Experiments with Cholesky Factorization on Clusters of SMPs

Siegfried Benkner[1]**, Dieter F. Kvasnicka**[2]**, Maria Lucka**[1]
[1] Institute for Software Science
University of Vienna, A-1090 Vienna, Austria
E–mail: sigi@par.univie.ac.at and lucka@par.univie.ac.at

[2] Institute for Materials Science
Vienna University of Technology
E–mail: dieter.kvasnicka@tuwien.ac.at

Abstract
Cholesky factorization of large dense matrices is an integral part of many applications in science and engineering. In this paper we report on experiments with different parallel versions of Cholesky factorization on modern high-performance computing architectures. For the parallelization of Cholesky factorization we utilized various standard linear algebra software packages and present performance results on SMP clusters and shared-memory cc-NUMA machines. Clusters of SMPs can be characterized as hybrid parallel architectures which combine the main architectural features of distributed-memory and shared-memory parallel computers. Although the availability of SMP clusters is increasing rapidly within the scientific computing community, currently no generally accepted programming model exists for these machines. As a consequence, most application developers utilize pure distributed-memory programming models, usually based on the message passing interface (MPI), and thus may miss a number of optimization opportunities offered by the shared-memory available within the nodes of a cluster. In order to address these issues, we have experimented with different parallelization strategies for Cholesky decomposition comparing pure message passing strategies to a hybrid parallelization strategy that combines message passing with shared-memory parallelization based on multi-threading.

1 Introduction

Cholesky factorization belongs to the basic numerical algorithms with numerous applications in scientific and engineering computing, ranging from computational fluid dynamic problems to financial modeling problems. Thus efficient parallel implementations of this code are required on parallel computer architectures. Our experience with Cholesky factorization stems from its use in the program package WIEN2k [1] which is used for quantum mechanical calculations of solids.

Modern parallel computer systems offer many possibilities for an efficient parallelization of linear algebra algorithms. However, the designers of new parallel algorithms are faced with the challenge of achieving the highest possible performance while ensuring portability across different computing platforms.

While in the last fifteen years the high-performance computing arena has been dominated by vector supercomputers and later by distributed memory machines, over the last several years, clusters of (symmetric) shared-memory multi-processors (SMPs) have become the most promising parallel computer architectures. These machines can be characterized as hybrid-parallel systems, combining the main features of distributed-memory (DM) and shared-memory (SM) parallel architectures. Clusters of SMPs offer several possibilities for an efficient parallelization of linear algebra algorithms, however, currently no generally accepted programming model exists for SMP clusters. Most application programs developed for SMP clusters are based on MPI [2, 3], a standard API for message-passing which has been designed for distributed-memory parallel architectures. However, MPI programs which are executed on clusters of SMPs usually do not directly utilize the shared-memory available within nodes and thus may miss a number of optimization opportunities. OpenMP [4], a directive based parallelization API for Fortran and C/C++, only supports shared-memory parallelization and thus is not applicable to SMP clusters. A promising approach for parallel programming on SMP clusters attempts to combine MPI with

OpenMP [4] in a single application [5, 6, 7]. This strategy aims to fully exploit SMP clusters by relying on data distribution and explicit message-passing between the nodes of a cluster, and on data sharing and multi-threading within nodes.

In this paper we compare distributed-memory, shared-memory, and hybrid parallelization strategies for Cholesky factorization using standard parallel linear algebra libraries. For this algorithm well established numerical library implementations are available. Another advantage is that there are no data dependencies in the program flow, leading to deterministic behavior, regardless of the actual implementation or exact hardware characteristics. The Cholesky factorization, which is usually written as either

$$A = LL^\top \quad \text{or} \quad A = U^\top U \tag{1}$$

is a factorization of the Hermitian, positive definite matrix A into a lower triangular matrix L or an upper triangular matrix U. This factorization is widely used for solving systems of equations or generalized eigenproblems (see, e. g. [8]). *Blocked* Cholesky factorization exhibits better locality of reference and is therefore preferred over the unblocked version on modern computer architectures (Anderson et al. [9], Ueberhuber [10]).

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top \\ 0 & L_{22}^\top \end{pmatrix} = \begin{pmatrix} L_{11}L_{11}^\top & L_{11}L_{21}^\top \\ L_{21}L_{11}^\top & L_{21}L_{21}^\top + L_{22}L_{22}^\top \end{pmatrix}$$

consists of the following three tasks.

1. Factorize $A_{11} = L_{11}L_{11}^\top$,
2. Solve the linear system $L_{21}L_{11}^\top = A_{21}$ for L_{21},
3. Update $A'_{22} := A_{22} - L_{21}L_{21}^\top$ and compute the Cholesky factorization of $A'_{22} = L_{22}L_{22}^\top$.

Three versions of this algorithm are (i) left-looking (Step 1 is applied recursively), (ii) right-looking (Step 3 is applied recursively), and (iii) "recursive" Cholesky factorization (Steps 1 *and* 3 are applied recursively; see Andersen et al. [9]). The right-looking algorithm applied to dense matrices is used in this paper in the subroutines of the linear algebra packages LAPACK and ScaLAPACK.

This paper is organized as follows: In Section 2 we give a brief overview of parallel computer architectures, their programming models and parallel linear algebra libraries. In Section 3 we describe shared-memory parallelization strategies for Cholesky factorization. Distributed-memory parallelization is discussed in Section 4 and in Section 5 we experiment with hybrid parallelization strategies. Performance results reported in these sections have been obtained on three different parallel computers. The paper closes with a brief summary and discussion of related work.

2 Parallel Architectures and Programming Models

In this section we briefly discuss the basic architectural features of high performance computer architectures. Instead of attempting to provide a complete classification, we discuss parallel architectures mainly with respect to their memory organization. Moreover, we describe the main features of three parallel machines that have been used for our experiments with different parallel versions of Cholesky factorization, and give a brief summary of the major parallel programming models and parallelization strategies.

2.1 Shared-Memory versus Distributed-Memory Machines

Depending on their memory organization modern parallel computing systems can be classified into shared-memory (SM) systems and distributed-memory (DM) systems.

In a shared-memory parallel computer multiple processors are connected to a large shared memory, usually by a bus or a crossbar network. These systems, often referred to as symmetric multiprocessors (SMPs), provide a *global address space* with uniform memory access (UMA) for each processor. Due to the availability of a global address space, SMP-like systems are relatively easy to program since the data space of a program can be organized in much the same way as on a conventional sequential system. However, the major drawback of these systems is their limited scalability since only a relatively small number of processors can be connected to a shared memory with reasonable technical effort and at affordable costs. For these reasons, most modern high-performance computing architectures are at some level based on physically distributed memory, in order to ensure scalability.

In a distributed-memory parallel computer, multiple processors, each with its own local memory, are connected via a fast interconnection network. Since these systems do not share a global memory, they offer virtually unlimited scalability and can be built much more cost effective than shared-memory systems. However, in contrast to shared-memory systems, a processor only has access to its local memory, while data on other processors are not directly accessible. If data from other processors are required, explicit data exchanges over the network, usually be means of explicit message passing, have to be performed. While a distributed memory organization is the key to scalability allowing thousands of processors to be connected to form a powerful parallel supercomputer, the lack of a global address space has important consequences on the complexity of application development and program efficiency. Since the time required to access data from a remote processor may be orders of magnitude higher than the time to access locally stored data, data locality becomes a key issue. Due to the non-uniform memory access (NUMA) characteristics, the distribution of data, with the goals of minimizing the delays caused by non-local data accesses while spreading the computational workload evenly, becomes a key issue in parallel application development.

Most distributed-memory parallel computers are nowadays based on commodity technology, e.g. PC clusters or clusters of workstations.

2.2 Scalable Shared Memory Machines and SMP clusters

In order to overcome the problems associated with programming distributed-memory machines, several variants of scalable shared-memory systems (also called distributed shared memory systems) have been developed. These systems provide a global address space on top of physically distributed memory by means of special hardware (and/or software) support. While these machines provide the programmer with a logically shared address space, they exhibit non-uniform memory access (NUMA) characteristics due to the underlying distributed memory organization. As a consequence, the organization of data has a strong influence on the efficiency or parallel programs. An import class of architectures that fall into this category are cc-NUMA (cache-coherent NUMA) machines.

Over the last several years, clusters of shared-memory parallel processors (SMPs) have become increasingly important. These machines can be characterized as hybrid-parallel systems, combining the main features of distributed-memory (DM) and shared-memory (SM) parallel architectures. SMP clusters consist of a number of nodes each equipped with multiple processors. Each processor has direct access only to the shared memory within its node, while data located on other nodes has to be accessed by means of explicit message passing over the interconnection network. Examples of such systems are multiprocessor clusters from SUN, SGI, IBM, a variety of multi-processor PC clusters, supercomputers like the NEC SX-5 or the Japanese Earth Simulator and the ASCI White machine. While clusters of SMPs offer virtually unlimited scalability, they increase the complexity of user applications development by forcing programmers to deal with shared-memory as well as with distributed-memory parallelization issues.

2.3 Machines used in our experiments

Our experiments with different parallel versions of Cholesky factorization have been performed on three different parallel machines, an SGI Origin, an IBM p690 and a Beowulf SMP PC cluster.

2.3.1 SGI Origin

The SGI Origin 2000 [11] is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor. The Origin system consists up to 512 nodes interconnected by a scalable network. Each node consists of one or two MIPS processors, up to 4GB of coherent memory and a connection to a portion of the XIO I/O subsystem. For operational purposes the system behaves as a single shared-memory computer with the operating system taking care of maintaining memory consistency. Thus any processor can use the total amount of memory. In addition to being used in shared-memory mode the computer can also be used as a traditional message passing computer. The SGI Origin 2000 at Parallab, University of Bergen, which we used in our experiments consists currently of 62 MIPS RISC R10000 64-bit processors each with 192 MB of memory, and running at 195MHz. The aggregate peak performance is 24.8 GFlops/s and the aggregate memory is 12 Gbyte.

2.4 Beowulf SMP Cluster

The Beowulf-cluster located at the University of Vienna, consists of one front-end with two 400MHz Pentium II processors, and 16 compute nodes connected via Fast Ethernet and Myrinet. Each node is equipped with four Pentium III Xeon processors (700MHz) and 2 GB RAM. This system is a typical SMP cluster where the 4 processors of a node have access to a shared memory, while data transfers across nodes have to be accomplished by means of message passing.

2.5 IBM pSeries 690

The IBM p690 system is composed of multiple nodes, each usually consisting of 32 Power4 processors running at 1.3 GHz and sharing 32-64 GB of memory. For our experiments we used an IBM p690 Regatta Turbo system at the University of Bergen, with 3 nodes and a total of 96 processors and 196 GB memory. The operating system is AIX. The disk subsystem is based on the IBM 7133 Model D40 with SSA technology and has a total of 5.8 Terabyte disk capacity. The system has currently the most powerful CPU's of the supercomputers in Norway in 2002.

2.6 Parallel Programming Models

Despite the tremendous advances in computer architecture, there is no commonly accepted parallel programming model. The predominant software tools for developing scientific and numerical applications for parallel systems are still conventional programming languages like Fortran and C with extensions for parallelization. There are mainly two trends in parallel programming, depending on how the address space of parallel systems is organized. On the one hand, the standard application programming interface (API) for message-passing, MPI [3], is widely used, mainly for distributed-memory systems. Recently, a standard API for shared-memory parallel programming based on multi-threading, OpenMP [4], has become available. While OpenMP is restricted to shared-memory architectures only, MPI programs can also be executed on shared-memory machines and clusters. As a consequence, most application developers utilize pure distributed-memory programming models, usually based on the message passing interface (MPI [2]), and thus may miss a number of optimization opportunities offered by the shared-memory machines or SMP clusters.

2.6.1 Message Passing

Most message passing applications are based on the Single Program Multiple Data (SPMD) paradigm, where a program is parameterized in such a way that it can be executed on multiple processors, each executing the same program on its own local portion of the data. An SPMD programs are parameterized in a way that they can be executed on multiple processors, each executing the same program on its own local portion of the data. SPMD programs are usually developed by manually transforming an existing sequential program into parallel form according to the following three steps. First, the user has to partition the data space among multiple processors by modifying the allocation strategy of arrays. Second, parallel execution has to be achieved by partitioning the computations in accordance with the chosen data distribution. Finally, communication required for non-local data accesses has to be realized by inserting calls to appropriate communication routines from the message passing library. In order to achieve high efficiency, computations have to be partitioned evenly across processors while attempting to minimize communication overheads.

While the availability of MPI on virtually all parallel machines ensures portability of codes across different parallel architectures, machine specific program optimizations performed for one architecture may lead to performance losses on other parallel architectures. Therefore, portability of codes while maintaining the performance characteristics of a program cannot always be achieved with MPI. Although the SPMD paradigm offers a convenient transition path from sequential to parallel applications, transforming a sequential program into a distributed-memory parallel program usually requires a major code rewrite and the maintenance of two separate program versions, a sequential and a parallel version.

Although MPI has been developed for distributed-memory systems it can be employed on shared memory systems as well. However, MPI programs which are executed on clusters of SMPs usually ignore the shared-memory aspect and treat such machines as pure distributed-memory systems. Since MPI does not allow the programmer to directly utilize the shared-memory available within nodes a number of optimization opportunities may be missed.

2.6.2 Multithreading

Parallelization in a shared memory context is based on the concept of multi-threading. Shared memory parallelization is less complex than parallelization for distributed memory. Due to the availability of a global address space, a distribution of a program's data space is not required. Most applications for shared-memory systems have been parallelized using multi-threading libraries, like for example pThreads. Recently, OpenMP, a standard application programming interface for shared-memory parallel programs has become available. OpenMP supports the parallelization of both C/C++ and Fortran programs based on various directives and associated library routines. OpenMP provides support for parallel execution and work-sharing (i.e., distribution of computations among threads), for sharing or privatizing of data, and for thread synchronization. In order to transform a sequential program into a shared-memory parallel program first, the computations have to be distributed among multiple threads (e.g. distribution of loop iterations among threads) and, second, appropriate synchronization techniques to ensure consistency of shared data have to be applied. The execution of OpenMP programs is based on the fork-join model of computation. Under this model, a single master thread starts to execute the program sequentially. When a parallel section is reached, the master thread spawns a team of threads which execute the subsequent code in parallel. At the end of a parallel section all threads synchronize and only the master thread continues execution.

Since OpenMP is based on directives, in many cases a single source code can be maintained for both the sequential and parallel version of a code. Moreover, since no data distribution is required, programs can be parallelized incrementally. While OpenMP offers a simple API for shared-memory systems, it should be noted that OpenMP programs cannot be executed on distributed-memory systems or on SMP clusters.

2.6.3 Hybrid Parallelization

A promising approach for parallel programming on SMP clusters is based on the combination of MPI and OpenMP in a single application. Such a strategy attempts to fully exploit the potential of SMP clusters by relying on data distribution across the nodes of a cluster, and on multi-threading within the shared memory of nodes. While such an approach allows optimizing parallel programs by taking the hybrid architecture of SMP clusters into account, applications written in such a way tend to become extremely complex.

Under the hybrid model, a program is executed on an SMP cluster by a set of parallel processes, each of which runs usually on a separate node. Each process allocates a portion of the program's data in shared memory. Communication across node processes is realized by means of appropriate MPI message-passing primitives. In order to exploit additional shared-memory parallelism, each MPI node process generates a set of threads which run concurrently in the shared address space of a node. Usually the number of threads spawned within node processes is equal to the number of processors available within nodes.

The hybrid parallelization strategy offers a number of advantages compared to the usual distributed-memory parallelization strategy as realized by most HPF compilers. The hybrid strategy reflects closely the topology of SMP clusters by exploiting distributed-memory parallelism across the nodes and shared-memory parallelism within nodes. It allows a direct utilization of the shared memory within nodes and usually requires less total memory than the DM parallelization strategy. For example, replicated data arrays, which are mostly accessed in a read only way, have to be allocated only once per node, while in the distributed-memory model replicated arrays have to be allocated by each process, resulting in multiple copies on each node. Moreover, the hybrid model usually allows a more efficient handling of communication. Since data is distributed only across nodes and communication is performed only by the master thread, the hybrid model results in less messages as well as larger messages. However, on certain architectures, such a strategy, where only the master thread performs communication, may reduce the overall communication bandwidth.

2.7 Parallel Linear Algebra Libraries

Numerical linear algebra is fundamental to most calculations in scientific computing and is often the most computationally intensive part of such calculations. For solving linear algebra problems the linear algebra packages such as LAPACK [9] and ScaLAPACK [12] are widely used and nowadays there are versions of both optimized for different computer architectures.

2.7.1 LAPACK

The LAPACK[13] library has been designed for high-performance workstations and shared-memory multiprocessors. LAPACK is an example of a mathematical software package whose highest- level components are portable, while machine dependences are hidden in lower-level modules [14]. Such a hierarchical approach contributes to the software portability across diverse parallel architectures. This library uses block-partitioned algorithms for performing dense and banded linear algebra computations. The performance of algorithms depends on the implementation of some low-level operations, such as the dot product or matrix-vector product. The algorithms are designed to use block matrix operations, such as matrix multiplication, in the innermost loops. For fastest possible performance LAPACK requires highly optimized block matrix operations that are already implemented on each machine. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). In present there are highly efficient machine-specific implementations of the BLAS available for high-performance computers. Although BLAS routines are available on the net, many computer vendors supply tuned and optimized versions of BLAS for their particular architecture. There is a new paradigm called ”Automated Empirical Optimization of Software” (AEOS) which is based on software packages that automatically adapt given software for given architecture. ATLAS (AUtomatically Tuned Linear Algebra Software) [15] is the application of this new paradigm for the production of highly efficient routines based on the BLAS routines.

2.7.2 ScaLAPACK

The ScaLAPACK (or Scalable LAPACK) library [12] has been designed for massively parallel SIMD machines, parallel vector computers and for distributed-memory parallel computers. ScaLAPACK is based on a Single-Program-Multiple-Data (SPMD) programming paradigm using explicit message passing for interprocessor communication. The sequential BLAS, the BLACS and the PBLAS are modules from which the higher level ScaLAPACK routines are built. The ScaLAPACK library includes a subset of LAPACK routines redesigned for distributed-memory parallel computers. It assumes that matrices are laid out in a two-dimensional block cyclic decomposition. The PBLAS (Parallel BLAS) are an extended subset of the BLAS for distributed memory-computers and operate on matrices distributed according to a block cyclic data distribution scheme. Like LAPACK, most ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. In the ScaLAPACK and PBLAS routines, all interprocessor communication occurs within the BLACS. The way in which a matrix is distributed over the processes has a major impact on the load balance and communication characteristics of the current algorithm and so largely determines its performance and stability [16]. The block-cyclic distribution of data is parameterized by four numbers P, Q, m_b and n_b, where $P \times Q$ is the process grid and $m_b \times n_b$ is the block size. Simply said the set of processes is mapped to a virtual mesh, where every process is naturally identified by its coordinates in this PxQ grid. This virtual machine is in fact part of a larger object defined by the BLACS and called a context. An general MxN matrix operand is first decomposed into m_b by n_b blocks starting at its upper left corner. These blocks are then uniformly distributed across the process mesh. According to [16] the choice of parameters P and Q influences the performance. The best performance could be achieved in cases where $1/2 \leq P/Q \leq 1$. This inequality is valid for pairs of P and Q integers shown in the Table 1. We have used the given values as input parameters for our test cases.

N	1	2	4	8	16	32
P	1	1	2	2	4	4
Q	1	2	2	4	4	8

Table 1: Values of P and Q chosen for the ScaLAPACK test cases.

3 Shared-Memory Parallelization with LAPACK

In this section we present time measurement for shared-memory parallel versions of Cholesky factorization on three different machines. While on all machines the same source code has been used, machine-specific implementations of the LAPACK library have been linked.

3.1 IBM p690

At the IBM p690 Regatta Turbo system the linear algebra software packages that provide the main routines of LAPACK are called ESSL and PESSL. The ESSL (Engineering and Scientific Subroutine Library) provides a collection of mathematical subroutines optimized for the Power4 architecture [17]. ESSL contains standard libraries for dense matrix linear algebra operations, i.e., BLAS (Levels 1, 2, 3), LAPACK and many other linear algebra subroutines. Parallel ESSL provides parallel versions of the subroutines and is a scalable mathematical subroutine library that supports parallel processing applications. PESSL supports also the Single Program Multiple Data (SPMD) programming model using either the Message Passing Interface (MPI) signal or the MPI threaded library. As it was mentioned before the ESSL library contains the BLAS (Level 1, 2, 3) subroutines and a subset of LAPACK. It contains actually libraries for dense matrix operations. The BLAS and LAPACK libraries enable a user-program to reach performance levels that approach the theoretical peak performance of the Power4 processor.

N	TH=1	TH=2	TH=4	TH=8	TH=16
1000	0.14	0.09	0.07	0.05	0.04
2000	0.95	0.53	0.33	0.25	0.15
3000	3.06	1.59	0.91	0.67	0.37
4000	6.97	3.62	2.00	1.46	0.81

Table 2: Elapsed times for shared-memory parallel version of Cholesky factorization on the IBM p690. TH denotes the number of threads (processors) and N the size of the matrix (double precision).

In our implementation, the parallel PESSL library was linked to the main routine and the number of threads was set by the environment variable $OMP\_NUM\_THREADS$. Table 2 illustrates the execution time depending on the number of threads TH. In order to compensate work-load fluctuations due to multiple users, all performance measurements have been performed multiple times. As can be seen from Table 2, the algorithm scales better for test cases with larger matrices, since the relative synchronization overhead decreases. For matrix size of 3000 and more elements (double precision) almost linear scaling can be achieved.

3.2 SGI Origin

On the SGI Origin 2000 there are two vendor-specific scientific mathematical libraries *complib.sgimath* and its parallel counterpart $complib.sgimath\_mp$ that both include LAPACK. The library *complib.sgimath* contains a collection of standard libraries such as Level 1,2,3 BLAS library, EISPACK, LINPACK, and LAPACK. The calling sequence of the sequential and parallel routines are identical, just the linking is different. Linking with *complib.sgimath* results in a sequential version and linking with $complib.sgimath\_mp$ in a parallel version. The number of parallel threads again is specified by setting the environment variable $OMP\_NUM\_THREADS$.

N	TH=1	TH=2	TH=4	TH=8	TH=16	TH=32
1000	1.22	0.63	0.39	0.26	0.27	0.48
2000	9.64	4.93	2.51	1.47	1.15	1.89
3000	28.78	15.07	7.81	4.53	3.41	5.04
4000	70.34	36.31	19.41	10.86	8.15	9.39

Table 3: Elapsed times for shared-memory parallel version of Cholesky factorization on the Origin. TH denotes the number of threads (processors) and N the size of the matrix (double precision). LAPACK routines were utilized by linking the library $complib.sgimath\_mp$.

The time measurements for different number of threads are presented in Table 3. As can be seen, for up to 8 threads (processors) good speed-ups are achieved. On 16 processors only minimal performance gains can be observed, while using 32 processors results in a slow-down.

3.3 Beowulf Cluster

On the Beowulf cluster at the University of Vienna the BLAS routines have been optimized by means of ATLAS [15]. Using ATLAS, it is possible to automatically optimize BLAS according to characteristics of the processor and memory hierarchy.

Because the Beowulf cluster consists of nodes with four processors, three different versions (for two, for three and for four threads) of the BLAS library have been generated by means of ATLAS.

N	sequential	TH=2	TH=3	TH=4
1000	0.78	0.51	0.39	0.35
2000	5.51	3.17	2.36	1.93
3000	17.95	9.81	7.06	5.81
4000	41.76	22.25	15.86	12.60

Table 4: Elapsed times for shared-memory parallel version of Cholesky factorization on the Beowulf cluster using LAPACK with ATLAS-optimized BLAS routines. TH denotes the number of threads (processors) and N the size of the matrix (double precision).

Table 4 shows the execution times for the shared-memory parallel version of Cholesky on a single node of the cluster for 2, 3, and 4 threads in comparison with a sequential version (libpgf77blas). Again on larger matrix sizes very good speed-ups can be achieved.

N	sequential	TH=2
1000	0.82	0.70
2000	6.04	4.54
3000	20.13	14.42
4000	47.60	33.03

Table 5: Elapsed times for shared-memory parallel version of Cholesky factorization on the Beowulf cluster. TH denotes the number of threads (processors) used and N the size of the matrix (double precision). LAPACK with single and double threaded ASCI Red libraries were used.

In Table 5 we show the performance results when using the ASCI Red libraries. These libraries have been originally developed for the ASCI Red machine, which is one of the fastest supercomputers consisting of 4816 nodes with two processors each. The comparison of the results for two threads in Tables 4 and 5 shows that the version optimized by means of ATLAS outperforms the ASCI Red library by approximately 30%.

4 Distributed-Memory Parallelization with ScaLAPACK

The ScaLAPACK software package is equipped with a set of drivers for performing performance experiments for different machine and problem sizes. In our experiments we used the driver routine pdlltdriver.f which handles Cholesky decomposition by means of the subroutine pdpotrf.f. In the LLT.dat input file the user specifies the size of problem, number of test cases, values P and Q determining the virtual process grid, etc.

4.1 IBMp690

For our experiments on the IBM p690 ScaLAPACK was compiled with the standard Fortran compiler. In order to enable processors within the same node to perform MPI communication via shared memory copies, special compiler options were used.

N	NP=2	NP=4	NP=8	NP=16	NP=32
1000	0.09	0.06	0.05	0.03	0.02
2000	0.89	0.31	0.19	0.13	0.10
3000	2.68	1.00	0.57	0.43	0.24
4000	7.73	2.60	1.37	0.77	0.49

Table 6: Distributed-memory version of Cholesky factorization using ScaLAPACK on the IBM p690. NP indicates the number of processes (processors) and N the matrix size.

In Table 6 performance results for the distributed-memory version of Cholesky factorization using ScaLAPACK are shown. For large matrix sizes good speed-ups are achieved. Comparing these results to the shared-memory version (see Table 2) shows that for two and four processes (or threads) the distributed memory version is significantly slower (up to a factor of 2). However, when eight or more threads are used, the distributed memory version becomes superior.

4.2 SGI Origin 2000

On the SGI Origin 2000 computer system a distributed version of our code was obtained by by linking the ScaLAPACK with the SGI Mathematical library *complib.sgimath* library, which contains a SGI's version of the BLAS library. Table 7 shows the performance results of the distributed-memory version of Cholesky on the Origin 2000. Again, for larger matrices very good speed-ups are obtained.

Comparing these results to the shared-memory version (see Table 1) shows that on up to eight processes (or threads) the distributed memory version is slower, but only by approximately 25%. For 16 processors with both versions almost the same performance is achieved, while on 32 processors the distributed memory version performs best.

N	NP=1	NP=2	NP=4	NP=8	NP=16	NP=32
1000	1.34	0.97	0.52	0.48	0.27	0.26
2000	10.08	6.60	3.12	2.54	1.57	1.15
3000	33.01	21.59	9.84	6.26	3.56	2.85
4000	79.67	48.01	23.45	14.09	7.79	4.9

Table 7: Distributed-memory version of Cholesky factorization using ScaLAPACK on the SGI Origin 2000 with *complib.sgimath*. NP indicates the number of processes (processors) and N the matrix size.

4.3 Beowulf cluster

For the distributed-memory version of Cholesky factorization on the Beowulf cluster we used the GM 1.5 with MPICH-1.2.1..7b message passing system and executed the code on different numbers of nodes and different numbers of processors per node.

N	N1:P2	N2:P1	N1:P4	N2:P2	N4:P1	N2:P4	N4:P2	N8:P1
1000	0.50	0.47	0.35	0.30	0.29	0.27	0.23	0.20
2000	3.73	3.49	2.28	2.04	1.93	1.49	1.28	1.16
3000	12.53	11.66	7.19	6.45	6.10	4.36	3.81	3.46
4000	29.96	27.67	16.25	14.58	13.89	9.42	8.29	7.56

Table 8: Distributed-memory version of Cholesky factorization using ScaLAPACK on the the Beowulf cluster for different combinations of nodes (Nx) and processors per node (Px).

The resulting time measurements are presented in the in Tables 8 and 9 for varying number of nodes and processors, where Nx,($x = 1, 2, 4, 8$) denotes the number of nodes used, and Py,($y = 1, 2, 4$) the number of processes (processors) per node. As can be seen the best performance can be achieved when every process runs on a separate node, which, for example, can be seen in the columns N2:P1 for two processes, N4:P1 for four processes. However, using such a strategy only one forth of the availbale processors in the machine are utilized. When more than one processor per node is utilized no linear speed-ups are obtained since these processors have to compete for shared resources (e.g. the memory bus).

N	N8:P2	N4:P4	N16:P1	N16:P2	N8:P4	N16:P4
1000	0.15	0.18	0.13	0.14	0.17	0.12
2000	0.77	0.91	0.69	0.56	0.67	0.43
3000	2.20	2.44	1.96	1.43	1.71	1.05
4000	4.74	5.44	4.25	2.96	3.48	2.12

Table 9: Distributed-memory version of Cholesky factorization using ScaLAPACK on the the Beowulf cluster for different combinations of nodes (Nx) and processors per node (Px).

If we compare the distributed-memory version to the shared-memory version (see Table 4), on a single node of the cluster, we can see that the SM version is slightly better.

5 Hybrid parallelization

In this section we report on experiments with hybrid-parallel versions of Cholesky factorization that combine MPI message-passing and OpenMP multi-threading. On the IBM p960 that we had access to, it was not yet possible to test a hybrid version of our code. Therefore we present here only results for the SGI Origin.

Tables 10 and 11 illustrate the time measurement results for varying number of processes NP when two and four threads were set by means of the environment variable $OMP\_NUM\_THREADS$. The number of processors on which the tests were running corresponded to the number NP.

In Table 10 performance results of the hybrid version on utilizing up to 16 MPI processes, each with two threads, is shown. In the Table NP denotes the total number of processors used, while the number of MPI processes corresponds to NP/2.

N	NP=2	NP=4	NP=8	NP=16	NP=32
1000	0.78	0.58	0.39	0.35	0.26
2000	3.68	2.82	1.83	1.49	0.97
3000	11.28	7.95	4.86	3.79	2.36
4000	25.08	16.84	10.74	8.25	6.08

Table 10: Hybrid-parallel version of Cholesky factorization using ScaLAPACK with 2-threaded library complib.sgimath. NP denotes the total number of processors (threads) utilized. The number of MPI processes corresponds to NP/2.

In Table 10 performance results of the hybrid version utilizing up to 8 MPI processes, each with four threads, is shown. In the Table NP denotes the total number of processors used, while the number of MPI processes corresponds to NP/4.

For the large matrix (4000 elements), the hybrid version, where each MPI process spawned two threads (see Table 10), delivers better results than the shared memory version (see Table 3). However this is not the case for the 4-threaded hybrid version.

N	NP=4	NP=8	NP=16
1000	0.86	0.53	0.45
2000	3.85	2.77	1.90
3000	11.66	7.87	4.87
4000	25.81	16.81	10.67

Table 11: Hybrid-parallel version of Cholesky factorization using ScaLAPACK with 4-threaded library complib.sgimath. NP denotes the total number of processors (threads) utilized. The number of MPI processes corresponds to NP/4.

6 Related Work and Conclusions

Several researchers have investigated the advantages of a hybrid programming model based on MPI and OpenMP against a unified MPI-only model. Cappelo [5] et al. investigated a hybrid-parallel programming strategy in comparison with a pure message-passing approach using the NAS benchmarks on IBM SP systems. In their experiments the MPI-only approach has provided better results than a hybrid strategy for most codes. They conclude that a hybrid-parallel strategy becomes superior when fast processors make the communication performance significant and the level of parallelization is sufficient. Henty [18] reports on experiments with a Discrete Element Modeling code on various S MP clusters. He concludes that current OpenMP implementations are not yet efficient enough for hybrid parallelism to outperform pure message-passing. Haan [6] performed experiments with a matrix-transpose showing that a hybrid-parallel approach can significantly outperform message-passing parallelization.

As our experiments have shown, there is no single parallelization strategy for Cholseky factorization that is best on all architectures. Even on a single machine it can be advantages to switch parallelizations strategies depending on the size of the problem. In order to obtain a good hybrid-parallel version based in a combination of LAPACK and ScaLAPACK additional work in the implementation and configuration of these libraries is required.

”It will be some time in the future before truly transparent [19], complete efficient numerical software is available for cluster computing. Likewise, effective programming development and analysis tools for cluster computing are becoming available but are still in early stages of development.”

Acknowledgement: The work described in this paper was supported by the Special Research Program SFB F011 ”AURORA” of the Austrian Science Fund. The authors would like to thank the Parallab University of Bergen, Norway for providing access to their computer resources and mainly the Jan-Frode Myklebust for his support and help.

REFERENCES

[1] P. Blaha, K. Schwarz, G. K. H. Madsen, D. F. Kvasnicka, J. Luitz, *WIEN2k, An Augmented Plane Wave + Local Orbitals Program for Calculating Crystal Properties*, Vienna University of Technology, Institute for Materials Chemistry, Vienna, Austria, 2001.

[2] W. Gropp, E. Lusk, A. Skjelum, *Using MPI*, MIT Press, Cambridge London, 1994.

[3] *http://www.mpi-forum.org/*

[4] *http://www.openmp.org/*

[5] F. Cappello and D. Etieble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, Dallas, November 2000.

[6] O. Haan. Matrix Transpose with Hybrid OpenMP / MPI Parallelization. Technical Report, http://www.spscicomp.org/2000/userpres.html#haan, 2000.

[7] R. D. Loft, S. J. Thomas, J. M. Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *Proceedings SC2001*, Denver, USA, November 2001.

[8] G. H. Golub, C. F. Van Loan, *Matrix Computations,* 3rd ed., Johns Hopkins University Press, Baltimore, 1996.

[9] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. C. Sorensen: LAPACK *Users' Guide*, 3rd edn. SIAM Press, Philadelphia PA, 1999.

[10] C. W. Ueberhuber: *Numerical Computation.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

[11] Laudon J., Lenoski D., *The SGI:Origin:A ccNUMA Highly Scalable Server* Silicon Graphics, Inc., 2011 North Shoreline Boulevard, Mountain View, California 94043

[12] L. S. Blackford et al., SCALAPACK *Users' Guide*, SIAM Press, Philadelphia, 1997.

[13] E.Anderson, Z.Bai, C.Bischof, S.Blackford, J.Demmel, J.Dongarra, J.Du Croz, A.Greenbaum, S.Hammarling, A.McKenney, and D.Sorensen. LAPACK *User's Guide*, SIAM, Philadelphia, PA, USA, third edition, 1999.

[14] J.J.Dongarra, V.Eijkhout. *Numerical Linear Algebra Algorithms and Software* Journal CAM (Numerical) Linear Algebra.Volume 31(4), 28 October 1999.

[15] R. C. Whaley, A. Petitet, J. J. Dongarra, *Automated Empirical Optimizations of Software and the ATLAS Project*, Parallel Computing, **27**(1-2), 2001.

[16] J.Choi, J. J. Dongarra, L. S. Ostruchov, A. P.Petitet, D. W. Walker and R. C. Whaley, *The design and implementation of the SCALAPACK LU, QR and Cholesky factorization routines*, ORNL/TM-12470, 1994.

[17] Parallel Engineering and Scientific Subroutine Library for AIX Version 2 Release 3, Guide and Reference http://www.rs6000.ibm.com/doc_link

[18] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, Dallas, November 2000.

[19] S. Browne, J. J. Dongarra, A. Trefethen , *Numerical Libraries and Tools for Scalable Parallel Computing*, Cluster Computing BOF at SC'99, Portlad OR, November, 1999.