Compiler and Runtime Support for Programming in Adaptive Parallel Environments[1]

Guy Edjlali, Gagan Agrawal,
Alan Sussman, Jim Humphries,
and Joel Saltz
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742, USA
{edjlali,gagan,als,humphrie,saltz}@cs.umd.edu

Abstract

For better utilization of computing resources, it is important to consider parallel programming environments in which the number of available processors varies at runtime. In this paper, we discuss runtime support for data parallel programming in such an adaptive environment. Executing programs in an adaptive environment requires redistributing data when the number of processors changes, and also requires determining new loop bounds and communication patterns for the new set of processors. We have developed a runtime library to provide this support. We discuss how the runtime library can be used by compilers of HPF-like languages to generate code for an adaptive environment. We present performance results for a Navier-Stokes solver and a multigrid template run on a network of workstations and an IBM SP-2. Our experiments show that if the number of processors is not varied frequently, the cost of data redistribution is not significant compared to the time required for the actual computation. Overall, our work establishes the feasibility of compiling HPF for a network of non-dedicated workstations, which are likely to be an important resource for parallel programming in the future.

1 Introduction

In most existing parallel programming systems, each parallel program or job is assigned a fixed number of processors in a dedicated mode. Thus, the job is executed on a fixed number of processors, and its execution is not affected by other jobs on any of the processors. This simple model often results in relatively poor use of available resources. A more attractive model would be one in which a particular parallel program could use a large number of processors when no other job is waiting for resources, and use a smaller number of processors when other jobs need resources. Setia *et al.* [15, 20] have shown that such a dynamic scheduling policy results in better utilization of the available processors.

There has been an increasing trend toward using a network of workstations for parallel execution of programs. A workstation usually has an individual owner or small set of users who would like to have sole use of the machine at certain times. However, when the individual

[1]This work was supported by ARPA under contract No. NAG-1-1485 and by NSF under grant No. ASC 9213821. The authors assume all responsibility for the contents of the paper.

users of workstations are not logged in these workstations can be used for executing a parallel application. When the individual user of a workstation returns, the application must be adjusted either not to use the workstation at all or to use very few cycles on the workstation. The idea is that the individual user of the workstation does not want the execution of a large parallel application to slow down the processes he/she wants to execute.

We refer to a parallel programming environment in which the number of processors available for a given application varies with time as an *adaptive* parallel programming environment. The major difficulty in using an adaptive parallel programming environment is in developing applications for execution in such an environment. In this paper, we address this problem for distributed memory parallel machines and networks of workstations, neither of which support shared memory. In these machines, communication between processors has to be explicitly scheduled by a compiler or by the user.

A commonly used model for developing parallel applications is the data parallel programming model, in which parallelism is achieved by dividing large data sets between processors and having each processor work only on its local data. High Performance Fortran (HPF) [13], a language proposed by a consortium from industry and academia and being adopted by a number of vendors, targets the data parallel programming model. In compiling HPF programs for execution on distributed memory machines, two major tasks are dividing work or loop iterations across processors, and detecting, inserting and optimizing communication between processors. To the best of our knowledge, all existing work on compiling data parallel applications assumes that the number of processors available for execution does not vary at runtime [5, 12, 23]. If the number of processors varies at runtime, runtime routines need to be inserted for determining work partitioning and communication during the execution of the program.

We have developed a runtime library for developing data parallel applications for execution in an adaptive environment. There are two major issues in executing applications in an adaptive environment:

- Redistributing data when the number of available processors changes during the execution of the program and,
- Handling work distribution and communication detection, insertion and optimization when the number of processors on which a given parallel loop will be executed is not known at compile-time.

Executing a program in an adaptive environment can potentially incur a high overhead. If the number of available processors is varied frequently, then the cost of redistributing data can become significant. Since the number of available processors is not known at compile-time, work partitioning and communication need to be handled by runtime routines. This can result in a significant overhead if the runtime routines are not efficient or if the runtime analysis is applied too often.

Our runtime library, called Adaptive Multiblock PARTI (AMP), includes routines for handling the two tasks we have described. This runtime library can be used by compilers for data parallel languages or it can be used by a programmer parallelizing an application by hand. In this paper we describe our runtime library and also discuss how it can be used by a compiler. We restrict our work to data parallel languages in which parallelism is specified through parallel loop constructs, like forall statements and array expressions. We present experimental results on two applications parallelized for adaptive execution by inserting our runtime support by hand. Our experimental results show that if the number of available processors does not vary frequently, the cost of redistributing data is not significant as compared to the total execution time of the program. Overall, our work establishes the the feasibility of compiling HPF-like data parallel languages for a network of non-dedicated workstations.

The rest of this paper is organized as follows. In Section 2, we discuss the programming model and model of execution we are targeting. In Section 3, we describe the runtime library we have developed. We briefly discuss how this runtime library can be used by a compiler in Section 4. In Section 5 we present experimental results we obtained by using the library to parallelize two applications and running them on a network of workstations and an IBM SP-2. In Section 6, we compare our work with other efforts on similar problems. We conclude in Section 7.

2 Model for Adaptive Parallelism

In this section we discuss the programming model and model of program execution our runtime library targets. We call a parallel programming system in which the number of available processors varies during the execution of a program an *adaptive* programming environment. We refer to a program executed in such an environment as an *adaptive* program. These programs should adapt to changes in the number of available processors. The number of processors available to a parallel program changes when users log in or out of individual workstations, or when the load on processors change for various reasons (such as from other parallel jobs in the system). We refer to *remapping* as the activity of a program adjusting to the change in the number of available processors.

We have chosen our model of program execution with two main concerns:

- We want a model which is practical for developing and running common scientific and engineering applications and,

- We want to develop adaptive programs that are portable across many existing parallel programming systems. This implies that the adaptive programs and the runtime support developed for them should require minimal operating system support.

We restrict our work to parallel programs using the Single Program Multiple Data (SPMD) model of execution. In this model, the same program text is run on all the processors and

```
Real A(N,N), B(N,N)

Do Time_step = 1 to 100
        Forall( i = 1:N, j = 1:N)
                A(i,j) = B(j,i) + A(i,j)
        EndForall
        ...
        More Computation involving A & B ..
        ...
Enddo
```

Figure 1: Example of a Data-Parallel Program

parallelism is achieved by partitioning data structures (typically arrays) between processors. This model is frequently used for scientific and engineering applications, and most of the existing work on developing languages and compilers for programming parallel machines uses the SPMD model [13]. An example of a simple data parallel program that can be easily transformed into a parallel program that can be executed in SPMD mode is shown in Figure 1. The only change required to turn this program into an SPMD parallel program for a static environment would be to change the loop bounds of the forall loop appropriately so that each processor only executes on the part of array A that it owns and then to determine and place the communication between processors for array B.

We are targeting an environment in which a parallel program must adapt according to the system load. A program may be required to execute on a smaller number of processors because an individual user logs in on a workstation or because a new parallel job requires resources. Similarly, it may be desirable for a parallel program to execute on a larger number of processors because a user on a workstation has logged out or because another parallel job executing in the parallel system has finished. In such scenarios, it is acceptable if:

- The adaptive program does not remap immediately when the system load changes and,
- When the program remaps from a larger number of processors to a smaller number of processors, it may continue to use a small number of cycles on the processors it no longer uses for computation.

This kind of flexibility can significantly ease remapping of data parallel applications, with minimal operating system support. If an adaptive program has to be remapped from a larger number of processors to a smaller number of processors, this can be done by redistributing the distributed data so that processors which should no longer be executing the program do not own any part of the distributed data. The SPMD program will continue to execute on all processors. We refer to a process that owns distributed data as an *active process* and a process from which all data has been removed as a *skeleton process*. A processor owning an active process is referred to as an *active processor* and similarly, a processor owning a skeleton process is referred to as a *skeleton processor*. A skeleton processor will still execute each parallel loop in the program. However, after evaluating the local loop bounds to restrict execution to local data, a skeleton processor will determine that it does not need to execute any iterations of the parallel loop. All computations involving writing into scalar variables will continue to be executed on all processors. The parallel program will use some cycles in the skeleton processors, in the evaluation of loop bounds for parallel loops and in the computations involving writing into scalar variables. However, for data parallel applications involving large arrays this is not likely to cause any noticeable slowdown for other processes executing on the skeleton processors. This model substantially simplifies remapping when a skeleton processor again becomes available for executing the parallel program. A skeleton processor can be made active simply by redistributing the data so that this processor owns part of the distributed data. New processes do not need to be spawned when skeleton processors become available, hence no operating system support is required for remapping to start execution on a larger number of processors. In this model, a maximal possible set of processors is specified before starting execution of a program. The program text is executed on all these processors, though some of these may not own any portions of the distributed data at any given point in the program execution. We believe that this is not a limitation in practice, since the set of workstations or processors of a parallel machine that can possibly be used for running an application is usually known in advance.

In Figure 2 we have represented three different states of five processors (workstations) executing a parallel program using our model. In the initial state, the program data is spread across all five processors. In the second state, two users have logged in on processors 0 and 2, so the program data is remapped onto processors 1,3 and 4. After some time, those users log off and another user logs in on processor 1. The program adapts itself to this new configuration by remapping the program data onto processors 0,2,3 and 4.

If an adaptive program needs to be remapped while it is in the middle of a parallel, much effort may be required to ensure that all computations n restart at the correct point on all the processors after remapping. The main problem is ensuring that each iteration of the (parallel) loop is executed exactly once, either before or after the remapping. Keeping track of which loop iterations have been completed before the remapping, and only executing those that haven't already been completed after the remapping, can be expensive. However, if the program is allowed to execute for a short time after detecting that remapping needs to be done, the remapping can be substantially simplified. Therefore, in our model, the adaptive program is marked with *remap points*. These remap points can be specified by the programmer if the program is parallelized by hand, or may be determined by the compiler if the program is compiled from

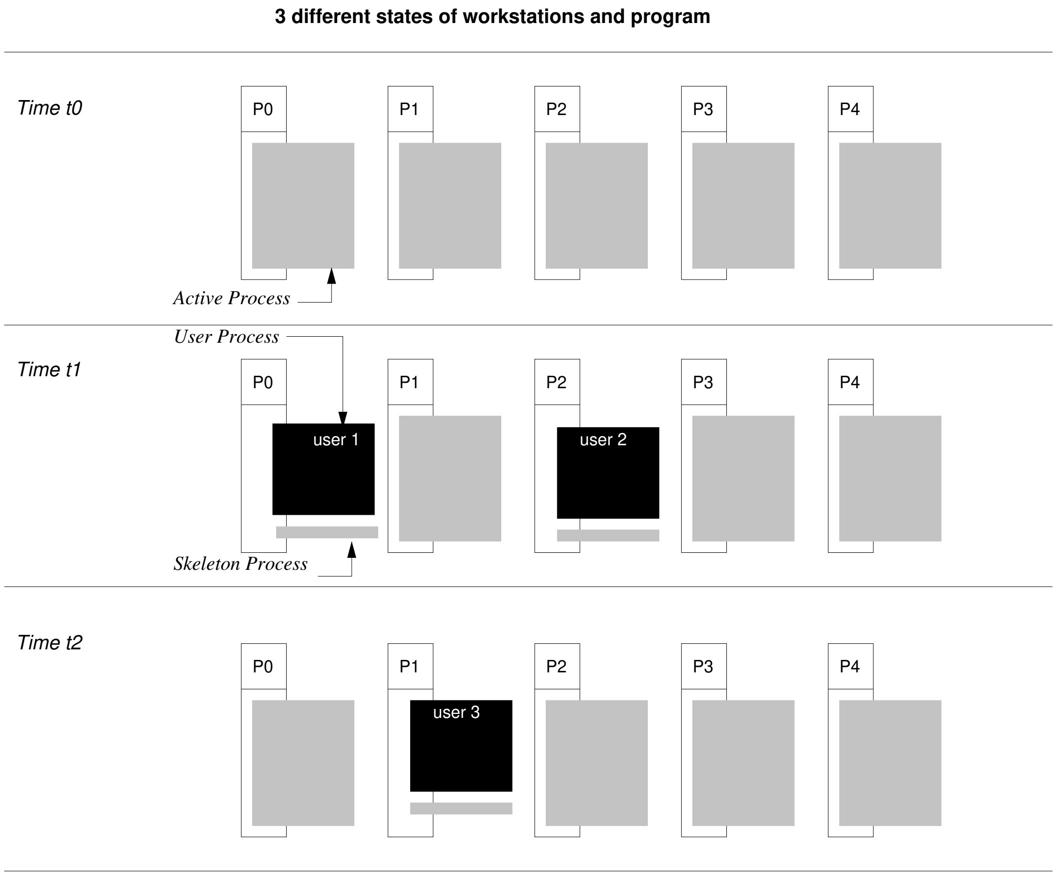

Figure 2: An Adaptive Programming Environment

a single program specification (e.g. using an HPF compiler). We allow remapping when the program is not executing a data parallel loop. The local loop bounds of a data parallel loop are likely to be modified when the data is redistributed, since a processor is not likely to own exactly the same data both before and after remapping. We will further discuss how the compiler can determine placement of remap points in Section 4.

At each remap point, the program must determine if there is a reason to remap. We assume a *detection* mechanism that determines if load needs to be shifted away from any of the processors which are currently active, or if any of the skeleton processors can be made active. This detection mechanism is the only operating system support our model assumes. All the processors synchronize at the remap point and, if the detection mechanism determines that remapping is required, data redistribution is done.

Two main considerations arise in choosing remap points. If the remap points are too far apart, that is if the program takes too much time between remap points, this may not be acceptable to the users of the machine(s). If remap points are too close together, the overhead of using the detection mechanism may start to become significant.

Our model for adaptive parallel programming is closest to the one presented by Proutty *et al.* [19]. They also consider data parallel programming in an adaptive environment, including a network of heterogeneous workstations. The main difference in their approach is that the responsibility for data repartitioning is given to the application programmer. We have concentrated on developing runtime support that can perform data repartitioning, work partitioning and communication after remapping. Our model satisfies the three requirements stated by Proutty *et al.*, namely withdrawal (the ability to withdraw computation from a processor within a reasonable time), expansion (the ability to expand into newly available processors) and redistribution (the ability to redistribute work onto a dynamic number of processors so that no processor becomes a bottleneck).

3 Runtime Support

In this section we discuss the runtime library we have developed for adaptive programs. The runtime library has been developed on top of an existing runtime library for structured and block structured applications. This library is called Multiblock PARTI [2, 21], since it was initially used to parallelize multiblock applications. We have developed our runtime support for adaptive parallelism on top of Multiblock PARTI because this runtime library provides much of the runtime support required for forall loops and array expressions in data parallel languages like HPF. This library was also integrated with the HPF/Fortran90D compiler developed at Syracuse University [1, 3, 5]. We discuss the functionality of the existing library and then present the extensions that were implemented to support adaptive parallelism. We refer to the new library, with extensions for adaptive parallelism, as Adaptive Multiblock PARTI (AMP).

3.1 Multiblock PARTI

This runtime library can be used in optimizing communication and partitioning work for HPF codes in which data distribution, loop bounds and/or strides are unknown at compile-time and indirection arrays are not used. Consider the problem of compiling a data parallel loop, such as a forall loop in HPF, for a distributed memory parallel machine or network of workstations. If all loop bounds and strides are known at compile-time and if all information about the data distribution is also known, then the compiler can perform work partitioning and can also determine the sets of data elements to be communicated between processors. However, if all this information is not known, then these tasks may not be possible to perform at compile-time. Work partitioning and communication generation become especially difficult if there are symbolic strides or if the data distribution is not known at compile-time. In such cases, runtime analysis can be used to determine work partitioning and generate communication. The Multiblock PARTI library has been developed for providing the required runtime analysis routines.

In summary, the runtime library has routines for three sets of tasks:

- Defining data distribution at runtime; this includes maintaining a distributed array descriptor (DAD) which can be used by communication generation and work partitioning routines.
- Performing communication when the data distribution, loop bounds and/or strides are unknown at compile-time and,
- Partitioning work (loop iterations) when data distribution, loop bounds and/or strides are unknown at compile-time.

A key consideration in using runtime routines for work partitioning and communication is to keep the overhead of runtime analysis low. For this reason, the runtime analysis routines must be efficient and it should be possible to reuse the results of runtime analysis whenever possible. In this runtime system, communication is performed in two phases. First, a subroutine is called to build a communication *schedule* that describes the required data motion, and then another subroutine is called to perform the data motion (sends and receives on a distributed memory parallel machine) using a previously built schedule. Such an arrangement allows a schedule to be used multiple times in an iterative code.

To illustrate the functionality of the runtime routines for communication analysis, consider a single statement *forall* loop as specified in HPF. This is a parallel loop in which loop bounds and strides associated with any loop variable cannot be functions of any other loop variable [13]. If there is only a single array on the right hand side, and all subscripts are affine functions of the loop variables, then this *forall* loop can be thought as copying a rectilinear section of data from the right hand side array into the left hand array, potentially involving changes of offsets and strides and index permutation. We refer to such communication as a regular section move [11]. The library includes a regular section move routine, *Regular_Section_Move_Sched*, that can analyze the communication associated with a copy from a right hand side array to left hand side array when data distribution, loop bounds and/or strides are not known at compile-time.

A regular section move routine can be invoked for analyzing the communication associated with any *forall* loop, but this may result in unnecessarily high runtime overheads for both execution time and memory usage. Communication resulting from loops in many real codes has much simpler features that make it easier and less time-consuming to analyze. For example, in many loops in mesh-based codes, only *ghost* (or *overlap*) cells [10] need to filled along certain dimension(s). If the data distribution is not known at compile-time, the analysis for communication can be much simpler if it is known that only *overlap* cells need to be filled. The Multiblock PARTI library includes a communication routine, *Overlap_Cell_Fill_Sched*, which computes a

```
Real *A, *B, *Temp
DAD *D                       DAD for A and B
SCHED *Sched

Num_Proc                =    Get_Number_of_Processors()
D                       =    Create_DAD(Num_Proc, ...)
Sched                   =    Compute_Transpose_Sched(D)
Lo_Bnd1                 =    Local_Lower_Bound(D,1)
Lo_Bnd2                 =    Local_Lower_Bound(D,2)
Up_Bnd1                 =    Local_Upper_Bound(D,1)
Up_Bnd2                 =    Local_Upper_Bound(D,2)

Do Time_step = 1 to 100
        Data_Move(B, Temp, Sched)
        Forall( i = Lo_Bnd1:Up_Bnd1,
                 j = Lo_Bnd2:Up_Bnd2)
                 A(i,j) = Temp(i,j) + A(i,j)
        EndForall
        ...
        More Computation involving A & B ..
        ...
Enddo
```

Figure 3: Example SPMD Program Using Multiblock PARTI

schedule that is used to direct the filling of overlap cells along a given dimension of a distributed array. The schedules produced by *Overlap_Cell_Fill_Sched* and *Regular_Section_Move_Sched* are employed by a routine called *Data_Move* that carries out both interprocessor communication (sends and receives) and intra-processor data copying.

The final form of support provided by the Multiblock PARTI library is to distribute loop iterations and transform global distributed arrays references into local references. In distributed memory compilation, the owner computes rule is often used for distributing loop iterations [12]. Owner computes means that a particular loop iteration is executed by the processor owning the left-hand side array element written into during that iteration. Two routines, *Local_Lower_Bound* and *Local_Upper_Bound*, are provided by the library for transforming loop bounds (returning, respectively, the local lower and upper bounds of a given dimension of the referenced distributed array) based upon the owner computes rule.

An example of using the library routines, to parallelize the program from Figure 1 is shown in Figure 3. The library routines are used for determining work partitioning (loop bounds) and for determining and optimizing communication between the processors. In this example, the data distribution is known only at runtime and therefore, the distributed array descriptor (DAD) is filled in at runtime. Work partitioning and communication is determined at runtime using the information stored in the DAD. The function *Compute_Transpose_Schedule()* is shorthand for a call to the *Regular_Section_Move_Sched* routine, with the parameters set to do a transpose for a two-dimensional distributed array. The schedule generated by this routine is then used by the *Data_Move* routine for transposing the array B and storing the result in the array Temp. Functions *Local_Lower_Bound* and *Local_Upper_Bound* are used to partition the data parallel loop across processors, using the DAD. The sizes of the arrays A, B and Temp on each processor depend upon the data distribution and are known only at runtime. Therefore, arrays A, B and Temp are allocated at runtime. The calls to the memory management routines are not shown in the figure. The code could be optimized further by writing specialized routines to perform the transpose operation, but the library routines are also applicable to more general forall loops.

The Multiblock PARTI library is currently implemented on the Intel iPSC/860 and Paragon, the Thinking Machines CM-5, the IBM SP1/2 and the PVM message passing environment for a network of workstations [8]. The design of the library is architecture independent and therefore it can be easily ported to any distributed memory parallel machine or any environment that supports message passing (e.g. Express).

3.2 Adaptive Multiblock PARTI

The existing functionality of the Multiblock PARTI library was useful for developing adaptive programs in several ways. If the number of processors on which a data parallel loop is to be executed is not known at compile-time, it is not possible for the compiler to analyze the communication, and in some cases, even the work partitioning. This holds true even if all other information, such as loop bounds and strides, is known at compile-time. Thus runtime routines are required for analyzing communication (and work partitioning) in a program written for adaptive execution, even if the same program written for static execution on a fixed number of processors did not require any runtime analysis.

Several extensions were required to the existing library to provide the required functionality for adaptive programs. When the set of processors on which the program executes changes at runtime, all active processors must obtain information about which processors are active and how the data is distributed across the set of active processors. To deal with only some of the processors being active at any time during execution of the adaptive program, the implementation of Adaptive Multiblock PARTI uses the notion of *physical numbering* and *logical numbering* of processors. If p is the number of processors that can possibly be active during the execution of the program, each such processor is assigned a unique physical processor number between 0 and $p-1$ before starting program execution. If we let c be the number of processors that are active at a given point during execution of a program, then each of these active processors is assigned a unique logical processor number between 0 and $c - 1$. The mapping between physical processor numbers and logical processor numbers, for active processors, is updated at remap points. The use of a logical processor numbering is similar in concept to the scheme used for processor groups in the Message Passing Interface Standard (MPI) [7].

Information about data distributions is available at each processor in the Distributed Array Descriptors (DADs). However, DADs only store the total size in each dimension for each distributed array. The exact part of the distributed array owned by an active processor can be determined using the logical processor number. Each processor maintains information about what physical processor corresponds to each logical processor number at any time. The mapping from logical processor number to physical processor is used for communicating data between processors.

In summary, the additional functionality implemented in AMP over that available in Multiblock PARTI is as follows:

- Routines for consistently updating the logical processor numbering when it has been detected that redistribution is required.
- Routines for redistributing data at remap points and,
- Modified communication analysis and data move routines to incorporate information about the logical processor numbering.

The communication required for redistributing data at a remap point depends upon the logical processor numberings before and after redistribution. Therefore, after it has been decided that remapping is required all processors must obtain the new logical processor numbering. The detection routine, after determining that data redistribution is required, decides upon a new logical processor numbering of the processors which will be active. The detection routine informs all the processors which were either active before remapping or will be active after remapping of the new logical numbering. It also informs the processors which will be active after remapping about the existing logical numbering (processors that are active both before and after remapping will already have this information). These processors need this information for determining what portions of the distributed arrays they will receive from which physical processors.

The communication analysis required for redistributing data was implemented by modifying the Multiblock PARTI *Regular_Section_Move_Sched* routine. The new routine takes both the new and old logical numbering as parameters. The analysis for determining the data to be sent by each processor is done using the new logical numbering (since data will be sent to processors with the new logical numbering) and the analysis for determining the data to be received is done using the old logical numbering (since data will be received from processors with the old logical numbering).

```
Compute Initial DAD, Sched and Loop Bounds

Do Time_step = 1 to 100
        If Detection() then Remap()
        Data_Move(B, Temp, Sched)
        Forall( i = Lo_Bnd1:Up_Bnd1,
                j = Lo_Bnd2:Up_Bnd2)
                A(i,j) = Temp(i,j) + A(i,j)
        EndForall
        ...
        More Computation involving A & B ..
        ...
Enddo

Remap()
        Real *New_A, *New_B

        New_NProc          =    Get_No_of_Proc_and_Numb()
        New_D              =    Create_DAD(New_NProc)
        Redistribute_Data(A, New_A, D, New_D)
        Redistribute_Data(B, New_B, D, New_D)
        D = New_D; A = New_A; B = New_B ;
        Sched              =    Compute_Transp_Sched(D)
        Lo_Bnd1            =    Local_Lower_Bound(D,1)
        Lo_Bnd2            =    Local_Lower_Bound(D,2)
        Up_Bnd1            =    Local_Upper_Bound(D,1)
        Up_Bnd2            =    Local_Upper_Bound(D,2)

End
```

Figure 4: Adaptive SPMD Program Using AMP

Modifications to the Multiblock PARTI communication functions were also required for incorporating information about logical processor numberings. This is because the data distribution information in a DAD only determines which logical processor owns what part of a distributed array. To actually perform communication, these functions must use the mapping between logical and physical processor numberings.

Figure 4 shows the example from Figure 3 parallelized using AMP. The only difference from the non-adaptive parallel program is the addition of the detection and remap calls at the beginning of the time step loop. The initial computation of the loop bounds and communication schedule are the same as in Figure 3. The remap point is the beginning of the time-step loop. If remapping is to be performed at this point, the function *Remap* is invoked. *Remap* determines the new logical processor numbering, after it is known what processors are available and creates a new Data Access Descriptor (DAD). The *Redistribute_Data* routine redistributes the arrays A and B, using both the old and new DADs. After redistribution, the old DAD can be discarded. The new communication schedule and loop bounds are determined using the new DAD. We have not shown the details of the memory allocation and deallocation for the data redistribution.

4 Compilation Issues

The examples shown previously illustrate how AMP can be used by application programmers to develop adaptive programs by hand. We now briefly describe the major issues in compiling programs written in an HPF-like data parallel programming language for an adaptive environment. We also discuss some issues in expressing adaptive programs in High Performance Fortran. As we stated earlier, our work is restricted to data parallel languages in which parallelism is specified explicitly. Incorporating adaptive parallelism in compilation systems in which parallelism is detected automatically [12] is beyond the scope of this paper.

In previous work, we successfully integrated the Multiblock PARTI library with a prototype Fortran90D/HPF compiler developed at Syracuse University [1, 3, 5]. Routines provided by the library were inserted for analyzing work partitioning and communication at runtime, whenever compile-time analysis was inadequate. This implementation can be extended to use Adaptive Multiblock PARTI and compile HPF programs for adaptive execution. The major issues in compiling a program for adaptive execution are determining remap points, inserting appropriate actions at remap points and ensuring reuse of the results of runtime analysis to minimize the cost of such analysis.

4.1 Remap Points

In our model of execution of adaptive programs, remapping is considered only at certain points in the program text. If our runtime library is to be used, a program cannot be remapped inside a data parallel loop. The reason is that the local loop bounds of a data parallel loop are determined based upon the current data distribution, and in general it is very difficult to ensure that all iterations of the parallel loop are executed by exactly one processor, either before or after remapping.

There are (at least) two possibilities for determining remap points. They may be specified by the programmer in the form of a directive, or they may be determined automatically by the compiler. For the data parallel language HPF, parallelism can only be explicitly specified through certain constructs (e.g.. forall statement, forall construct, independent statement [13]). Inside any of these constructs, the only functions that can be called are those explicitly marked as *pure* functions. Thus it is simple to determine, solely from the syntax, what points in the program are not inside any data parallel loop and therefore can be remap points. Making all such points remap points may, however, lead to a large number of remap points which may occur very frequently during program execution, and may lead to significant overhead from employing the detection mechanism (and synchronization of all processors at each remap point).

Alternatively, a programmer may specify certain points in the program to be remap points, through an explicit directive. This, however, makes adaptive execution less transparent to the programmer.

Once remap points are known to the compiler, it can insert calls to the detection mechanism at those points. The compiler also needs to insert a conditional based on the result of the detection mechanism, so that, if the detection mechanism determines that remapping needs to be done, then calls are made both for building new Distributed Array Descriptors and for redistributing the data as specified by the new DADs. The resulting code looks very similar to the code shown in the example from Section 3, except that the compiler will not explicitly regenerate schedules after a remap. The compiler generates schedules anywhere they will be needed, and relies on the runtime library to cache schedules that may be reused, as described in the next section.

4.2 Schedule Reuse in the Presence of Remapping

As we discussed in Section 3, a very important consideration in using runtime analysis is the ability to reuse the results of runtime analysis whenever possible. This is relatively straightforward if a program is parallelized by inserting the runtime routines by hand. When the runtime routines are automatically inserted by a compiler, an approach based upon additional runtime bookkeeping can be used. In this approach, all schedules generated are stored in hash tables by the runtime library, along with their input parameters. Whenever a call is made to generate a schedule, the input parameters specified for this call are matched against those for all existing schedules. If a match is found, the stored schedule is returned by the library. This approach was successfully used in the prototype HPF/Fortran90D compiler that used the Multiblock PARTI runtime library. Our previous experiments have shown that saving schedules in hash tables and searching for existing schedules results in less than 10% overhead, as compared to a hand implementation that reuses schedules optimally [1].

This approach easily extends to programs which include remapping. One of the parameters to the schedule call is the Distributed Array Descriptor(DAD). After remapping, a call for building a new DAD for each distributed array is inserted by the compiler. For the first execution of any parallel loop after remapping, no schedule having the new DADs as parameters will be available in the hash table. New schedules for communication will therefore be generated. The hash tables for storing schedules can also be cleared after remapping to reduce the amount of memory used by the library.

4.3 Relationship to HPF

In HPF, the *Processor* directive can be used to declare a processor arrangement. An intrinsic function, *Number_of_Processors*, is also available for determining the number of physical processors available at runtime. HPF allows the use of the intrinsic function *Number_of_Processors* in the specification of a processor arrangement. Therefore it is possible to write HPF programs in which the number of physical processors available is not known until runtime. The *Processor* directive can appear only in the specification part of a scoping unit (i.e. a subroutine or main program). There is no mechanism available for changing the number of processors at runtime.

To the best of our knowledge, existing work on compiling data parallel languages for distributed memory machines assumes a model in which the number of processors is statically known at compile-time [5, 12, 23]. Therefore, several components of our runtime library are also useful for compiling HPF programs in which a processor arrangement has been specified using the intrinsic function *Number_of_Processors*. HPF also allows *Redistribute* and *Realign* directives, which can be used to change the distribution of arrays at runtime. Our redistribution routines would be useful for implementing these directives in an HPF compiler.

5 Experimental Results

To study the performance of the runtime routines and to determine the feasibility of using an adaptive environment for data parallel programming, we have experimented with a multiblock Navier-Stokes solver template [22] and a multigrid template [17]. The multiblock template was extracted from a computational fluid dynamics application that solves the thin-layer Navier-Stokes equations over a 3D surface (multiblock TLNS3D). The sequential Fortran77 code was developed by Vatsa *et al.* at NASA Langley Research Center, and consists of nearly 18,000 lines of code. The multiblock template, which was designed to include portions of the entire code that are representative of the major computation and communication patterns of the original code, consists of nearly 2,000 lines of F77 code. The multigrid code we experimented with was developed by Overman *et al.* at NASA Langley. In earlier work, we hand parallelized these codes using Multiblock PARTI and also parallelized Fortran 90D versions of these codes

No. of Procs.	Time per Iteration	Cost of Remapping to			
		12 procs.	8 procs.	4 procs.	1 proc.
12	2213	-	3024	3740	6757
8	2480	3325	-	3715	9400
4	3242	2368	2755	-	6420
1	8244	2548	5698	5134	-

Figure 5: Cost of Remapping (in ms.): Multiblock code on Network of Workstations

using the prototype HPF/Fortran 90D compiler. In both these codes, the major computation is performed inside a (sequential) time-step loop. For each of the parallel loops in the major computational part of the code, the loop bounds and communication patterns do not change across iterations of the time-step loop when the code is run in a static environment. Thus communication schedules can be generated before the first iteration of the time-step loop and can be used for all time steps in a static environment.

We modified the hand parallelized versions of these codes to use the Adaptive Multiblock PARTI routines. For both these codes, we chose the beginning of an iteration of the time-step loop as the remapping point. If remapping is done, the data distribution changes and the schedules used for previous time steps can no longer be used. For our experiments, we used two parallel programming environments. The first was a network of workstations using PVM for message passing. We had up to 12 workstations available for our experiments. The second environment was a 16 processors IBM SP-2.

In demonstrating the feasibility of using an adaptive environment for parallel program execution, we considered the following factors:

- the time required for remapping and computing a new set of schedules, as compared to the time required for each iteration of the time-step loop,
- the number of time steps that the code must execute after remapping to a greater number of processors to effectively amortize the cost of remapping, and
- the effect of skeleton processes on the performance of their host processors.

On the network of Sun workstations, we considered executing the program on 12, 8, 4 or 1 workstations at any time. Remapping was possible from any of these configurations to any other configuration. We measured the time required for one iteration of the time-step loop and the

No. of Procs.	Time per Iteration	Cost of Remapping to				
		16 procs.	8 procs.	4 procs.	2 procs.	1 proc.
16	59.2	-	33	49	86	159
8	91.5	34	-	54	88	156
4	139.5	47	53	-	96	160
2	215.8	78	85	95	-	171
1	526.8	143	152	156	173	-

Figure 6: Cost of Remapping (in ms.): Multiblock code on IBM SP-2

No. of Procs.	Time per Iteration	Cost of Remapping to			
		8 procs.	4 procs.	2 procs.	1 proc.
8	93.9	-	14	20	36
4	134.4	18	-	22	29
2	206.6	19	23	-	29
1	308.4	33	33	36	-

Figure 7: Cost of Remapping (in ms.): Multigrid code on IBM SP-2

No. of Proc.	No. of Time-steps for Amortizing when remapped to			
	12 proc.	8 proc.	4 proc.	1 proc.
12	-	-	-	-
8	12.4	-	-	-
4	2.3	3.6	-	-
1	0.4	1.1	1.0	-

Figure 8: No. of Time-steps for Amortizing Cost of Remapping: Multiblock code on Network of Sun Workstations

cost of remapping from one configuration to another. The experiments were conducted at a time when none of the workstations had any other jobs executing. The time required per iteration for each configuration and the time required for remapping from one configuration to another are presented in the Figure 5. In this table, the second column shows the time per iteration, and columns 3 to 6 show the time for remapping to 12, 8, 4 and 1 processor configuration, respectively. The remapping cost includes the time required for redistributing the data and the time required for building a new set of communication schedules. The speed-up of the template is not very high because it has a high communication to computation ratio and communication using PVM is relatively slow. These results show that the time required for remapping for this application is at most the time required for 4 time steps.

Note that on a network of workstations connected by an Ethernet, it takes much longer to remap from a larger number of processors to a smaller number of processors than from a small number of processors to a large number of processors. e.g. the time required for remapping from 8 processors to 1 processor is significantly higher than the time required for remapping from 1 processor to 8 processors. This is because if several processors try to send messages simultaneously on an Ethernet, contention occurs and none of the messages may actually be sent, leading to significant delays overall. Instead, if a single processors is sending messages to many other processors, no such contention occurs.

We performed the same experiment on a 16 processor IBM SP-2. The results are shown are in Figure 6. The program could execute on either 16, 8, 4, 2 or 1 processors and we considered remapping from any of these configurations to any other configuration. The templates obtains significantly better speed-up and the time required for remapping is much smaller. The super-linear speed-up noticed in going from 1 processor to 2 processors because on 1 processor, all data cannot fit into the main memory of the machine. In Figure 7, we show the results from the multigrid template. Again, the remapping time for this routine is reasonably small.

Another interesting tradeoff occurs when additional processors become available for running the program. Running the program on a greater number of processors can reduce the time required for completing the execution of the program, but at the same time remapping the program onto a new set of processors causes additional overhead for moving data. A useful factor to determine is the number of iterations of the time-step loop that must still be executed so that it will be profitable to remap from fewer to a greater number of processors. Using the timings from Figure 5, we show the results in Figure 8. This figure shows that if the program will continue run for a several more time-steps, remapping from almost any configuration to any other larger configuration is likely to be profitable. Since the remapping times are even smaller on the SP-2, the number of iterations required for amortizing the cost of remapping will be even smaller.

In our model of adaptive parallel programming, a program is never completely removed from any processor. A skeleton process steals some cycles on the host processor, which can potentially slow down other processes that want to use the processor (e.g. a workstation user who has just logged in). The skeleton processes do not perform any communication and do not synchronize, except at the remap points. In our examples, the remap point is the beginning of an iteration of the time-step loop. We measured the time required per iteration on the skeleton processors. Our experiments show that the execution time on skeleton processers is always less than 10% of the execution time on active processers. For the multiblock code, the time required per iteration for the skeleton processors was 4.7 ms. and 30 ms. on the IBM SP-2 and Sun-4 workstations, respectively. The multigrid code took 11 ms. per iteration on the IBM SP-2. We expect, therefore, that a skeleton process will not slow down any other job run on that processor significantly (assuming that the skeleton process gets swapped out by the operating system when it reaches a remap point).

6 Related Work

In this section, we compare our approach to other efforts on similar problems.

Condor [14] is a system that supports transparent migration of a process (through check-pointing) from one workstation to another. It also performs detection to determine if the user of the workstation on which a process is being executed has returned, and also looks out for other idle workstations. However, this system does not support parallel programs; it considers only programs that will be executed on a single processor.

Several researchers have addressed the problem of using an adaptive environment for executing parallel programs. However, most of these consider a task parallel model or a master-slave model. In a version of PVM called Migratable PVM (MPVM) [6], a process or a task running on a machine can be migrated to other machines or processors. However, MPVM does not provide any mechanism for redistribution of data across the remaining processors when a data parallel program has to be withdrawn from one of the processors.

Another system called User Level Processes (ULP) [18] has also been developed. This system provides light-weight user level tasks. Each of these tasks can be migrated from one machine to another, but again, there is no way of achieving load-balance when a parallel program needs to be executed on a smaller number of processors. Piranha [9] is a system developed on top of Linda [4]. In this system, the application programmer has to write functions for adapting to a change in the number of available processors. Programs written in this system use a master-slave model and the master coordinates relocation of slaves. There is no clear way of writing data parallel applications for adaptive execution in all these systems.

Data Parallel C and its compilation system [16] have been designed for load balancing on a network of heterogeneous machines. The system requires continuous monitoring of the progress of the programs executing on each machine. Experimental results have shown that this involves a significant overhead, even when no load balancing is required [16].

7 Conclusions and Future Work

In this paper we have addressed the problem of developing applications for execution in an adaptive parallel programming environment, meaning an environment in which the number of processors available varies at runtime. We have defined a simple model for programming and program execution in such an environment. In the SPMD model supported by HPF, the same program text is run on all the processors, remapping a program to include or exclude processors only involves remapping the (parallel) data used in the program. The only operating system support required in our model is for detecting the availability (or lack of availability) of processors. This makes it easier to port applications developed using this model onto many parallel programming systems.

We have presented the features of Adaptive Multiblock PARTI, which provides runtime support that can be used for developing adaptive parallel programs. We described how the runtime library can be used by a compiler to compile programs written in HPF-like data parallel languages for adaptive execution. We have presented experimental results on a hand parallelized Navier-Stokes solver template and a multigrid template run on a network of workstations and an IBM SP-2. Our experimental results show that adaptive execution of a parallel program can be provided at relatively low cost, if the number of available processors does not vary frequently.

Acknowledgements

We would also like to thank V. Vatsa and M. Sanetrik at NASA Langley Research Center for providing access to the multiblock TLNS3D application code. We will also like to thank John van Rosendale at ICASE and Andrea Overman at NASA Langley for making their sequential and hand parallelized multigrid code available to us.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[2] Gagan Agrawal, Alan Sussman, and Joel Saltz. Efficient runtime support for parallelizing block structured applications. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 158–167. IEEE Computer Society Press, May 1994.

[3] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 1995. To appear. Also available as University of Maryland Technical Report CS-TR-3143 and UMIACS-TR-93-94.

[4] R. Bjornson. *Linda on Distributed Memory Multiprocessors.* PhD thesis, Yale University, 1991.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[6] Jeremy Casas, Ravi Konuru, Steve W. Otto, Robert Prouty, and Jonathan Walpole. Adaptive load migration systems for PVM. In *Proceedings Supercomputing '94*, pages 390–399. IEEE Computer Society Press, November 1994.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, Computer Science Dept., University of Tennessee, April 1994. Also appears in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994.

[8] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[9] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the Sixth International Conference on Supercomputing*, pages 417–427. ACM Press, July 1992.

[10] Michael Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[11] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[12] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[13] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[14] M.Litzkow and M.Solomon. Supporting checkpointing and process migration outside the Unix kernel. *Usenix Winter Conference*, 1992.

[15] Vijay K. Naik, Sanjeev Setia, and Mark Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings Supercomputing '93*, pages 824–833. IEEE Computer Society Press, November 1993.

[16] N.Nedeljkovic and M.J.Quinn. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience*, 5(4), 1993.

[17] Andrea Overman and John Van Rosendale. Mapping robust parallel multigrid algorithms to scalable memory architectures. In *Proceedings of 1993 Copper Mountain Conference on Multigrid Methods*, April 1993.

[18] R.Konuru, J.Casa, R.Prouty, and J.Walpole. A user-level process package for PVM. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 48–55. IEEE Computer Society Press, May 1994.

[19] R.Prouty, S.Otto, and J.Walpole. Adaptive execution of data parallel computations on networks of heterogeneous workstations. Technical Report CSE-94-012, Oregon Graduate Institute of Science and Technology, 1994.

[20] Sanjeev Setia. *Scheduling on Multiprogrammed Distributed Memory Parallel Machines*. PhD thesis, University of Maryland, Aug 1993.

[21] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.

[22] V.N. Vatsa, M.D. Sanetrik, and E.B. Parlette. Development of a flexible and efficient multigrid-based multiblock flow solver; AIAA-93-0677. In *Proceedings of the 31st Aerospace Sciences Meeting and Exhibit*, January 1993.

[23] Hans P. Zima and Barbara Mary Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993. In Special Section on Languages and Compilers for Parallel Machines.