TR/PA/92/89

Linear Algebra Calculations on a virtual shared memory computer

Patrick R. Amestoy[1], Michel J. Daydé[2,4], Iain S. Duff[1,3], Pierre Morère[2],

October 9, 1992

Abstract

We evaluate the impact of the memory hierarchy of virtual shared memory computers on the design of algorithms for linear algebra. On classical shared memory multiprocessor computers, block algorithms are used for efficiency. We study here the potential and the limitations of such approaches on globally addressable distributed memory computers. The BBN TC2000 belongs to this class of computers and will be used to illustrate our discussion.

The BBN TC2000 is a virtual shared memory multiprocessor with up to 512 nodes. Each node contains one RISC processor (a Motorola 88100) and 16 MBytes of memory. The originality of the BBN TC2000 comes from its interconnection network (Butterfly switch) and from its globally addressable memory. Memory references can be either remote or local to one node. The memory hierarchy consists of the disks, the remote memory, the local memory of each node, the local cache of the 88100, and the internal registers of the processor.

We describe the implementation of Level 3 BLAS and examine the performance of some of the LAPACK routines. The impact of the number of processors with respect to the choice of the variants of classical matrix factorizations (for example KJI, JKI, JIK for the **LU** factorization) is discussed. We also study the factorization of sparse matrices based on a multifrontal approach. The ideas introduced for the parallelization of full linear algebra codes are applied to the sparse case. We discuss and illustrate the limitations of this approach in sparse multifrontal factorization.

We show that the speed-ups obtained on the BBN TC2000 for the class of methods presented here are comparable to those obtained on more classical shared memory computers, such as the Alliant FX/80, the CRAY-2, and the IBM 3090/VF and we explain why our approach can be extended to other virtual shared memory multiprocessors.

[1]CERFACS, 42 av. G. Coriolis, 31057 Toulouse Cedex, France
[2]ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse CEDEX, France
[3]Also Rutherford Appleton Laboratory, OXON OX11 0QX, England.
[4]Also visiting scientist in the Parallel Algorithms Group at CERFACS

1 Introduction

One key issue in the design of high performance computers is to balance the three components: speed of computation, memory bandwidth and size of memory. Faster processors need higher memory bandwidth to a larger memory in order to perform close to their peak performance. In order to achieve a balance using today's processor chips the most cost effective solution is to use hierachical memories (registers, small intermediate memories and/or multi-level cache memories, main memory, disk, ...). The effective bandwidth is improved by interleaving and pipelining access to the memories and by having multiple ports or multi-word memory paths. Efficiency then relies on some kind of "regular" access to the memory and on statistically good locality of the data. It is necessary to design software in order to exploit the hardware properly. On multiprocessor computers, the shared memory paradigm has the advantage, from a user viewpoint, of making all the memory space addressable in a transparent way. This can be extremely convenient when porting codes between parallel computers. However, this does not mean that we have fast access to all the memory available. On classical shared memory multiprocessors (CRAY Y-MP, CRAY C-90, CONVEX C380, IBM 3090/6VF, NEC SX3 ..), uniform access time to the memory is provided. However this usually requires complex and costly memory and processor-to-memory interconnection technologies. A cost effective way of designing shared memory parallel computers with a high number of processors is to attach a local memory to each processor and to make it globally addressable. This type of memory organization is often called virtual shared memory. In this case, the cost of accessing an element in shared memory depends on its locality. This is why such memory organization is also called non-uniform shared memory. Examples of such architectures are the BBN computers (GP1000 and TC2000), the KSR1 from Kendall Square Research, the DASH, the MEMNET, the PLUS, the IBM RP3, and the TERA systems. Most of these systems generally combine a global memory accessible by all the processors and a faster but smaller local memory or local cache. Data transfers between processors are done through the shared memory. The crucial aspect for performance on this class of computers is that remote references should be avoided as much as possible.

We do not pretend here that virtual shared memory computers are superior to all other kinds of computers. However, we believe that the strength of this class of computer comes from its capability of combining efficiently the best features of *both* shared *and* distributed memory. Furthermore, other manufacturers, such as Intel (with the Paragon), have announced the availability of some level of shared memory on their computers and the massively parallel projects from CONVEX and CRAY are likely to possess this feature. This emphasizes the importance of software development for this type of parallel computer.

We discuss in this paper an efficient implementation of linear algebra kernels on virtual shared memory computers. The BBN TC2000 computer will be used to illustrate our discussion. We also comment on the potential of the algorithms presented here on other virtual shared memory computers such as the Kendall Square Research multiprocessor.

The BBN TC2000 is a MIMD multiprocessor based on the Motorola 88100 RISC processor. The TC2000 has from 4 to 512 processor nodes. Each processor node of the configuration includes from 4 to 16 Mbytes of memory, one 88100 processor, and two 88200 cache management units. The size of each cache is 16Kbytes, one is dedicated to data, the other to instructions. The peak performance of the 88100 is 20 Mflops using single precision, and 10 Mflops using double precision. The nodes communicate through a high performance switch (the butterfly switch). It provides transparent access by each processor to all locations in memory, whether local to a processor or remote on another processor. Each node has a switch interface, the memory of the individual nodes forms a pool of shared memory which is accessible by all the processor nodes. Thus this architecture is hybrid in the sense that it exhibits features of both shared and distributed memory architectures. Therefore efficient code design for this type of computer retains aspects of both programming paradigms : communication and synchronization are effected through the shared memory, but the cost of accessing local data (physically on the same node as the processor) versus remote data makes data locality crucial. We use the BBN Fortran programming language that includes extensions such as the parallel do construct and extensions for distributing the data. We discuss the memory organization of the BBN further in Section 2.

In this paper, we consider the implementation of full and sparse linear algebra kernels. Block algorithms are used to obtain efficiency on traditional shared memory computers and we study here the potential and limitations of tuned implementations of such algorithms on the BBN TC2000. We evaluate the impact of the memory hierarchy of the TC2000 on the design of algorithms. We have developed a parallel version of the Level 3 BLAS kernels (Dongarra, Du Croz, Duff, and Hammarling (1990)) by expressing them in terms of matrix-matrix operations and operations involving triangular blocks (Daydé, Duff, and Petitet (1992)). We show that this blocking technique provides a natural way for parallelizing the kernels and allows efficient exploitation of the memory hierarchy on the BBN TC2000. In the case of full linear algebra, we examine the performance of some of the LAPACK codes using a preliminary version of single and double precision Level 3 BLAS that we have developed. The impact of the number of processors on the algorithmic choices, especially on the choice of the variants of classical matrix factorizations is discussed. This leads us to conclusions on the efficiency and the possible limits of using parallel matrix-matrix kernels in full linear algebra on architectures with a large number of processors. In sparse linear algebra, we study **LU** factorization based on a multifrontal approach (see Duff and Reid (1983, 1984)). The very nature of a sparse system means that many equations are not directly linked to each other in the sense that they do not have a variable in common. This gives scope for parallelism because several pivots can be chosen and can perform their updates simultaneously. The multifrontal algorithm uses an *elimination tree* which reflects the parallelism arising from the sparsity. The parallelism from the elimination tree is combined with a node-level parallelism that exploits ideas used in full linear algebra. This second level of parallelism involves blocking techniques controlled by machine dependent user selectable parameters (see Amestoy and Duff (1992b)). Parallel versions of the multifrontal method have been developed on a range of shared memory computers (see Amestoy (1991)) including the Alliant FX/80, the CRAY-2, the CRAY-YMP, and the IBM 3090/6VF. Although the BBN TC2000 computer can be used as a shared memory computer, we show, in this paper, that we must modify the original algorithm, used with success on shared memory computers, to make better use of the memory hierarchy of the BBN TC2000.

We first recall, in Section 2, the main memory features of the BBN TC2000. We describe in Section 3 the parallel implementation of the BLAS routines. In Section 4, we show that with an efficient design of the BLAS routines a large part of the parallelism involved in full linear algebra methods can be captured. Performance analysis of LAPACK routines for **LU** and Cholesky factorization is used to illustrate the discussion. The same ideas are applied in Section 5 to sparse matrix factorization. We show that, in this case, significant performance enhancement can be obtained by modifying the original shared memory algorithm. In Section 6, we consider other virtual shared memory computers such as the Kendall Square Research multiprocessor. We explain why the algorithms presented in this paper are suitable for this class of computers. In our concluding section, we summarize the results presented here and describe the direction of our future research.

2 Memory organization on the BBN TC2000

The memory hierarchy on the BBN TC2000 consists of the remote memory, the local memory, the local cache of the 88100, and the internal registers of the 88100. The crucial aspects of this memory organization are the cost of accessing remote versus local memory, and the use of the caching mechanism. This is illustrated in Table 2.1 where we study the influence of data locality on the average memory access time.

Data can be logically declared as *private* or *shared*. Data that are declared shared are allocated to the physical memory of a single node. Therefore, memory contentions will occur when a shared array is frequently accessed by all the processors. To overcome this problem, there exist extensions to the Fortran language which enable the user to distribute the elements of shared arrays over all the nodes. This can be done by declaring an array as *shared scattered*, in which case the columns of the array are allocated to different nodes a column at a time. Unfortunately this way of storing arrays is not Fortran compatible since there is no uniform relationship between entries in successive columns. We thus do not use this data distribution scheme. We use instead another extension *shared interleaved arrays* since, in this case, the array is distributed over all the nodes of the machine by pieces of size the length of the cache lines (16 bytes, or four single or two double precision words). In other words, the local node memories can be regarded as similar to banks of a traditional shared memory computer. In the CERFACS configuration, we use 26 of the 30 processors for our computations. For each of the 26 processors, we have allocated 2 out of 16 Mbytes of memory to interleaved shared data. This yields 52 Mbytes of shared interleaved data on the physical memory of the computer. Note that this memory configuration can be modified (by software) when rebooting the computer.

The 88200 is in charge of managing the data and the instruction caches. There is no hardware or software policy for maintaining cache coherency so cache consistency issues must be handled by the programmer. Shared data are by default non-cachable, but the programmer can override the default cache policies and manage the cache coherency. Any processor can have both private data that is inaccessible to any other processor and shared local data that is local to the processor but can be accessed by any other processor. When accessed by another processor we term this data shared remote. Three cache policies can be used by the programmer: *uncached*, *writethru*, and *copyback*. With a writethru strategy, data modified in the cache are updated immediately in the memory. In this way, the main memory is always up-to-date. Note that, in this case, we can have different versions of the same data located in different caches. Using copyback, modified cache lines are updated

	Number of processors							
	1	2	4	8	12	16	20	24
READING								
private	0.35							
shared data								
uncached	0.56	1.26	2.22	5.40	8.74	12.04	15.41	18.50
cached (writethru)	0.36	0.57	0.82	1.99	3.25	4.46	5.72	6.90
cached (copyback)	0.36	0.57	0.83	1.99	3.22	4.42	5.73	6.88
interleaved data								
uncached	2.00	2.01	2.01	2.05	2.10	2.17	2.25	2.33
cached (writethru)	0.80	0.80	0.81	0.81	0.83	0.84	0.86	0.88
cached (copyback)	0.80	0.81	0.81	0.81	0.82	0.84	0.86	0.89
WRITING								
private	0.50							
shared data								
uncached	0.51	1.24	2.70	6.44	10.37	14.31	18.29	21.85
cached (writethru)	0.68	1.62	3.54	8.48	13.63	18.84	24.13	28.82
cached (copyback)	0.54	1.12	2.55	5.96	9.53	13.14	16.77	20.08
interleaved data								
uncached	1.99	1.99	2.01	2.11	2.21	2.33	2.46	2.61
cached (writethru)	2.59	2.60	2.66	2.81	2.97	3.14	3.32	3.51
cached (copyback)	1.72	1.73	1.76	1.86	1.98	2.11	2.24	2.38

Table 2.1 Average time in microseconds for accessing one element of a 1,000,000 single precision real array.

in memory only when they have to be removed from the cache (for example when flushing a cache line). Therefore, the memory is not always up-to-date. Private data are always cached using copyback. Fortran extensions are designed to specify the caching mechanism and to force flushing of the caches. Note that the copyback policy is generally the more efficient since it causes less traffic between the caches and the memory. Problems of cache consistency can occur if one processor modifies data within its cache, while another possesses an old copy of the same data. It can happen both using writethru and copyback, the only solution is to invalidate all the old copies of the data. Using copyback the situation is even more complex, since when the cache line updates the memory, all the locations corresponding to the cache line are updated. Thus, if two processors modify different data within the same cache line there is an inconsistency. Therefore, we must also guarantee that the processors work on independent cache lines when we use the copyback mechanism.

To illustrate the influence of the data locality on the access time, we show in Table 2.1 the access times for reading or writing a single precision real (32-bit) array of size 1,000,000. We examine the most common combinations of types: private, shared (local or remote) and shared interleaved and of cache strategies: uncached, writethru and copyback. Timings in Table 2.1 show that the data locality and the cache strategy are key issues that must be taken into account to reach good performance. The average time for writing a shared cached element using the writethru policy is higher than when not caching the element since using writethru the element is modified both in the cache and in the memory, while it has to be modified only in the memory when using the uncached policy.

The TC2000 is a virtual memory system and reducing memory paging is crucial when dealing with large amounts of data. As soon as the amount of data exceeds the capacity of the node memory, swapping occurs and causes great overheads (the cost of accessing one memory page of size 8 Kbytes on disk is typically one hundredth to one tenth of a second). Memory paging has an important effect on data storage. If the amount of local data (either private or local shared data) exceeds the capacity of one node, it is preferable to declare it as interleaved. The main advantage of interleaving is that the amount of physical memory allocated for storing interleaved memory pages can be specified explicitly, node by node, and set up at the boot time. Therefore, interleaved memory can be set up in such a way that a high amount of memory pages can be physically stored in the memory nodes, thus avoiding swapping with disk.

3 Level 3 BLAS on the BBN TC2000

This work is a logical continuation of previous studies on the implementation of Level 3 BLAS on a Transputer network (Berger, Daydé, and Morère (1991)) and on the design of a blocked parallel version of Level 3 BLAS for various shared memory vector multiprocessors (see Daydé and Duff (1991), and Daydé, Duff, and Petitet (1992)).

We have developed two versions of the Level 3 BLAS : one serial and one parallel. In our parallel version, the assumption is that the arrays involved in the calculations are shared. We advise the user to interleave shared data (see Table 2.1). In the serial version, we have not made any assumptions, but it is clear that it is preferable to use these kernels on private data.

We have used our blocked version of Level 3 BLAS (Daydé, Duff, and Petitet (1992)), as a platform both for the serial and the parallel versions of Level 3 BLAS for the TC2000. Our implementation combines the use of blocking and loop-unrolling techniques. It is based on the use of the matrix-matrix multiplication kernel GEMM. The calculations within the Level 3 BLAS are partitioned across submatrices so that they can be expressed as sequences of operations involving matrix-matrix multiplications and operations involving triangular matrices. The size of the submatrices is controlled by a parameter that depends on the characteristics of the target machine. Here the main parameter to be considered is the size of the local cache.

The parallel Level 3 BLAS we have developed manage the cache coherency. We have inserted cache flushes and invalidates in the codes so that arrays can always be cached even when they are in shared memory. In our experiments, we ensure that the arrays start at the beginning of a cache line (for example they could be declared within a shared, interleaved, copyback COMMON) and that their leading dimension is a multiple of the cache line length, so that we can guarantee that the processors work on independent cache lines and we can therefore use the copyback strategy. Normally, however, we advise the use of the safer writethru strategy.

3.1 Serial Level 3 BLAS

We first implement a tuned serial version of the Level 3 BLAS (plus some tuned Level 2 BLAS kernels such as GEMV and TRSV) for single and double precision real matrices. These codes are targeted to be efficient when data are local to the processor in charge of the calculations (but it is not necessary). The calculations are blocked in order to make use of the GEMM kernel. This blocking allows an efficient reuse of data held in the cache. Depending on the kernel and on the precision (single or double) we determine empirically the best value of the blocking parameter (typical values are 32 or 64).

	Standard Fortran	Tuned Fortran no duplication			Tuned Fortran duplication	
M=N=K	Local	Local	sh.int.wr.	sh.int.cop	sh.int.wr.	sh.int.cop.
Single						
32	4.6	8.7	6.9	8.5	3.5	3.7
64	4.0	8.1	5.9	6.7	5.3	5.5
96	3.5	8.2	6.0	6.9	5.6	5.8
128	3.5	8.2	6.0	7.0	6.0	6.1
256	3.6	8.2	6.0	7.0	6.4	6.4
512	3.5	7.8	5.3	6.0	6.5	6.6
Double						
32	2.4	3.2	2.5	2.9	2.0	2.0
64	1.8	3.2	2.4	2.7	1.9	2.0
96	1.8	3.2	2.4	2.7	1.9	2.0
128	1.8	3.0	2.1	2.3	2.0	2.0
256	1.8	2.7	1.6	1.7	2.4	2.4
512	1.9	2.7	1.5	1.6	2.4	2.4

Table 3.1.1 : Performance in Mflops of serial matrix-matrix multiplication.

We use tuned Fortran code to perform operations on the submatrices. The tuning of the Fortran code is based on loop-unrolling. Our intention is to use the internal registers of the 88100 efficiently. Practically, it means that we use temporary variables in Fortran (we would declare register variables using C). We copy references to array elements to these temporary variables. The loops are unrolled and we try to minimize the number of dependencies within the calculations so that the hardware can pipeline sequences of independent load/stores, floating-point additions and multiplications simultaneously. As there is no special hardware for triadic operations we avoid the use of this type of operation. Another key factor for minimizing traffic on the interconnection network (that is for decreasing the proportion of remote references) is to copy the most frequently referenced shared data into private data storage. This data duplication is one of the main mechanisms used in the design of our parallel BLAS. It can also be beneficial when a uniprocessor code has to deal with a large amount of data that causes memory paging. In this case, we use shared interleaved data instead of local data (either private or shared), and we duplicate data between shared interleaved and private memory. Nevertheless, copying data between private and shared (remote or interleaved) memory represents a non negligible cost that makes it efficient only for large enough matrices. In Table 3.1.1 we show the performance of matrix-matrix multiplication using Fortran code for square matrices. The performance is reported for different data storage schemes, with or without data duplication and illustrates many of the issues we have just discussed. The benefit of using tuned code on a single processor is obvious although the limitations of the memory speed of the BBN are well illustrated by the fact that the uniprocessor performance is less than half of the peak performance of 20Mflops for the 88100 in single precision. It could be expected that single precision might be twice as fast as double precision but the results in the table show the difference to be sometimes closer to a factor of three. This is because the 88100 is basically a 32-bit processor and the pipelining of the arithmetic is not as efficient in double precision. Clearly it is better to have all the data on the processor (local) than to have it distributed and equally the copyback strategy gives better results if it can be safely used. When data duplication is used the caching strategy has less influence because the private data is always copyback cached. The duplication of data is advantageous for large matrices but the overhead of copying makes this strategy less efficient for small arrays.

3.2 Parallel version of Level 3 BLAS

We have designed parallel versions of Level 3 BLAS to use shared interleaved or non-interleaved matrices (both cached or uncached). As in the serial version, the calculations are blocked across submatrices. We have seen in Daydé, Duff, and Petitet (1992) that loop level parallelism can be sufficient to parallelize our blocked implementation of the BLAS. We have used the same approach on the BBN using parallel loop extensions. In order to map the calculations on the memory hierarchy efficiently (to avoid remote accesses), the submatrices are first copied to the private memory of each processor (in small working arrays), then the calculations are performed and the results copied back to the shared memory. Note that the calculations are organized in order to reuse the submatrices copied to the private memory as much as possible. The maximum value of the blocking parameter is dependent on the cache size but its value depends both on the size of the problem as well as the number of processors available. Therefore, depending on the number of processors available and on the size of the matrices involved, the operations are carried out on 16-by-16, 32-by-32, or 64-by-64 submatrices.

Kernel	Precision	uniproc.	Number of processors					
			1	2	4	8	16	24
GEMM	single	7.8	6.6	13.4	26.2	52.1	98.8	124.4
	double	2.7	2.5	4.9	9.7	19.2	37.2	47.0
SYMM	single	6.8	6.2	12.4	24.9	49.5	92.3	123.6
	double	2.0	2.5	5.0	9.8	19.3	38.2	51.9
SYRK	single	7.8	4.7	8.6	15.1	30.2	54.7	54.3
	double	1.7	1.7	4.0	7.8	15.7	29.2	29.2
SYR2K	single	7.7	5.0	9.6	18.2	36.5	66.3	66.1
	double	1.6	1.8	4.4	8.6	17.6	33.2	33.3
TRMM	single	7.1	6.0	12.0	23.9	47.6	92.8	114.9
	double	2.5	2.5	5.0	9.9	19.5	38.2	51.6
TRSM	single	7.6	5.3	10.6	21.4	41.6	80.7	103.6
	double	2.6	2.2	4.4	8.7	17.2	33.1	45.3

Table 3.2.1 Performance in Mflops of parallel BLAS kernels using 512-by-512 matrices.

The BBN parallel Fortran extensions allow the parallelization of several nested loops simultaneously. For example the two nested loops on indices i and j can be parallelized simultaneously using the *DEPTH* option for the BBN construct *PARALLEL DO. REPLICATE* ensures that the values of variables in its list are available to all processors. As an example, we give below a code for a parallel copy.

```
      PARALLEL DO, DEPTH(2), REPLICATE(n)
      LOCAL i, j
         DO 20 j = 1, n
            DO 10 i = 1, n
               A(i, j ) = B( i, j )
 10         CONTINUE
 20      CONTINUE
      END PARALLEL
```

The parallelization of the GEMM kernel can be done by partitioning the calculations by submatrices (the same for SYMM). Therefore, the calculation will be parallelized at the level of the two loops 70 and 50 that are used to organize the calculations across submatrices of the matrix C. The code fragment has the following structure :

```
*
*           Form  C := alpha*A*B + beta*C.
*
            PARALLEL DO, DEPTH(2), REPLICATE(.....)
            LOCAL I, J, L, JB, LB, IB, ...
*
* Sectioning loops
*
               DO 70 I = 1, M, NBLIG
                  DO 60 J = 1, N, NBCOL
                     IB = MIN( M - I + 1, NBLIG )
                     JB = MIN( N - J + 1, NBCOL )
*
*  Copy submatrix of C into private memory
* .................................
*
                     DO 50 L = 1, K, NBLIG
                        LB = MIN( K - L + 1, NBLIG )

*
*  Copy submatrix of A and B into private memory
* .................................
*
*  and perform serial matrix-matrix multiplications
*  on the private memory of each processor
*
                        CALL SGEMM_SERIAL('N', 'N', IB, JB, LB,... )
```

```
 50                    CONTINUE
*
* Update the submatrix of C in the shared memory
* ................................
*
 60                 CONTINUE
 70              CONTINUE
*
*    End of parallel matrix-matrix multiplication
*
*
           END PARALLEL
```

For the other kernels, we simply parallelize the outer loop on the columns, but we try to balance the amount of calculation by distributing columns evenly across the processors when dealing with triangular blocks.

We present in Table 3.2.1 a summary of the performance we have obtained with our parallel version of BLAS on 512-by-512 matrices. Note that we only use Fortran programming. The matrices are all shared interleaved copyback. This data storage does not always correspond to the most efficient uniprocessor code. Therefore, we also report the performance of the uniprocessor code (using local shared copyback matrices). Most kernels show a good speed-up reflecting that of GEMM which is used on submatrices. The rank update codes (SYRK and SYR2K) are not so well tuned. The difference in performance of single and double precision reflects that of Table 3.1.1. The size of the matrices is not large enough to reach peak performance. However, we have obtained a rate of 150 Mflops using 24 processors for SGEMM on a 1536-by-1536 matrix.

4 Using parallel Level 3 BLAS in full linear algebra

We study, in this section, the parallelization of the LAPACK codes corresponding to blocked **LU** factorization (GETRF) and blocked Cholesky (POTRF) factorization. Parallelism is only exploited within the Level 3 BLAS. We use our tuned version of the BLAS. All the matrices are declared as shared interleaved copyback.

We show, in Table 4.1, the performance of all the variants of **LU** factorization. We use the left-looking variant (JKI variant) for the factorization of a block column since it is the most efficient from our experience. Note that this unblocked code only calls Level 1 and Level 2 BLAS. The best uniprocessor version corresponds to the version using shared interleaved copyback data because the amount of memory space required for a 2000-by-2000 matrix exceeds the amount of physical memory on one node. Therefore, if we use local data (either private or shared copyback), memory paging cannot be avoided. Both the Crout and the right-looking versions have a higher percentage of GEMM operations than the left-looking variant (Dongarra, Duff, Sorensen, and van der Vorst (1991)). The right-looking variant updates larger and more square matrices and therefore is more favourable for the Level 3

Variant	Precision	Number of processors					
		1	2	4	8	16	24
Left	single	5.6	10.4	17.9	26.9	35.4	37.2
	double	2.2	4.1	7.2	11.4	15.2	18.4
Crout	single	6.1	11.0	19.1	31.8	47.1	55.1
	double	2.3	4.2	7.4	12.2	18.7	22.3
Right	single	5.4	10.1	18.1	31.1	49.1	60.6
	double	2.0	3.7	6.7	11.3	18.8	24.1

Table 4.1 Performance in Mflops of blocked **LU** factorization using 2000-by-2000 matrices.

BLAS operations (Daydé and Duff (1991)). It is thus the most efficient version for blocked **LU** factorization.

Variant	Precision	Number of processors					
		1	2	4	8	16	24
Left (I)	single	5.3	10.3	19.3	33.6	48.7	47.4
	double	2.1	4.2	8.3	14.3	21.8	22.0
Top (J)	single	6.1	11.6	21.4	38.6	67.6	91.3
	double	2.3	4.4	8.3	14.8	25.9	35.9
Right (K)	single	5.0	9.5	16.0	30.9	57.1	57.0
	double	1.9	3.9	8.2	16.5	31.8	32.0

Table 4.2 Performance in Mflops of blocked Cholesky factorization using 2000-by-2000 matrices on the BBN TC2000.

In Table 4.2, we present the performance of the LAPACK codes corresponding to blocked Cholesky factorization. Parallelism is only exploited within the Level 3 BLAS. We use our tuned version of the BLAS. All the matrices are declared as shared interleaved copyback. The results correspond to all the variants of Cholesky factorization. The unblocked code used for factorizing a diagonal block is the top-looking variant (J variant) since it is the most efficient. The block J variant of the Cholesky factorization is the most efficient because it is the only one calling GEMM (see for example Bischof (1990) and Mayes and Radicati di Brozolo (1989)).

Amdahl's law is given by :

$$S_p = p/(f + (1 - f) * p)$$

where p is the number of processors, S_p is the theoretical speed-up computed for p processors, and f the fraction of code that is parallelized. The speed-up S_p is bounded by $S_p < 1/(1 - f)$. We use these formulae to compute theoretical speed-ups calculated from the percentage of time spent in the Level 3 BLAS when using serial Level 3 BLAS and local matrices. We give values for these ideal speed-ups on different numbers of processors and also the value for the speed-up if the number of processors tends to infinity. We compare these predictions with the speed-ups obtained using parallel Level 3 BLAS on shared interleaved copyback matrices in Table 4.3. We have used the J variant of the Cholesky factorization and the right-looking variant of the **LU** factorization using single precision in this table.

Code	Speed-up	Number of processors					
		2	4	8	16	24	infinite
LU	theoretical	1.9	3.4	5.8	8.8	10.6	18.3
	actual	1.2	2.0	3.2	4.5	5.3	
Cholesky	theoretical	2.0	3.9	7.7	14.9	21.7	212.8
	actual	1.4	2.5	4.3	7.1	8.4	

Table 4.3 Comparison of theoretical and actual speed-up for blocked **LU** factorization and blocked Cholesky factorization using 1000-by-1000 matrices on the BBN TC2000.

Note that we have considered here the best uniprocessor version since all data are local to the processor (and the amount of storage does not exceed the physical memory of one node) while the parallel BLAS versions use shared interleaved copyback arrays. Therefore, the theoretical speed-ups reported do not take into account the additional time delay of remote accesses due to the interleaved memory. The Cholesky factorization has a higher percentage of Level 3 BLAS operations than **LU** factorization which explains the higher speed-ups achieved in the Cholesky factorization. Note that with a serial version using shared interleaved copyback matrices, the unblocked part of the computation (and the pivoting in the **LU** factorization) are relatively more penalized than the tuned Level 3 BLAS using duplication in private memory. As a consequence the percentage of time spent in the Level 3 BLAS decreases and the theoretical infinite speed-ups for **LU** and Cholesky decrease from 18.3 and 212.8 to 8.1 and 181.8 respectively. On the other hand, the uniprocessor reference is in this case slower and the real speed-ups of our parallel codes increase from 5.3 and 8.4 to 6.4 and 10.1 on **LU** and Cholesky respectively in a 24-processor environment.

5 Sparse multifrontal method

5.1 Introduction

We consider, in this section, the direct solution of large sparse sets of linear equations on the BBN TC2000 computer. The algorithm is based on a multifrontal approach (Duff and Reid (1983, 1984)). In a multifrontal approach, parallelism is exploited by using the sparsity through an elimination tree. Each edge of the elimination tree corresponds to an assembly while each node is associated with an elimination process on a full frontal matrix. The parallelism from the elimination tree can thus be combined with a node-level parallelism that exploits ideas used in full linear algebra (see Section 4 and Amestoy, Daydé, and Duff (1989)). For the elimination process, we use a row oriented (frontal matrices are stored by row) adaptation of the right-looking variant (KIJ-SAXPY) (see for example Daydé and Duff (1991)). At the k-th step of this block form, a block row of the factors is computed and the corresponding transformations, based on Level 3 BLAS kernels, are applied to the remaining reduced matrix. This second level of parallelism involves block spawning controlled by machine dependent user selectable parameters. This approach has been successfully implemented on a range of shared memory multiprocessors including the Alliant FX/80, the IBM 3090/VF, the CRAY 2 and the CRAY Y/MP (see Amestoy and Duff (1992b) and Amestoy (1991)).

We discuss, in this section, different strategies for adapting our shared memory multifrontal method for the TC2000. Clearly the access time depends on the physical locality of the data. Two fairly straightforward implementations of the shared memory multifrontal code are described in Section 5.2. In Section 5.3, we increase the amount of computations performed on private data without modifying the original algorithm. We apply the techniques used for the parallelization of full linear algebra code (see Section 4). Shared data are then moved to private memory before computation and are stored back to shared memory after computation. Based on the analysis of the results obtained in Section 5.3, we introduce, in Section 5.4, modifications of both the multifrontal code and the multifrontal algorithm that will lead to our best version of the multifrontal method on the BBN TC2000.

		multifrontal factorization			
		(1)		(2)	
Computer	nprocs	Mflops	(speed-up)	Mflops	(speed-up)
Alliant FX/80	8	15	(1.9)	34	(4.3)
IBM 3090E/3VF	3	83	(1.9)	105	(2.4)
IBM 3090J/6VF	6	126	(2.1)	227	(3.8)
CRAY-2	4	316	(1.8)	404	(2.3)
CRAY Y-MP	6	529	(2.3)	1119	(4.8)

Table 5.1.1 Performance summary of the multifrontal factorization on matrix BCSSTK15.
-in columns (1) we exploit only parallelism from the tree;
-in columns (2) we combine the two levels of parallelism.

A medium size sparse matrix, *BCCSTK15* from the Harwell-Boeing set (see Duff, Grimes, and Lewis (1989)), will be used to illustrate our discussion. The matrix is of order 3948 with 117816 nonzeros. A minimum degree ordering is used in the analysis and the number of floating-point operations for the factorization is 443 million. We use double precision (64-bit) arithmetic. We recall, in Table 5.1.1, the performance obtained on a range of shared memory multiprocessors. The speed-ups mentioned in Table 5.1.1 will be used as a reference point for our discussion. For a detailed analysis of these results one should refer to Amestoy (1991).

5.2 Shared memory based multifrontal algorithm

In our first two implementations of the code, the BBN TC2000 is used as a classical shared memory multiprocessor computer. The versions of the multifrontal code presented in this section are thus fairly straightforward adaptations of the shared memory multifrontal code in Amestoy (1991). In *version 1* of the code, all data are declared shared. Note that data declared shared are physically allocated (see Section 2) to the memory of the processor declaring them for the first time. As a result, most data are located on the physical memory of the processor activating all the parallel tasks. Therefore, with version 1, we only use a small part of the bandwidth of the interconnecting network. One way to overcome this problem is to uniformly distribute the shared data over the physical memory of the computer. This is done using interleaving (see Section 2) so that the data are always distributed over all processors independently of the number of processors actually used for the factorization. This leads to *version 2* of the code. In both versions we use uniprocessor tuned versions of the shared BLAS library described earlier. We also use the optimal value of the block parameters for controlling the second level of parallelism on the Alliant FX/80 (see Amestoy (1991)).

Nb of procs	1	2	3	4	6	8	16	20	24	26
version 1										
(1)	328	226	217	186	190	191	198	171	166	206
(2)		227	170	151	117	105	103	100	99	95
version 2										
(1)	490	330	261	247	221	218	225	224	215	214
(2)		300	213	179	129	117	102	98	84	83

Table 5.2.1 Performance summary of two fairly straightforward implementations of the multifrontal method (in seconds) on matrix BCSSTK15.
-in rows (1) we exploit only parallelism from the tree;
-in rows (2) we combine the two levels of parallelism.

We show, in Table 5.2.1, timings obtained for the numerical factorization of our test matrix (BCSSTK15) from the Harwell-Boeing set (Duff, Grimes, and Lewis (1989)). We see, in Table 5.2.1, that, on one processor, version 1 is much faster than version 2 where shared data are uniformly distributed over the physical memory of the TC2000. However, we observe that the uniprocessor Megaflop rate of version 1 (1.35 Mflops) is much slower that what could have been expected from the performance of the tuned uniprocessor Level 3 BLAS (see Table 3.1.1). Using version 1 of the code, most data are local to the physical memory of one processor of the BBN TC2000. 2 Mbytes of physical memory on each processor are reserved for the interleaved shared memory and 1 Mbytes for the operating system. A maximum of 13 Mbytes of physical memory can thus be used by version 1 of the code. Unfortunately, to factorize our test matrix, we need more than twice this amount of shared memory (see Amestoy and Duff (1992a)). As a consequence, page swapping on the disk cannot be avoided. This explains the relative low uniprocessor performance obtained with version 1.

Moreover, we see in Table 5.2.1 that the difference in performance between the first two versions reduces when we increase the number of processors. With version 2, shared data are distributed over all the processors. We thus decrease the average data access time by increasing the number of processors used for numerical factorization and we obtain better speed-ups with version 2 than with version 1. Furthermore, the comparison with the speed-ups obtained on shared memory multiprocessors (see Table 5.1.1) shows that, with version 2, a large part of the potential parallelism of the tree is exploited. Although the speed-up is relative to a low uniprocessor performance, it shows that the average memory access time to interleaved data is not very sensitive to an increase in the number of processors and in the memory traffic. It thus confirms the timings obtained in Table 2.1 for interleaved data.

We have seen, in this section, that with shared data as in version 1, we only use a small part of both the physical memory and the bandwidth of the computer. This results in small multiprocessor speed-ups and in page swapping problems that will increase with the size of the problem. With version 2, we overcome these problems by interleaving the shared data over the physical memory of all the processors. This leads to a slower uniprocessor code having a good parallel behaviour.

5.3 Private BLAS based multifrontal method

Clearly, the first two ways of implementing the multifrontal method on the BBN TC2000 do not take into account the main characteristics of the computer sufficiently. The multifrontal numerical factorization is based on an elimination tree where the nodes correspond to assembly and elimination operations. Most of the floating-point operations involved during factorization come from the node elimination step. We can thus apply the ideas introduced in Section 4 for full linear algebra codes to the sparse multifrontal method. In *version 3* of the multifrontal code, shared blocks of data involved in the block KIJ-SAXPY scheme are moved to temporary private working arrays prior to computation. At the end of the computation, performed in private memory, we then update the original shared block of data. To enhance performance, all shared data are cached with a writethru strategy and interleaving is used to smooth the access time to shared data. Note that, to prevent cache inconsistency problems, cache flush instructions must be inserted in the code.

Furthermore, we can reduce the amount of data transfers between private and shared memory when the elimination process at a node, k say, does not generate additional parallelism. This can be detected during the assembly process of node k and will only depend on the choice of the blocking parameters controlling the second level of parallelism. In this case, the frontal matrix is assembled in a private working array on which we perform all the elimination operations. Note that at the end of the elimination process, we must store in shared memory the **LU** factors and the block of data, the so called *contribution block*, that is needed when assembling the frontal matrix of the father of node k.

On the other hand, if a node elimination process generates parallel tasks based on block spawning then we assemble the corresponding frontal matrix in the shared interleaved memory. Data moves are then required during the elimination step to perform Level 3 BLAS kernels on private data. In this case, to enhance the performance of the kth step of the KJI-SAXPY scheme, we also copy the block row on which elimination operations are performed to a private working array. In this way, all the Level 2 BLAS based operations of the KJI-SAXPY scheme are performed in the private area. Note that once the current block row has been factored, we need to update the corresponding block in the shared area before starting the parallel updating of the remaining rows in the frontal matrix.

In a non-uniform memory environment, the parallelization of a node process implies overheads of communication over and above those for a sequential node elimination step. We thus need a good control of the second level of parallelism to prevent speed-down due to communication overheads. We have noticed that the optimal block size to parallelize the KJI-SAXPY blocked scheme is smaller than the optimal block size for a uniprocessor environment. The main reason is that, by reducing the block size used for the optimal sequential processing of a given node (typically 32), we increase the potential parallelism of the node elimination even if we have to increase the overhead of data transfer to and from the private areas. Note that, to keep good average Mflop rates during computation we must also maintain a minimum task granularity (i.e. a minimum block size). We have found experimentally that, for a parallel node process, a good compromise is reached with a block size for the KIJ-SAXPY algorithm equal to 10. The number and the granularity of the parallel tasks generated during node-level parallelism are controlled by several user-defined parameters including the above block size. We show in Table 5.3.1 the timings corresponding to the best values of the block parameters.

Nb of procs	1	2	3	4	6	8	16	20	24	26
version 3										
(1)	216	145	113	105	94	91	87	84	84	85
(2)		174	124	95	75	63	48	42	39	38

Table 5.3.1 Performance summary in seconds on matrix BCSSTK15.
-in row (1) we exploit only parallelism from the tree;
-in row (2) we combine the two levels of parallelism.

The results presented in Table 5.3.1 clearly illustrate the gain coming from the modifications of the code both in terms of speed-up and performance.

In row (1) of Table 5.3.1, we have assumed that the private working arrays are large enough to store the largest frontal matrix. In this case, most floating-point operations are performed in private memory. Only the contribution blocks and the **LU** factors need to be stored in the shared interleaved memory. Note that, with Matrix BCSSTK15, the largest frontal matrix is of order 533. It thus requires 2.2Mbytes of memory which is less than the private memory of a processor (13Mbytes). The uniprocessor timing presented in Table 5.3.1 is thus a good uniprocessor time reference. We notice that the speed-ups coming from the use of only tree parallelism (see row (1) in Table 5.3.1) are comparable to the speed-ups obtained on shared memory computers (see Table 5.1.1). On larger test problems, we do not want to store frontal matrices that do not fit in the memory of a processor in private memory because of disk swapping issues (see Section 2). However, in practice, large frontal matrices create node-level parallelism and are anyway stored in shared interleaved memory.

We also observe, in Table 5.3.1, that, although the second level of parallelism nicely supplements that from the elimination tree, it does not provide all the parallelism that one could expect (see Table 5.1.1). The second level of parallelism can even introduce speed down with a small number of processors as can be seen in columns 3 and 4 of Table 5.3.1. On a small number of processors, the gain coming from the node-level parallelism does not balance the overhead due to data transfer between private and shared memory. Reducing the overhead due to data transfers, creating more parallelism, and controlling the node-level parallelism better are important issues that we need to address to obtain higher and more stable performance.

5.4 Tuning the multifrontal method for the BBN TC2000

In version 3 of the code, we applied the ideas used for the implementation of full linear algebra kernels to our sparse multifrontal method. Private working arrays were introduced as temporary storage for computationally intensive blocks of data. We thus implicitly assumed, in version 3, that it is always beneficial to move from shared to private memory blocks of data involved in the Level 2 and Level 3 BLAS kernels of the KJI-SAXPY scheme. We show, in this section, how we can modify both the implementation and the algorithm used in version 3 to control the overhead due to memory transfers and to increase the parallelism of the method.

We use a block KJI-SAXPY algorithm to factorize the frontal matrices. We thus systematically combine a triangular solver (DTRSM) with a matrix-matrix multiplication (DGEMM). In version 3 of the code, all Level 2 and Level 3 BLAS routines were performed on private data. This process often involves data transfers between the shared interleaved memory and the private memory. From a comparison of the performance using duplication and no duplication on small matrices in Table 3.1.1, we see that it is not always beneficial to force all Level 3 BLAS kernels to be performed in private memory. In version 4 of the code, the decision to copy shared blocks of data involved in the BLAS kernels is based on a simplified model using the average memory access times mentioned in Table 2.1 and the average Megaflop rates mentioned in Table 3.1.1.

To create more parallelism during a node process and to use data locality better, we have modified the algorithm used in version 3. We overlap the parallel updating involved at step k of the KJI-SAXPY scheme with the elimination of the block row at step $k+1$ in a similar way to that done by Daydé and Duff (1991). Based on the value of the blocking parameters introduced for version 3 (not modified in version 4), we know during the assembly phase if the node elimination step will create parallel tasks. We assume, in the following discussion, that the second level of parallelism is used during the updating operations involved at step k of the KJI-SAXPY algorithm. In this case the frontal matrix is assembled in shared interleaved memory. Let us denote by *master task*, the task in charge of the elimination of the k-th block row. Let nb be the size of the block row of the KJI-SAXPY algorithm. At step k of the KJI-SAXPY algorithm, the master task works on a private copy of the kth block row. Once the block row is factored, the master task updates the original values of the block row in shared memory. The updating of the remaining rows of the frontal matrix can then be performed in parallel. The master task moves to private memory the first nb rows of the remaining frontal matrix, performs the updating of the first nb rows, and finally starts step $k+1$ of the KJI-SAXPY algorithm. At the same time the *slave tasks* update equal-size block rows of the rest of the frontal matrix. This will be referred to as *pipelining* of the blocked KJI elimination steps. We use the average memory access times mentioned in Table 2.1 and the average Megaflop rates mentioned in Table 3.1.1 to estimate the time spent in the master and in the slaves tasks. Assuming that we have an infinite number of processors, we compute the ideal number of tasks involved in the parallel updating process. If the number of tasks is larger than the number of physical processors, then we slightly increase the number of rows updated by the master. The blocking parameters are then used to check that each slave task processes a minimum number of rows. Because of the combined parallelism and/or because of the varying memory access times, the master task can finish before the end of all its slaves tasks. In this case, we postpone the updating operations of step $k+1$ until the end of all the slave tasks. Note that we have kept our dynamic scheduling approach (see Duff (1989) and Amestoy (1991)) so that the processor used by the master task is freed and can perform other ready-to-be-activated tasks.

Nb of procs	1	2	3	4	6	8	12	16	20	24	26
version 4											
(1)	216	143	113	103	94	89	85	84	84	84	85
(2)		155	107	86	68	55	44	36	33	32	31

Table 5.4.1 Performance summary in seconds on matrix BCSSTK15.
-in row (1) we exploit only parallelism from the tree;
-in row (2) we combine the two levels of parallelism.

We show in Table 5.4.1 results obtained with version 4 of the multifrontal method. In row (1) of Table 5.4.1, private working arrays are larger than the largest frontal matrix and most of the computation is performed in private memory. The modifications introduced in version 4 of the code have thus little effect on the performance of the code (compare row (1) of Table 5.4.1 with row (1) of Table 5.3.1). Note that further improvements can be expected in row (1) of Table 5.4.1 on matrices with larger fronts not fitting in our private working array. In this case, a frontal matrix is allocated in shared memory and the selective duplication done with version 4 will provide improvements over version 3. The gains with respect to version 3 of the code are clear when the second level of parallelism is combined with the parallelism from the tree (compare row (2) of Table 5.4.1 with row (2) of Table 5.3.1). The speed-ups obtained with the second level of parallelism (see row (2) in Table 5.4.1) on a relatively large number of processors (typically more than 8) are noticeably better than those obtained in the previous version of the code. The comparison with the speed-ups obtained on an 8-processor shared memory Alliant FX/80 (see columns (2) in Table 5.1.1) also shows that version 4 of the multifrontal code exploits well the second level of parallelism. With version 4, we finally reach 14.3 Mflops (in double precision) which corresponds to almost three times the best performance obtained with the best straightforward shared memory implementation of the multifrontal code.

6 Implementing linear algebra kernels on virtual shared memory computers

Our aim in this section is to show how our approach, described in the previous sections, can be extended to other virtual shared memory computers without significant programming effort. In our discussion we mainly focus on the KSR multiprocessor since that is one of the only other machines of this class currently commercially available.

We believe that virtual shared memory multiprocessors are likely to become an important class of parallel computers both for hardware reasons (distributed memory is a cost-effective way of designing memory for a high number of processors), but also for software reasons, because they may combine the best features of both shared and distributed memory. They especially provide the advantage of allowing the easy porting of codes originally designed for traditional shared memory computers (note that it does not mean that the code will achieve a high performance since the performance mainly depends on how efficiently the data are mapped onto the memory system of the computer). This probably explains why most of the recently announced massively parallel computers are likely to have some kind of virtual shared memory.

The crucial parameter on virtual shared memory architectures is the ratio of time for local to remote data accesses which makes data locality crucial and may highly penalize codes with not good enough locality of references (this is even more true on distributed memory architectures). For this reason data duplication between shared and private memory is used on the BBN TC2000 to increase the amount of local memory references done by each processor. Moreover, especially in the multifrontal code, priority for allocating a new task to a processor was given to the processor holding local copies of the data required for that calculation. This is probably the main difference in the code design between shared memory and virtual shared memory computers : whenever it is possible, the processor holding local copies of data, should be the one in charge of performing the computational tasks accessing these data the most frequently.

The KSR1 from Kendall Square Research is a shared memory multiprocessor. Each processor has 32 Mbytes of memory and up to 32 processors are connected on a ring. Bigger configurations are obtained by connecting several rings together on an additional ring. The processor is superscalar and has a peak performance of 40 Mflops. The local memories of the KSR are managed as local caches so that a remote reference to an element causes the data to be transferred to the processor memory (and subsequent references to the data will be local). Prefetch and post-store instructions are available to the programmer in order to decrease the latency of accessing remote elements. More details about the KSR computer can be found in Dunigan (1992). This paper includes comparisons with other computers such as the Sequent, the BBN TC2000, the IBM RS/6000-530, the Intel IPSC/860, and the Intel Delta.

The KSR has a faster processor than the BBN TC2000 and a longer remote memory latency. The memory hierarchy (number of internal registers, cache size), and the shared memory management are different. In particular, there is no need to copy data between the private and the shared memory since the KSR automatically turns a remote reference into a local reference. This may result in higher performance even though the memory latency is higher on the KSR than on the BBN. The tuning of the codes, described in the previous sections, is controlled by a set of machine dependent parameters which can be easily modified to take into account the memory hierarchy of the KSR computer. We can, for example, adapt the block size and the depth of loop-unrolling to the size of the local cache and the number of internal registers. We can also automatically modify the task granularity and thus control the degree of parallelism and the load balancing. Note that this can be useful on most of the computers whether they are shared, virtual shared, or distributed memory based computers. It becomes more crucial when the number of processors increases and when the data transfer ratios between the various levels of memory are higher. Note that, although it is not relevant on the KSR computer, the process of copying data between private and shared memory will be very useful on shared memory computers with the concept of private data. Data allocation is one of the main difficulties on purely distributed memory multiprocessors (especially for the multifrontal code) and the implementation of codes on virtual shared memory multiprocessors such as the BBN

TC2000 has already pointed out similar problems. One could say that the implementation of codes on virtual shared memory machines is an intermediate step from shared to purely distributed memory multiprocessors.

On the BBN TC2000, the data allocation is static and it is therefore easy to identify which processor holds local data, while on the KSR there is no concept of private data and remote data, once referenced, are transferred into the local memory. This ALLCACHE memory management will require careful blocking of the calculations in order to guarantee an efficient reuse of local data and to avoid unnecessary remote references.

Clearly, some additional effort would be required to implement our linear algebra codes on other virtual shared memory computers. However, we believe that the techniques used to design efficient linear algebra codes on the BBN TC2000 address the important issues thus minimizing such additional effort.

7 Concluding remarks

The advantage of the virtual shared memory available on the TC2000 is that it provides a very convenient means for porting library codes from classical shared memory architectures. We have been able to run a subset of the LAPACK library using both serial and parallel versions of the Level 3 BLAS. We have also successfully applied the same ideas to the porting of a sparse multifrontal code. Most of the tuning done on the BBN TC2000 was aimed at efficiently mapping the code into the memory hierarchy within one node while avoiding remote memory references. Finally, we have indicated how our approach could be extended to other virtual shared memory multiprocessors such as the KSR computer.

In Table 7.1 we compare the speed-ups obtained on parallel matrix-matrix multiplication on the BBN TC2000 and other shared memory multiprocessors : the Alliant FX/80, the CRAY-2, and the IBM 3090 models E and J (see Daydé and Duff (1991)). The speed-ups obtained on the BBN TC2000 can be successfully compared to those obtained on the other computers even if the performance achieved is not always comparable.

Computer	Number of processors						
	2	3	4	6	8	16	24
BBN TC2000	1.9	2.7	3.7	5.4	7.4	14.3	18.1
IBM 3090E-400	-	2.9	-	-	-	-	-
CRAY-2	-	-	3.7	-	-	-	-
IBM 3090J-600	-	-	3.8	5.1	-	-	-
Alliant FX/80	-	-	-	-	6.6	-	-

Table 7.1 Speed-ups obtained on parallel GEMM on 512-by-512 matrices using 64-bit arithmetic.

We have shown that blocking techniques provide a natural way for parallelizing the kernels and that they allow an efficient exploitation of the memory hierarchy of the BBN TC2000. We have noticed, for example, that our parallel version of matrix-matrix multiplication

(GEMM) on matrices of order greater than 1000 can achieve a performance rate of 150 Mflops and 55 Mflops using single precision and double precision arithmetic respectively.

In full linear algebra, we have obtained good performance using simple Level 3 BLAS based parallelism. The main limitation of our approach is due to the fact that, with Level 3 BLAS based parallelism, we cannot keep a high number of processors busy. Increasing the size of the matrices factorized would of course increase the percentage of Level 3 BLAS operations. However, it is clear that a parallelization over the kernels (see for example Daydé and Duff (1991) and Dackland, Elmroth, Kågström, and Van Loan (1991)) needs to be implemented to increase the parallelism of the methods and to reduce the amount of data transfers between private and shared interleaved memory. To some extent, this idea has been applied in the design of our final shared version of the multifrontal method (version 4). With a Level 3 BLAS based parallelization of the second level of parallelism (version 3) we reach a maximum speed-up of 5.7 while, with version 4 of the code, we obtain a speed-up of 7.

Note that much work still needs to be done on the sparse multifrontal method. Experiments on a wider range of test problems must be performed to generalize our conclusions. We also need to study the influence of numerical pivoting on the performance of the code. Further improvements of the sparse multifrontal code will involve a more distributed type of algorithm. To address a large number of processors, we also probably need to study clustering heuristics of the elimination tree.

References

P. R. Amestoy and I. S. Duff, (1992a), *Memory allocation issues in sparse multiprocessor multifrontal methods*, Technical Report TR/PA/92/83, CERFACS, Toulouse, France.

P. R. Amestoy and I. S. Duff, (1992b), *MUPS: parallel package for solving sparse unsymmetric sets of linear equations*, Technical Report, CERFACS, Toulouse, France. To appear.

P. R. Amestoy, M. J. Daydé, and I. S. Duff, (1989), *Use of Level 3 BLAS kernels in the Solution of Full and Sparse Linear Equations*, in Proceedings of the European Symposium on High Performance Computing, J. Delaye and E. Gelenbe, eds., Elsevier Science Publishers B.V., 19–31.

P. R. Amestoy, (1991), *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*, PhD Thesis TH/PA/91/2, CERFACS, Toulouse, France.

P. Berger, M. J. Daydé, and P. Morère, (1991), *Implementation and Use of Level 3 BLAS kernels on a Transputer T800 Ring Network*, Technical Report TR/PA/91/54, CERFACS, Toulouse, France.

C. H. Bischof, (1990), *Fundamental Linear Algebra Computations on High-Performance Computers*, Technical Report MCS-P150-0490, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan, (1991), *Parallel block matrix factorizations on the shared memory multiprocessor IBM 3090 VF/600*, Technical Report Report UMINF-91.07, University of Umeå, Sweden.

M. J. Daydé and I. S. Duff, (1991), *Use of Level 3 BLAS in LU factorization in a multiprocessing environment on three vector multiprocessors, the ALLIANT FX/80, the CRAY-2, and the IBM 3090/VF*, Int. J. of Supercomputer Applics., **5**, 92–110.

M. J. Daydé, I. S. Duff, and A. Petitet, (1992), *A Parallel Block Implementation of Level 3 BLAS Kernels for MIMD Vector Processors*, Technical Report TR/PA/92/74, CERFACS, Toulouse, France.

J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, (1990), *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., **16**, 1–17.

J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, (1991), *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia.

I. S. Duff and J. K. Reid, (1983), *The multifrontal solution of indefinite sparse linear systems*, ACM Trans. Math. Softw., **9**, 302–325.

I. S. Duff and J. K. Reid, (1984), *The multifrontal solution of unsymmetric sets of linear systems*, SIAM J. Matrix Anal. and Applics., **5**, 633–641.

I. S. Duff, R. G. Grimes, and J. G. Lewis, (1989), *Sparse matrix test problems*, ACM Trans. Math. Softw., **15**, 1–14.

I. S. Duff, (1989), *Multiprocessing a sparse matrix code on the Alliant FX/8*, J. Comput. Appl. Math., **27**, 229–239.

T. Dunigan, (1992), *Kendall Square multiprocessor : early experiences and performance*, Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

P. Mayes and G. Radicati di Brozolo, (1989), *Portable and efficient factorization algorithms on the IBM 3090/VF*, in Proceedings international conference on supercomputing, New-York: ACM, 263–270.