Extending OpenMP for NUMA machines

John Bircsak, Peter Craig, RaeLyn Crowell,
Zarka Cvetanovic, Jonathan Harris*,
C. Alexander Nelson and Carl D. Offner
Compaq Computer Corporation, 110 Spitbrook Road, Nashua, NH 03062, USA

This paper describes extensions to OpenMP that implement data placement features needed for NUMA architectures. OpenMP is a collection of compiler directives and library routines used to write portable parallel programs for shared-memory architectures. Writing efficient parallel programs for NUMA architectures, which have characteristics of both shared-memory and distributed-memory architectures, requires that a programmer control the placement of data in memory and the placement of computations that operate on that data. Optimal performance is obtained when computations occur on processors that have fast access to the data needed by those computations. OpenMP – designed for shared-memory architectures – does not by itself address these issues.

The extensions to OpenMP Fortran presented here have been mainly taken from High Performance Fortran. The paper describes some of the techniques that the Compaq Fortran compiler uses to generate efficient code based on these extensions. It also describes some additional compiler optimizations, and concludes with some preliminary results.

1. Introduction

As computer vendors increase both the amount of memory and the number of processors that can be configured within a single system, the bandwidth of the connection between processors and the memory they share has become a critical bottleneck that limits the scaling of parallel applications.

As a result, a number of vendors are designing systems partitioned into smaller modules. Within each module is a local memory and a small number of processors which have very high speed access to that local memory. The modules are interconnected so each processor can access the memory located in other modules – there is logically one global memory. However, access to such memory in other modules (measured both by latency and bandwidth) is slower. These systems are referred to as Non-Uniform Memory Architectures, or NUMA. Commercial examples include: Compaq's AlphaServer GS80, GS160, and GS320 systems, the HP 9000 V-class systems, IBM's NUMA-Q architecture, and SGI's Origin 2000 systems.

NUMA systems can deliver high performance for OpenMP [1] applications if the placement of data and computations is such that the data needed by each thread is local to the processor on which that thread is executing. Techniques for automatically placing data and computations, while effective in some cases, do not work well for all programs. Accordingly, Compaq has added a set of directives to its Fortran for Tru64 UNIX that extend the OpenMP Fortran API. Using these directives, the programmer can specify the placement of data and computations onto memories for optimal performance.

2. Motivation

Consider the red-black code in Fig. 1.

Each page of the array X must be stored in one of the physical memories of the NUMA system. The processors on the same module as that page have very fast access to it; the processors on other modules have slower access to it.

In the program above, the schedule clause of each DO directive controls the assignment of iterations to threads. But the program does not associate the threads with the portions of the array X that they will be accessing. For example, if all of array X is stored in the memory of one module, but the threads of the program execute on processors on all of the modules, then only the few threads executing on the module containing array X benefit from the high speed local access. All other threads compete for the limited bandwidth of the remote access path to that memory, which limits application scaling and increases execution time.

If the storage for array X was divided into pieces, with each piece placed into the module of the thread

*Corresponding author. Tel.: +1 603 884 1779; Fax: +1 603 884 0153; E-mail: Jonathan.Harris@compaq.com.

ISSN 1058-9244 / $8.00 © 2000, IEEE. Reprinted with permission from Proceedings of IEEE Supercomputing 2000, 4–10 November 2000, Dallas, Texas, USA.

```
program redblack
parameter(n = 8000, tolerance = 0.001)
integer i, j
logical converged
real x(0 : n+1, 0 : n+1), t, diff
call initialize(x)
converged = .false.
do while(.not. converged)
    diff = 0.0
    !Do the red iterations:
    !$omp parallel private(i, j, t)
    !$omp do schedule(static) reduction(+:diff)
    do j = 1, n
        do i = 2 − iand(j, 1), n, 2
            t = 0.25 ∗ (x(i−1, j) + x(i+1, j) + x(i, j−1) + x(i, j+1))
            diff = diff + (t − x(i, j))∗∗2
            x(i, j) = t
        end do
    end do
    !Do the black iterations:
    !$omp do schedule(static) reduction(+:diff)
    do j = 1, n
        do i = 1 + iand(j, 1), n, 2
            t = 0.25 ∗ (x(i−1, j) + x(i+1, j) + x(i, j−1) + x(i, j+1))
            diff = diff + (t − x(i, j))∗∗2
            x(i, j) = t
        end do
    end do
    !$omp end parallel
    converged = sqrt(diff) .le. tolerance
end do
print *, x
end program redblack
```

Fig. 1. An OpenMP Fortran code for a red-black computation.

using that piece, then all of the threads would benefit from high speed local access to their data. This universal local access would result in a dramatic improvement in scaling and execution time.

2.1. *Automatic Methods and Operating System Features*

A simple way to spread an application's data across the memories of a NUMA system is to make the operating system's virtual memory subsystem aware of the distinct physical memories as it chooses which physical pages to use to satisfy page faults. The system can use different algorithms to determine in which memory a given virtual page will be stored:

The round robin algorithm effectively arranges the memories in a circular list and maps each consecutive virtual page to the next memory in the list. This eliminates the bottleneck that occurs when all of the data is stored in a single memory. However, it does not specifically place pages of data on the memory modules of their referencing threads.

The first touch algorithm chooses a physical page on the module with the thread that first caused a page fault for that page. This causes each page to be located on the module that makes the first reference to it.

A more dynamic approach to page placement is automatic page migration. In this approach statistical information is kept about the number of references to a page from each module in the NUMA system. If references from a module not containing the physical page predominate, then the operating system migrates the physical page to the referencing module. Researchers have presented encouraging results using this technique; however experience has shown that it is challenging to tune automatic page migration such that it responds rapidly enough without incurring excessive overhead.

Each of these techniques has limited information about how the application uses the data:

- The round robin algorithm uses no information about the program whatsoever, and so there is no reason to expect a page to exist on the module with the thread that is accessing it.
- The first touch algorithm only uses information based on where a page was first referenced. It knows nothing about subsequent references, which might well be by different threads on different modules.
- Automatic page migration uses statistical information but often suffers because it must wait until sufficient data has been gathered to justify migrating a page.

On the other hand, skilled parallel programmers can easily identify the computationally intensive parts of their application, so they are already aware of the pattern in which the threads will access the data. Such knowledge is already required to tune applications for optimum cache performance.

This knowledge can be used to direct the compiler to place data in the modules for fast access by threads. The remainder of this paper describes extensions to OpenMP to allow programmers to accomplish this placement.

3. An extended OpenMP language

This paper presents two related, but different, sets of directives for extending OpenMP for NUMA systems: user-directed page migration and user-directed data layout.

3.1. User-directed page migration directives

Many technical applications have a small number of loop nests which account for a large percentage of the execution time. Furthermore, when these loops are parallelized, each thread often makes repeated access to the same sections of memory.

The MIGRATE_NEXT_TOUCH directive provides a simple data migration solution that will often yield nearly optimal performance in these cases. Its syntax is:

!dec$ migrate_next_touch (variable$_1$, . . ., variable$_n$)

For each variable in the list, this directive specifies a range of pages starting with the page that contains the first byte of the variable, and ending with the page that contains the last byte of the variable. When execution reaches the MIGRATE_NEXT_TOUCH directive, each of the indicated pages is marked for migration. The next time a thread references one of these pages, the page will migrate to the module of the referencing thread. In the typical application, the user will insert a MIGRATE_NEXT_TOUCH directive just before the key parallel loops. This marks for migration each shared array used in the loop. For example, the red-black program in Fig. 1 would only need the following directive inserted prior to the do while loop:

!dec$ migrate_next_touch (X)

This user-directed form of page migration avoids the delays associated with gathering statistics for automatic page migration. The overhead of migrating the pages is small compared to the savings of time achieved by making most of the references local.

If you inspect your MIGRATE_NEXT_TOUCH directive and discover that the data in the array mentioned in the directive will never be used (because it will all be overwritten before it is read), then a PLACE_NEXT_TOUCH directive can be used instead. This directive remaps the virtual pages without copying the original data in the array. Its syntax is:

!dec$ place_next_touch (variable$_1$, . . . , variable$_n$)

For each variable in the list, this directive specifies a range of pages including all pages that are entirely contained within the storage for the variable. Partial pages at the start or end of the storage for the variable are not included. The next time a thread references data in one of these pages, the operating system assigns a physical page to it that is located on the module of the referencing thread. The previous contents of the page are not moved to the new location, and hence the contents of the page are undefined. The **Place_Next_Touch** directive provides a highly efficient method for distributing an output array over a set of modules.

3.2. User-directed data layout directives

When there is not a sustained association between threads and the data they reference, the overhead of migrating pages may be unacceptable. Furthermore, when pages are large relative to the declared extents of arrays, distribution with page granularity may be too coarse. To overcome this limitation, an OpenMP programmer can make use of compiler support for distributing individual elements of arrays onto the memories.

OpenMP has a rich language for specifying thread-parallel computations, but its storage model assumes a single uniform memory architecture and provides no facilities for laying out data onto multiple distinct memories. Further, the schedulable computation entity, a loop iteration, cannot be scheduled in a way that recognizes the data accesses made by that iteration. High Performance Fortran (HPF) [3], developed by a consortium of users and vendors, contains a carefully designed set of directives to solve exactly these problems. Compaq has added HPF's data layout directives to Compaq Fortran for Tru64 UNIX.

The programmer uses these data layout directives to specify how to divide arrays into pieces and how to assign those pieces to the memories. The programmer will use knowledge of the algorithm to place these pieces so that, to the extent allowed by the algorithm, all pieces needed by a loop iteration are in the same memory unit. Based on the data layout and the data references in the loop, the compiler then identifies the module on which each iteration should be executed and assigns the loop iterations to threads executing on those modules.

As a matter of terminology, the data layout of an array is also called the *mapping* of the array. That is, the data layout itself is referred to as a mapping—it describes the function that maps array elements to memory locations on different modules. An array to which mapping directives apply is called a *mapped array*.

3.2.1. Data layout directives

The simplest and most important data layout directive is the DISTRIBUTE directive. This directive specifies a mapping pattern of data objects onto memories. It is used with the keywords BLOCK, CYCLIC, or $*$, which specify the distribution pattern.

The use of the DISTRIBUTE directive is best explained by examining some example distributions. Consider the case of a 16×16 array A in an environment with 4 memories. Here is one possible specification for A:

real $A(16, 16)$
!dec$ distribute $A(*,$**block**$)$

Figure 2 shows this distribution.

The asterisk (*) for the first dimension of A means that the array elements are not distributed along the first (vertical) axis. In other words, the elements in any given column are not divided up among different memories, but assigned entirely to one memory. This type of mapping is called a "collapsed" or "serial" distribution.

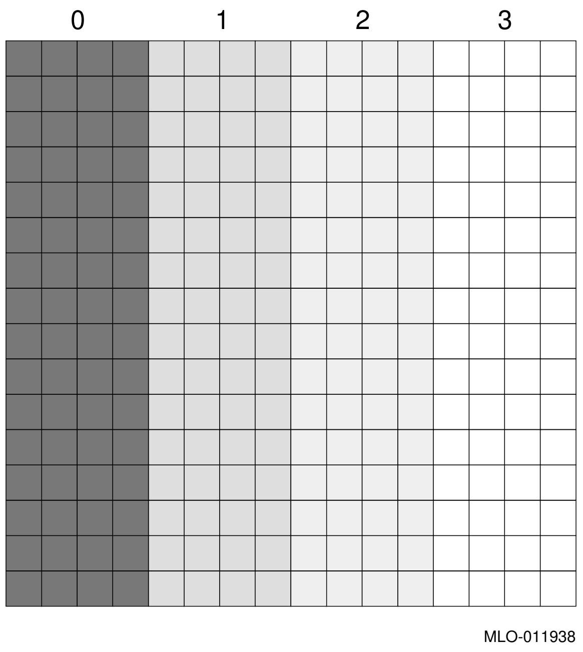

Fig. 2. Distributing a 16×16 array (*, BLOCK). The shading indicates into which memory each array element is mapped.

The BLOCK keyword for the second dimension means that for any given row, the array elements are distributed over each memory in large blocks. The blocks are of approximately equal size with each memory assigned to only one block. As a result, A is broken into four contiguous groups of columns with each group assigned to one memory.

Another possibility is (*, CYCLIC) distribution. As in (*, BLOCK), the elements in any given column are assigned as entirely to one memory. The elements in any given row, however, are dealt out to the memories in round-robin order, like playing cards dealt out to players around the table. When elements are distributed over n memories, each memory, starting from a different offset, contains every nth column. Figure 3 shows the same array and memory arrangement as Fig. 2, distributed (*, CYCLIC) instead of (*, BLOCK).

The pattern of distribution is figured independently for each dimension: the elements in any given column of the array are distributed according to the keyword for the first dimension, and the elements in any given row are distributed according to the keyword for the second dimension. For example, in an array distributed (BLOCK, CYCLIC), the elements in any given column are laid out in blocks, and the elements in any given row are laid out cyclically, as in Fig. 4.

Figure 5 shows an example array distributed (BLOCK, BLOCK).

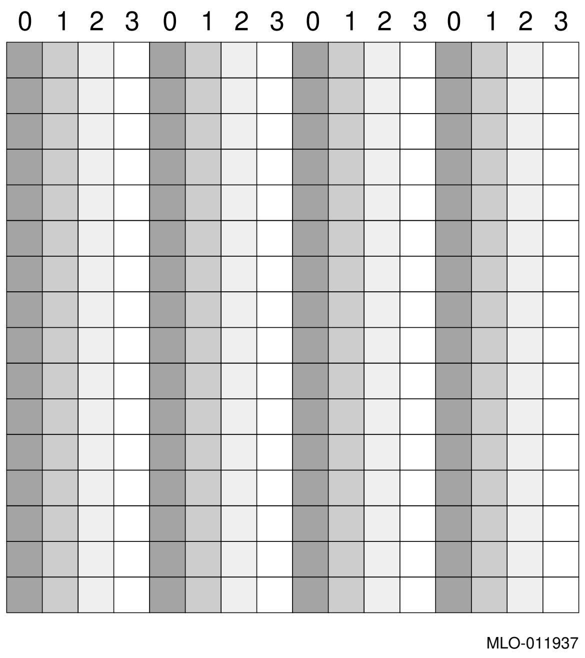

Fig. 3. Distributing a 16 × 16 array (*, CYCLIC). The shading indicates into which memory each array element is mapped.

A BLOCK, BLOCK distribution divides the array into large rectangles. The array elements in any given column or any given row are divided into two large blocks. In the current example, memory 0 gets A(1:8, 1:8), memory 1 gets A(9:16, 1:8), memory 2 gets A(1:8, 9:16), and memory 3 gets A(9:16, 9:16).

3.2.2. *Deciding on a distribution*

There is no simple rule for computing data distribution, because optimal distribution is highly algorithm-dependent. When the best-performing distribution is not obvious, it is possible to find a suitable distribution through trial and error, because the DISTRIBUTE directive affects only the performance of a program (not the meaning or result). In many cases, however, you can find an appropriate distribution simply by answering the following questions:

– Does the algorithm have a row-wise or column-wise orientation?
– Does the calculation of an array element make use of distant elements in the array, or – like the red-black computation in Fig. 1 – does it need information primarily from its near neighbors in the array?

If the algorithm is oriented toward a certain dimension, the DISTRIBUTE directive can be used to map the data appropriately. For example, Fig. 2 shows that a (*, BLOCK) distribution is vertically oriented. A (BLOCK, *) distribution, in contrast, is horizontally oriented.

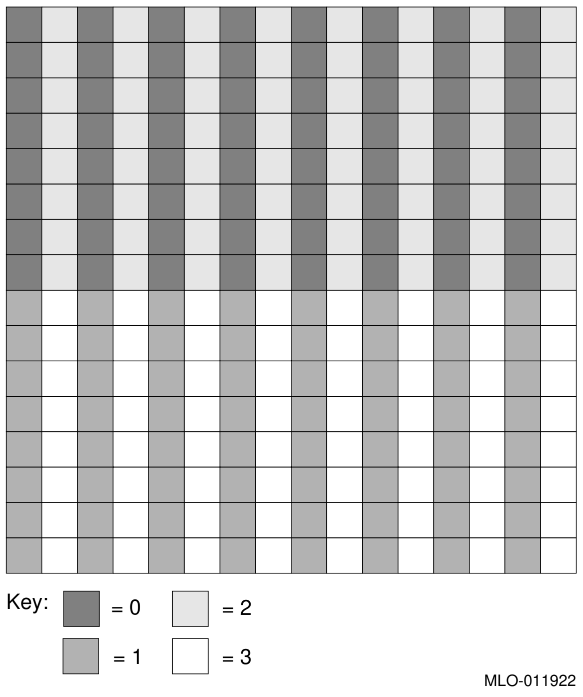

Fig. 4. Distributing a 16 × 16 array (BLOCK, CYCLIC). The shading indicates into which memory each array element is mapped.

Nearest-neighbor calculations generally run faster with a BLOCK distribution. This is because the computations needed to update the array will reference data that resides in the same memory in most cases.

When the calculation of an array element requires information from distant elements in the array, a CYCLIC distribution is frequently faster because of load-balancing considerations.

3.2.3. *An example: LU decomposition*

We take as an example the standard textbook form of LU decomposition, without pivoting. (Adding pivoting is important in ensuring numerical stability, but does not materially change the analysis we present here.)

Figure 6 shows the Fortran code, written with an OpenMP directive (beginning with **!$omp**) specifying that the j loop will iterate in parallel over the columns of the array.

Figure 7 shows how the algorithm progresses. On iteration k of the outer loop, the lightly shaded column of elements below $a(k, k)$ is normalized by dividing each element by $a(k, k)$. This is the operation performed by

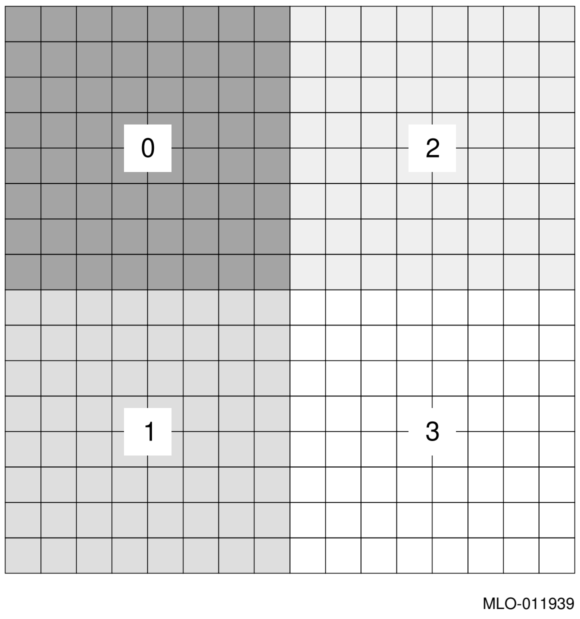

Fig. 5. Distributing a 16×16 (BLOCK, BLOCK). The shading indicates into which memory each array element is mapped.

```
program LU
integer, parameter ($n = 1024$)
real(kind =8) :: $a(n, n)$
!dec$ distribute ($*$, cyclic) :: $a$
do $k = 1, n$
    ! Column normalization
    do $m = k + 1, n$
        $a(m, k) = a(m, k)/a(k, k)$
    end do
    ! Submatrix modification
    !dec$ omp numa
    !$omp parallel do private($i$)
    do $j = k + 1, n$
        do $i = k + 1, n$
            $a(i, j) = a(i, j) - a(i, k) * a(k, j)$
        end do
    end do
end do
end program LU
```

Fig. 6. LU decomposition without pivoting, with a data layout directive added.

the inner loop on m (labeled "Column normalization") inside the k loop. Next, the darker shaded square of elements to the right of that column is updated by taking each element $a(i, j)$ and subtracting the product of the elements $a(i, k)$ and $a(k, j)$. This update can be performed in parallel.

The update step accesses data symmetrically across both dimensions of the matrix a. The normalization step, on the other hand, accesses data down a single column of the matrix. For this reason, it seems most efficient to map the data so that each column lives entirely on one memory. In that way, the normalization step can be carried out as a purely local pipelined computation. Parallelism will come about by distributing the columns to distinct threads.

To map the columns of the array a to distinct processors, we could use either a (*, BLOCK) or a (*, CYCLIC) distribution. Since the update step accesses distant elements in the array, little advantage would be gained from a block distribution. On the other hand, there is much to be gained from using a cyclic distribution in the case of our algorithm. To see why, see Fig. 8, which depicts the computation with (*, BLOCK) distribution (the illustration shows a 16 by 16 array distributed over four memories).

Computation is done on progressively smaller submatrices in the lower right-hand corner of the array. The first panel of the figure shows the first iteration of the DO loop, in which the entire array is worked on in parallel by all the threads. The second panel shows the seventh iteration, by which time the threads assigned to memory 0 are completely idle, because none of the elements of the submatrix are stored in on memory 0. The third panel of the figure shows the eleventh iteration of the DO loop, by which time the threads assigned to memories 0 and 1 are idle. The fourth panel shows the fifteenth iteration, where only the threads assigned to memory 3 are working, with all the other threads idle. For most of time spent in the DO loop, one or more threads are left idle.

In contrast, a (*, CYCLIC) distribution uses threads on all four memories up until the point when only 3 out of 16 columns remain to be completed (see Fig. 9). This load balancing consideration makes a (*, CYCLIC) distribution the far better choice for this algorithm.

3.2.4. Element granularity vs. page granularity data layout

In some cases the data layout directives specify that two elements within the same page should be placed in different memory units. However, virtual memory hardware and software can only place entire pages in a memory unit. Consequently the compiler must either choose one memory in which to place both elements, or it must rearrange the order of elements in the virtual address space to keep the two elements on distinct pages. The former approach is called page-granularity layout because the granularity of the data being placed is a

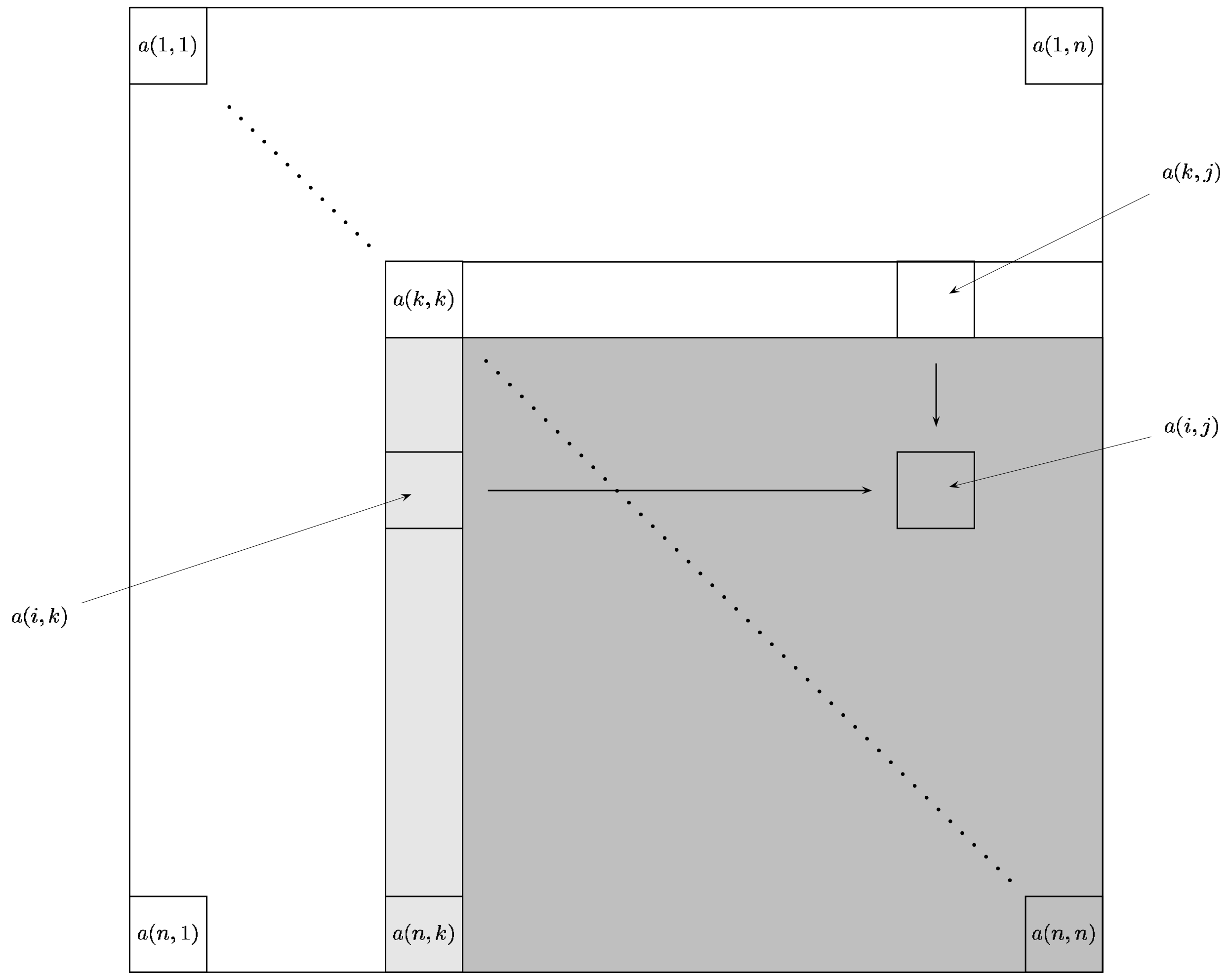

Fig. 7. Computations in the kth iteration of the outer LU decomposition loop.

page. The latter approach is called element-granularity layout because the granularity of the data being placed is the individual array element.

Some Fortran programs depend upon the standard ordering of array elements. This may occur when an EQUIVALENCE statement is used to map another symbol onto the storage for the array or when the array is passed as an argument in a call where the subroutine declares the dummy argument with a different shape. When arrays are used in this manner, the compiler is not permitted to rearrange the order of elements, thus element-granularity cannot be used.

When permitted, element granularity is preferred because it provides precise placement of data. This is important in cases where the extent of an array dimension is such that the ratio of the number of pages needed to store that dimension is small relative to the number of memories assigned to that dimension. This can occur when many dimensions of an array are distributed, and the number of memories on the target machine is large. Large page sizes exacerbate this problem. Finally, element granularity has a more efficient implementation, particularly for mapped local variables in subprograms, and it enables other optimizations which could not be performed while maintaining the original order of elements.

Element and page granularity data layout are described in more detail below in Section 4.3.

Compaq's new directives include NOSEQUENCE, used to enable element granularity layout for the specified arrays. Page granularity is the default. NOSEQUENCE can be used to override this default either for one or more arrays or for an entire program or subprogram.

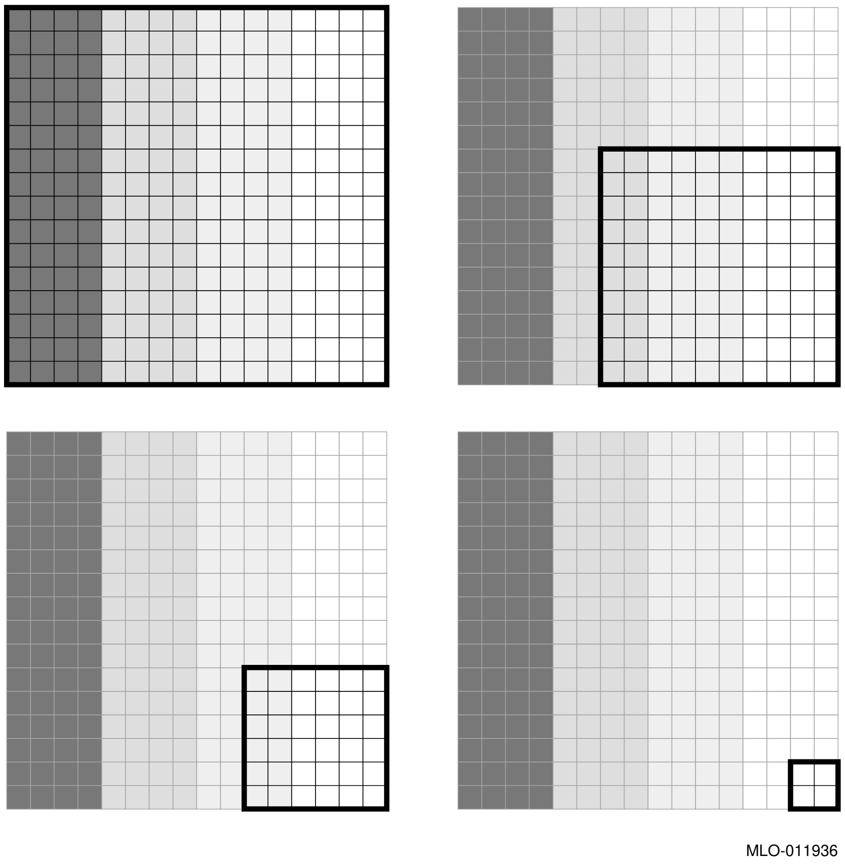

Fig. 8. LU Decomposition with (*, BLOCK) distribution.

3.2.5. Placing computations

The OMP NUMA directive immediately before an OMP PARALLEL DO loop (see Fig. 6) tells the compiler to schedule each iteration of the loop onto a thread that is associated with the memory containing the data to be used in that iteration.

In the previous examples, such a proper assignment of loop iterations to memory units is obvious. With more complicated loops, however, it is not always so obvious. For this reason, we have added the ON directive, which allows programmers to specify the desired assignment. Consider the following example:

```
integer, parameter:: (n = 1000)
real, dimension(n):: X, Y, Z, Q, R, S
!dee$ distribute (block):: X, Y, Z, Q, R, S
!dec$ omp numa
!$omp parallel do
do i = 1, n − 1
  !dec$ on home (x(i + 1))
  X(i + 1) = Y(i) + Z(i + 1)
  !dec$ on home (q(i))
  Q(i) = R(i) + S(i)
end do
```

In this example, there is no single obvious choice for an assignment of iterations to memory units. The ON HOME directives indicate that for the assignment to X, iteration i should execute on the module containing $X(i + 1)$, and that for the assignment to Q, iteration i should occur on the module containing $Q(i)$. The ON directive is not required by the language as compilers can determine some location for the computation, but it is useful for cases where the compiler's choice is not optimal.

3.2.6. Other directives

In addition to the MIGRATE_NEXT_TOUCH user-directed migration directive, the language also includes a MIGRATE_TO_OMP_THREAD directive. This directive causes a set of pages to be placed in the module executing the specified thread.

While much user-directed data layout can be performed simply with the DISTRIBUTE directive described previously, most real programs will need addi-

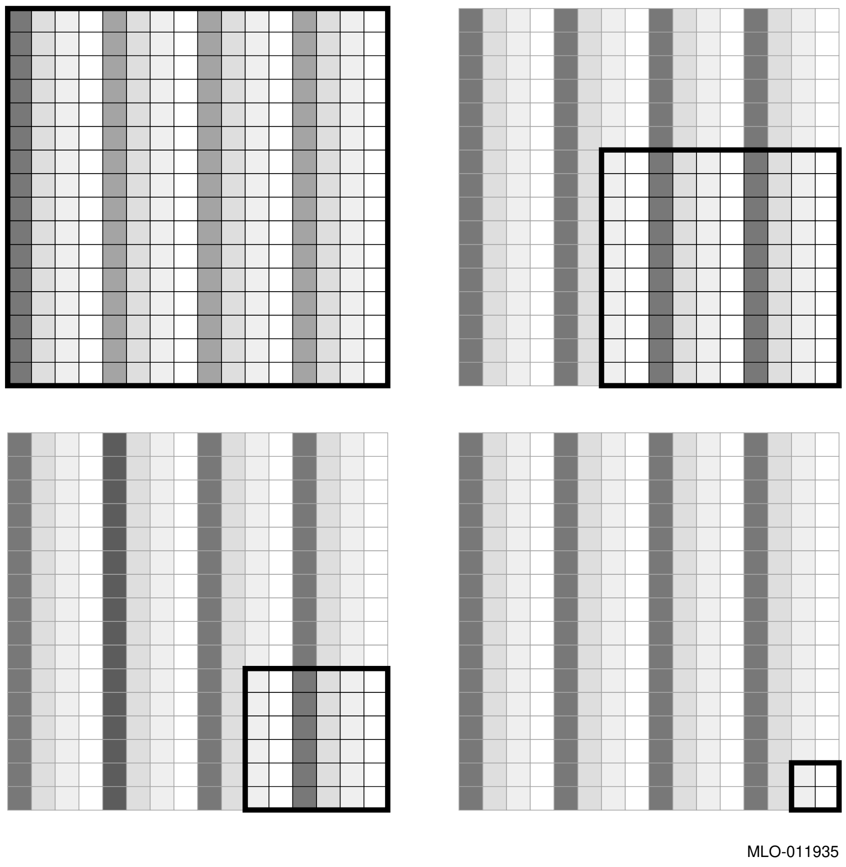

Fig. 9. LU Decomposition with (*, CYCLIC) distribution.

tional capabilities. Compaq Fortran for True64 Unix therefore includes the following additional directives:

SEQUENCE/NOSEQUENCE The SEQUENCE directive (which is the default) specifies that an array is page-granularity, rather than element-granularity. The NOSEQUENCE directive can be used to override this default for an array, or for a group of arrays, or for each array in a program unit.

MEMORIES The MEMORIES directive is equivalent to the HPF PROCESSORS directive. It provides a way to specify that the memories of the machine are to be thought of as an array, where the rank of the array and the extent in each dimension are given in the directive. For instance,

!dec$ memories M(4,5,4)
double precision, dimension (1000, 1000, 1000):: A
!dec$ distribute (**cyclic, block, block**) **onto** M:: A

ALIGN This directive can be used for the following purposes:

1. Aligning a 1-dimensional array with successive columns or rows of a 2-dimensional array. This could be useful, for instance, when calling a subroutine iteratively on the columns of an array. In such a case, the 1-dimensional dummy argument would be explicitly aligned by such a directive with a particular column, that column changing on each invocation of the subroutine.
2. Specifying that an array is *replicated.* A replicated array is an array that has an identical copy on each memory. It is useful for data that is written once (or seldom) and read often. Since corresponding elements on each memory must have the same values, it can be thought of as a privatized variable with shared semantics. It is the compiler's responsibility to make sure that any change to the array is reflected in every copy of the array.

The ALIGN directive can also be used to indicate that an array is *partially replicated.* For instance, the array might be distributed over one dimension of a 2-dimensional array of memories, and then replicated over the other dimension.

ON In general, the compiler decides how to distribute iterations of a NUMA PARALLEL loop nest over memories based on the mapping of an array within the loop nest. There are some cases in which the compiler may not have enough information to pick the right distribution of iterations. For instance, there may be more than one array, with slightly different mappings; or there may be an array with an indirection vector. In such cases, the ON directive may be used to tell the compiler exactly how to distribute the iterations over memories.

TEMPLATE A template is used to define a virtual array (i.e., one that takes up no storage) which can be mapped and with which other arrays can be aligned. Suppose for example, that we have a subroutine with three dummy arguments A, B, and C, each passed in as assumed shape arrays. We might have declarations like this at the top of the subroutine:

!dec$ template T(1000, 2000, 1000)
!dec$ distribute (**block, block, cyclic**):: T
double precision, dimension (:,:,:):: A, B, C
!dec$ align (i, j, k) **with** $T(i, j, k)$:: A, B, C

In this example, arrays A, B, and C may not have the same extents, since they are assumed shape dummies. However, the mapping directives guarantee that corresponding elements of A, B, and C are aligned – that is, they live on the same memory. For instance, $A(10, 15, 20)$, $B(10, 15, 20)$, and $C(10, 15, 20)$ (assuming all these elements exist) all live on the same memory.

This is a guarantee to the subroutine in which these dummies are declared. It is the responsibility of the compiler to make sure that these directives are honored and to remap the actual arguments if needed to ensure this.

REALIGN, REDISTRIBUTE These two directives can be used in the executable code to dynamically change the mapping of an array in the program. The compiler then generates code to move the elements of the array to their new specified locations.

Compaq Fortran also contains facilities to allow the passing of mapped arrays to functions and subroutines. Variations of the ALIGN and DISTRIBUTE directives allow the programmer to specify:

```
real, dimension(100,200) :: A, B
!$omp do parallel
do j = 1, 200
    !$omp do parallel
    do i = 1, 100
        A(i,j) = B(i,j)
    end do
end do
```

Fig. 10. A two-dimensional loop nest.

– that the mapping of a dummy argument is known to be mapped in the same way as the actual, or
– that the compiler must remap the incoming argument if necessary so that it has the specified mapping. (This was already mentioned above in discussing the TEMPLATE directive.)

Using these capabilities requires an explicit interface in the code. Such an explicit interface is most easily provided automatically by having the subroutine be contained in a module.

In addition, the INHERIT directive can be used to specify that a dummy argument is coming in with a mapping that will not be known until run-time.

4. Some compiler internals

This section is addressed to people with some knowledge of compiler technology and the techniques that compilers use to generate code for distributed-memory machines. Such techniques have been exhaustively studied, particularly in the context of High Performance Fortran. A general introduction to this area can be found in the paper [5]; this paper also establishes some of the terminology used internally in the Compaq Fortran compiler.

For the most part, this section of the paper therefore describes the changes that are made to distributed-memory processing to target a shared-memory NUMA architecture. It also describes some new optimizations that the compiler can introduce in this context.

4.1. Data space and iteration space

Let us use a simple example, shown in Fig. 10.

The notions of data and iteration space are represented as internal data structures in the compiler. They are used by the compiler to manage parallelism.

Each array has associated with it a *data space*, which holds information about the shape of the array and how the array is laid out over the memories of the machine.

In this example, the data space for both arrays A and B is two-dimensional, with extents 100 and 200. As yet, we have not stated anything about where the data is located.

Each computation (for instance, an addition, or an assignment) has an associated compiler data structure called *iteration space*, which represents the shape of the computation and where in the machine the computation is performed. In Fig. 10, the iteration space has the same rank and extents as the data space. In general though, this does not have to be the case. For instance, here is an example that initializes a boundary column of an array:

```
!$omp do parallel
do  $i = 1, 100$
  $A(i, 1) = 1$
end do
```

Here the data space of the array is 2-dimensional, but the iteration space of this computation is 1-dimensional. Another example is:

```
!$omp do parallel
do $j = 1, 100$
  !$omp do parallel
  do $i = 1, 200$
    $A(i, j) = B(i)$
  end do
end do
```

In this computation the data space of array B is 1-dimensional, but the iteration space of the computation is 2-dimensional.

4.2. Locating data and computations

The MEMORIES directive, described in Section 3.2.6, enables the programmer to specify to the compiler that the machine is to be regarded logically as a multi-dimensional array of memories, each with its own set of processors. The extent of this array of memories in each dimension is specified by this directive.

The compiler then has to associate elements of each mapped array with locations in the various logical memories. We refer to this process as "mapping an array". This mapping is based on information contained in the mapping directives, such as ALIGN and DISTRIBUTE, that apply to a given array. The mapping happens conceptually in two steps:

1. Data space is mapped coordinatewise onto an unbounded multidimensional array of virtual memories[1] using the information contained in a combination of ALIGN and TEMPLATE directives. In the common case, this mapping is implicit and no programmer directives are necessary. However, these directives are available when explicit control is needed.
2. The virtual memories are mapped coordinatewise onto the logical memories using information contained in a DISTRIBUTE directive.

Computations are similarly mapped onto these logical memories by the same two-step process, except that there are no ALIGN or DISTRIBUTE directives that apply directly to computations. Rather, the iteration space of a parallel computation is normally mapped by the compiler without explicit directives, based on the mappings of the data being accessed.

There are instances in which the compiler may not be sure which array to use in determining the mapping of the iteration space. As described in Sections 3.2.5 and 3.2.6, in such cases the programmer can insert an ON directive to specify the array to be used.

The relation of iteration space mapping to data space mapping is discussed in more detail in [5].

4.3. Element granularity versus page granularity

To show the effects of element granularity and page granularity mappings, let us assume that we have an array A declared as follows:

```
real $A$(5,5)
!dec$ distribute $A$ (block, block)
```

An element granularity mapping of the array A would lead to the data layout shown in Fig. 11. In this figure, we have pretended that that page size in our machine is 4 array elements in size. This is of course much smaller than any real page size, but we do this only to illustrate how pages are mapped to memories.

Note that there is some wasted memory space with element granularity mapping. This space consists of some empty locations "inside" the array and up to about a page in each memory.

The corresponding page granularity mapping is achieved by first laying out the array conceptually in array element order as in Fig. 12. Then one element from each page is picked, and the entire page is assigned to the physical memory to which that element would be mapped in an element granularity mapping. If we pick the first element of each page for instance, we get the layout shown in Fig. 13.

memory 0	memory 1	memory 2	memory 3
(1,1)	(4,1)	(1,4)	(4,4)
(2,1)	(5,1)	(2,4)	(5,4)
(3,1)	—	(3,4)	—
(1,2)	(4,2)	(1,5)	(4,5)
(2,2)	(5,2)	(2,5)	(5,5)
(3,2)	—	(3,5)	—
(1,3)	(4,3)	—	—
(2,3)	(5,3)	—	—
(3,3)	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—

Fig. 11. The array $A(5,5)$ distributed (**block, block**) over 4 memories, using element granularity with a page size of 4, with page boundaries indicated.

While this mapping causes many array elements to be assigned to a memory other than the one specified by the mapping directive, much of this problem is really due to edge effects and is greatly diminished as the ratio of the array dimensions to the page size increases.

We expect, in fact, that page granularity mapping will in many cases be quite acceptable. When converting old Fortran code to NUMA machines, it may be all that is needed. Nevertheless, it is in general true that better performance can be expected with element granularity than with page granularity.

4.4. Generating code

There are two main tasks involved in generating code for mapped objects: adjusting the subscripts and distributing loop iterations over threads.

4.4.1. Adjusting subscripts

An element granularity array may not have its elements stored in memory in array element order, as we have seen above. Therefore the subscripts in such an array need to be modified to reflect this. Let us take as an example an array A declared as in Fig. 14.

[1] In [5] virtual and logical memories are called virtual and logical processors, respectively.

(1,1)
(2,1)
(3,1)
(4,1)
(5,1)
(1,2)
(2,2)
(3,2)
(4,2)
(5,2)
(1,3)
(2,3)
(3,3)
(4,3)
(5,3)
(1,4)
(2,4)
(3,4)
(4,4)
(5,4)
(1,5)
(2,5)
(3,5)
(4,5)
(5,5)
—
—
—

Fig. 12. The array $A(5,5)$, in array element order, showing how array elements get allocated to pages when sequence and storage association are honored.

Here our machine consists of 30 memories arranged conceptually in a $2 \times 3 \times 5$ array. Each dimension of the array A is mapped to a corresponding dimension of that array. We have made the subscripts in the array 0-based for convenience in this example, but this is not necessary in general.

The first dimension of the array has extent 200. It is mapped BLOCK onto a memory array with extent 2 in the corresponding dimension. Therefore, for each fixed pair of values i_2 and i_3, the memory

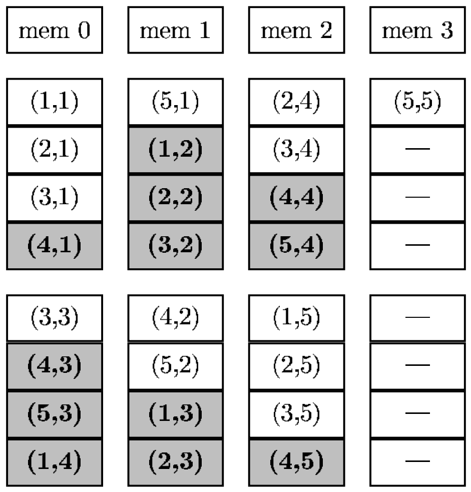

Fig. 13. The array $A(5,5)$ distributed (**block, block**) over 4 memories, but now with sequence and storage association honored, showing how pages are allocated to memories. The 12 shaded array elements are assigned to memories that are not those specified by the mapping directive.

```
real, dimension(0:199, 0:239, 0:299) :: A
!dec$ memories M(2, 3, 5)
!$omp distribute(block,block,block) onto M :: A
```

Fig. 14. A 3-dimensional array distributed (BLOCK, BLOCK, BLOCK) over a 3-dimensional array of memories.

holding element $A(0, i_2, i_3)$ holds the 100 elements $A(0{:}99, i_2, i_3)$, and another memory holds the 100 elements $A(100{:}199, i_2, i_3)$. We denote this value of 100 by *num_folds*$_1$. Similar calculations are performed in each dimension, giving us the values

$$num\_folds_1 = 100$$

$$num\_folds_2 = 80$$

$$num\_folds_3 = 60$$

The product of all the *num_folds/*$_j$ (in this case, 480000) is denoted by *num_folds* (without any subscript). This is the number of storage locations needed on each memory to hold the entire array A.

The memories in the machine are numbered from 0 to 29. We denote a particular memory number by the symbol $\overline{v}$.

This is explained further in [5]. A virtual memory address is denoted by v. Such an address is thought of as having a component $\overline{v}$ denoting which memory it refers to, and a second component $\hat{v}$ denoting the address in that memory. In a distributed-memory machine, $\hat{v}$ is broken up into coordinates that become the modified subscripts of the array, while $\overline{v}$ is used for generating message-passing code. In a (shared-memory) NUMA machine on the other hand, $\overline{v}$ and $\hat{v}$ need to be combined to form one global memory address. The analysis, however, is largely the same.

We can think of $\overline{v}$ as having 3 coordinates $(\overline{v}_1, \overline{v}_2, \overline{v}_3)$ corresponding to the three dimensions of the memory array. These coordinates are given simply by

$$\overline{v}_1 = \overline{v} \bmod 2$$

$$\overline{v}_2 = \left\lfloor \frac{\overline{v}}{2} \right\rfloor \bmod 3$$

$$\overline{v}_3 = \left\lfloor \frac{\overline{v}}{2*3} \right\rfloor \bmod 5$$

and $\overline{v}$ is retrieved from its coordinates simply by

$$\overline{v} = \overline{v}_1 + 2*\overline{v}_2 + 2*3*\overline{v}_3$$

With these conventions, the values $\overline{v}_j$ corresponding to an array element $A(i_1, i_2, i_3)$ in our example be computed simply as follows:

$$\overline{v}_j = \left\lfloor \frac{i_j}{num\_folds_j} \right\rfloor$$

An array element $A(i_1, i_2, i_3)$ then has its subscripts adjusted so it becomes $A(\lambda_1, \lambda_2, \lambda_3)$, where

$$\lambda_1 = i_1 \bmod num\_folds_1 + \overline{v} * num\_folds$$

$$\lambda_2 = i_2 \bmod num\_folds_2$$

$$\lambda_3 = i_3 \bmod num\_folds_3$$

If we were generating code for a distributed-memory machine, the term $\overline{v} * num\_folds$ in λ_1 would not appear. (The other terms represent the components of $\hat{v}$, as described above.) Each section of the array would be stored in the same set of virtual offsets on each of the distributed memories. Since this is a shared-memory machine, however, we have to ensure that these memory locations are distinct. Adding $\overline{v}{*}num\_folds$ to the first subscript of A does that. Of course, the page placement code for the array A then has to be written to map the pages of A to the correct memories, in conformance with this convention.

These subscripts can also be written as

$$\lambda_1 = i_1 - \overline{v}_1 * num\_folds_1 + \overline{v} * num\_folds$$

$$\lambda_2 = i_2 - \overline{v}_2 * num\_folds_2$$

$$\lambda_3 = i_3 - \overline{v}_3 * num\_folds_3$$

In a parallel construct that is local (i.e., each thread accesses only array elements living on its own memory) this second version is preferable since all the terms on the right except i_j will be determined by the memory and will be loop invariants. This avoids an expensive mod operation in the subscript calculation.

Of course the formulas for subscript transformations can be rather more complex, particularly if the programmer has used ALIGN statements. The formulas also change for different distributions such as CYCLIC or CYCLIC(n). But extending these subscript transformations to cover such cases is straightforward, if tedious, and the general idea remains true: the expressions that are generated are highly optimizable and cause no noticeable performance degradation.

4.4.2. Distributing loop iterations

The directive OMP NUMA (see Fig. 6) is used to indicate that the immediately following PARALLEL DO loop is to have its iterations assigned to threads in a NUMA-aware manner. Such a loop is called a NUMA loop.

In generating code for a NUMA loop, the compiler binds each thread to a memory of the machine. It distributes the iterations of the loop to these threads as if by a STATIC schedule. The particular schedule is determined by the mapping of the iteration space of the loop. This mapping is always taken to be an element granularity mapping – that is, the loop iterations are distributed exactly in accordance with the mapping directives. This is because, even if the data were mapped with page granularity, it would involve too much runtime overhead to make the distribution of loop iterations exactly match this mapping, as can be seen, for instance, in Fig. 13. Thus, even in a trivial assignment such as

```
!dec$ numa
!$omp parallel do
do $i = 1, n$
  $A(i) = i$
end do
```

where the iteration space is derived from the mapping of the array A, the assignments to $A(i)$ will only really be guaranteed to be local if A is given an element granularity mapping. If A is a page granularity array, some of the assignments to $A(i)$ will be executed by threads on remote memories.

Loop iterations are conceptually assigned in two stages:

Stage 1: The iterations are distributed over the memories of the machine.

Stage 2: The iterations assigned to each memory are distributed over the threads executing on that memory.

```
!dec$ omp numa
!$omp parallel do
do $i_3 = LB_3, UB_3, S_3$
    !dec$ omp numa
    !$omp parallel do
    do $i_2 = LB_2, UB_2, S_2$
        !dec$ omp numa
        !$omp parallel do
        do $i_1 = LB_1, UB_1, S_1$
            $A(i, j, k) = A(i, j, k) + T$
        end do
    end do
end do
```

Fig. 15. Example of nested parallel loops.

```
!$omp parallel
do $i_3 = LB_3^{loc}, UB_3^{loc}, S_3^{loc}$
    !$omp parallel
    do $i_2 = LB_2^{loc}, UB_2^{loc}, S_2^{loc}$
        !$omp parallel
        do $i_1 = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
            $A(\lambda_1, \lambda_2, \lambda_3) = A(\lambda_1, \lambda_2, \lambda_3) + T$
        end do
    end do
end do
```

Fig. 16. Parallel loop nest after stage 1. Note that this is just a view of the internal representation in the compiler. The PARALLEL DO directives have been replaced by PARALLEL directives, since the distribution of loop iterations is already being made explicit in the loop bounds.

Stage 1: Distributing iterations over memories In the first stage, the iterations are distributed over the memories of the machine. This is done by parametrizing the loop bounds for the iterations of the loop to be executed by a single thread by the number of the memory on which that thread is executing. Let us consider for example the loop nest in Fig. 15. Let us assume that the array A has the same mapping as we used before in Fig. 14.

Figure 16 shows how this first step transforms the loop nest, where the new bounds LB_j^{loc}, UB_j^{loc} and so on are expressions in terms of

– the mapping of the iteration space of the assignment, which is derived from the data space shape and mapping of the array A, and
– the value *my_memory*, which takes values in the interval

$$[0 \ldots \langle\text{number of physical memories}\rangle - 1]$$

If we assume that the iteration space of this loop nest is inferred from the distribution of the array A (as would certainly be the case in this example), then, the local iteration bounds are computed as follows:

$$LB_1^{loc} = (my\_memory \bmod 2) * 100$$
$$UB_1^{loc} = (my\_memory \bmod 2 + 1) * 100 - 1$$
$$S_1^{loc} = 1$$
$$LB_2^{loc} = \left(\frac{my\_memory}{2} \bmod 3\right) * 80$$
$$UB_2^{loc} = \left(\frac{my\_memory}{2} \bmod 3 + 1\right) * 80 - 1$$
$$S_2^{loc} = 1$$
$$LB_3^{loc} = \frac{my\_memory}{6} * 60$$
$$UB_3^{loc} = \left(\frac{my\_memory}{6} + 1\right) * 60 - 1$$
$$S_3^{loc} = 1$$

The modified subscripts, assuming element granularity, are

$$\lambda_1 = i_1 - \overline{v}_1 * num\_folds_1 + \overline{v} * num\_folds$$
$$\lambda_2 = i_2 - \overline{v}_2 * num\_folds_2$$
$$\lambda_3 = i_3 - \overline{v}_3 * num\_folds_3$$

With page granularity, the subscripts (in this example) are unchanged:

$$\lambda_1 = i_1$$
$$\lambda_2 = i_2$$
$$\lambda_3 = i_3$$

Of course, all these formulas depend on the particular mapping. For instance, if the mapping of A were (**block, cyclic, cyclic**(10)), or if A were given a non-trivial alignment, the formulas would look quite different. In addition, in order to manage **cyclic**(n) distributions, an additional loop would be generated for each dimension distributed **cyclic**(n). But there is nothing new in this – it is exactly what HPF compilers have traditionally done for distributed memory, with slight modifications only in how subscripts are treated.

Stage 2: Distributing iterations in a memory over threads In the second stage, the iterations assigned to each memory of the machine are distributed over the threads that execute on that memory.

Let us denote the number of threads per memory by tpm. We will assume that the memory number – the number returned by my_memory – is given by

```
!$omp parallel
do $i_3 = LB_3^{loc}, UB_3^{loc}, S_3^{loc}$
  !$omp parallel
  do $i_2 = LB_2^{loc}, UB_2^{loc}, S_2^{loc}$
    !$omp parallel
    do $i_1 = \overline{LB}_1^{loc}, \overline{UB}_1^{loc}, \overline{S}_1^{loc}$
      $A(\lambda_1, \lambda_2, \lambda_3) = A(\lambda_1, \lambda_2, \lambda_3) + T$
    end do
  end do
end do
```

Fig. 17. Parallel loops after stage 2: inner loop threadized.

$$my\_memory = \left\lfloor \frac{my\_thread()}{tpm} \right\rfloor$$

For instance, if $tpm = 4$, then threads 0–3 are assigned to memory 0, threads 4–7 to memory 1, and so on.

In our implementation all the threads in a memory are distributed at one loop level (rather than having the threads conceived as a multi-dimensional space whose dimensions are parceled out over various loop levels). Let us call this process "threadizing".

Figure 17 shows the generated code in Fig. 16 when the inner loop is threadized.

The new quantities $\overline{LB}_1^{loc}$, $\overline{UB}_1^{loc}$, and $\overline{STR}_1^{loc}$ can be computed in two different ways:

The first way is to distribute the iterations on a memory in a block fashion over the threads assigned to that processor:

$$\overline{LB}_1^{loc} = LB_1^{loc} + (my\_thread \bmod tpm) * \left\lceil \frac{\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor + 1}{tpm} \right\rceil * S_1^{loc}$$

$$\overline{UB}_1^{loc} = \min \begin{cases} UB_1^{loc} \\ LB_1^{loc} + \left((my\_thread \bmod tpm + 1) * \left\lceil \frac{\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor + 1}{tpm} \right\rceil - 1 \right) * S_1^{loc} \end{cases}$$

$$\overline{STR}_1^{loc} = S_1^{loc}$$

The second way is to distribute the iterations on a memory in a cyclic fashion over the threads assigned to that processor:

$$\overline{LB}_1^{loc} = LB_1^{loc} + S_1^{loc} * (my\_thread * tpm)$$

$$\overline{UB}_1^{loc} =$$

$$\begin{cases} \overline{LB}_1^{loc} + \left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{tpm * S_1^{loc}} \right\rfloor * tpm * S_1^{loc} \\ \text{if } my\_thread \bmod tpm \\ \leqslant \left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor \bmod tpm \\ \overline{LB}_1^{loc} + \left(\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{tpm * S_1^{loc}} \right\rfloor - 1 \right) * tpm * S_1^{loc} \\ \text{otherwise} \end{cases}$$

$$\overline{STR}_1^{loc} = tpm * S_1^{loc}$$

The code in the case that the outer loop is threadized is entirely similar. The only difference is that the bounds of the outer loop are modified, rather than those of the inner loop.

No modification of this needs to be made to handle imperfectly nested loops. The design handles such loop nests as is.

We decide which loop to threadize as follows: In general, the outermost NUMA loop is threadized. However, the programmer can specify that a loop has few iterations by giving it the "shortloop" attribute. Shortloops will not be threadized unless there is no alternative, and the compiler searches for the outermost non-shortloop to threadize. Since the NUMA loops form a forest, each tree has to be handled separately, and since shortloop declarations can be scattered throughout this tree, some care has to be taken in the compiler to handle this correctly. The actual algorithm used is as follows:

1. Perform a depth-first walk of each numa loop tree. Some of the loops are marked S ("shortloop"). On each path down from the root, mark the first non-S loop as M ("go multiple"), and don't continue the walk further down the tree at that node. On the way back up the tree, mark each ancestor of an M node as A ("above M").
2. Perform a second depth-first walk of the each numa loop tree. For each node N,
   a) if N is an M node, stop the recursion at this point.
   b) if N is an A node (and so in particular not an M node), recurse.
   c) if N is an S node but not an A node, mark it as M and stop the recursion at this point.

There are no other possibilities. For the root of the tree must be either an S node (and possibly also an A node) or an M node. If any other node N is unmarked, then it must not have been reached in the first walk, which means that there is an M loop above it, and this in turn means that it is not reached in the second walk.

Thus at the end of Stage 2, code has been generated that causes the threads to execute the NUMA loop nest in parallel, with each iteration executed by exactly one thread.

4.5. An HPF-style optimization for loop nests

Creating and destroying threads can be expensive. Therefore, our implementation creates one thread for each physical processor in the machine. Each thread is bound to its processor, and this binding persists throughout the program. OpenMP teams are created from these existing threads.

When a nest of parallel loops is encountered, the OpenMP language specifies that a new team starts up at each level of the nest. This can be somewhat inefficient. However, in many cases our compiler can determine that it would not change the observable semantics of the program if all the threads were started up at the outermost loop and the compiler simply managed those threads explicitly at each level. Here is an example:

```
real, dimension (n, n):: B
real, dimension (n):: A
!dec$ distribute (block, block):: B
!dec$ align A(i) with B(i, *)
!dec$ numa
!$omp parallel do
do i = 1, n
  A(i) = ...
  !dec$ numa
  !$omp parallel do
  do i = 1, n
    B(i, j) = ... A(i) ...
  end do
end do
```

The array A is partially replicated over the second dimension of B. In a straightforward OMP implementation, each iteration of the outer loop would be executed by exactly one thread. This thread, which assigns to $A(i)$ say, would have to assign to all the replicated versions of $A(i)$, and of necessity this would have to be done serially since it is all done by one thread. Each such thread would then spawn a sub-team which would execute the inner loop over j.

However, each thread executing an iteration of the inner loop *could have* executed the assignment to the replicated instance of $A(i)$ that it subsequently used. So we can obtain greater parallelism by starting up all the threads on entry to the outer loop. The assignments to all the replicated values of A are done in parallel, following which the inner loop just executes in parallel as it would in any case.

This optimization can also be useful even when there is only one loop in the nest. Here is an example:

```
real, dimension (n, n):: B
!dec$ distribute (block, block):: B
!dec$ numa
!$omp parallel do
do i = 1, n
  B(i, n) = 1
end do
```

In this example a slice of the array B is being initialized. A 1-dimensional team of threads could in principle be constructed to perform this initialization. The problem is that the set of threads that have data local to them (let us call this the "natural" set of threads for this loop) may not include the master thread because the master thread may be bound to a processor on a module not containing any data in this slice.

This natural set of OpenMP threads therefore could not by itself constitute an OpenMP *team* of threads because it does not include the master thread. So in this case, our compiler creates a team consisting of all the threads, and guards the assignment statement so that only those threads that live on logical memories owning elements of the slice $B(:, n)$ participate in the assignment statement.

4.6. Explicit stack management for element-granularity mapped arrays

Let us call an array declared in a subprogram that is not a dummy argument a *local* array. Thus, all automatic arrays are local arrays, but a local array might have constant bounds and therefore not be automatic. Unfortunately, this definition of "local" conflicts with the standard Fortran definition, but we use it only here.

Mapped local arrays have to be allocated on entry to the subprogram in which they are defined and deallocated on exit from that subprogram. Such arrays are typically allocated on the stack. Thus, on entry to the subprogram, code has to be generated to map the the virtual pages of the stack to the appropriate memories based on the mapping of the local variables. Furthermore, if two different subprograms contain local variables with different mappings, the virtual stack pages will have to be remapped on each entry. This is a time-consuming process. There is, however, a way that we can overcome this problem, at least in the case of element-granularity local arrays:

At the start of the program some storage is allocated on each memory. We can think of this as a collection of "local" stacks that together constitute a separate compiler-managed stack. Note that these separate "local" stacks are in global memory, not in thread-local storage; an array allocated over these stacks needs to be globally visible to all threads in the subprogram. For convenience in addressing, this collection of local stacks is allocated as a continuous range of global addresses.

On entry to each subprogram, each element-granularity mapped local array will have storage reserved for it in each of these local stacks. Only as much storage will be reserved in each local stack as is necessary to hold the local section of the array. Since the array is assumed to be element-granularity, we do not have to be concerned about the fact that the pages allocated to an array are not in array element order or that an array may not use up all of a page before skipping to a page on another memory.

Mapping directives are not honored for arrays in a scope that is entered from within a parallel region. This constraint means that each such stack will only be manipulated by one thread (i.e., the single thread that initially starts executing the program); as a result, there will be no problem maintaining the integrity of these stacks.

The main advantage of this technique is that the pages never have to be remapped. Once the original allocation has been done, the operating system is never involved in this stack management. Since we are managing this memory as a stack (or a set of stacks), allocation and deallocation are quite efficient.

A secondary advantage of this technique is that we do not waste any memory. These arrays are allocated even more efficiently than element-granularity mapped arrays that could not be placed on the stack.

5. Results

Preliminary performance measurments of our NUMA extensions have been gathered on an AlphaServer GS320 system running Version 5.1 of Tru64 UNIX. We measured several variants of a simple LU decomposition without pivoting. The pattern of access to the data appears in Section 3.2.3. This is a good example of a program in which the pattern of access to the data changes frequently.

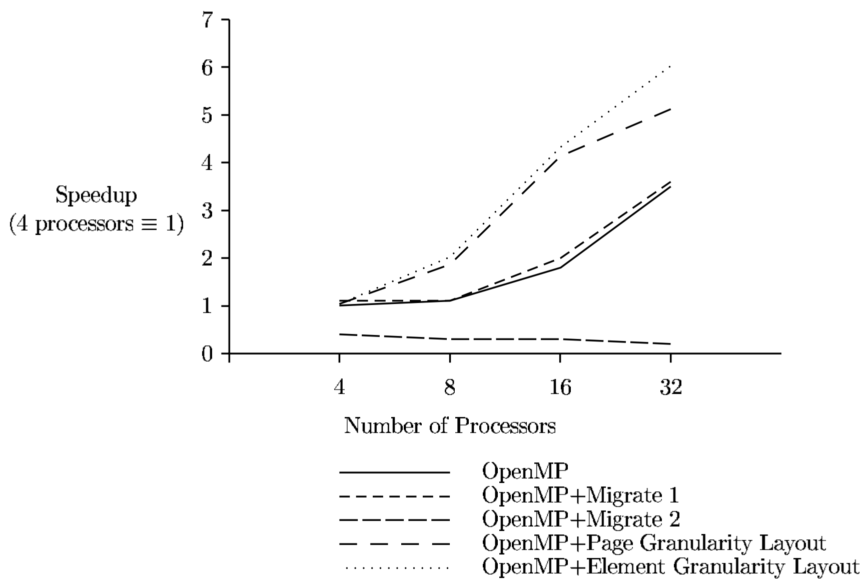

Fig. 18. Speedup for the LU example. Timing was measured for 4, 8, 16, and 32 processors. Speedup is relative to the 4 processor standard OpenMP time.

The variants of the program that were measured are referred to as follows:

OpenMP	The program with standard OpenMP directives only.
OpenMP+ Migrate	The standard OpenMP program with one MIGRATE_NEXT_TOUCH directive added. We measured two different versions, with the directive in different places.
OpenMP+ Layout	The standard OpenMP program with data layout directives added. We measured two different versions, one with page granularity and one with element granularity.

Figure 18 shows the performance of the four versions of LU that were measured. Notice that OpenMP+Layout significantly outperforms the standard OpenMP version. The data layout directives coupled with the NUMA-aware scheduling of iterations causes the workload to be spread across the threads while executing each iteration on a thread that is near the data needed by that iteration. Note also that as expected, element granularity performs somewhat better than page granularity, although both significantly outperform the other versions.

OpenMP+Layout also outperforms OpenMP+ Migrate. To understand this, consider the two places in the program where you could insert the MIGRATE_NEXT_TOUCH directive. The first place (which is denoted by Migrate 1) is outside the outer loop. This causes the pages to migrate near the thread that uses them on the first iteration and then remain there. However, the second and later iterations have many nonlocal references because the threads access different columns than they did on the first iteration. The same problem occurs regardless of the SCHEDULE clause specified on the PARALLEL DO directive.

The second place for the MIGRATE_NEXT_TOUCH directive (denoted by Migrate 2) is inside of the outer loop. Here the pages are migrated to the correct places on each iteration of the outer loop, and the references are all local. However, the overhead of migrating the pages on every iteration of the outer loop adds significantly to execution time.

6. Comparisons with other implementations

The Silicon Graphics Origin 2000 Fortran compiler [4] implements a set of data distribution directives supporting element-granularity layout (which they call "reshaped" distribution) and page-granularity layout (which they call "regular" distribution). In addition, there is a REDISTRIBUTE directive to move the pages containing an array to correspond to a different distribution. There is also an AFFINITY directive, essentially a combination of the ON HOME and OMP NUMA directives described above. Both the Fortran and C compilers support a directive like the migrate_to_omp_thread described above, which they call "page_place". The Origin 2000 compilers do not support the migrate_next_touch capability.

Compaq's new directives deal with the issues that arise when passing distributed arrays as arguments to subroutines and functions. The ALIGN directive, which specifies how to lay out arrays relative to other arrays, is needed to handle mapped array sections as actual arguments and for describing array replication; this directive is not part of Origin 2000 Fortran. The INHERIT directive facilitates writing library code which must accept arguments whose distributions vary from call to call. Origin 2000 Fortran does not use directives on page granularity dummy arguments; it clones the subroutine for each different distribution of actual arguments in the program.

Compaq's new directives also provide support for the full Fortran 95 language, including array sections, pointers, and derived types. The data layout directives also provide the necessary information to properly optimize the use of array syntax for NUMA machines.

7. Conclusions

The extensions to OpenMP described in this paper allow OpenMP programmers to easily write efficient code for NUMA architectures. User-directed page migration provides a simple way to place data in modules whose threads access it, and this migration will be effective for many real applications.

User-directed data layout, a more powerful capability, can do the following:

- It can place objects smaller than a page in specific memories.
- It allows better control of iteration scheduling.
- It can partially or fully replicate objects.
- It allows flexible handling of subroutine interface issues.
- It has certain implementation efficiencies not available with user-directed page migration.

The performance results show the effectiveness of both approaches. The directives, in either case, do not add significant complexity to the source code.

References

[1] OpenMP Architecture Review Board, OpenMP Fortran Application Program Interface, Version 1.1, Available online at http://www.openmp.org/specs/mp-documents, November 1999.

[2] International Organization for Standardization and International Electrotechnical Commission, *International Standard Programming Language Fortran*, ISO/IEC 1539-1:1997, December 1997.

[3] High Performance Fortran Forum, High Performance Fortran language specification, Version 2.0, Available from the Center for Research on Parallel Computation, Rice University, Houston, TX, and on-line as http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/files/hpf-v20.ps.gz, January 1997.

[4] Silicon Graphics Inc, MIPSPro Fortran 90 Commands and Directives Reference Manual, Document number 007-3696-003.

[5] C.D. Offner, A data structure for managing parallel operations, in: *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume II: Software Technology*, IEEE Computer Society Press, Available on-line at http://www.cs.umb.edu/˜offner, January 1994, pp. 33–42.