Data Distribution Support on Distributed Shared Memory Multiprocessors

Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic
Silicon Graphics Computer Systems
Mountain View, CA

Jennifer M. Anderson
Digital Western Research Lab
Palo Alto, CA

Abstract

Cache-coherent multiprocessors with distributed shared memory are becoming increasingly popular for parallel computing. However, obtaining high performance on these machines requires that an application execute with good data locality. In addition to making effective use of caches, it is often necessary to distribute data structures across the local memories of the processing nodes, thereby reducing the latency of cache misses.

We have designed a set of abstractions for performing data distribution in the context of explicitly parallel programs and implemented them within the SGI MIPSpro compiler system. Our system incorporates many unique features to enhance both programmability and performance. We address the former by providing a very simple programming model with extensive support for error detection. Regarding performance, we carefully design the user abstractions with the underlying compiler optimizations in mind, we incorporate several optimization techniques to generate efficient code for accessing distributed data, and we provide a tight integration of these techniques with other optimizations within the compiler. Our initial experience suggests that the directives are easy to use and can yield substantial performance gains, in some cases by as much as a factor of 3 over the same codes without distribution.

1 Introduction

Cache-coherent shared memory multiprocessors are attractive for parallel programming since they provide a uniform view of memory, with inter-processor communication specified implicitly through load and store operations to shared memory locations. To enable scaling beyond bus-based machines, modern multiprocessors typically contain a large number of processing nodes each with one or more processors and a portion of main memory connected through a scalable interconnection network. This class of machines is termed *CC-NUMA* (cache-coherent non-uniform memory access) and includes several commercially available systems, such as Convex Exemplar, Sequent STiNG, and SGI Origin-2000.

Although global memory is uniformly accessible by all the processors, remote memory latencies are typically much larger than local memory latencies (e.g., 2-3 times on the Origin-2000). While processor caches can exploit temporal locality on both local and remote data, many applications, such as those without temporal reuse or with working sets larger than the cache, are unable to benefit from cache locality alone. To obtain high performance on such applications, it is often necessary to distribute the data structures in the program so that the cache misses of each processor are more likely to be satisfied from local rather than remote memory.

In this paper we describe a set of data distribution abstractions for CC-NUMA multiprocessors. We have designed these abstractions as a set of directives that allow the programmer to manually control the distribution of array data structures in explicitly parallel programs. We provide a small set of abstractions that are easy to use, yet expressive enough for real applications. Our directives are integrated with existing mechanisms for exploiting loop-level parallelism. Furthermore, the directives are designed keeping in mind the compiler's ability to generate efficient code for accesses to distributed data. Taken together, our abstractions enable the programmer to exploit loop-level parallelism and exercise fine control over both data distribution and computation scheduling. We have implemented these directives in the SGI MIPSpro7.1 commercial compiler system targeting the Origin-2000 multiprocessor.

The primary concern when distributing data on CC-NUMA architectures is that physical placement of data must be performed in units of an operating system page (16 KB on the Origin-2000). This can lead to false sharing at the page level when multiple data items that we wish to place in local memories of distinct processors happen to lie within the same page. In such situations it becomes necessary to move data objects within the virtual address space of the process so that they now belong to distinct pages. However, moving data within the virtual address space of a process is not always legal and must be done carefully to be correct. Performing these transformations safely and efficiently is a key component of our approach.

The main contributions of this paper are the following. We present a set of abstractions for controlling loop scheduling and data distribution on CC-NUMA architectures, and describe the implementation of these abstractions within a production compiler. The unique aspects of our system include extensive error-detection features, support for separate compilation across multiple files, optimization techniques to generate efficient code for distributed arrays, and tight integration of data distribution optimizations with other loop-level optimizations in the compiler.

The rest of the paper is organized as follows. Section 2 provides an overview of the Origin-2000 architecture. Section 3 contains a

PLDI '97 Las Vegas, NV, USA

detailed description of our directives for parallelism and data distribution. In Section 4 we describe the implementation of these directives within the compiler and the runtime system. Section 5 describes our scheme for automatically propagating distribution directives across subroutine calls. Section 6 outlines our support for error detection. Section 7 presents the optimizations performed by the compiler to efficiently support these constructs. We present performance results on some applications in Section 8, discuss related work in Section 9, and offer concluding remarks in Section 10.

2 Overview of the Origin-2000

Figure 1 shows the high-level architecture of the Origin-2000 [LL97]. Two 195 MHz MIPS R10000 processors together with a portion of the shared memory are connected through a hub, and multiple such nodes are connected together in a hypercube through a switch-based interconnect.. Each processor has separate on-chip instruction and data caches (32KB each, 32-byte line size), and a unified off-chip cache (typically 1-4MB, 128-byte line size), all two-way set associative. The hub maintains cache coherence across processors using a directory-based invalidation protocol. The latency of a miss to local memory is about 70 processor cycles, while a miss to the remote memory of another processor ranges from 110 to 180 cycles.

The operating system on the Origin-2000 supports data allocation at the granularity of a physical page (16Kbytes). It provides a default first-touch page allocation policy where a page is allocated from the local memory of the processor incurring the page fault, as well as an optional round-robin policy where pages are allocated in a round-robin fashion across processors.

3 Programming Model

In this section we provide a detailed description of our programming model [SGI96]. We first give a brief overview of the directives to exploit loop-level parallelism. We then describe our data distribution directives, with particular emphasis on our approach to overcoming the page granularity limitations. Finally, we describe the mechanisms to control the scheduling of parallel loops.

3.1 Existing Directives for Loop Parallelization

The code segment below illustrates our commonly used directive for specifying parallelism.

```
c$doacross local(i) shared(n, A)
do i = 1, n
   A(i) = 2*i
enddo
```

The *doacross* directive specifies that all the iterations of the *i* loop can be executed concurrently. The *local* and *shared* clauses specify that each iteration should have a local instance of the variable *i*, while the variables *A* and *n* should be shared across all the iterations and can be accessed directly through shared memory references. It is assumed that all iterations are fully concurrent with an implicit barrier at the end of the *doacross* loop. The

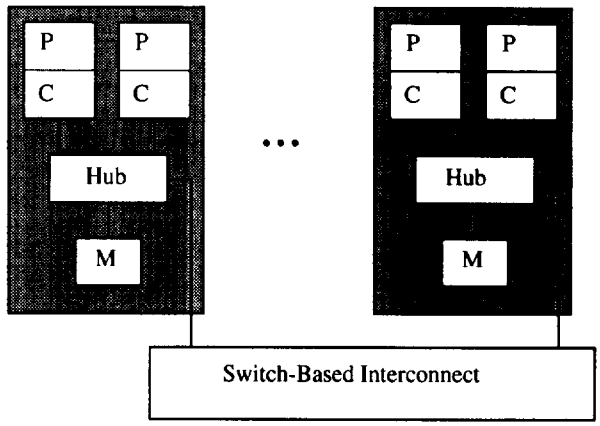


Figure 1. Origin-2000 Architecture Overview.

partitioning of iterations across processors may be controlled with an optional *schedtype* clause, and additional synchronization, if required, must be explicitly specified by the programmer using constructs such as locks and barriers.

We also support the parallel execution of nested loops through the *nest* clause on a *doacross* directive. For instance, in the following example all iterations in the *(i,j)* iteration space can be executed concurrently.

```
c$doacross nest (i,j) local (i,j) shared (m,n,B)
do i = 1, n
   do j = 1,m
      B(j,i) = i+j
   enddo
enddo
```

3.2 Extensions for Data Distribution

Our data distribution support focuses on regular distributions of arrays in Fortran. Our directives are similar to the basic data distribution directive in HPF (High Performance Fortran) [Lov93, KLS+94, HPF2-97], as shown below.

```
real*8 A(m, n, . . .)
c$distribute A (<dist>, <dist>, . . .)
```

<*dist*> may be one of *block, cyclic, cyclic(<expr>)*, or *, with the same meaning as in HPF.

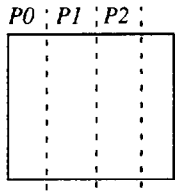
Distribution may be specified for both global and local arrays, including dynamically sized local arrays. There are no restrictions on the size or the assigned number of processors for an array dimension. The number of processors in each distributed dimension is determined at program start-up time, which enables the same executable to run with different number of processors. Furthermore, *distribute* directive can contain an optional *onto* clause specifying how the total number of processors should be assigned across multiple distributed array dimensions.

We consider two possible ways of supporting array distribution. The simple approach attempts to map each processor's portion of the distributed array onto physical pages allocated from within the local memory of that processor (*regular* distribution). This

approach is simple and easy to implement, but is limited by the underlying page granularity. The aggressive approach reorganizes the layout of the array within the virtual address space of the process, thereby overcoming the page-granularity limitations and guaranteeing the desired distribution (*reshaped* distribution). However, changing the layout of an array is not always legal since it may violate storage layout assumptions in the program. Furthermore, it can incur additional array addressing overhead.

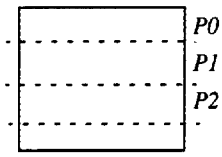
The choice between these two implementations depends on the size and layout of individual portions of the distributed array and can vary for different arrays within the same program. For instance, in the example

```
real*8 A (1000, 1000)
c$distribute A (*, block)
```



an individual portion is a single contiguous piece of size $8*10^6/P$ bytes, assuming Fortran style column-major layout and P processors. Since an individual portion may be much larger than a page, regular distribution may be sufficient for this array. On the other hand, in the example

```
real*8 A (1000, 1000)
c$distribute A (block, *)
```



an individual portion is still $8*10^6/P$ bytes, but due to the column-major layout of the array each contiguous piece is only $8*10^3/P$ bytes, significantly less than a page. In such situations reshaped data distribution is desirable.

As we can see, the choice between regular and reshaped distribution may depend on array bounds and the number of processors, which are typically symbolic values not known at compile time. Rather than leave the implementation choice to the compiler, we allow the programmer to choose between regular and reshaped distributions using the directives

```
c$distribute
c$distribute_reshape
```

A program can contain both *distribute* and *distribute_reshape* arrays. A particular array, however, must be declared either *distribute* or *distribute_reshape* (or neither, of course) for the duration of the program, and cannot be dynamically switched between the two kinds of distributions.

3.2.1 Restrictions on Data Reshaping

Array reshaping is legal only under certain conditions related to storage and sequence assumptions about that array and to passing arrays as subroutine arguments.

First, a reshaped array cannot be equivalenced to another array, either explicitly through an *equivalence* statement or implicitly through multiple declarations of a *common* block. If an array in a *common* block is reshaped, then each declaration of that *common* block must (a) contain an array at the same offset within the *common* block, (b) declare that array with the same number of dimensions, each of the same size, and (c) specify the same reshaped distribution for the array.

The second restriction concerns passing a reshaped array as an argument to a subroutine. If the entire array is passed as an argument, then the number of dimensions and the size of each dimension in the actual and the formal parameter must match exactly. On the other hand, passing an element of a reshaped array[1] is treated as passing a portion of a distributed array (the size and shape of the portion depend on the array distribution, as illustrated by the example below). The called procedure treats the incoming parameter as a non-distributed (i.e., standard Fortran) array. The declared bounds on the formal parameter are required not to exceed the size of the distributed array portion passed in as the actual argument.

For instance, in the example

```
real*8 A(1000)
c$distribute_reshape A (cyclic(5))
do i=1,1000,5
   call mysub (A(i))
enddo
end
subroutine mysub (X)
real*8 X(5)
• • •
end
```

the main program calls subroutine *mysub* once for each portion (5 elements) of the reshaped array *A*. The declared size of the formal parameter *X* in *mysub* can therefore be at most 5 elements.

Given these restrictions, it can be difficult for the programmer to correctly reshape arrays in a large application. We make this task easier in multiple ways. First, the programmer does not need to specify reshaped distributions on formal parameters of subroutines; *distribute_reshape* directives need to be supplied only at array definition points, and the compiler automatically propagates them down a call chain (Section 5). Second, we provide extensive error-detection support (both compile-time and runtime) to enforce the consistency of reshaped arrays (Section 6). Finally, we provide a rich set of intrinsics for traversing the individual portions of a distributed array [SGI96]. Taken together, these features make it significantly easier to distribute data in large applications.

3.3 Dynamic Data Redistribution

Dynamic data redistribution may be useful when an application needs a different distribution on the same array in two distinct phases of the program. We therefore provide the following *redistribute* directive

---

1. We treat call sub(A) as passing the entire array A, and call sub(A(0)) and call sub (A(i)) as passing an element of A to sub.

Original loop (with affinity for a distributed array of size n):

*c$doacross affinity (i) = data(A(s*i+c))*
do i=LB, UB, step

Block distribution: ($b = \left\lceil \frac{n}{P} \right\rceil$)

do p = 0, P-1
do i = $max(LB, \left\lceil \frac{pb-c}{s} \right\rceil)$, $min(UB, \left\lfloor \frac{(p+1)b-c-1}{s} \right\rfloor)$, *step*

Cyclic distribution: (s=1. the expressions for s>1 are omitted for brevity)

do p = 0, P-1
do i = $LB + ((p - LB - c) \bmod P)$, *UB, P*

Block-cyclic: (cyclic(k))

do p = 0, P-1
do j = $\left\lceil \frac{(LB)s+c+1}{kP} \right\rceil - 1, \left\lceil \frac{(UB)s+c+1}{kP} \right\rceil - 1$
do i = $max(LB, \left\lceil \frac{pk-c+kPj}{s} \right\rceil)$, $min(UB, \left\lfloor \frac{(p+1)k-c-1+kPj}{s} \right\rfloor)$, *step*

Figure 2. Implementing Affinity Scheduling.

c$redistribute A (<dist>, <dist>, . . .)

where *<dist>* may be one of *block, cyclic, cyclic(<expr>)*, or *. Redistribute is an executable statement that has global effect. We do not allow redistribution of reshaped arrays - as discussed later, dynamic reshaping severely restricts compiler optimizations of reshaped data distributions and can result in inefficient code.

3.4 Affinity Scheduling

Along with data distribution we allow the user to control the scheduling of the iterations of a parallel loop across processors through an optional affinity clause on a *doacross* directive (see Section 3.1).

```
real*8 A(n)
c$distribute A(block)
c$doacross local (i) shared(n, A) affinity (i)=data(A(i))
do i=1,n
   A(i) = i*i
enddo
```

The affinity clause specifies that iteration *i* should execute on the processor that contains element *A(i)* of the distributed array *A* (the distribution can be either regular or reshaped). Simple linear expressions such as *A(p*i+q)* are also allowed, but *p* and *q* must be literal constants, with *p* non-negative.

In summary, a programmer can use the *distribute* and *distribute_reshape* directives to distribute the key arrays in the program, along with the *affinity* clause on *doacross* loops to ensure that each loop iteration and the data referenced in that iteration are collocated on the same processor.

4 Implementation of Basic Features

We now describe the implementation of our programming model. We focus on reshaped arrays and give only a brief overview of the implementation of the other features.

4.1 Affinity Scheduling

We implement affinity scheduling by transforming the loop into a doubly (or triply) nested loop, where the outermost loop traverses the processors in the distributed dimension, and the inner loop(s) traverse each processor's elements of the distributed array. Our transformations, similar to those described by Hiranandani et al. [HKM+93], are reproduced in Figure 2.

4.2 Regular Distributions

Regular data distribution only affects the underlying page allocation. Its implementation, therefore, is simply an operating system call to allocate the physical pages for each portion of the distributed array from within the local memory of the corresponding processor. This system call is the only OS support required to implement both regular and reshaped data distribution, and it overrides the default first-touch page allocation policy.

This system call for page allocation is generated within the compiled code for local arrays. For *common* block arrays the object file is annotated with information about array dimensions and their specified distributions. At program start-up time the runtime library reads this information and makes the appropriate operating system call. Finally, a *redistribute* directive also translates into a runtime call to remap the pages for the array.

4.3 Reshaped Distributions

The implementation of the *distribute_reshape* must guarantee the desired distribution and is allowed to change the layout of the array in the virtual address space.

Rather than simply padding the array portions up to the next page boundary, we transform a reshaped array into a *processor-array* with each element of the processor-array in turn pointing to the array elements belonging to that processor (see Figure 3).

For a single reshaped dimension a reference *A(i)* is transformed as shown in Table 1 (the transformed reference is shown in C syntax). Our transformation of a reference is similar to that described by Anderson, Amarasinghe, and Lam [AAL95]. The first dimension of each reference is the index into the processor array (the processor containing element *i*), while the remaining dimensions index into the processor's portion of the array to locate element *i*. In this table *N* is the size of the array dimension, *P* is the number of processors assigned to that dimension, *b* is the size of a processor portion for a *block* distribution, and *k* is the chunk size for a *cyclic(k)* distribution. The transformation for a reshaped array distributed in multiple dimensions is a simple composition of this basic scheme.

Distribution	*Transformed Reference*
Block	A[i/b] [i%b]
Cyclic	A[i%P] [i/P]
Cyclic (k)	A[(i/k)%P] [i/(Pk)] [i%k]

Table 1. Transformation of Reshaped Array References

This implementation scheme allows us to manage storage for reshaped arrays in a space-efficient fashion. Since each processor's portion of a distributed array can be allocated independently of the other portions, each processor allocates a pool of storage from the shared heap, maps the pages for this pool of storage from within its local memory, and allocates its portion of each reshaped array from this pool of memory. We can therefore avoid padding the ends of each portion up to a page boundary.

As shown in Table 1, a transformed reference to a reshaped array contains integer divide and remainder operations (*div* and *mod*). These operations are extremely expensive on modern microprocessors - our optimizations for reducing their impact is the subject of Section 7.

5 Propagating Reshape Directives

As mentioned in Section 3, when a reshaped array is passed as an argument to another subroutine, we automatically propagate the *distribute_reshape* directive to the called subroutine.

There are two main issues in supporting this feature. The first is that it should work correctly with separate compilation, so that we can propagate the directive to subroutines that may be defined in other, separately compiled files. We do so in a way that is similar to a C++ template instantiation mechanism [Str94]. Briefly, for each user source file, the compiler maintains a shadow file

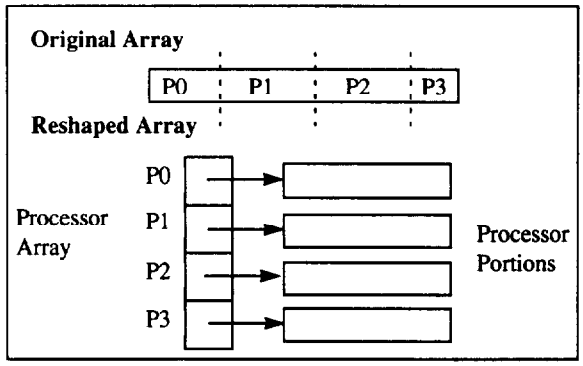


Figure 3. Transformation of a Reshaped Array.

into which it inserts an entry each time a reshaped array is passed as an argument to a subroutine. When the linker is called with all the object files, it first invokes a *pre-linker*, which examines all the object files and the corresponding shadow files, and propagates each directive to the called subroutine. Since the pre-linker is called with all the object files, it has a global view of the user program and can successfully propagate directives across files.

The second issue is that a subroutine may be invoked from multiple places with a different distribution on the same parameter. Compiling the subroutine to dynamically handle multiple incoming distributions may result in substantial runtime overhead. Instead, we clone a copy of the subroutine for each distinct combination of *distribute_reshape* directives on its parameters. Although this results in code expansion, the generated code is more efficient, since each cloned copy of the subroutine can be optimized at compile time for the particular combination of incoming distributions. Furthermore, in practice we expect the number of distinct distributions to be small.

The actual cloning of the subroutine is implemented as follows. The compiler updates the shadow file with (a) the name of each subroutine defined in the file (along with any *distribute_reshape* directives propagated into the subroutine), and (b) the name of each subroutine call in the file that contained a reshaped array as an actual argument. The pre-linker examines all the shadow files, matching subroutine invocations with subroutine definitions with respect to *distribute_reshape* directives on the parameters. For each invocation without a matching definition the pre-linker inserts a *request* into the shadow file for the desired subroutine instance and invokes the compiler again on that file. The compiler in turn examines the corresponding shadow file and creates the requested clones (if any) of each subroutine.

Finally, we avoid unnecessary cloning by removing requests from the shadow file for each definition that does not have a matching call. This is useful when the user removes a subroutine invocation from the program, leaving a now redundant request in the shadow file from the previous compilation.

Overall, this mechanism allows us to transparently clone multiple instances of a subroutine, one for each incoming combination of *distribute_reshape* directives on the subroutine parameters. We do so by transparently reinvoking the compiler at link time to compile a new clone of a subroutine. The first compilation of a

program can potentially result in several recompilations as the *distribute_reshape* directives are propagated all the way down the call graph of the program. However, subsequent compilations incur a recompilation only if a new cloning request is generated.

6 Error-Detection Support

Our error-detection support is geared towards enforcing the restrictions on reshaped arrays outlined in Section 3.2.1. The mechanisms for error detection include compile-time, link-time, and runtime checks. Among the compile-time checks, disallowing the equivalencing of reshaped arrays is straightforward and performed during the compilation of each subroutine. Checking that *common* blocks are declared consistently across all files is performed at link time - we annotate the shadow file (described in Section 5) with an entry for every declaration of a *common* block, along with the information (shape, size, and distribution) about each reshaped array (if any) in that *common* block. When the pre-linker is invoked, it reads all the entries and verifies that all declarations of each *common* block (one for each subroutine that uses it) are consistent with each other. Specifically, it verifies that each reshaped array in a *common* block appears at the same offset within the *common* block and with the same shape, size, and distribution in *all* declarations of that *common* block. This rule flags a link-time error only for inconsistent *common* blocks with reshaped arrays - *common* blocks without reshaped arrays are not affected.

Our optional runtime checks are useful when a reshaped array (or portion thereof) is passed as an argument to a subroutine. They are used to verify that the shape and size of the declared formal parameter are consistent with those of the incoming actual argument. These checks are implemented as follows. At each subroutine invocation with a reshaped array (or a portion thereof) passed as an argument, we take the address being passed in and use it as an index into a runtime hash table to store information about the actual argument. For the entire array we store the shape and size of the array, while for an array portion we store just the size of that portion. Upon entry to each subroutine, we take the incoming value for each parameter and use it as an index into the hash table described above. If an entry is found then the incoming argument is either a reshaped array or a portion thereof. In either case, we compare the information found in the hash table with the declared shape and size of the formal parameter, generating a runtime error in case of a mismatch.

The overhead for the runtime checks includes (i) adding an entry to the hash table each time a reshaped array is passed as an argument to a subroutine (and removing the entry upon return from the subroutine), and (ii) performing a lookup of the hash table at subroutine entry for each parameter to the subroutine (we optimize the second part to some extent by performing this lookup only for formal parameters that are declared to be arrays).

Overall, these runtime checks are extremely useful in catching errors in reshaped distributions. Such errors are otherwise extremely difficult to detect, since they are not easily distinguished from other algorithmic or coding errors.

7 Optimizing Reshaped Array References

As we saw earlier in Section 4.3, computing the address of an element of a reshaped array requires expensive div and mod operations - on a R10000 processor a 32-bit integer divide takes about 35 cycles. Optimizing these operations is crucial for high performance and is the subject of this section.

Given a reference to an element of a distributed array, the div operation determines which processor contains the element, while the mod operation determines the offset within that processor's portion of the array. When the reference is known to be local to the processor then the div operation can simply be replaced by the processor number. When the offsets within each processor's portion are strictly increasing then the mod operation can be replaced by an addition. Our basic optimization approach, therefore, consists of loop tiling and peeling to create inner loops that reference a single portion of the reshaped array and do not require any div and mod operations; this is similar to the approaches proposed by Anderson, Amarasinghe, and Lam [AAL95] and Hiranandani et al. [HKM+93]. Furthermore, several additional optimizations are necessary to ensure that other compiler optimizations are not adversely affected by reshaped arrays; we describe each of these below.

7.1 Tiling and Peeling for Reshaped Arrays

The tiling transformations to create kernels over portions of a distributed array are exactly the same as those presented for affinity scheduling in Section 4.1. The original loop is transformed into either two (for *block* and *cyclic* distributions) or three (for *cyclic(k)* distributions) loops. In either case, the outer loop, called the *processor tile loop*, indexes through the processors in the distributed dimension, while the inner loop(s) traverse the elements within a portion of the distributed array *A*.

We illustrate these with a few examples.

```
real*8 A(n)
c$distribute_reshape A(block)
do i = 1, n
   A(i) = i
enddo
```

Before optimization, this loop has the form

```
do i = 1, n
   A[i/b][i%b] = i
enddo
```

After the tiling optimization, the loop is transformed into

```
do p = 0, P-1
   lb = MAX((p*b+1), 1)
   ub = MIN(((p+1)*b), n)
   local_index = lb % b
   do i = lb, ub
        A[p][local_index] = i
        local_index = local_index + 1
   enddo
enddo
```

As we can see, the mod operation has been moved out of the innermost loop so that we now need only P rather than n mod operations, while the div operation has disappeared altogether.

If there are references to elements in the neighboring portion of a distributed array, then we peel iterations from the inner loop. For instance, in the code

```
do i = 2, n-1
   A(i) = (A(i-1) + A(i) + A(i+1))/3
enddo
```

we peel one iteration from each end of the inner loop, which again results in an innermost loop without any div or mod operations.

```
do p = 0, P-1
   lb = MAX(p*b, 2)
   ub = MIN(((p+1)*b-1, n-1)
   if (lb .LE. ub) then
        A[(lb-1)/b][(lb-1)%b] = (A[lb/b][lb%b] +
          A[(lb-1)/b][(lb-1)%b] +
          A[(lb+1)/b][(lb+1)%b]) / 3
   endif
   if (lb+1 .LE. ub-1) local_index = lb % b
   do i= lb+1, ub-1
        A[p][local_index] = (A[p][local_index-1] +
          A[p][local_index] + A[p][local_index+1])/3
        local_index = local_index + 1
   enddo
   if (lb .LT. ub) then
        A[(ub-1)/b][(ub-1)%b] = (A[ub/b][ub%b] +
          A[(ub-1)/b][(ub-1)%b] +
          A[(ub+1)/b][(ub+1)%b]) / 3
   endif
enddo
```

We increase the applicability of the above transformations in three ways. First, besides parallel loops with data affinity, we apply them to other loops that reference reshaped arrays, such as serial loops and parallel loops without user-declared affinity. For each loop we examine the reshaped array references in that loop and use a simple heuristic to determine the desired tiling and peeling of the loop that will result in the fewest div and mod operations. However, this optimization is not always possible: whereas tiling for reshaped arrays is always legal for (a) parallel loops, and (b) serial loops with *block* distributions (since the iterations are still executed in the same order as the original loop), for *cyclic* or *cyclic(k)* distributions this transformation changes the order in which the iterations are executed and is therefore subject to data dependence constraints.

Second, for loops such as

```
do i = 1, n
   A(i+c*k) = • • •
enddo
```

(c is a constant and k is a loop-invariant variable) we *skew* the loop by (c*k). This converts references like *A(i+c*k)* to *A(i)*, which enables subsequent tiling and peeling.

Third, having tiled a loop based on one array, we can simultaneously optimize references to other reshaped arrays that match the first array in size and distribution.

These optimizations are successful only when the index expressions of reshaped references are of the simple form $s*i+c$, with s and c being literal constants. More complex index expressions are not optimized and always incur the indexing overhead.

7.1.1 Loop Interchange

We can tile multiple loops in a nest if they reference reshaped arrays. In the code

```
real*8 A(n, n)
c$distribute_reshape A(block, block)
do j = 1, N
   do i = 1, N
        A(i,j) = i+j
   enddo
enddo
```

since A is distributed in both dimensions, we tile each of the j and i loops, generating a loop structure as follows:

```
do p_j = 1, P_j-1
   do j = . . .
        do p_i = 1, P_i-1
          do i = . . .
            • • •
          enddo
        enddo
   enddo
enddo
```

We try to interchange the j and p_i loops so that the processor tile loops (p_i and p_j) are outermost and the actual data loops (i and j) are innermost. We can then hoist div and mod operations within the p_i loop out of the j loop, which results in fewer such operations. This interchange is always legal for parallel loops within the *doacross-nest* directive (see Section 3.1), but is subject to the same legality constraints as normal loop interchange for sequential loops.

7.2 Hoisting and CSE of Array Index Expressions

After performing the optimizations described above, we examined the generated code for some small examples, but found that the address computation for reshaped arrays was still inefficient. There were three main reasons.

The first problem was the inability of the scalar optimizer to perform code hoisting and common subexpression elimination (CSE) across the index expressions of reshaped arrays. A reshaped reference generates indirect loads (from the processor array) and div and mod operations, all of which are, in general, unsafe operations that cannot be speculated. As a result, they cannot be moved out of an *if* clause or a *do* loop. Since these operations are always safe in the context of reshaped arrays, we fixed this problem by hoisting them out of loop nests and condi-

tional statements directly during the transformation of reshaped array references.

The second problem was poor CSE across index expressions in the presence of subroutine calls. For instance, the index expression for a reshaped array may contain references to the block size b of a distributed dimension. Although b is initialized at the array definition point and never modified, the compiler must assume that the subroutine call could modify b and must reload it from memory after each call. We solved this problem by marking such variables as constant within the compiler internal representation.

The third problem was that subsequent optimizations of the transformed code were affected by the indirect loads generated for accesses to reshaped arrays, since an indirect memory reference can in general be aliased to any other data in the program. We solved this problem by marking the base symbol of the array as aliased only to other references using the same base array.

7.3 DIV/MOD using Floating-Point Arithmetic

While an integer divide takes about 35 cycles on the MIPS R10000 processor and is not pipelined, the corresponding floating-point operation takes 11 cycles. We therefore simulate the integer divide in software using the floating-point unit. In addition to reducing the cost of the basic div and mod, the software scheme often enables additional hoisting of the reciprocal of the operands.

7.4 Integration with Other Optimizations

Within the overall MIPSpro7.1 compiler, we process reshaped arrays as part of the loop-nest optimizer. The order of the optimizations is as follows:

1. Loop skewing, tiling, interchange, and peeling for reshaped arrays.
2. Regular loop-nest optimizations (e.g., fusion, fission, interchange, cache and register tiling).
3. Transformation of reshaped array references, including optimization of reshaped references in the inner loops, and hoisting indirect loads and div and mod operations.
4. CSE across index expressions of reshaped arrays.

The primary benefit of this approach is that the loop transformations for reshaped arrays are done early, presenting the code in a convenient form for the regular loop-level optimizer. Since the processor tiles are in place and often interchanged to be the outermost within a loop nest, the loop-nest optimizer can perform single processor optimizations as it would on normal, single processor code. Furthermore, by delaying the transformation of reshaped arrays references we maintain them in a reasonable form during the regular loop-level transformations.

8 Performance Results

We present performance results from one benchmark application (LU from the NAS 2.1 parallel benchmark suite) and two computation kernels (matrix transpose and 2-D convolution). Our results are obtained on a 128-processor Origin-2000 with a 4MB secondary cache per processor and 16 GB of memory. In addition to measuring the overall performance, we use the hardware counters on the MIPS R10000 to analyze our results [ZLT+96].

8.1 LU

We converted the original MPI version of the LU benchmark to a shared memory version using *doacross* directives for parallelism and *distribute_reshape* directives for data distribution. The primary data structures are two 4-dimensional arrays that we distribute in a *(*,block,block,*)* fashion, based on the parallel partitioning of the program.

Optimization	*Time (secs)*
Reshape, no optimizations	*83.91*
Reshape, tile and peel	53.26
Reshape, tile and peel, hoist	46.23
Original code without reshaping	45.71

Table 2. Effect of Reshape Optimizations.

We first evaluate just the basic effectiveness of our reshaped array addressing optimizations. We do so by comparing the code with and without reshaping running on a single processor, so that we can focus on just the reshaped array addressing overhead. As shown in Table 2, the basic code with reshaping, compiled at -O3 but without optimizations for reshaped arrays, ran very poorly. Tiling and peeling for reshaped arrays helped tremendously, followed by hoisting of indirect loads and div and mod operations, which enabled CSE across index expressions. Most importantly, the final version of the code ran nearly as efficiently as the original code without reshaping.

Figure 4 shows the relative performance of four versions of LU on the class C input (the arrays are of size (5,166,166,166)). Two instances are without any data distribution directives: one uses the first-touch policy and the other uses round-robin page placement. (Data is initialized in parallel in this application.) The other two instances are with regular and reshaped data distribution directives respectively.

As shown by the results, the performance of this application is determined by bandwidth rather than latency considerations. Since all four versions spread the data across the machine (although differently), they all achieve good performance. The parallel initialization of data is quite effective, with first-touch outperforming both round-robin and regular distribution (the latter two exhibit nearly identical performance). Furthermore, only reshaping can effectively provide the desired *(*,block,block,*)* distribution and obtains the best performance on 64 processors, although the improvements over first-touch are modest (6%).

All four instances exhibit superlinear speedup because (a) the large data set size (360MB) exceeds the amount of memory on a single node (about 250MB) resulting in remote memory references even in the uniprocessor case, and (b) the larger aggregate

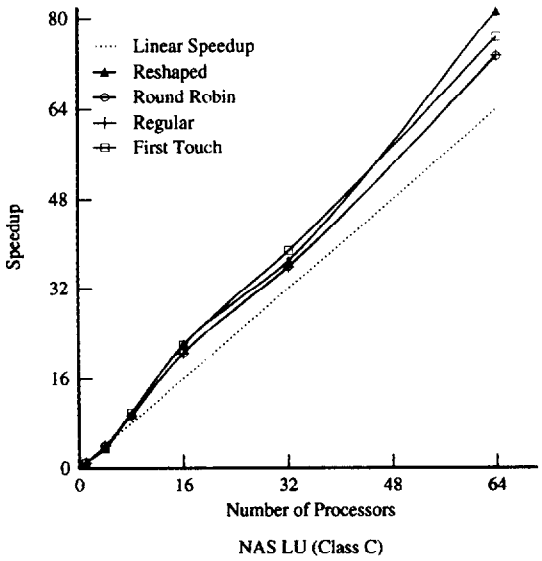


Figure 4. Performance of NAS-LU (Class C).

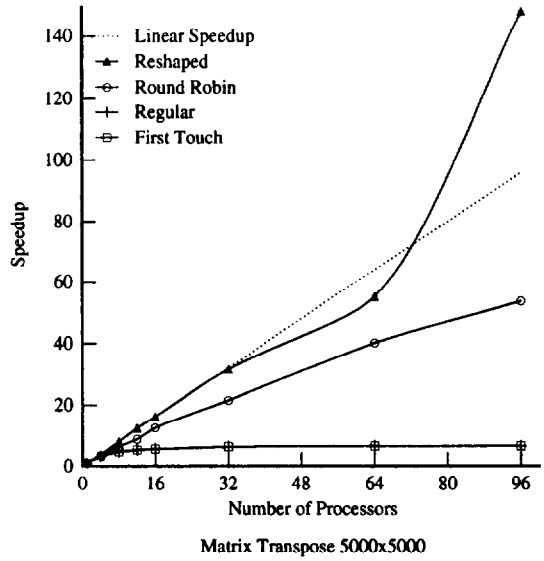


Figure 5. Performance of Matrix Transpose.

cache at high processor counts improves the cache hit rate. We counted the cache misses using the hardware counters on the processor, and found that the total number of secondary cache misses decreased by a factor of three from 1 to 16 processors.

8.2 Matrix Transpose

We next examine the performance of a parallel matrix transpose, in which we iterate several times over the following loop nest.

```
c$distribute A (*,block), B(block,*)
c$doacross local(i,j)
do i=1,m
   do j=1,m
      A(j,i) = B(i,j)
   enddo
enddo
```

Figure 5 presents the speedup of four versions of the code over a serial version on an input of size 5000x5000. The four versions are first-touch allocation, round-robin placement, regular distribution, and reshaped distribution. Data initialization is performed serially in this application. The matrix with the *(block,*)* distribution cannot be distributed properly without reshaping; consequently both first-touch policy and regular distribution result in most of the data being allocated from within the memory of one or two nodes. These nodes become a bottleneck for accesses to this matrix, which results in extremely poor performance. Round-robin placement outperforms these two versions by distributing the data uniformly across the nodes and better utilizing the network bandwidth.

The reshaped version obtains the best performance: by reshaping even the row-distributed matrix so that each processor's portion is contiguous in memory we are able to ensure that almost all misses are satisfied locally. A secondary effect is the reduction in TLB misses with reshaping: since the reshaped version uses all the data in a page, it uses much fewer pages compared to the other versions. Using the hardware performance counters, we found that at 32 processors the round-robin version spends about 15% of its time in TLB misses, while the reshaped version needs less than half that time.

At moderate processor counts the reshaped version outperforms the round-robin version by 30-50%. At a little over 64 processors the reshaped version starts to improve superlinearly as the application gets increasing benefits from cache reuse. The two arrays require a total of 400 MB of memory while our system has 4 MB of secondary cache per processor. At large processor counts this data begins to fit within the secondary caches, and the number of cache misses decreases substantially for the reshaped version (a total of 244M misses compared to 702M for the round-robin version at P=96). One might expect similar effects for the other versions, but random cache interference will prevent them from fully utilizing the cache until much larger processor counts. Data reshaping ensures that each processor's portion is contiguous in virtual memory, and the OS page-coloring algorithm tries to ensure that contiguous virtual addresses do not map into conflicting physical addresses. As a result, cache interference is greatly reduced for the reshaped version of the code.

8.3 2-D Convolution

We next examine the performance of a 2-D convolution. As shown in the code below, we exploit either one or two levels of parallelism with distributions of *(*,block)* and *(block,block)* respectively.

```
c$distribute A(*,block), B(*,block)
c$doacross local (i,j) affinity(j) = data(A(i,j))
do j=2,n-1
   do i=2,n-1
      A(i,j)=(B(i-1,j)+B(i,j-1)+B(i,j)+B(i,j+1)+B(i+1,j))/5
   end do
end do
```

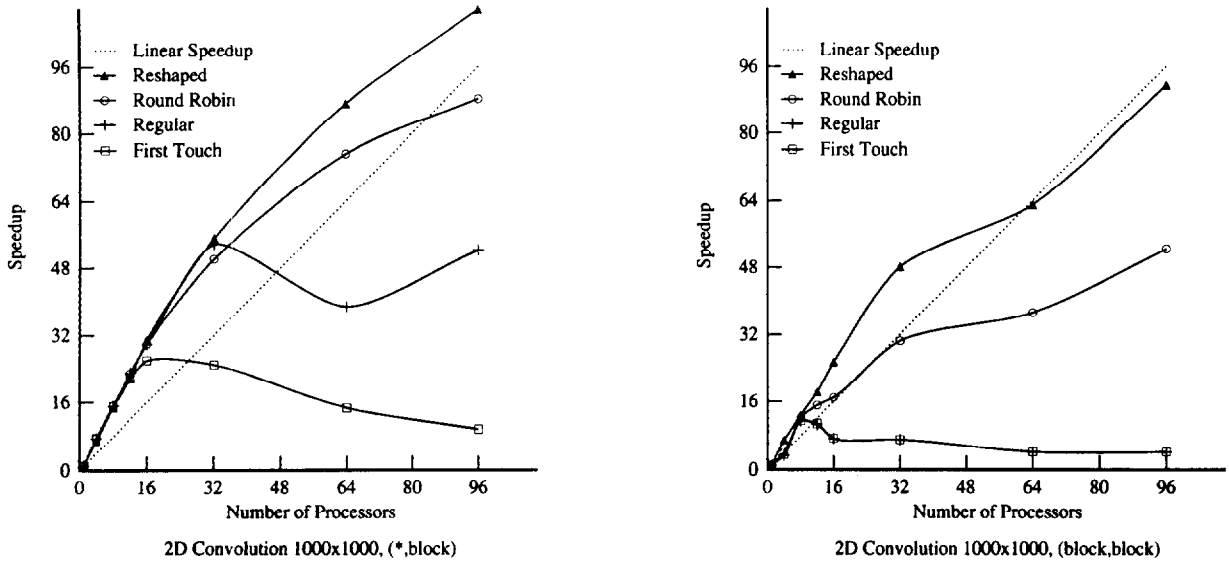


Figure 6. Performance of 2-D Convolution (1000x1000).

The following exploits two levels of parallelism

```
c$distribute A(block,block), B(block,block)
c$doacross nest (j,i) local (i,j) affinity(j,i)=data (A(i,j))
do j=2,n-1
   do i=2,n-1
      A(i,j)=(B(i-1,j)+B(i,j-1)+B(i,j)+B(i,j+1)+B(i+1,j))/5
   end do
enddo
```

We present results with the first-touch policy, round-robin page placement, regular distribution, and reshaped distribution. Due to serial initialization all the data is placed on just a few nodes with the first-touch scheme. Our results are relative to the serial version of the code and presented on two different input sets: 1000x1000 (Figure 6) and 5000x5000 (Figure 7).

With a single level of parallelism we obtain successive improvements over first-touch allocation with regular, round-robin, and then reshaped distribution. Regular distribution benefits are due to memory locality alone. On the smaller input round-robin placement outperforms regular distribution for increasing number of processors, even though the cache miss behavior remains unchanged. We believe this is because the round-robin policy results in more uniform page distribution and better bandwidth utilization as compared to regular distribution. For the 1000x1000 case the performance of regular distribution actually becomes chaotic at large processor counts - each processor's portion becomes progressively smaller, increasing page-level false sharing. Since a page requested by multiple processors is simply allocated from within the local memory of the processor to last request the page, this often results in a poor distribution. Reshaping reduces the degradation due to edge effects at page boundaries and achieves the best performance. However, these edge effects are less important for larger problem sizes - for the 5000x5000 case (the left graph in Figure 7) regular distribution performs as well as reshaped distribution. With 96 processors the application data (400MB) fits completely within the processor caches and both round-robin and regular distribution outperform the reshaped version (which incurs some reshaping overhead).

When we exploit two levels of parallelism both first-touch and regular distribution perform equally poorly, since with two-dimensional blocks the array layout suffers severely from false sharing over both cache lines and pages. Round-robin placement is an improvement due to better bandwidth utilization, but reshaping is the only option for such distributions and, as expected, performs much better. For larger processor counts in the 5000x5000 case, two-level parallelism outperforms the single level due to better communication/computation ratio.

Finally, the data set size (16MB for the smaller input, 400MB for the larger input) exceeds the 4MB cache of a single processor, which leads to superlinear speedups for higher processor counts with larger aggregate cache size.

8.4 Summary

Overall, these performance results illustrate the need for providing both regular and reshaped distribution. As demonstrated by the convolution code on the bigger input, regular distribution is perfectly adequate when the individual portions of a distributed array are large. Reshaped distributions, on the other hand, are useful when data needs to be distributed at a finer granularity, such as the *(block, block)* distribution in the convolution code. Besides improving memory locality, reshaping the layout of an array can also improve cache behavior by improving spatial locality and reducing false sharing across cache lines.

9 Related Work

Several approaches have been proposed in the literature to improve data locality, including operating-system-based page migration, compiler-based data distribution, as well as programming languages such as HPF.

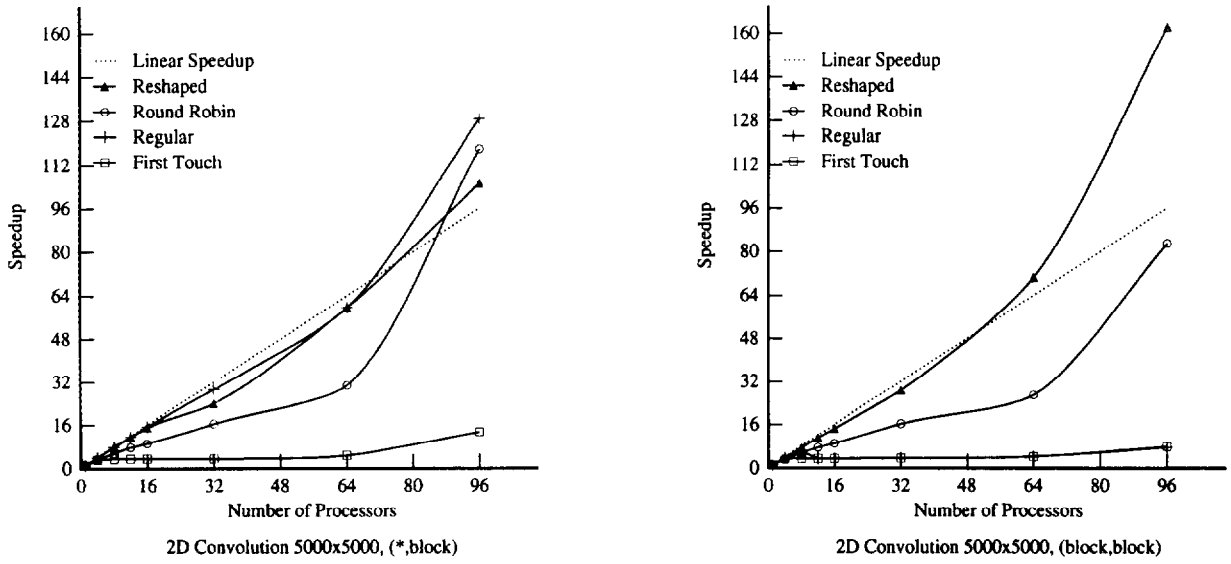


Figure 7. Performance of 2-D Convolution (5000x5000).

Operating-system-based approaches [VDG+96] use runtime statistics, such as TLB or cache misses, to identify the processor incurring the most cache misses to a page and migrate the page to the local memory of that processor. While transparent to the user, the main limitations of this approach are that it migrates data at the granularity of an entire page, incurs runtime overhead and must therefore be conservative, and infers application behavior only indirectly through per-page statistics.

Compiler-based approaches [AAL95, AnL93, GuB92, KeK95, SSP+95] to data distribution are typically integrated with the automatic detection of loop-level parallelism. These approaches are attractive since they are entirely transparent to the user and, in contrast to OS-guided approaches, perform data distribution based on a static analysis of the data reference patterns in the program. However, many codes are not amenable to such automatic compiler analysis and require explicit programmer intervention.

Finally, programming languages such as HPF [Lov93, KLS+94, HPF2-97], Fortran-D [HKK+91], and Vienna Fortran [CMZ92], provide a variety of data distribution directives. We limit our comparison to HPF, the most widely known of these languages. HPF was originally designed as an alternative to message passing for programming distributed address space machines such as the IBM SP-2. In HPF the user typically writes a serial program and annotates it with data distribution information. Based on this distribution, the compiler partitions and schedules the program for parallel execution and manages the communication and synchronization between processors. Programming in the message passing paradigm can be quite difficult, and the primary attraction of HPF is a familiar shared memory programming model with all the communication managed automatically by the implementation.

Our model is similar to HPF in many respects: our data distribution directives are similar to those provided in HPF, our *doacross* directive for expressing loop-level parallelism is the same as the *independent* directive in HPF, while our *affinity* clause corresponds to the *on* directive, recently introduced in HPF-2.0 [HPF2-97]. Furthermore, our restrictions related to storage and sequence association on reshaped arrays are similar to those imposed on distributed arrays in HPF.

In contrast to HPF, however, our model was designed for cache-coherent shared address space machines and therefore differs in several fundamental ways. When programming in HPF on a distributed address space machine, data distribution is necessary to obtain parallel execution. In contrast, on a CC-NUMA machine data distribution is performed as an optional optimization over and above exploiting parallelism. On such machines, because of global cache-coherent shared memory, traditional multiprocessing codes written with loop-level parallel directives continue to run correctly in parallel without requiring data distribution. Because remote latencies are relatively low (compared with those on distributed address space machines), these programs often run well without any modifications. Data distribution on these machines is purely a performance enhancement to overcome the additional penalty of a remote reference over a local reference. For our abstractions, therefore, a simple programming model and an efficient implementation were absolutely critical: if we were not careful, the overhead of managing data distribution could very quickly outweigh any gains from memory locality.

The differing goals are reflected in our specific designs. First, we provide two distinct types of data distribution, regular and reshaped, whereas HPF provides only reshaped distributions. We believe it is important to have both types of distributions on CC-NUMA architectures: distributions that do not have page granularity problems can use regular distribution, thereby avoiding the legality restrictions and the array addressing overhead of reshaped arrays, while distributions that suffer from page granularity limitations can use reshaping.

Second, we provide only the core data distribution constructs, and omit most of the advanced HPF features. For instance, we do

not support dynamic redistribution of reshaped data. Also, we provide a simpler model for passing reshaped arrays as arguments to subroutines. HPF allows mismatched distributions on actual and formal parameters, and it provides three different kinds of distribution directives on formal array parameters. The implementation must remap the data (if necessary) at runtime if the directive is *prescriptive*, compile the called subroutine to accept any incoming distribution if the directive is *transcriptive*, and assert the specified distribution of the actual parameter if the directive is *descriptive*. In contrast, we automatically propagate distributions down the call chain, cloning routines as necessary. An HPF implementation cannot always propagate distributions at compile time since HPF permits dynamic redistribution of data. Because of these simplifications in our programming model, the distributions of all reshaped arrays are always known at compile time. This makes it much more likely that we will optimize the implementation of reshaped array references.

10 Conclusions

In this paper we have presented a set of abstractions for distributing data on CC-NUMA multiprocessors. Our abstractions are easy to use and provide a simple model for data reshaping. Our unique error-detection features and propagation of distribution directives across subroutine calls enable the user to safely use reshaped distributions. Efficiency has been a prime concern, and we have carefully avoided features that could hinder compiler optimizations of data distribution (such as dynamic data reshaping). We have implemented these abstractions within the SGI MIPSpro7.1 compiler and incorporated several optimization techniques to improve the efficiency of the generated code. Our initial experience has been encouraging: the abstractions allow the user to focus on identifying the desired distribution based on the characteristics of the application, and leave the implementation details to the compiler.

Acknowledgments: We thank Jeff McDonald who helped us with the LU application, Chau-Wen Tseng who participated in the initial stages of this work, and Seema Hiranandani and the anonymous referees who offered many useful comments on earlier drafts of this paper.

Bibliography

[AAL95] J. M. Anderson, S. P. Amarasinghe and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166-178, July 1995

[AnL93] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112-125, June 1993.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1), pages 31-50, Fall 1992.

[GuB92] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. In *Transactions on Parallel and Distributed Systems*, 3(2), pages 179-193, March 1992.

[HPF2-97] High Performance Fortran Language Specification, Version 2.0, January 1997. Available by anonymous ftp from softlib.rice.edu: /pub/HPF.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C-W. Tseng. An Overview of the Fortran-D Programming System. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Springer-Verlag, August 1991.

[HKM+93] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced Compilation Techniques for Fortran D. Technical Report CRPC-TR93338, Center for Research on Parallel Computation, October 1993.

[KeK95] K. Kennedy and U. Kremer. Automatic Data Layout for High Performance FORTRAN. In *Proceedings of Supercomputing '95*, December 1995.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. The High Performance Fortran Handbook. MIT Press, Cambridge MA, 1994.

[LL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[Lov93] D. B. Loveman. High Performance Fortran. In *IEEE Parallel and Distributed Technology, Systems & Applications*, 1(1), pages 25-42, February 1993.

[SSP+95] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert and S. Chatterjee. Efficient Distribution Analysis via Graph Contraction. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Processing*, August 1995.

[SGI96] Silicon Graphics, *MIPSpro Fortran-77 Programmer's Guide*, Document number 007-2361-004, 1996.

[Str94] B. Stroustrup. The Design and Evolution of C++. Addison Wesley, Reading MA, 1994.

[VDG+96] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 279-289, October 1996.

[ZLT+96] M. Zagha, B. Larson, S. Turner, M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Supercomputing 96*, Pittsburgh PA, November 1996.