RC 20665 (12/19/96)
Computer Science/Mathematics

IBM Research Report

Run-time Support for Dynamic Processor Allocation in HPF Programs

Samuel Midkiff, José E. Moreira, Vijay K. Naik

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

This page intentionally left blank.

Run-time Support for Dynamic Processor Allocation in HPF Programs

Samuel Midkiff — *midkiff@watson.ibm.com*
José E. Moreira — *moreira@watson.ibm.com*
Vijay K. Naik — *vkn@watson.ibm.com*

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

In this paper, we discuss a research prototype HPF compiler and a run-time system that allow dynamic processor allocation from HPF programs. Using dynamic processor allocation, an HPF program can be executed on a varying number of processors during a single run. This can improve both program efficiency and overall system utilization of the parallel system on which the program is executing. Performance data is presented to show that dynamic processor allocation in HPF programs can be supported efficiently.

This page intentionally left blank.

Run-time Support for Dynamic Processor Allocation in HPF Programs

Samuel Midkiff
midkiff@watson.ibm.com

José E. Moreira
moreira@watson.ibm.com

Vijay K. Naik
vkn@watson.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

In this paper, we discuss a research prototype HPF compiler and a run-time system that allow dynamic processor allocation from HPF programs. Using dynamic processor allocation, an HPF program can be executed on a varying number of processors during a single run. This can improve both program efficiency and overall system utilization of the parallel system on which the program is executing. Performance data is presented to show that dynamic processor allocation in HPF programs can be supported efficiently.

1 Introduction

Application-level dynamic processor management is becoming an important issue as large-scale parallel and distributed systems have emerged as cycle servers and throughput engines. Large complex applications are being developed to take advantage of multi-Gigaflop capabilities and, at the same time, dozens of users simultaneously share system resources. During the course of an application execution, there may be significant variations in the exploitable parallelism as well as in the computational and memory requirements. If a program runs for days or weeks, acquiring resources just in time and releasing them when not needed is an effective technique in improving the load balance and program efficiency. As resource management capabilities have become more sophisticated, facilities for application-level dynamic processor allocation are becoming available. This in turn is influencing parallel programming paradigms; for example, the recently released MPI-2 standard [7] allows for dynamic spawning of new processes. In this paper, we describe a technique for making the dynamic processor allocation capabilities available to programs based on High Performance Fortran (HPF) paradigm.

Informally, one of the goals of HPF is to allow programmers to write programs that are single threaded and maintain global name space. HPF programs provide sufficient information to the compiler so that parallel code can be automatically generated for architectures with distributed memory and nonshared address space. In general, the HPF programmer is relieved from the cumbersome task of scheduling computations onto processors and need not be concerned with the number of processors on which the program may be executed. Thus, at least in principle, only the compiler needs to be concerned with the concept of dynamic processor allocation. However, for performance reasons, and sometimes for eliminating any ambiguity in the parallel code, there are cases where the programmer needs to provide hints to the compiler regarding scheduling and controlling the distribution of computations among processors. The recently incorporated extensions to HPF [4] such as the directives `ON` and `TASK_REGION`, the intrinsic `ACTIVE_NUM_PROCS`, and the extrinsic `HPF_SERIAL` all serve as examples for the need for strong hints and advice from the programmer. In a similar vein, dynamic processor allocation decisions cannot always be left up to the compiler. Dynamic processor management requires some knowledge about the granularity of the change in parallelism, the longevity of the change (*i.e.*, whether the change would be sustained over an interval), and the cost effectiveness of processor allocation operations. The state-of-the-art for the compiler technology is such that programmer supplied hints are necessary for determining where and when in a program processors can be allocated and deallocated.

In this paper, we propose a set of extensions to HPF directives that allow programmers to specify points in a program where the number of processors allocated can be altered. We refer to such points in the program as *reconfiguration points* and our proposed extensions as the *processor reconfiguration directives*. We describe the semantics for these directives, explain how they relate to the existing directives, and discuss the restrictions on the use of these directives.

At a program specified reconfiguration point, the compiler generates code so that the run-time system can negotiate with the underlying resource management system for acquiring or releasing processors. The compiler also generates code so that, if there is a change in the number of processors, data is redistributed accordingly and the rest of the code executes correctly and efficiently. As a proof-of-principle, we have implemented compiler-level dynamic processor allocation operations based on program-level processor reconfiguration directives. We have also developed a run-time system that can perform the necessary operations for modifying the processor partition and the affected data distributions, and for updating any run-time descriptors that are functions of the total number of processors allocated to the program.

For the compiler implementation, we modified the *pHPF* compiler – a research prototype `HPF` compiler developed at IBM Research [3]. We also modified the run-time system associated with the pHPF compiler so it can interact, using an API for dynamic processor management, with the *Distributed Resource Management System (DRMS)*. DRMS is an advanced resource management system, also developed at IBM Research, that allows applications to acquire and release resources dynamically, on parallel platforms such as the IBM SP2. For this work, we made use of many of the low-level functions provided by the *ARTS* run-time system [8], which was developed to enable reconfiguring SPMD-style parallel programs on DRMS, and modified to support HPF programs[1].

In this paper, we discusses the approach we have taken to allow dynamic processor allocation in HPF programs, by enabling them to run under DRMS. We discuss some of the salient points of our implementation work. Finally, using two model programs, we present some performance data to test and benchmark dynamic processor allocation from HPF programs on an IBM SP2. Our results indicate that dynamic processor allocation and program reconfiguration can be accomplished in an efficient manner.

The rest of this paper is organized as follows. Section 2 presents the DRMS programming model and discusses how it applies to HPF. Section 3 discusses how the HPF compiler generates code according to this programming model. Section 4 consists of a performance evaluation of dynamic processor allocation in HPF programs. Section 5 discusses some related work, and, finally, Section 6 presents our conclusions.

2 The DRMS Programming Model

We support dynamic processor allocation in HPF programs through the `RESIZE` DRMS directive. The example subroutine of Figure 1 performs a Jacobi relaxation on an $n \times n$ grid `u`. At each iteration of the `do`-loop, one relaxation step is performed. A `RESIZE` directive is at the beginning of the loop body. DRMS directives are prefixed with `c$DRMS$`, and are ignored by regular HPF compilers.

The `RESIZE` directive in Figure 1 says that the application partition size can be changed at that point (the beginning of the iteration) and specifies a set of valid partition sizes on which the application can be executed. This particular `RESIZE` specifies a range of valid sizes, from a lower bound `lb` to an upper bound `ub` in multiplicative steps of `st` (*i.e.*, `lb`, `lb*st`, `lb*st*st`, ..., `ub`). The possible variations of `RESIZE` are shown in Table 1. Further information on the `RESIZE` directive is available in [8].

The actual allocation of physical processors at a `RESIZE` is performed by the DRMS resource control and scheduling system. Each time a `RESIZE` statement is executed, a query is made to test for a possible change in the number of processors. If the DRMS scheduler decides to *reconfigure* the application (*i.e*, change its processor partition) then a new processor partition (larger or smaller than the old one) is created, and

[1] *DRMS, ARTS, pHPF* and the *pHPF* run-time were designed and implemented at IBM T.J. Watson Research Center. Some development on the *pHPF* run-time was done at the IBM Tokyo Research Laboratory.

```
      subroutine JACOBI
      ...
      real*8, dimension(N,N) :: U, SOL
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: U, SOL
      ...
      do I = 1, M
! beginning of iteration is an SOP
c$DRMS$  RESIZE lb:ub:*st
! Perform the relaxation
         U(2:N-1,2:N-1) = H_SQ + 0.25*(U(1:N-2,2:N-1) +
                    U(3:N,2:N-1) + U(2:N-1,1:N-2) + U(2:N-1,3:N))
! Compute error (reduction operation)
         ERR = sum(abs(U(2:N-1,2:N-1) - SOL(2:N-1,2:N-1)))
      end do
      ...
      end
```

Figure 1: HPF program with dynamic processor allocation.

Table 1: Different forms of the `RESIZE` directive.

Resize form	Range of partition sizes specified
`RESIZE` l:u:+s	$(l, l+s, l+2s, \ldots, u)$
`RESIZE` l:u:*s	$(l, l \times s, l \times s^2, \ldots, u)$
`RESIZE` l:u:^s	$(i^s \mid l \leq i^s \leq u, i = 1, 2, \ldots)$
`RESIZE` $< l, v >$	$(v[1], v[2], \ldots, v[l])$

program execution continues on the new partition from the statement immediately following the `RESIZE`.

An HPF program with `RESIZE` directives follows the *DRMS programming model*. This model is based on the concepts of *schedulable and observable quanta* (SOQs) and *schedulable and observable points* (SOPs). Execution of a parallel application consists of the execution of a sequence of SOQs. Each SOQ executes on a fixed processor partition. The boundaries between SOQs are defined by SOPs which mark the transition from one SOQ to the next. Only at an SOP is the processor partition allocated to a job modified. This enables the following SOQ to execute on a different number of processors. We call an SOP that actually causes a change in the processor partition a *reconfiguration point*.

The global data set of the application, which is problem-specific, is preserved across an SOP. This enables computation to proceed smoothly from one SOQ to another. However, in many situations a change of processor partition encourages or even forces a change of mapping between the data and the address spaces of the participating processors. To support changes in the processor partition, an SOQ is organized into four sections: (i) the *resource section* specifies the processor requirements for execution of the SOQ; (ii) the *data section* specifies the mapping between the data set and the processor partition actually allocated to the SOQ; (iii) the *control section* specifies values for control variables that will dictate the flow of computation in each processor of the partition; (iv) finally, the *computation section* specifies the computation and communication operations performed by each processor.

HPF programs are well-suited for transformation and execution under the DRMS programming model. HPF programs are written assuming a single address space, and therefore the global data set is precisely defined. The DRMS `RESIZE` directive identifies SOPs and implements the resource section of an SOQ. It is the responsibility of a DRMS-enabled HPF compiler to then generate the appropriate data, control, and computation sections for the SOQ. These issues are discussed in the next section.

3 HPF Compiler Support for Dynamic Program Reconfiguration

In this section, we describe the enhancements we made to the `pHPF` compiler. The enhanced compiler generates code that allows the run-time system to interact with the DRMS resource management system for dynamically acquiring and releasing processors. The code generated also uses the run-time system to make the necessary changes to data and control distributions whenever there is a change in the number of processors allocated to the program. In the following discussion, we refer to the original compiler as the *pHPF compiler* and the modified version as the *drms-pHPF compiler*.

3.1 The `pHPF` compiler

The `pHPF` compiler implements a subset of full HPF, providing support for block and cyclic distributions, `LOCAL` procedures, and the full Fortran90 language. The compiler does not support the `REDISTRIBUTE` and `REALIGN` HPF directives, but supports redistribution and realignment at subroutine boundaries. At the subroutine boundary, if the distribution specified by the interface block is different than the current distribution, then the data is redistributed immediately before entering the subroutine, and immediately upon leaving the subroutine. Thus, within a given program unit, the distribution and alignment information is essentially static. This allows `pHPF` to do advanced optimizations without supporting the "dataflow" propagation of distribution information. (See [10] for details.) The `pHPF` compiler also supports a single multiprocessor grid per program unit.

`pHPF` efficiently support the execution of programs on processor partitions specified at load time. This is done by generating underlying SPMD code that has its control structure parameterized on the actual number of processors. If the number of processors, or other terms in the expression, are constants then normal constant-folding optimizations simplify the resulting code. Because distributions and alignments are static within a program unit, the `pHPF` compiler, when entering a routine, can compute various attributes about the distribution of arrays and the processor arrangement once, and use these values throughout the program unit. This control information includes: (i) the rank of the processor grid; (ii) the extent of each dimension of the processor grid; (iii) the upper and lower bounds of the local portion of each distributed array; and (iv) array dimension to processor grid dimension mapping information.

The pre-computation and caching of this information allows efficient execution of the body of the program unit. However, because the control information is a function of the number of processors, the caching scheme poses a problem when the number of processors allocated to a programs is changed dynamically at run-time. In the following, we describe how this problem is overcome in the `drms-pHPF` compiler.

3.2 The `drms-pHPF` compiler

We have enhanced the `pHPF` compiler to recognize the reconfiguration points specified by the `RESIZE` directives and to generate code that can take advantage of the dynamic processor allocation facilities offered by DRMS. For normal HPF related operations, the compiler generates calls to the standard `pHPF` run-time system. For processor reconfigurations related operations, the compiler generates calls to the `drms-pHPF` run-time system. The `drms-pHPF` compiler generates code that keeps the interactions with the two run-time systems transparent to each other. During program execution, when at a reconfiguration point, the program queries the `drms-pHPF` run-time system to check if the number of processors allocated to the program has changed. In case of a reconfiguration, the control is passed to the `drms-pHPF` run-time system which redistributes the data over the new set of processors. After that the control is passed back to the program and possibly to the `pHPF` run-time system. Control information is recomputed, to reflect the new number of processors. The new control information guarantees that computation is divided appropriately for the new processor configuration.

During the compilation phase, the compiler identifies the `RESIZE` construct, and transforms it to executable code as shown in Figure 2. This SPMD code runs on all processor in the partition assigned to the program. The call to the `drms-pHPF` run-time routine `drms_hpf_resize_mul()` queries the resource control system for a change in the number of processors. If there is a change, the program returns from the call already executing on the new number of processors, and the RESZD flag is set to **true**.

```
RESIZE l:u:*s      →      call drms_hpf_resize_mul(l,u,s,RESZD)
                          if (RESZD) {
                              call drms_hpf_pgrid_extents(pgrid_id, drms_pgrid_extents)
                              data section
                              control section
                          }
```

Figure 2: Stylized resize code generated by the compiler.

To complete the changes in program state when there is a reconfiguration, the body of the **if** is executed. The new shape of the processor grid, as it applies to the program unit, is obtained from the `drms-pHPF` run-time system trough the call to drms_hpf_pgrid_extents(). The code inside the **if** also includes the data and control sections of the new SOQ. The data section consists of redeclarations of distributed arrays to the `drms-pHPF` run-time system, which redistributes the data to conform to the new specification. The control section consists of a recomputation of the control information, as performed in the beginning of the program unit. Note that the computation code generated by the `drms-pHPF` compiler (outside the **if**) is not changed, since it is already parameterized on the control information.

3.3 Restrictions

To provide dynamic processor allocation facilities to HPF programs, and to make these reconfigurable, we have targeted the DRMS programming model and made use of the DRMS application run-time system. As a result, we have inherited some DRMS behavior into executable code generated by our compiler. The DRMS `RESIZE` operation, as defined in this paper, is a serializing operation: the SOQs delimited by the `RESIZE` have to be executed in the proper order, and all processors must participate in a reconfiguration operation. The impact of this behavior on HPF programs can be seen in Figure 3(a). Even if the operations a(i) = f(i) are independent, the loop cannot be parallelized because of the serializing effect of `RESIZE`. The loop would be compiled correctly (into sequential code) by our compiler, but a loss of performance would result.

```
do i = 1,n
    RESIZE l:u:*s
    a(i) = f(i)
end do
```

(a)

```
do j = 1,n,b
    RESIZE l:u:*s
    do i = j,j+b
        a(i) = f(i)
end do
```

(b)

Figure 3: Example of `RESIZE` inside a potentially parallel loop.

We do not see this behavior of `RESIZE` as a major problem. It is expected that resizes will most often be used between coarse-grain phases of a computation, when resource changes are necessary to efficiently utilize the system, rather than within a parallel phase. Thus, in the most common case, the situation described above will not arise. If a single parallel loop runs for a long time, it may be convenient or necessary to allow dynamic processor allocation during the execution of the loop. One possible solution for this situation is to strip-mine the loop, as shown in Figure 3(b). The single parallel loop is decomposed as a sequence of parallel inner loops. Each instance of the inner loop can be treated as an SOQ, and reconfiguration can be performed between instances.

4 Performance Evaluation

We conducted a set of experiments to evaluate the performance of our initial implementation of dynamic processor allocation for HPF. The experiments were performed on a 32-processor IBM SP2 with thin nodes. Each processing element of this SP2 is an IBM RS/6000 model 390 processor, with 64 kbyte data cache, 66.5 MHz clock speed, and 128 Mbyte of RAM. The processors are interconnected via a high-performance switch. Time intervals were measured, in our experiments, using a real-time clock with resolution better than 1 μs.

As benchmarks, we used the Jacobi relaxation code (Figure 1), with 1024×1024 and 2048×2048 grid sizes (JB1024 and JB2048). We also used an HPF version of `tomcatv`, with a problem size of 128 (TOM128). Each execution consisted of 200 *outer-loop* iterations, with all parallelism exploited within an iteration. A `RESIZE` statement was placed at the beginning of the loop body, and a reconfiguration was actually performed every 10 iterations. At least 40 sample points were taken for each reconfiguration operation, which required running each benchmark multiple times.

Our performance measurements focused on the *reconfiguration time*, the *redistribution time*, and the *impact on normal operation*. *Reconfiguration time* is the elapsed time from the moment an application reaches a reconfiguration point until the the application is ready to continue on the new processor partition. One component of the reconfiguration time is the *redistribution time*: the time it takes to rearrange the data so that computation can continue on the new processor partition. The other components of the reconfiguration time include the time to change the processor partition and the time to restart the application on the new processor partition. The details of these system-level operations are discussed in [8]. We also measure three individual components of the redistribution time directly: (i) the *compute time* is the time taken to compute the slices of arrays that have to be exchanged among processor, in order to perform the data rearrangement; (ii) the *buffer time* is the time taken in copying these slices to and from intermediate data buffers, local to each processor, used in performing the actual exchange; (iii) finally, the *message time* is the time for actual data exchange among processors using message-passing. To determine the impact of dynamic processor allocation on program execution we compare the *steady-state* performance of HPF programs with and without the `RESIZE` directive. The steady-state performance is measured as the iteration time when the application is running on a fixed processor partition, discarding any transients due to cold cache and TLB misses. Those transients occur on the first iteration of each application, and on the first iteration after a reconfiguration (because of data rearrangement).

Our results for reconfiguration, redistribution, and impact on normal operation are summarized in Table 2. Results are grouped by application and the "operation" column identifies the particular reconfiguration. The notation $P_1 \rightarrow P_2$ denotes a reconfiguration from P_1 (source) to P_2 (target) processors. Timing results are listed in the form of mean and standard deviation ($\mu \pm \sigma$) of our samples. The "reconfigure" and "redistribute" columns list the total reconfiguration time and redistribution component, in seconds. The "rate" column list the *redistribution rate* for the operation. The redistribution rate is defined as the actual amount of distributed data in the target partition, divided by the redistribution time, divided by the number of processors in the smallest (rate limiting) partition. The redistribution rate is in units of Mbyte *per* second *per* processor. The "reconfigure", "redistribute", and "rate" columns characterize the intrinsic cost of a reconfiguration. Columns "first", "drms", and "hpf" characterize the impact on normal operation. Column "first" lists the execution time of the first iteration after a reconfiguration, running on the target processor partition. Columns "drms" and "hpf" list the steady-state iteration execution times for the DRMS-HPF (t_{DRMS}) and vanilla HPF (t_{HPF}) versions of the application, when executed on the target processor partition. Iteration times are in milliseconds. The "%" column is computed as $100 \times (t_{\mathrm{DRMS}} - t_{\mathrm{HPF}})/t_{\mathrm{HPF}}$, and measures how much slower the DRMS-HPF version is compared to the vanilla HPF version.

Figure 4 shows a breakdown of redistribution times for selected reconfiguration operations. The notations $P_1 : P_2$ at the top of each vertical bar denotes a reconfiguration from P_1 to P_2 processors. The number at the top of each vertical bar represents the total time of each redistribution. The fraction of each component (compute, buffer, message) is indicated by different shadings, and the unaccounted time is grouped into the category "other".

Table 2: Results for reconfigurations under DRMS.

bench	operation	reconfigure (s) $\mu\pm\sigma$	redistribute (s) $\mu\pm\sigma$	rate MB$/s/$PE	first (ms) $\mu\pm\sigma$	drms (ms) $\mu\pm\sigma$	hpf (ms) $\mu\pm\sigma$	%
	4→8	2.86±0.29	0.76±0.04	8.06	83±19	46±5	45±4	2
	8→4	2.57±0.21	0.79±0.03	7.75	119±12	85±4	85±1	0
	8→16	4.41±1.24	0.52±0.02	5.91	61±10	27±7	26±4	4
JB1024	16→8	3.45±0.72	0.54±0.02	5.66	82±11	46±5	45±4	2
	8→32	8.47±2.54	0.55±0.02	5.76	55±10	18±3	16±2	10
	32→8	10.31±1.42	0.57±0.02	5.38	84±12	46±5	45±4	2
	16→32	10.20±3.32	0.51±0.02	3.07	54±11	18±3	16±2	10
	32→16	9.41±2.93	0.52±0.01	2.99	62±11	27±7	26±4	4
	4→8	4.77±0.17	2.58±0.06	9.42	208±17	160±9	159±8	1
	8→4	4.47±0.16	2.65±0.08	9.11	370±66	322±8	321±8	0
	8→16	4.69±0.53	1.48±0.08	8.25	126±14	87±10	86±8	1
JB2048	16→8	4.23±0.80	1.54±0.06	7.86	197±11	160±9	159±8	1
	8→32	11.12±2.37	1.43±0.05	8.60	87±16	50±9	48±10	3
	32→8	8.87±2.55	1.49±0.06	8.14	198±12	160±9	159±8	1
	16→32	12.56±2.12	1.04±0.04	5.90	89±11	50±9	48±10	3
	32→16	6.11±2.23	1.05±0.04	5.83	123±10	87±10	86±8	1
	4→8	4.43±1.00	0.25±0.01	1.12	433±11	424±11	424±11	0
	8→4	3.43±0.56	0.25±0.00	1.01	873±14	855±10	855±9	0
	8→16	4.68±0.93	0.39±0.01	0.43	240±50	220±11	218±10	1
TOM128	16→8	3.74±0.54	0.40±0.02	0.36	439±16	424±11	424±11	0
	8→32	9.39±2.72	0.53±0.02	0.42	150±44	123±12	122±13	1
	32→8	9.35±0.97	0.52±0.02	0.27	440±17	424±11	424±11	0
	16→32	10.83±2.70	0.71±0.03	0.16	137±26	123±12	122±13	1
	32→16	8.59±3.44	0.70±0.04	0.12	233±11	220±11	218±10	1

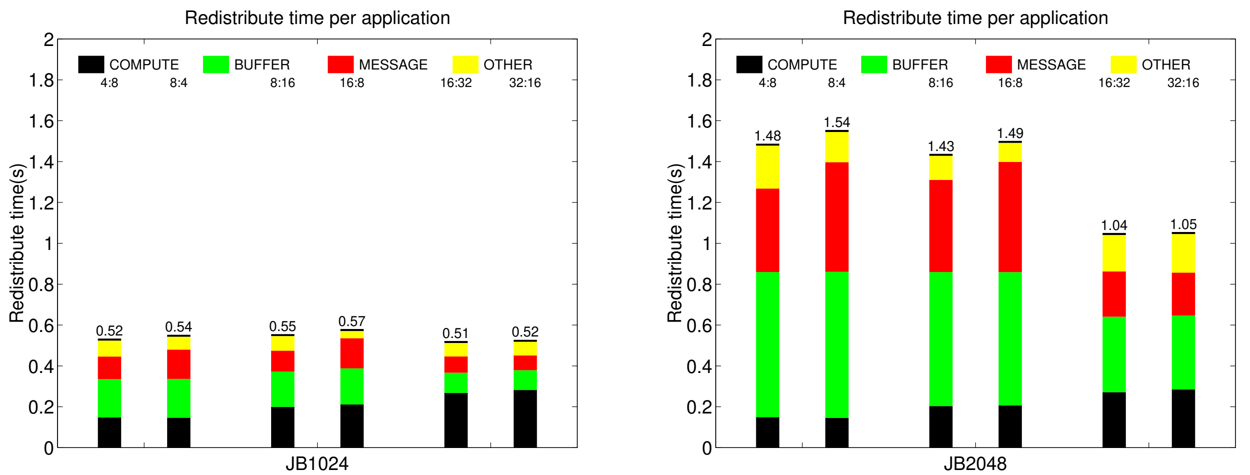

Figure 4: Components of redistribution times.

We note, from Table 2, that the steady-state performance of the DRMS-HPF and vanilla HPF versions are very similar. The DRMS-HPF version is usually $1ms$ slower per iteration, which is approximately the cost of a "null" `RESIZE` statement (*i.e.*, one that does not change the processor partition). Therefore, the time difference is the cost of querying the run-time system for a possible reconfiguration. The additional flexibility of dynamic processor allocation had no measurable effect on the optimization of the application computations. Where the $1ms$ cost of `RESIZE` is significant (e.g. JB1024 on 32 processors), the application code can be easily changed so that a `RESIZE` is performed only once every m iterations, where m is chosen appropriately to control the granularity of computation between `RESIZE`s. The first iteration after a `RESIZE` can execute significantly slower that in steady-state (JB1024 on $16 \rightarrow 32$ reconfiguration), but this penalty is a very small fraction of the total reconfiguration cost, and the relative penalty decreases as the problem size increases.

Most of the reconfiguration time is consumed by the system-related operations previously mentioned. The reconfiguration time grows significantly and becomes less stable (higher variance) when 32 processors are in either the source or target partition, indicating potential bottleneck in the scalability of our implementation. We note, however, that production codes can run for several hours on a large number of processors, and a reconfiguration time of $10s$ (or even a minute) is an insignificant part of these applications.

The redistribution time is a small part of the total reconfiguration time Except for the case of a large problem size on a small number of processors (JB2048 on $4 \rightarrow 8$ and $8 \rightarrow 4$ reconfiguration). The redistribution time is also very stable (low variance), because the data rearrangement is done in user mode and the processors involved are dedicated exclusively to the application. For any given problem size, the "rate" column shows that the redistribution rate drops as the number of processors increases (in either the source or target partition). Because there is less data per processor, there is less data to amortize the fixed cost, and the efficiency drops. The rates are particularly low for TOM128 because of its relatively small global data size (~ 1 Mbyte) as compared to JB1024 (~ 25 Mbyte) and JB2048 (~ 100 Mbyte).

More insight on the redistribution time can be gained from Figure 4. From the plots for JB1024 and JB2048 we note that the "compute" component does not depend on the problem size. This is expected because the computation of the slices that have to be exchanged depends only on the complexity of the data distributions and number of processors involved. This complexity grows as the number of processors involved increases, as can be observed from the plots. This behavior of the compute time can also create scalability problems, specially for small problem sizes on large number of processors (JB1024 on $16 \rightarrow 32$ and $32 \rightarrow 16$ reconfiguration). A possible solution is to "cache" the result of such computations, so that they only have to be don once in the lifetime of the application for each type of reconfiguration. Another solution is for the compiler, when possible, to perform these computation and store the results in the program. We also observe that the "buffer" and "message" components of redistribution exhibit a more scalable behavior. Their time increase with the problem size, as there are more data to move, and decrease as the partition size increases (note that the smaller of the target and source partitions limits the rate of redistribution).

5 Related Work

The importance of dynamic processor allocation as a way to improve application and system performance in multiprocessors has been recognized in [2, 6, 9, 11]. The effectiveness of dynamic processor allocation has been demonstrated for the DRMS environment in particular, with a load of reconfigurable SPMD applications, in [5].

The Adaptive Multiblock PARTI (AMP) system [1] supports dynamically reconfigurable applications, including HPF programs. In AMP an application is spawned on the maximum number of processors it can run. By appropriately redistributing data and computations, AMP can control the subset of processors actively participating in the execution. The other processors just follow the general control flow of the program and are, thus, more lightly loaded. Such an approach results in interference among applications in multi-user environments. This problem is particular severe in systems that do not support sharing processors among applications and in the case of gang-scheduling.

Execution of parallel applications on time-variant processor partitions can also be supported entirely at the system level. It is possible to time-multiplex virtual processors on top of physical processors, and then support reconfiguration by migrating the virtual processor. Such an approach is described in [6]. The disadvantage of such approaches, in general, are the need to multiplex and schedule the execution of many threads on one processor, and the negative impact on memory references caused by having each thread work on a small section of the global data space.

Demonstrating the growing importance of application-level dynamic processor management, the MPI-2 proposal [7] supports dynamic process creation and management. The facilities provided by MPI-2 are, in general, at a lower level of abstraction than the DRMS `RESIZE` directives, specially when applied to HPF programs. With MPI, the programmer is responsible for rearranging data and computations, if it desired to use a different number of processors in a phase of a computation. With DRMS-HPF, the user just has to acquire the proper number of processors and the rest is taken care by the compiler and run-time system.

6 Conclusions

Our previous work with reconfigurable SPMD applications has shown that dynamic processor allocation can have a significant beneficial impact on the performance of a parallel system. The ability to reconfigure HPF applications make a larger range of applications available for execution on systems that support dynamic processor allocation, such as DRMS. Because of the simpler structure of HPF programs relative to many SPMD programs, and because the HPF distribution information can be automatically passed on to the DRMS programs, we found the conversion of HPF programs to run under the DRMS environment to be straightforward.

We wish to emphasize that the benefits provided by DRMS – increased system utilization, increased flexibility in scheduling long running jobs, and more efficient use of processors by a single job – all are achieved with very little overhead on individual applications. The actual modifications to the run-time libraries and `pHPF` compiler were straightforward, and should not be an impediment to implementing similar functionality on other systems. Therefore, the approach described in this paper is efficient, straightforward to implement in system software, simple for the application writer to use, and provides good performance.

Finally, in this work we have focused on using features of the DRMS run-time system to support dynamic processor allocation in HPF programs. We would like to note that the DRMS run-time system can also be used to implement many of the HPF 2.0 approved extensions [4], including: `REDISTRIBUTE` executable directives, persistent remapping of `DYNAMIC` distributed arrays across subprogram interfaces, mapping of distributed arrays to processor subsets, `GEN_BLOCK` and `INDIRECT` distributions, and user declared shadow widths. The DRMS run-time system was designed to support these features in explicitly parallel SPMD codes. The same features can be supported in HPF programs by properly adapting the compiler to use the DRMS run-time system services.

Acknowledgements: This work is partially supported by NASA under the HPCCPT-1 Cooperative Research Agreement No. NCC2-9000.

References

[1] Edjlali, E., Agrawal, G., Sussman, A., and Saltz, J., *Data parallel programming in an adaptive environment*, in Proceedings of 9th International Parallel Processing Symposium, Santa Barbara, CA, April 1995.

[2] Gupta, A., Tucker, A., and Urushibara, S., *The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications*, in Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1991.

[3] Gupta, M., Midkiff, S., Schonberg. E., Seshadri, V., Shields, D., Wang, K.-Y., Ching, W.-M., and Ngo, T., *An HPF compiler for the IBM SP2*, in Proceedings of Supercomputing '95, IEEE Press, 1995.

[4] High Performance Fortran Forum, *High Performance Fortran Language Specifications*, October 1996. Version 2.0.δ.

[5] Konuru, R. B., Moreira, J. E., and Naik, V. K., *Application-assisted dynamic scheduling on large-scale multicomputer systems*, in Proceedings of Second International Euro-Par Conference (Euro-Par'96), Lyon, France, vol. 1124 of Lecture Notes in Computer Science, August 1996, pp. II:621–630.

[6] McCann, C., Vaswami, R., and Zahorjan, J., *A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*, ACM Transactions on Computer Systems, 11 (1993), pp. 146–178.

[7] MESSAGE PASSING INTERFACE FORUM, *MPI-2: Extensions to the Message-Passing Interface*, November 1996. DRAFT.

[8] Moreira, J. E., Naik, V. K., and Konuru, R. B., *Designing reconfigurable data-parallel applications for scalable parallel computing platforms*, Tech. Rep. RC 20455, IBM Research Division, May 1996. Submitted for publication.

[9] Naik, V. K., Setia, S. K., and Squillante, M. S., *Processor allocation in multiprogrammed, distributed-memory parallel computer systems*, Tech. Rep. RC 20239, IBM Research Division, October 1995. To appear in Journal of Parallel and Distributed Computing.

[10] Palermo, D. J., *Compiler techniques for optimizing communication and data distributi on for distributed-memory multicomputers*, PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[11] Polychronopoulos, C., *Multiprocessing versus multiprogramming*, in Proceedings of the 1989 International Conference on Parallel Processing, vol. II, August 8-12 1989, pp. 223–230.