Parallel, structured and adaptive mesh refinement codes - Compiler support for easing the programmer's burden

Nikos Chrisochoides, Induprakas Kodukula, Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853.
{nikosc,prakas,pingali}@cs.cornell.edu

June 24, 1997

Abstract

To be written later..

Outline:

1) Problem domain 2) Library approach 3) Why library approaches are limited 4) Proposed compiler support 1) Automatic parallelization 2) Memory locality enhancement 3) Automatic load-balancing 5) Conclusions Phases of the solver

1 Introduction

In this white paper, we propose a compiler-based framework for Semi-structured methods such as adaptive mesh refinement and multigrid methods. It is difficult to implement these methods efficiently even on a sequential machine; parallelism adds an order of magnitude overhead to the complexity.

The computation in semi-structured methods is characterized by irregularly organized regular computations on the underlying data. The underlying data is specified as **grid components** which are organized in an irregular fashion in a **grid hierarchy** which itself changes dynamically. The computation in the application consists of stencil operations for relaxation on grid components, and interpolation and projection operations for transferring data between grid components at different levels of the grid hierarchy. The computation is regular (that is, data access functions are affine functions of surrounding loop indices). Parallelism is obtained by distributing the grid components in the hierarchy among various processors. However, this also leads to communication between various grid components. The volume and patterns of communication depend critically on the data distribution. The irregular and evolving nature of the application also leads to load imbalance, requiring redistribution of data at run time.

Even though the underlying computation is dense, HPF-like compiler technology is inadequate for AMR applications. There are two reasons for this:

- Block-cyclic distributions are inadequate to obtain proper load balance for these applications. Obtaining good load balance requires grid components to be distributed in complex ways among the processors, as explained below.
- The communication between various grid components is determined by the connectivity of the grid hierarchy. Due to the irregular nature of this connectivity, communication patterns are irregular.

Because of the inadequacy of present compiler technology, several libraries have been developed to make the application programmer's job easier. The most important of these are Multi-block PARTI [1], P++/AMR++ [3], LPARX [2], DAGH [7]. All these libraries hide the nature of the data distribution from the application programmer as much as possible. They provide constructs such as **forall** loops to enable application writing at a high level, and provide library calls to take care of parallelization issues. Each of these libraries makes certain assumptions regarding how the underlying data is distributed, which affects application performance. **Multi-block PARTI** [1] uses block-cyclic distributions as in HPF, but allows data arrays to be mapped to a subspace of all the computing processors. This works well for Multigrid codes, but for adaptive mesh refinement, the limitations of HPF apply here as well. **AMR++** [3] is an AMR class library layered on top of a parallel array library P++. AMR++ treats P++ as a black box and uses the distributions that P++ provides. These distributions reduce load-imbalance by allowing arrays to be distributed by columns of variable size, but are still limited in expressiveness. **LPARX** [2] allows data arrays to be distributed in irregularly shaped and irregularly sized blocks onto processors. While in principle it is possible in LPARX for multiple blocks of data to be assigned to one processor, in practice there is only one data block per processor. For adaptive mesh refinement, LPARX helps in reducing the communication between grid components at the same level, but the communication between grid components at different levels increases. Finally, **DAGH** [7] uses a space-filling curve enumeration to distribute the blocks of an array onto processors. Space filling curves ensure spatial locality, which means that this distribution policy reduces communication between grid components at different levels of the grid hierarchy.

For the rest of this document, we confine our attention to **single block** problems, where the underlying algorithm is an explicit method. While we expect to generalize our approaches to work for multi-block problems, as well as implicit schemes, that is not the focus of this paper.

2 Library approach

In this section, we describe the library philosophy of supporting adaptive mesh refinement. We point out several limitations that will server as the motivation for the rest of the paper.

3 Compiler and Run-time support

In this section, we describe the problems that we propose to solve using compiler technology that we have developed elsewhere. In the context of adaptive mesh refinement codes, we focus our attention on three different problems. We believe that support for automatic parallelization is essential to ease the programmer's burden in the development of adaptive mesh refinement codes. We have identified three problems that we believe to be crucial in this respect. We discuss each of these problems below.

3.1 Support for general distributions

As mentioned before, each of the various libraries support a wide variety of different data distributions. For example, DAGH chooses a data distribution that minimizes the communication between different levels of the grid hierarchy. LPARX on the other hand chooses to minimize the communication at a given level of the grid hierarchy. While specific choices of distributions are appropriate when the underlying library is being developed for a specific problem or hardware characteristics in mind, it does serve to limit the programmer's options. If a programmer wishes to make use of a different class of data distributions for whatever reason, they must reimplement their application using a different library which provides the necessary flexibility. This is all the more vexing as the underlying sequential algorithm in the program is the same, with only the data distribution changing from one problem instance to another.

Specifically, support for generalized distributions should provide the following:

- Distribution Specification
- Local storage allocation
- Communication Set generation
- Communication Scheduling
- Execution of local iterations

3.2 Memory management for deep hierarchies

Shared memory multiprocessors and clusters of these appear to be the latest stage in hardware evolution currently. Inside each multiprocessor, some bus-based cache-coherence mechanism is used. However, to overcome the limitations on scalability of the bus-based architecture, many smps are connected by a high speed interconnect to form clusters of symmetric multiprocessors. Each processor in these clusters can access data in its own cache, its private memory and memory from other smps as well. The entire memory in the system appears as a hierarchy to a processor. For program execution to be efficient, a processor must make good use of the memory hierarchy, and exploit locality of reference. This is especially crucial because as one goes from one level of the memory hierarchy to the next, the access time increases by a factor of 5-10 or even more. In addition, restructuring code to exploit memory locality is a non-trivial task, one made even more complex by the presence of parallelism. Clearly, compiler support is essential.

3.3 Support for load balancing

Adaptive mesh refinement codes have the fundamental property that their execution path is unknown and unpredicatble at compile time. In particular, the distribution of work and data to processors cannot be completely determined at compile time. As additional levels of refinement occur, the computation to be performed over different pieces of the grid hierarchy changes dynamically. Since the refinement can occur completely independently of the current distribution of the grid hierarchy, the underlying grid hierarchy needs to be redistributed to maintain good load balance. At the same time, the resulting change in communication patterns also needs to be kept in mind when redistributing the load. For the programmer to accomplish this task well adds an order of magnitude complexity to the initial programming effort, and it is desirable to minimize this effort using sophisticated compiler and run-time support.

4 Proposed Compiler support

In this paper, we suggest that compiler technology we have developed elsewhere can be adapted to provide solutions to each of the three problems we have mentioned before. Specifically, we believe that the **data-centric** approach [6] to compilation that we have developed in the context of locality enhancement for memory hierarchies is the key to our proposed compiler support.

4.1 Data-centric Transformations

The ultimate result of the orchestration is, of course, a transformed program with the desired data reuse, but to get that program, the tool would reason directly about the desired data flow rather than about the control flow of the program. A useful analogy is signal processing. The input and the output of signal processing is a signal that varies with time, and in principle, all processing can be done in the time domain. However, it is often more convenient to take a Fourier transform of the signal, work in the frequency domain and then take an inverse Fourier transform back into the time domain.

4.1.1 Data Shackle

In the rest of the paper, the phrase **statement instance** refers to the execution of a statement for given values of index variables of loops surrounding that statement.

Definition 1 *A* **data shackle** *is a specification in three parts.*

- *We choose a data object and divide it into* **blocks***.*
- *We determine a sequence in which these blocks are "touched" by the processor.*
- *For each block, we determine a set of statement instances to be performed when that block is touched by the processor. However, we leave unspecified the order of enumeration of statement instances within this set.*

We now look at this in detail.

- For the case of adaptive mesh refinement applications, we choose to divide up the underlying arrays into blocks using the fully general block distribution. With each array is an associated data structure called the **distribution descriptor** that keeps track of the blocks of the underlying array on each processor. More details on this can be found in [4].
- The second step is also specified with the distribution descriptor. We note that each block is completely specified by the indices of two diagonally opposite corners of the block, which can be thought of as an integer tuple. The distribution descriptor enumerates a series of such tuples, and the order of traversal of the blocks is the order in which the distribution descriptor enumerates the integer tuples.
- The final step is to specify the statement instances that should be performed when a block is scheduled. From each statement `S`, we choose a **single** reference `R` of the array that is being blocked. For now, we assume that an identical reference is chosen from every statement.

The rationale for the term "data shackle" should now be clear. One thinks of an instrument like a pantograph in which a master device visits the blocks of data in lexicographic order, while a slave device shackled to it is dragged along some trajectory in the iteration space of the program. It is also convenient to be able to refer to statement instances "shackled" to a particular block of data.

We note that the above sequence of steps merely constitutes a specification of the desired code to be executed. Details of how executable code can be produced is described in detail in [4, 6].

5 Data-centric Compilation support for AMR

In this section, we discuss the plan of attack for the problems related to adaptive mesh refinement mentioned earlier using the **data-centric** approach. We take each problem in turn, and demonstrate how the data-centric approach makes the programmer's job considerably easier.

5.1 Support for general Distributions

As mentioned earlier, one of the key problems to be solved when generating efficient code for adaptive mesh refinement, is to be able to generate parallel code in the presence of more complex distributions than currently available in HPF2 [5]. In previous work [4], we have demonstrated how to provide efficient solutions to the parallelization process in the presence of fully generalized block distributions. In particular, we have demonstrated efficient solutions to the following problems:

- Local storage allocation
- Communication Set generation
- Execution of local iterations

The steps in the adaptive mesh refinement solver consist of three distinct phases:

- Integration
- Prolongation of values from fine meshes to coarse meshes
- Restriction of values from coarse meshes to fins meshes

These computations consist of **do-all** loop nests, and affine array references. Since these loops are **do-all**, we do not address the problem of communication placement. We also do not directly address the complexity of specifying the fully general block distribution. However, it seems to be the case that the distributions used at the start of execution of an adaptive mesh refinemnt program are fairly simple. The full generality of the fully general block distribution is necessary only at later stages of the program execution, but at that stage, it is not directly visible to the programmer. The compiler (with the cooperation of an underlying run-time system) manipulates the complex forms of this distribution at run time with no intervention from the programmer.

5.2 Memory management for Deep Hierarchies

We developed the **data-centric** approach in our attempt to exploit locality in deep memory hierarchies for scientific programs [6]. AMR applications certainly fall in th class of programs that we can handle using our approach (since the programs have dense loops with affine array access functions). However, there are two additional sources of complexity for AMR applications.

- One is the presence of parallelism. While our previous work does not explicitly talk about parallelism, we believe that it can co-exist easily with parallelization techniques. We are exploring the approach of applying the memory-locality enhancement transformations on the SPMD version of the parallelized code (i.e. after parallelization has already been performed). This is work in progress.
- The second complexity arises because the grain of computation in AMR applications in the SPMD program depends on the size of the array block being computed on. This poses a problem for locality enhancing transformations. Computation on a large array block (which does not fit into cache, for instance), needs to be transformed for more levels of memory hierarchy than computation on a small block (which may fit into a small level of memory hierarchy). We believe that this is more of an engineering issue rather than a fundamental problem in the whole approach.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. On efficient runtime support for multiblock and multigrid applications: Regular section analysis. Technical Report CS-TR-3140 and UMIACS-TR-93-92, University of Maryland, Department of Computer Science and UMIACS, October 1993.

[2] Scott B. Baden, Scott R. Kohn, and Stephen J. Fink. Programming with lparx. Technical Report CS94-377, Department of Computer Science, University of San Diego, 1994.

[3] D Balsara, M Lemke, and D Quinlan. AMR++, a c++ object oriented class library for parallel adaptive refinement fluid dynamics applications. In **American Society of Mechanical Engineers, Winter Annual Meeting, Annaheim, CA**, volume 157, pages 413–433, 1992.

[4] Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali. Compiler and run-time support for semi-structured applications. In **International Conference on Supercomputing**. ACM SIGARCH, 1997.

[5] High Performance Fortran Forum. High performance fortran language specification, version 2, specification. http://www.crpc.rice.edu/HPFF/hpf2/index.html, October 1996.

[6] Induprakas Kodukula, Ahmed Nawaaz, and Keshav Pingali. Data-centric multi-level blocking. In **Programming Languages, Design and Implementation**. ACM SIGPLAN, 1997.

[7] Manish Parashar and James C. Browne. Distributed dynamic data-structures for parallel adaptive mesh refinement. In **HiPC**, 1995.