.

6 Conclusions

The difficulty of efficiently handling out-of-core irregular problems limits the performance of distributed-memory multiprocessors. Since coding out-of-core version of an irregular problem might be a very difficult task and virtual memory does not perform well in irregular programs, there is a need for compiler-directed explicit I/O approach.

In this paper, we have presented a preliminary design for addressing these problems which is based on an HPF compilation system and the advanced runtime support CHAOS+. We have evaluated the effectiveness and feasibility of this approach on an out-of-core irregular kernel and compared its performance with the corresponding in-core versions supported by virtual memory. The results achieved are encouraging.

References

1. G. Bachler and R. Greimel. Parallel CFD in the Industrial Environment. *UNICOM Seminars*, London, 1994.
2. P. Brezany and V. Sipkova. Coupling Parallel Data and Work Partitioners to the Vienna Fortran Compilation System. In *Proceedings of the Conference EUROSIM – HPCN Challenges 1996*. North Holland, Elsevier, June 1996.
3. P. Brezany, A. Choudhary, and M. Dang. Parallelization of Irregular Out-of-Core Applications for Distributed-Memory Systems. *Proc. of HCPN 97, Vienna*, April 1997, Springer-Verlag, LNCS 1225.
4. P. Brezany, A. Choudhary, and M. Dang. Parallelization of Irregular Codes Including Out-of-Core Data and Index Arrays. In *Proceedings of the conference "Parallel Computing 1997 - PARCO'97"*, North Holland, Elsevier, April 1998.
5. B. M. Chapman, P. Mehrotra, and H. P. Zima. Extending HPF for advanced data parallel applications. *TR 94-7*, Univ. of Vienna, 1994.
6. A. Choudhary, et al. PASSION: Parallel and Scalable Software for Input-Output. *CRPC-TR94483*, Rice University, Houston, 1994.
7. A. Choudhary, C. Koelbel, and K. Kennedy. Preliminary Proposal to Provide Support for OOC Arrays in HPF. *Document of the HPFF*, Sep. 7, 1995.
8. T. H. Cormen and A. Colvin. ViC*: A Preprocessor for Virtual-Memory C*. TR: PCS-TR94-243, Dept. of Computer Science, Dartmouth College, Nov. 1994.
9. C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel Loops on Distributed Machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, pages 1097–1119, IEEE Comp. Soc. Press, April 1990.
10. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, McLean, VA*, pages 110-118, February 1995.
11. R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 382–391, Manchester, July 1994.
12. R. Ponnusamy, et al. A Manual for the CHAOS Runtime Library. *Technical Report*, University of Maryland, May 1994.