OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK libraries

Dr. C. Addison, Dr. Y. Ren and Dr. M. van Waveren
Fujitsu European Centre for Information Technology,
Hayes, UK

Abstract: Dense linear algebra libraries need to cope efficiently with a range of input problem sizes and shapes. Inherently this means that parallel implementations have to exploit parallelism wherever it is present. While OpenMP allows relatively fine grain parallelism to be exploited in a shared memory environment it currently lacks features to make it easy to partition computation over multiple array indices or to overlap sequential and parallel computations. The inherent flexible nature of shared memory paradigms such as OpenMP poses other difficulties when it becomes necessary to optimise performance across successive parallel library calls. Notions borrowed from distributed memory paradigms, such as explicit data distributions help address some of these problems, but the focus on data rather than work distribution appears misplaced in an SMP context.

1 Introduction

The BLAS and LAPACK libraries [1] are widely used by scientists and engineers to obtain good levels of performance on today's cache-based computer systems. Distributed memory analogues, the PBLAS and ScaLAPACK [2] have been developed to assist people on this type of parallel system. Shared memory (SMP) variants tend to only be available from hardware vendors (e.g. Intel's Math Kernel Library, [3]) or from library companies such as NAG Ltd. or Visual Numerics Ltd.

Consistent with this pattern, Fujitsu recently released its first SMP version of the parallel BLAS and LAPACK libraries for its PRIMEPOWER series. What makes this release interesting is that it was written exclusively using OpenMP, rather than a special purpose thread library. In the course of developing these libraries, several issues arose concerning OpenMP. Some of these issues can be handled by careful use of OpenMP directives. Other issues reveal weaknesses in the Version 1.1 specification that are addressed in the Version 2 specification. Still other issues reveal weaknesses that are inherent in this approach to parallelisation and may be difficult to resolve directly.

In the rest of this paper, we present a brief overview of the SMP environment on the Fujitsu PRIMEPOWER. We then discuss some of the basic library issues surrounding the BLAS and how we have resolved these using OpenMP. Parallel performance in LAPACK routines is often obtained through a sequence of calls to parallel BLAS and by masking sequential computations with parallel ones. The latter requires splitting thread families into groups. The OpenMP Version 1.1 specification [8] does not support this particularly well. OpenMP Version 2 [9] has better support and some attractive extensions have been proposed (e.g. the suggestions in [4]) that make this approach simpler. Finally, many LAPACK routines have kernels that consist of a sequence of consecutive BLAS calls within a loop. When these calls operate on just vectors, or perform matrix-vector type operations, they are sensitive to the migration of data from one processor's cache to another and by the overheads that result from making each BLAS call a separate parallel region. Avoiding such overheads is not always possible and the paper concludes by examining some of the limitations that are inherent to OpenMP.

2 SMP programming on the Fujitsu PRIMEPOWER

The Fujitsu PRIMEPOWER is an SMP system that supports up to 128 processors in a Solaris environment [6]. The current processor employed is the SPARC64 IV. This is a SPARC V9 architecture compliant processor that is similar to Sun's UltraSPARC processor series. The SPARC64 IV contains extensive support for out-of-order and speculative execution. It has a fused floating-point multiply-add as well as a separate floating-point add pipeline. Each processor has 128 Kbytes of data cache and a similarly sized instruction cache. There is also a unified second level cache of 8 Mbytes. In September 2001, the top clock speed of these processors was 675 MHz, so the achievable peak performance is 1350 Mflop/s.

Multi-processor systems are built from system boards that have up to 4 processors and 16 Gbytes of memory. There is a maximum of 8 such boards in a node (cabinet) and then up to 4 nodes can be connected together via a high-speed crossbar. The

system has nearly uniform memory access across its potential 512 GBytes of memory. As the SPEC OpenMP benchmarks, [7], show, it is possible to obtain parallel speed-ups using the full 128 processor configuration on non-trivial applications.

The parallel programming environment is provided by Fortran and C compilers that support OpenMP Version 1.1[1]. Both compilers also have extensive support for the automatic parallelisation of user codes. Fujitsu's *Parallelnavi* batch environment, accessible via NQS, binds threads to processors, processors to jobs and provides support for 4 MByte pages. These all reduce performance variations relating to system and other user activity. Therefore, provided there are one or two processors available for systems' use and for handling basic interactive facilities, user jobs run on effectively dedicated processors.

3 Designing OpenMP parallel BLAS

One of the challenges in providing parallel BLAS and LAPACK routines is that most BLAS routines contain assembler kernels. Therefore OpenMP parallelism must lie outside of these kernels. This effectively introduces yet another level of blocking within the routines. The practical aspects of this and related issues are illustrated by the general matrix by matrix multiplication routine `dgemm`. This family of multiplications also forms the kernel around which all the other matrix-matrix BLAS operations are constructed, see [10].

The basic operation that `dgemm` supports is: $\mathbf{C} \leftarrow \alpha\,\mathbf{AB} + \beta\,\mathbf{C}$, where **C** is a general *m* by *n* matrix, **A** is a general *m* by *k* matrix and **B** is a general *k* by *n* matrix. Both α and β are scalars. In addition either **A** or **B** can be transposed, with a consistent change in dimensionality. Each member of this family of four operations is highly parallel. When *m* and *n* are sufficiently large, an effective solution is to partition the problem into an appropriate number of sub-matrices and perform each sub-matrix multiplication in parallel. With 4 threads one partition would be

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

This then leads to:

$$C_{11} = \alpha A_{11} \times B_{11} + \alpha A_{12} \times B_{21} + \beta C_{11},$$
$$C_{12} = \alpha A_{11} \times B_{12} + \alpha A_{12} \times B_{22} + \beta C_{12},$$
$$C_{21} = \alpha A_{21} \times B_{11} + \alpha A_{22} \times B_{21} + \beta C_{21},$$
$$C_{22} = \alpha A_{21} \times B_{12} + \alpha A_{22} \times B_{22} + \beta C_{22},$$

where these sub-matrix operations are independent of one another and can be performed by separate calls to the sequential `dgemm` on different threads. The only challenge is to ensure that the number of threads allocated to a dimension is proportional to *m* and *n* and that the sub-blocks are large enough that near peak sequential performance is obtained. Since OpenMP has no equivalent to the High Performance Fortran (HPF) notion of processor arrays with shape [11], the library writer must map the 2-D thread partitioning onto the 1-D array of thread identifiers. This is not difficult, but the mapping clutters the code and makes it slightly harder to maintain. This is particularly relevant when one recalls that this mapping must be performed separately for each variant of the operation because the partitioning of the matrices **A** and **B** across sequential calls depends on whether they are transposed or not.

Performance of `dgemm` on the PRIMEPOWER is good. Single processor performance using a 562.5 MHz system on 500 by 500 to 1000 by 1000 matrices is around 1 Gflop/s. On 16 processors, the performance is around 12 Gflop/s on the same sized problems and on 64 processors, the performance of `dgemm` on 500 by 500 to 1000 by 1000 matrices is around 32 Gflop/s.

The strategy of partitioning BLAS operations into a series of independent sequential BLAS calls has proven effective. However, the performance of the matrix-vector and vector-vector BLAS routines is sensitive to whether the matrix was already in cache (the "hot-cache" case) or not (the "cold-cache" case). This will be discussed in more detail at the end of this paper.

4 Building OpenMP LAPACK routines on top of OpenMP BLAS

One of the design decisions in LAPACK was to make extensive use of the matrix-matrix BLAS in order to block computations and thereby make better use of data in cache, [1]. It was also felt, with some justification, that the performance of major LAPACK computation routines would improve simply from the use of SMP versions of the BLAS routines. While the operations performed between matrix blocks tend to parallelise well, the operations performed within blocks tend to be sufficiently fine grain that performance is mediocre sequentially and scales poorly.

A classical way to remove such sequential bottlenecks is to overlap the sequential computations on one processor with different parallel computations performed by the remaining

[1] Version 2.0 Fortran will be available in 2002.

processors.

Consider the pseudo-code for the main block of the LU-decomposition routine `dgetrf` as shown in Figure 1.

```
          do j = 1, min( m, n ), nb
             jb = min( min( m, n )-j+1, nb )
 *
 *           Factor diagonal and subdiagonal blocks and test for exact
 *           singularity.
 *
             call dgetf2( m-j+1, jb, a( j, j ), lda, ipiv( j ), iinfo )
 *
 *           Adjust info and the pivot indices. (Code not shown!)
 *
 *
 *           Apply interchanges to columns 1:j-1. (Code not shown!)
 *
             if( j+jb.le.n ) then
 *
 *              Apply interchanges to columns j+jb:n. (Code not shown!)
 *
 *
 *              Compute block row of U.
 *
                call dtrsm( 'left', 'lower', 'no transpose', 'unit', jb,
      $                     n-j-jb+1, one, a( j, j ), lda, a( j, j+jb ),
      $                     lda )
                if( j+jb.le.m ) then
 *
 *                 Update trailing submatrix.
 *
                   call dgemm( 'no transpose', 'no transpose', m-j-jb+1,
      $                        n-j-jb+1, jb, -one, a( j+jb, j ), lda,
      $                        a( j, j+jb ), lda, one, a( j+jb, j+jb ),
      $                        lda )
                end if
             end if
          end do
```

Figure 1 - Pseudo-code for `dgetrf`

The operations performed within `dgetf2` and the pivot updates are best performed on a single processor. The routines `dtrsm` and `dgemm` are BLAS routines that operate on large parts of the matrix and that tend to perform well in parallel. Observe that the first `nb` columns of the trailing matrix will be the panel used for factorisation with the next value of `j`. Therefore this factorisation could be overlapped with the remainder of the update of the trailing matrix. Indeed, it is possible to do better than this, as is shown in Figure 2 with a segment of a variant of `dgetrf` containing OpenMP directives.

It is useful to distinguish between the names of the sequential BLAS called from within a parallel region (as in Figure 2) and the parallel BLAS called from a sequential region (as in Figure 1), but the functionality of the routines is identical. The pseudo-code in Figure 2 allows the factorisation of the second and subsequent panels to be overlapped with the updating of the remainder of the matrix. The code will only work if thread 0 updates at least `A(j:m,j:j+jb-1)` prior to factoring this same block. Further notice that the partitioning in the `dl_gemm` call is only over columns, which will limit scalability on small to medium problems.

When the problem size is large enough and the number of threads small enough for a column decomposition to be appropriate, then the computation performed in thread 0 can almost be totally masked by the operations in the other threads. Consider using 4 threads and a problem size of 1000. A paper and pencil study that only considers floating point operations suggests that the parallel BLAS leads to a speed-up of around 2.3. Overlapping the panel LU with matrix multiplication leads to a theoretical speed-up of around 3.8. Measured speed-up is less because of OpenMP overheads and because the rate of floating point computation is also relevant. On a 300 MHz PRIMEPOWER, a speed-up of 2.3 on 4 processors has been observed on the 2000 by 2000 system using parallel BLAS only, which rises to 3.6 when overlapping is also used. These are compared against the base LAPACK code with tuned BLAS.

```
        jb = min( min( m, n ), nb )
        call dgetf2( m, jb, a( 1, 1 ), lda, ipiv, info )

        jmax = min((n/nb-1)*nb,m)

*$OMP PARALLEL default(shared) private(range_n,i,low_n,jb)
         n_pmax = omp_get_num_threads()-1

         do j = nb+1, jmax, nb

          jb = min(jmax-j+1, nb )
*$OMP DO schedule(static)
          do i_n=0,n_pmax
*
*          Compute range_n and low_n for each value of i_n (Not shown)
*
           call dlaswp( range_n, a( 1, low_n ),lda,j-nb,j-1,
     $                      ipiv, 1 )

           call dl_trsm( 'left', 'lower', 'no transpose', 'unit',
     $                   nb, range_n,one, a( j-nb, j-nb ), lda,
     $                   a( j-nb, low_n ), lda )
*
           call dl_gemm( 'no transpose', 'no transpose', m-j+1,
     $                       range_n, nb, -one, a( j, j-nb ), lda,
     $                       a( j-nb, low_n ),lda,one,
     $                       a( j, low_n ),lda )

           if (i_n .eq. 0) then

            call dgetf2(m-j+1, jb, a( j, j ), lda, ipiv( j ), iinfo )
*
*           Adjust INFO and the pivot indices. (Code not shown!)
           end if
          end do
*$OMP END DO
*$OMP MASTER
           call dlaswp(nb, a(1,j-nb),lda, J, J+JB-1, IPIV, 1 )
*$OMP END MASTER

         end do
*$OMP END PARALLEL
*
*        Finish by updating then factoring a(jmax+1:m,jmax+1:n)
*
```

Figure 2 - OpenMP overlapped `dgetrf`

A comparison of performance over a wider range of problems is shown in Figure 3. When only floating point operations are considered, this strategy appears effective, so that good performance on 8 threads is possible on the 1000 by 1000 problem.

Improving scalability further runs into limitations of the OpenMP Version 1.1 specification. It also makes the code much more complicated. Effectively, three work groups of threads are desired. The first group contains thread 0. The second group (empty in the code of Figure 2, but probably just 1 or 2 threads) consists of threads that update a part of `A(j:m,j:j+jb-1)` and then proceed to update a part of `A(j:m,j+jb:n)`. The third group of threads just updates a portion of `A(j:m,j+jb:n)`. The goal is to distribute the matrix update across *all* threads subject to the constraint that thread 0 has additional processing to perform when factoring `A(j:m,j:j+jb-1)`.

When there are a larger (say 16 or more) number of threads, it is desirable to partition the matrix multiply performed with the main thread group by both rows and columns. This is only possible if the operations can be synchronised properly. For instance, before a part of the matrix multiplication can be performed, all earlier operations (e.g. the call to `dl_trsm` to update the sub-matrix that will form **B** in the subsequent matrix multiply) must have updated all the relevant parts of the sub-matrices. A block of columns that is partitioned

among several threads for the matrix multiply will be composed from several column blocks that were updated independently in the previous call to `dl_trsm`. Therefore explicit synchronisation is required.

5. The second thread group would perform the `dl_gemm` call over independent rectangular regions that covered its portion of the trailing matrix. Barrier synchronisation is required to prevent threads making `dl_gemm` calls before the relevant `dl_trsm` call has been completed.
6. The first thread group would first perform its `dl_gemm` calls over the first `nb` columns of the trailing matrix. The thread group would then split into two, with the group's master thread performing the panel LU decomposition over the first `nb` columns while the remaining threads made `dl_gemm` calls to update its portion of the trailing matrix.
7. The nested regions would end and there would be a barrier synchronisation performed between the two "outer" threads.

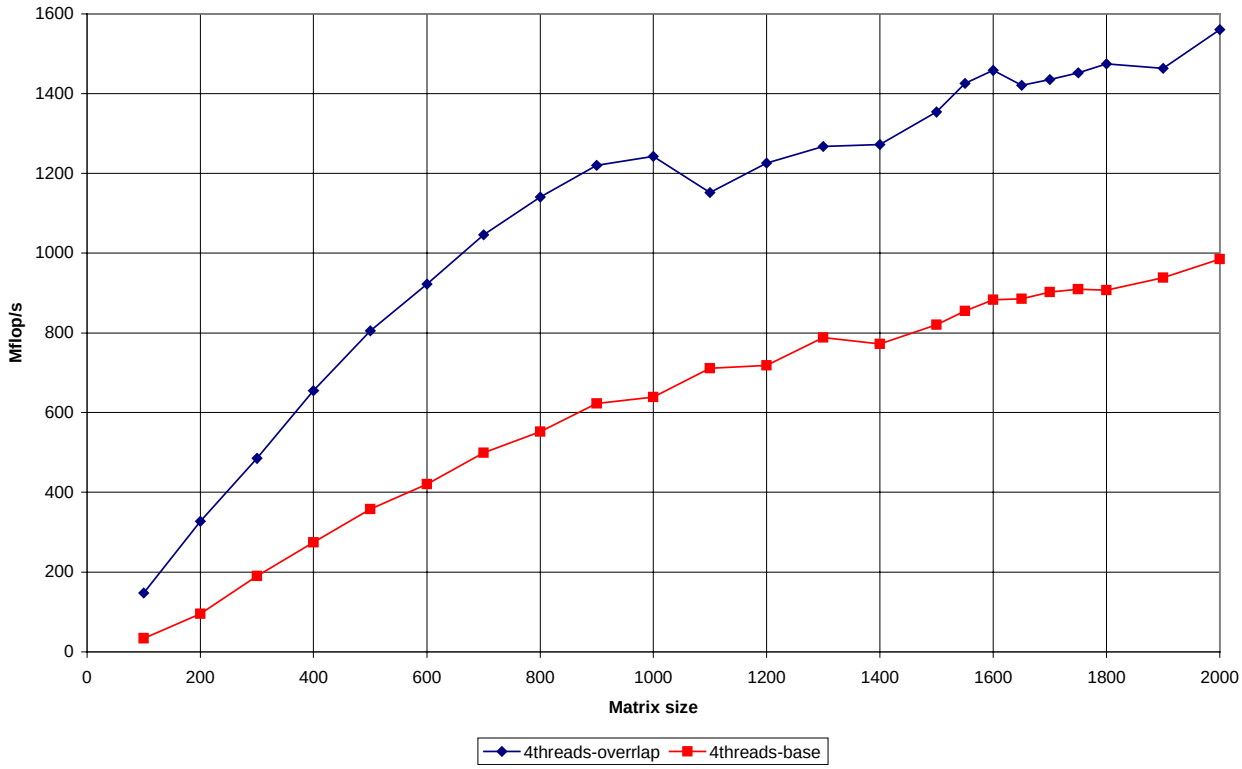

Figure 3 - Tuned versus base parallel dgetrf on a 4 processor 300 MHz SPARC64 III system

With the OpenMP Version 1.1 specification, this is only possible through the use of several sets of locks.

OpenMP Version 2 removes the need for locks because it allows the number of threads used within a parallel region to be specified at run time and barrier synchronisation can be used within a parallel region. The specification of the 3 thread groups mentioned above could be performed as follows:

1. At the outer most parallel region the `NUM_THREADS` attribute is set to 2.
2. The row and columns indices over which the different thread groups operate are determined.
3. The two nested parallel regions are defined.
4. Calls to `dlaswp` and `dl_trsm` are performed. Partitioning would be over columns only. The first thread group would take at least the first `nb` columns of the trailing matrix. The second group would take the balance of the columns.

The pseudo-code that implements the above steps is shown in Figure 4. The code has been simplified by hiding the computation of the partition values and by using a parallel do loop to differentiate the actions taken by the two outer threads. Also note the implicit assumption in step 5 that the column partitionings used for `dlaswp` and `dl_trsm` do not lead to any dependencies between the 2 thread groups when calls to `dl_gemm` are performed.

```
        jb = min( min( m, n ), nb )
        call dgetf2( m, jb, a( 1, 1 ), lda, ipiv, info )

        jmax = min((n/nb-1)*nb,m)

*$OMP PARALLEL default(shared) private(range_n,low_n,range_m,low_m,m_th,nth)
         mn_th = omp_get_num_threads()-1

         do j = nb+1, jmax, nb
          jb = min(jmax-j+1, nb )
*         Determine m_th (number row blocks), n_th (number column blocks),
*          tot_th(0), tot_th(1) (total number of threads in each group)

*$OMP DO schedule(static)
          do i_n=0,1
*$OMP PARALLEL NUM_THREADS(tot_th(i_n))
*
*          Determine range_n, low_n for the following 2 operations
*
            call dlaswp( range_n, a( 1, low_n ),lda,j-nb,j-1,
     $                   ipiv, 1 )
            call dl_trsm( 'left', 'lower', 'no transpose', 'unit',
     $                   nb, range_n, one, a( j-nb, j-nb ), lda,
     $                   a( j-nb, low_n ), lda )
*$OMP BARRIER
*          Determine range_n, low_n, range_m, low_m for dl_gemm

           call dl_gemm( 'no transpose', 'no transpose', range_m,
     $                     range_n, nb, -one, a( low_m, j-nb ), lda,
     $                     a( j-nb, low_n ),lda, one,
     $                     a( low_m, low_n ),lda )

           if (i_n .eq. 0) then
*$OMP BARRIER
*           Reset values of partition variables (low_n etc.) for special phase
*$OMP MASTER
            call dgetf2(m-j+1, jb, a( j, j ), lda, ipiv( j ), iinfo )
*
*           Adjust INFO and the pivot indices.
            range_m = 0; range_n=0; low_m=m; low_n=n
*$OMP END MASTER
            call dl_gemm( 'no transpose', 'no transpose', range_m,
     $                     range_n, nb, -one, a( low_m, j-nb ), lda,
     $                     a( j-nb, low_n ),lda, one,
     $                     a( low_m, low_n ),lda )

           end if
*$OMP END PARALLEL
          end do
*$OMP END DO
*$OMP MASTER
           call dlaswp(nb, a(1,j-nb),lda, J, J+JB-1, IPIV, 1 )
*$OMP END MASTER

         end do
*$OMP END PARALLEL
*
*        Finish by updating then factoring a(jmax+1:m,jmax+1:n)
```

Figure 4 - Simplified pseudo-code for `dgetrf` using OpenMP Version 2

Similar scalable performance issues can be found in many other LAPACK routines. Essentially the difficulty is that the Version 1.1 OpenMP specification does not offer sufficient flexibility in the way in which thread groups can be defined. At present, synchronising the activity of a sub-set of threads in a parallel region requires the use of locks, which can become very cumbersome. This problem has not been resolved in the Version 2.0 specification with recursive algorithms, which are becoming increasingly important in linear algebra operations such as factorisation, see [12]. There are also new algorithms to support, such as the divide and conquer algorithms for the symmetric tridiagonal eigenvalue problem and singular value

decomposition included in LAPACK Release 3.0. Possibly something like the thread groups proposed in [4] or the work queues proposed in [5] might be required in a future OpenMP specification.

Another difficulty with writing efficient OpenMP LAPACK routines relates to the overheads associated with several successive calls to parallel BLAS routines within one loop of an LAPACK routine. For instance, in the main loop of the symmetric tridiagonalisation routine `dlatrd` there is a sequence of 5 calls to matrix-vector BLAS, followed by 3 calls to vector BLAS. Each of these creates its own parallel region and each defines how many threads are appropriate for the operation and how work is partitioned among these threads. In a given call to `dlatrd`, this sequence of calls of BLAS routines is executed about 64 times, so that at least 512 different parallel regions are created. Even though the overheads associated with creating a new parallel region are low, the accumulated overheads of this many different regions impact the performance on smaller problems.

The current solution to this problem is to create special "in-parallel" versions of the relevant BLAS routines. These routines are written assuming that a parallel region has already been created (i.e. they are using "orphaned directives"). It is then possible to have only one parallel region for the entire calling routine. While this reduces the overheads of creating the parallel regions, there is no mechanism within OpenMP by which the partitioning of work among threads within these various routines can be organised to maximise the reuse of data that is already in the cache of particular processors. This can be a major performance difficulty.

5 Desired: OpenMP standards to support cache reuse

Cache reuse is related to the discussion of controlling data distribution on NUMA systems in OpenMP, see [13] and [14], but it is not identical. For example, the cache line (typically 64 bytes) is the important unit of ownership on a uniform memory access (UMA) system like the PRIMEPOWER while the page (typically 8192 bytes or larger) is the more important on NUMA systems. In many scientific applications, data is statically defined within pages so on NUMA systems this induces a data distribution across processors. Latencies to access remote data elements are orders of magnitude higher than access latencies to local data elements and the memory hierarchy is such that exploiting locality is critical as is communicating blocks of data to amortise the remote memory access cost. In such situations, it is helpful to treat distribution as an attribute of the data. This has been proposed as a model for OpenMP, see [13], and is also a useful model in a distributed memory / shared index space environment for languages such as HPF.

In a UMA environment or in a NUMA system with effective dynamic page management, there is a more dynamic and much finer grain view of data ownership by processors. Rather than distribution being an attribute of the data, it might be more useful to regard the partitioning of index spaces among threads as an attribute of the operator, with data residing in a cache line on a particular processor being a side effect. The objective in this setting is to minimize the differences in index space partitions between successive parallel loops. Alternatively, if the differences in index space partitions are known between parallel loops, a less demanding objective is to define a prefetch strategy that allows the cache-resident data to be loaded consistent with the second index space partition while executing over the first index space partition. The latter approach complements dynamic page management on a NUMA system.

The performance problems due to different partitionings and hence data lying in the "wrong" cache can be severe. Consider the code fragment for a parallel rank-1 update. This is the core operation performed in the BLAS routine `dger`.

```
*$OMP PARALLEL DO schedule(static)
*$OMP& default(shared) private(i)
      do j=1,n
        do i = 1,m
          a(i,j) = a(i,j) + x(i)*y(j)
        end do
      end do
*$OMP END PARALLEL DO
```

This is a highly parallel operation that should scale well for a range of problem sizes. However, parallel performance is heavily dependent on what precedes and follows this parallel region. For instance, if the array `a` is defined immediately beforehand using a single thread and the array is small enough that it can fit into that processor's Level 2 cache then parallel performance will be terrible. It will be faster to perform the update on the original processor.

If the array is defined in an earlier parallel region using a partitioning similar to that used in the above code fragment, and if the matrix fits into the collective Level 2 caches of the processors involved, then parallel performance will be excellent.

Optimal cache use cannot be determined just from information about current data locality and the next

operation to be performed. Suppose that several consecutive rank-1 updates were performed after the array had been initialised in a sequential section and that the array was small enough that it would fit into the collective Level 2 cache of the processors involved. A local decision to maximize cache reuse by limiting parallelism to a single thread would be the right decision with a single rank-1 update, but it would certainly be the wrong decision if there were 50 updates.

The rank-1 update also provides an example of how information from the calling program to the called routine can improve cache reuse. It is possible for the rank-1 update to be parallelised across both array dimensions and so that the actual partitioning used could be chosen to fit well with the partitionings in earlier and subsequent parallel sections while still using all available threads.

The adaptability of the parallel rank-1 update (as well as that of matrix multiplication and several other operations) suggests that HPF-style data distribution directives would be useful. If the data has been distributed among processes[2] sensibly, then many parallel BLAS routines will work well just by inheriting this distribution. However, this thinking misses the critical point – these routines are called from within larger applications and it is when defining effective data distributions for these applications that the limitations of static data distributions become clear.

5.1 Data distribution directives – cache reuse at a price

Consider LU decomposition as discussed in Section 4. With static data distributions using HPF or MPI, this application requires a doubly block-cyclic data distribution, see [2] for a justification. The blocking factor needs to be consistent with that required for good performance from the single processor matrix multiplication. The cyclic distribution is required to provide a degree of load balance among the processes. The block cyclic distribution is performed over both rows and columns of the matrix in order to have scalable matrix multiply performance. However, the induced 2-D process grid forces the "panel" factorisation (corresponding to calling `dgetf2` in Figure 1) to be performed over multiple processes. This reduces performance except on very large systems. The block cyclic data distribution also makes it difficult to overlap this factorisation with updates to the rest of the matrix. The data blocks are the same size, but the amount of computation required over the sub-group of blocks in the current panel is larger because of the panel LU factorisation. Furthermore, this sub-group changes as the computation proceeds.

Compare these difficulties with those involved in performing LU decomposition with OpenMP. If the panel factorisation is not overlapped with other computation, then the code in Figure 1 becomes parallel by providing parallel BLAS. The matrix multiply will be partitioned over both row and column indices in the call to `dgemm`, so asymptotically the code will behave well, but panel factorisation will be a bottleneck for many practical problem sizes. The panel factorisation can be overlapped with other computation, as shown in Figure 2. Performance is now limited by the 1-D partitioning in the matrix multiply. An optimal matrix-multiplication that keeps the work distributed evenly among threads can be combined with the overlap of the panel factorisation using locks or Version 2.0 features as in Figure 4. The resulting code will work well on a range of problem sizes with a large number of threads.

To summarize, data distribution directives are useful on distributed memory systems. Data distribution directives also promote cache reuse. The performance benefits from better cache reuse can be more than offset by a lack of scalability when the computation performed on data blocks changes dynamically. On balance, it seems this approach will lead to sub-optimal performance on a uniform memory access system such as the PRIMEPOWER. When something like the dynamic page management discussed in [14] is employed on NUMA systems, the programming issues reflect those of a UMA system. In other words, better-balanced performance over a wider range of problem sizes can be obtained with OpenMP Version 2.0 and a focus on the operations involved rather than with static or quasi-dynamic data distributions.

5.2 Cache reuse – design and policy standards?

If explicit data distributions are not a viable solution to cache reuse, then what is? Perhaps compilers should become "BLAS-aware" so that potential performance problems can be flagged and possibly fixed during compilation. Clear documentation about the parallelisation strategy used in each routine is one essential way to avoid pitfalls such as alternating between sequential and

[2] When explicit data distributions are imposed, the computation units become more heavy weight, which is conveyed by referring to them as processes rather than threads.

parallel sections. It would be useful if library providers could agree upon a standard format and terminology for index partitioning information.

There may be a need for information to be available at run time. One possibility would be for a library of parallel routines to include a "partitioning inquiry" function. Given a routine name, a valid set of input arguments and the number of threads, this function could return a descriptor that defined how the index space of the input and output arrays was partitioned among the threads. Notice that the intention is to provide this information for a specific instance of a routine invocation. For example, the way in which the array indices are partitioned among threads in a call to `dgemm` depends not only on the value of the arguments `N`, `M` and `K` but also on whether array `A` or array `B` is transposed and how many threads are available. Given this information, it might be possible for the writer of the calling program to organise the computation done at this level to reduce the amount of cache migration that will result from calls to a particular routine.

While this idea has merits, there are many difficulties with it. Firstly, there is the need for all of the partitioning algorithms employed in a routine such as `dgemm` to be accessible from the inquiry function. This also implies that the control structure of each routine is reproduced. When a parallel library routine was written, would it be possible to generate automatically a "shadow" routine that could generate the information required by the inquiry function? How general is the problem of cache migration, does it extend much beyond linear algebra? Would inquiry functions provide the information required by a user? How complex does the library routine have to become before this type of information cannot be provided or is of limited assistance in improving performance? Is there merit in standardising the format of descriptors, possibly to the extent that they become part of the OpenMP specification? With the limitations of Fortran 77, would there be merit in using external routines to generate a 2-D partitioning for a given number of rows and columns? A library would have its own suite of partitioning routines, but users could be given the specification for such routines so that they could provide appropriate partitioning routines for their application.

If runtime inquiry functions are too cumbersome, could the required information on cache use be encoded into performance analysis information? This might allow users to be alerted to performance problems related to cache reuse with suggestions on how to address these problems.

6 Conclusions

OpenMP provides a convenient means by which users can exploit SMP parallelism. However, obtaining good SMP performance on more than a handful of processors requires careful attention being paid to all of the standard parallelisation issues. OpenMP provides mechanisms to address most of these issues, but the Version 1.1 specification leads to code that is more cumbersome and harder to maintain than is desirable. The Version 2 specification addresses several of these limitations. While OpenMP provides the flexibility and low overheads to exploit loop parallelism, it lacks facilities to optimise the performance of a sequence of such parallel loops by exploiting data cache-locality. This reduces the benefits of library routines that deal with vector-vector or matrix-vector type operations.

There is a case for including index partitioning as an attribute of the arguments in calls to routines that contain parallel loops, but it is not clear what the most appropriate level of detail would be for this attribute. There might also be benefits in organising parallel routines so that one call option was to determine the index partitioning but not perform any further computation. However, when parallel routines have a fixed Fortran 77 interface, the problems become more difficult. One possibility would be to move the partitioning into separate routines and then document the interface to these routines so that users could write their own customized versions.

7 References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Third Edition, SIAM, Philadelphia, PA, 1999.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[3] Intel Limited, Intel Math Kernel Library, Version 5.0, 2001, http://developer.intel.com/software/products/mkl/index.htm

[4] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta and N. Navarro, "OpenMP Extensions

for Thread Groups and Their Run-time Support", *International Workshop on Languages and Compilers for Parallel Computers* (LCPC'00), New York (USA), August 2000.

[5] S. Shah, G. Haab, P. Petersen and J. Throop, "Flexible Control Structures for Parallelism in OpenMP". In *1st European Workshop on OpenMP*, Lund (Sweden), September 1999.

[6] N. Izuta, T. Watabe, T. Shimizu and T. Ichihashi, "Overview of the PRIMEPOWER 2000/1000/800 Hardware", *Fujitsu Scientific and Technical Journal*, Vol. **36**, No. 2, pp.121-127, December, 2000, (http://magazine.fujitsu.com/us/vol36-2/paper03.pdf).

[7] SPEC Organization, "Standard Performance Evaluation Corporation OpenMP Benchmark Suite", June 2001, (http://www.spec.org/hpg/omp2001).

[8] OpenMP Architecture Review Board, *Open MP Fortran Application Program Interface 1.1*, November, 1999, http://www.openmp.org/specs/mp-documents/fspec11.pdf.

[9] OpenMP Architecture Review Board, *Open MP Fortran Application Program Interface 2.0*, November, 2000, http://www.openmp.org/specs/mp-documents/fspec2.pdf.

[10] B. Kågström, P. Ling and C. Van Loan. "GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark", LAPACK Working Note 107, University of Tennessee, CS-95-315, October, 1995.

[11] C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, Massachusetts, 1994.

[12] R. C. Clint and J. Dongarra, "Automatically Tuned Linear Algebra Software", LAPACK Working Note 131, University of Tennessee, CS-97-366, 1998.

[13] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, C. D. Offner, "Extending OpenMP for NUMA machines", *Proc. Supercomputing 2000*, Dallas, November, 2000.

[14] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta and E. Ayguadé, "Is Data Distribution Necessary in OpenMP?", *Proc. Supercomputing 2000*, Dallas, November, 2000.