Portable Parallel Implementation of BLAS3

appeared in Concurrency Practice and Experience, Vol. 6, No. 5, pp. 411-459, 1994

Dganit Amitai
Amir Averbuch
Ronen Friedman
Eran Gabber

Department of Computer Science
School of Mathematical Sciences
Sackler Faculty of Exact Sciences
Tel-Aviv University
Tel-Aviv 69978, Israel

Abstract

Multiprocessors systems offer fast enough capabilities for handling large numerical tasks, such as real-life linear algebra problems. Yet, rogramming such systems has proven itself to be awkward, error prone and architecture specific.

One successful method for alleviating this problem, a method that worked well in the case of the massively pipelined supercomputers, is to use subprograms libraries. Those libraries are built to efficiently perform some basic operations, while hiding low-level system specifics from the programmer. Efficiently porting a library to a new hardware, be it a vector machine or a shared-memory or message-passing based multiprocessor, is a major undertaking. It is a slow process that requires an intimate knowledge of the hardware features and optimization issues.

We propose a scheme for the creation of *portable* implementations of such libraries. We present an implementation of BLAS (Basic Linear Algebra Subprograms), a highly used standard linear algebra library. We use VMMP (A Virtual Machine for MultiProcessors) — a set of services tuned for both shared memory and message-passing based MIMD machines [Gabber 1990]. Software created using this scheme is automatically efficient on both categories of MIMD machines, and on any hardware VMMP has been ported to. Additional level of abstraction is achieved using the programming language C++, an object oriented language [Eckel 1989, Stroustrup 1986]. For the programmer who is using BLAS-3, it is hiding both the data structures used to define linear algebra objects, and the parallel nature of the operations performed on these objects.

We developed a code of BLAS on top of VMMP. This code was run without any modifications on two shared memory machines — the commercial Sequent Symmetry and the experimental Taunop. Performance results for this implementation are given. The speedup of the BLAS routines, tested on twenty-two processors of the Sequent,

was in the range of 8.68 to 15.89. Application programs (e.g. Cholesky factorization) using the library routines achieved similar efficiency.

Key-Words:
BLAS, parallel, MIMD programming, C++

1 Introduction

1.1 Background

Many scientific and engineering problems cannot be effectively solved by direct methods. Their solution requires numerical methods. These problems arise in a variety of technical and scientific disciplines. Though they appear to be widely separated, a very large category of them can be characterized by local relationships on grid structures, that are translated into large systems of equations.

Handling these large grids requires a computing power several order of magnitude higher than what is currently available. Traditionally, large numerical problems are handled by supercomputers, which are very fast machines that are highly pipelined. Although the speed of supercomputers processors has increased by almost a factor of 100 over the last decade, the technology used to build them is approaching some theoretical limits of size and speed. It is evident that — at least until some technological breakthrough is found — single processors will not achieve the desired processing speeds. The only open way to accomplish the high demanding tasks is to use multiple processors.

The hardware technology needed to build large scale parallel machines is mostly available, but the use of such advanced hardware is limited by the lack of software. Software development for parallel machines, although started some thirty years ago, is slowed by the following obstacles:

Human thinking: it is difficult for (most) humans, whose thinking process is serial in its higher levels, to directly design parallel software systems.

Algorithms: to use parallel processors efficiently, one must make use of efficient parallel algorithms. This is especially true when dealing with message passing systems, where communication takes much of program execution times. However, most of the algorithms in use are inherently serial. Some of these can be adapted to parallel use; others must be replaced by new algorithms.

Lack of portability: there are many approaches to the design of parallel systems. As a result — there are many types of parallel hardware, each type using its own architecture, and mandating a different set of considerations in algorithm design. Thus, software solutions developed for one system, cannot be easily ported to another.

There are two main categories of solutions to these problems, when linear algebra problems are considered. One approach is to supply the user with easy to use access to the multiprocessing capabilities of the hardware and the operating system. There exist, in different stages of research and use, a few parallel languages that are designed to do just that

— they enable the user to use higher level constructs for parallelism and synchronization, avoiding some of the complexity of parallel coding. The higher level language constructs are compiled into the appropriate hardware-specific or operating system specific commands.

Parallel languages ease the implementation of parallel algorithms and as they are usually available on more than one architecture, promote the portability of applications. Code generated using these languages is potentially efficient, due to its gross-grained parallelism. In this case, medium to large blocks of code are executed independently by each processor between consecutive synchronization points. Gross-grained parallelism has the advantage of low synchronization overhead, as there are less synchronization points. More than that, the programmer has full control over the parallel implementation of his algorithm, and can make full use of the hardware capabilities. However, parallel languages underline the first problem mentioned above, the difficulty of conceiving programs that operate more than one processor simultaneously.

A second category of solutions, the code libraries, deals mainly with this problem, using finer-grained parallelism. Most of the advanced (vectorized, multiprocessors and multivector) computer systems provide optimized mathematical libraries. The routines in the libraries implement basic mathematical operations, and are tuned to efficiently use the hardware capabilities. This way, the programmer is supplied with efficient subprograms that hide the parallel nature of the machine.

Such libraries of routines were initially specific to the hardware. To promote the portability of applications, a new approach to linear algebra libraries was successfully adopted — the use of standard libraries as an applications standard building block. The efficiency achieved by the use of optimized libraries is maintained.

A good example for a standard library is BLAS — Basic Linear Algebra Subprograms [Lawson *et al.* 1979, Dongarra *et al.* 1988a, Dongarra *et al.* 1990a]. BLAS is a common set of basic Linear Algebra routines. It is widely used, being ported to many advanced architectures and being used in the implementation of LINPACK[Louter-Nool 1989]. Application code based on calls to BLAS is portable to any machine with an implementation of this widely available library. Not so BLAS itself. To make an *efficient* use of BLAS using a parallel machine, BLAS code must be modified and adapted to the hardware architecture. This is not a trivial task — it might even require a complete rewriting of the code.

1.2 Overview

Trying to alleviate this problem of portability, we have created an implementation of BLAS which makes use of VMMP[Gabber 1990]. Here the low level parallel operations are handled by VMMP. This BLAS implementation is unencumbered by any parallel or hardware specific constructs. Thus, it is portable to any machine running VMMP.

Portability, ease of use and the desire to encapsulate linear algebra constructs led to the choice of the object oriented language C++ [Eckel 1989, Stroustrup 1986] as the implementation language. For the programmer who is using BLAS-3, the use of C++ has added one more level of abstraction, by hiding both the data structures used to define linear algebra objects, and the parallel nature of the operations performed on these objects.

Section 2 introduces BLAS, its various levels, and VMMP. Our primary target was the implementation of 3-level BLAS. Section 4 details this implementation. During implementation, we had to consider the special nature of the VMMP "virtual machine". These performance issues are presented in section 5.1, along with results in section 5. Section 6 contains some well known applications, for which the efficiency and ease of use of our implementation can be demonstrated.

2 BLAS and VMMP

2.1 Introduction to BLAS

2.1.1 BLAS-1

BLAS (Basic Linear Algebra Subprograms) is the name of three packages of subprograms for basic operations of numerical linear algebra. The first of these packages, Level-1 BLAS, was created between the years 1973 and 1977 [Lawson *et al.* 1979]. It is a set of 38 subprograms operating on vectors and performing dot product (uv), vector plus a scalar times a vector ($u + \alpha v$), Givens transformations, vector norm ($\|v\|$), copy and swap, vector scaling (αv) and finding the largest element in a vector.
BLAS-1 was created with the following goals in mind:

- to serve as a conceptual aid to the design and coding of mathematical programs, by supplying a set of common basic building blocks.
- to aid the portability of the designed code. The Level-1 BLAS library was intended to be used as a common ground for applications, replacing mathematical libraries specific to hardware or site.
- to improve performance. As a significant part of the execution time of linear algebraic programs might be spent in a few low level loops, optimizing these operations can save a lot of execution time.
- to improve code readability, by the use of mnemonics for basic operations. BLAS-1 routine names are such mnemonics (e.g., *sswap()* for vectors swapping, or *dcopy()* for copying).
- to ensure the proper handling of special algorithmic and implementation-derived properties of the low level operation. To ensure the proper handling of error conditions.

BLAS-1 gained widespread acceptance, and has become a de facto standard for the elementary vector operations.

2.1.2 BLAS-2

Level-1 BLAS is not suited for newer computer architectures. This problem was first identified on *vector* machines. It was shown, e.g. by Dongarra and Eisenstat (and, using similar arguments, in [Dongarra *et al.* 1984]), that the vector-vector operations of BLAS-1 are not the best way to improve the performance of higher-level code [Dongarra and Eisenstat 1984].

Working on the vector-vector level precludes, for example, the efficient reuse of machine vector registers in SAXPY ($Ax + y$) operations. The optimization on vector machines should be performed at least in the level of matrix-vector operations, to provide more efficient register allocation and management, and more efficient memory bandwidth usage.

Dongarra, Du Croz, Hammarling and Hanson identified a set of elementary *matrix-vector* operations that occur frequently in the implementation of common linear algebra algorithms [Dongarra *et al.* 1988a]. These operations, that make better use of vector registers, were developed into a set of FORTRAN routines, Extended BLAS (also known as Level-2 BLAS or BLAS-2), described in [Dongarra *et al.* 1988a, Dongarra *et al.* 1988b]. BLAS-2 is indeed superior to BLAS-1 on vector machines (see, for example, [Louter-Nool 1989]), and replaced it in many mathematical libraries (like LINPACK)

2.1.3 BLAS-3

BLAS-2, which is one level higher than BLAS-1, provides the vectorization needed to serve as a tool for creating efficient linear algebra programs on vector processors. But again it fails when applied to newer hardware architectures that use *hierarchical memory* systems: having a fast cache, some local memory of moderate speed, and a slower global (shared) memory. Data at the higher levels of the memory hierarchy can be accessed immediately, whereas data on the lower (slower) levels cannot be accessed fast enough to feed the arithmetic units. Thus, to be efficient, algorithms on hierarchical memory machines must have a reasonable amount of *data locality* [Gallivan *et al.* 1987]. Level-2 BLAS routines are optimized for vector processing, i.e. for generating long vectors as input for the vector pipeline. This is not the way to go where data locality is an issue. A better approach here is to partition the problem data (the matrices) into blocks (submatrices, usually), providing each processor with different data blocks to work on. It is assumed that those blocks are small enough to fit higher in the memory hierarchy — in local memory, or better still, in the processor's cache.

Optimization according to this approach cannot be performed on the matrix-vector level. The full structure of the matrix is needed as input, mandating the use of matrix-matrix operations. BLAS-3, the last set of basic routines, is defined in these terms. The BLAS-3 routines handle single and double precision, real and complex, numbers. All of the routines are low-level, in the sense that there is, for example, no linear system solving routine. These are provided by higher-level libraries, e.g. LINPACK. On the other hand, symmetric and triangular matrices are handled specifically, for performance's sake.

In [Berry *et al.* 1986] there is a comparison of BLAS-2 and BLAS-3 based LU factorization algorithms, tested on the CEDAR that is a hierarchical memory parallel computer. The BLAS-3 version (using a block LU algorithm) is shown to run about 3.5 times faster for matrices of size not less than 512. Similar performance results on other systems can be found in [Anderson *et al.* 1990], and are explained in terms of data locality in [Bischof 1990].

2.1.4 The Model Implementation

The model implementation, mentioned in some of the articles referenced, is a pure FORTRAN implementation of the different BLAS levels [Dongarra *et al.* 1988b, Dongarra *et al.* 1990b]. It is available from the mail-server NETLIB. The model implementation uses FORTRAN, and is thus highly portable but not optimized. It may be used as a basis to derive a machine-specific optimized implementation from.

2.2 VMMP

The VMMP (Virtual Machine for MultiProcessors) is a software package, which provides a coherent set of services for parallel application programs running on diverse MIMD multiprocessors, both shared memory and message passing [Gabber 1990]. It is intended to simplify parallel program writing and to promote portable and efficient programming. VMMP ensures high portability of application programs by implementing the same services on all target multiprocessors. It may also serve as a common intermediate language between very high level language compilers and various target multiprocessors.

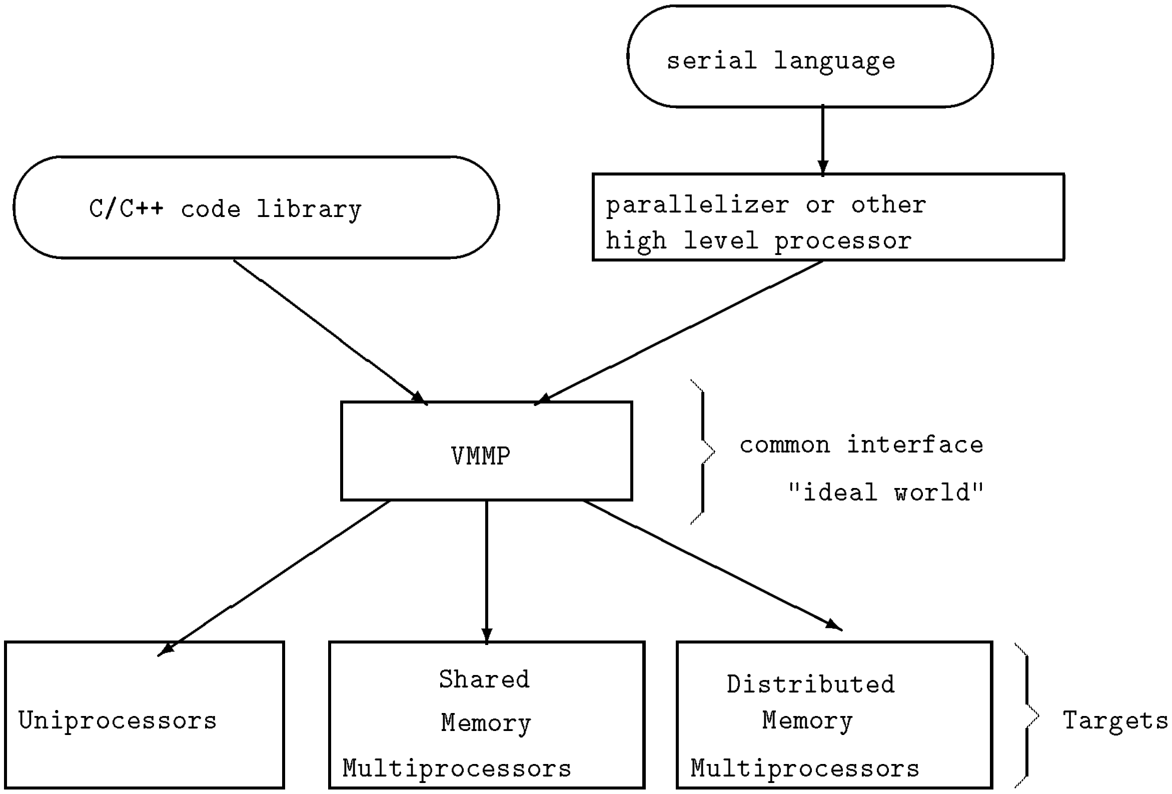

Figure 1: VMMP usage

VMMP supports two programming models: a *virtual tree machine* for *tree computations*, and a cooperating set (*crowd*) of processes for *crowd computations* (the later was chosen for our BLAS-3 implementation):

Tree computations are a general computation strategy, which solves the problem by breaking it into several simpler subproblems, that are solved recursively. The solution process looks like a tree of computations, in which the data flows down from the root into the leaves and solutions flow up towards the root. Some examples to tree computations are divide and conquer algorithms, game tree search and combinatorial algorithms.

Crowd computations are used for algorithms that are defined in terms of a set of cooperating processes. Each of these processes is an independent computational entity, which communicates and synchronizes with its peers. The processes communicate and synchronize mainly by message passing. A crowd may create a tree computation and vice versa.

A basic tool used by both models is the *shared (memory) object*. Shared objects provides a flexible communication mechanism in addition to parameters and message passing. Shared objects may be regarded as virtually shared memory areas, which contain any type of data, and can be accessed only by secure operations. Shared objects can be created, deleted, passed by reference, read and written.

The shared objects are intended to distribute data, collect results and collect global results by some associative and commutative operations. Other uses of the shared objects, such as random updates to locations of a large object, are discouraged, because the implementation of these operations incur high overhead on a message passing multiprocessor. The user must manipulate the shared objects only through VMMP services, which guarantee the correctness of the operations. Figure 2 illustrates the relationship among VMMP services.

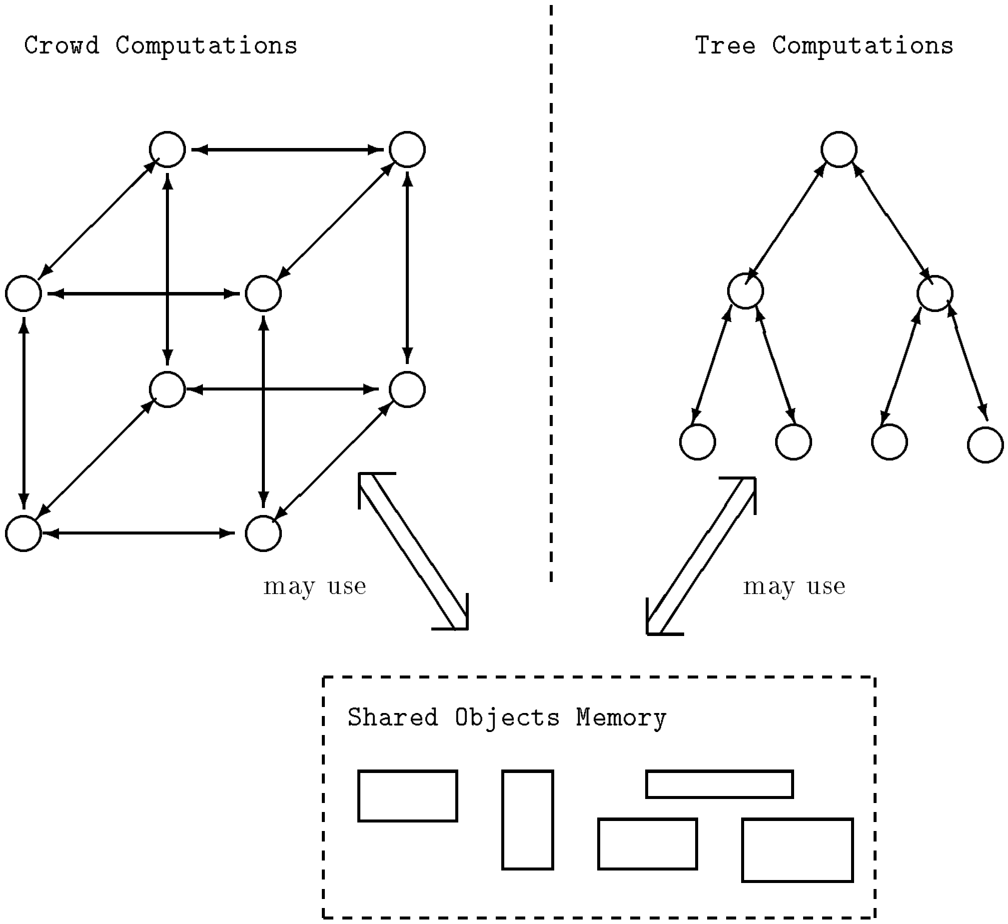

Figure 2: VMMP services

2.3 VMMP Services

VMMP provides the abstract machine interface for the convenient use of tree computations, crowd computations and shared memory objects. As such, VMMP does not support base level services like semaphores and barriers. The use of its services ensures the efficient implementation on a wide variety of target machines, and the ability to effectively hide implementation details from the user or the application library developer.

The VMMP services may be used to create, synchronize and terminate crowd processes. These processes can communicate using message passing services, or through the use of the above-mentioned shared objects. Shared objects may be created, initialized, updated and deleted. VMMP supports several data distribution strategies by some crowd operations and by shared objects operations.

Crowd services are used in the following manner: VMMP creates an initial process, which executes the user's main program. This process may spawn other processes using the `Vcrowd` service, which creates a *crowd* of cooperating processes running concurrently with the father process. The `Vwait` service forces the father process to wait until the termination of all its child processes in order to assure correct output results.

Crowd processes receive input and produce output using *shared objects*. Shared objects are created and initialized by the `Vdef` service. The `Vset` service change the value of a shared object and `Vget` retrieves the current value. Other services used will be detailed in the code examples later on. All VMMP services are documented in [Gabber 1990].

Table 3 shows the efficiency of VMMP's services. The following table lists the execution time required for some VMMP services on our eight-processor NS32532 based system. The low overhead of VMMP, compared to similar services provided by an operating system, is quite evident.

# of processes	1	2	3	4	5	6
Crowd creation	0.6	0.7	1.6	1.2	1.4	1.7
Crowd synchronization through *Vcombine()*	0.1	0.5	0.8	1.8	2.1	3.5
Concurrent fetch from shared memory (0.25MB)	70.7	126.7	212.3	281.3	356.0	426.7
Concurrent store of equal parts into shared memory (total of 128KB)	38.5	31.9	31.5	31.8	32.3	31.1
Concurrent *Vsend()* of equal messages (total of 128KB)	—	21.3	68.3	136.0	238.0	348.3

Figure 3: Time (in milliseconds) consumed by VMMP Services

Originally developed on MMX [Gabber 1988], an experimental shared memory multiprocessor built at Tel-Aviv University, VMMP is currently supported on the ACE, a shared memory multiprocessor built at IBM T.J. Watson Research Center running Mach, and on four computers running versions of UNIX: SUN3 workstations, Sequent machines, IBM RTs and National Semiconductor's VME532 systems. It is currently being ported to a network of Transputers.

3 Implementing BLAS-2

3.1 BLAS-2 — Not for VMMP

BLAS-2 routines (already mentioned in 2.1.2) deal with matrix-vector operations for various matrix kinds and various number representations. These routines perform $O(n^2)$ operations on $O(n^2)$ data. In other words — an element of the problem matrix is involved in only $O(1)$ (one or two, at the most) operations. This basic fact prohibits the use of VMMP to implement BLAS-2 routines:

A VMMP implementation routine copies its data into a shared memory object — to be used by the crowd processes (see 2.2) — and back. The crowd process that actually performs the mathematical operation, must also copy its slice of data from the shared memory object to its processors' local memory, and back. All together, we have at least four data moves in and out of slow memory (the shared memory object). Thus — the number of overhead operations performed on each element is of the same order as (and even bigger than) the number of computations.

The following section shows the performance results achieved for some BLAS-2 routines in a trial BLAS-2/VMMP implementation. Note that we do get a good speedup in the computations execution, but VMMP's overhead changes the picture.

3.2 Trial Implementation Performance Results

3.2.1 Routine Names

The conventions used to name BLAS-2 functions can be found in [Dongarra *et al.* 1988a]. We have created a parallel implementation of BLAS-2 routines for double-precision general matrices, i.e. — **dgemv()**, **dsymv()**, **dsymv()**, **dtrmv()**, **dtrsv()**, **dger()**, **dsyr()** and **dsyr2()** (see below).

3.2.2 Argument Conventions

The argument names in the calling sequences listed below should be interpreted as follows:

- **a** — A matrix. Its exact dimensions depend upon the specific routine and the other arguments.
- **lda** — number of data elements in one row of a's memory. It is the number of memory locations between the first element of one row and the first element of the next. This number can be obtained using the supplied method *mem_cols()*.
- **m,n** — the matrix' dimensions.
- **x,y** — vectors.
- **incx, incy** — the storage spacing (stride) of the vectors elements in the memory array.
- **trans** — "t"/"T" means the argument matrix should be transposed.

- **uplo** — "l" means only the lower triangle of a symmetric or triangular matrix should be referenced. "u" - the upper triangle.
- **diag** — "u" means the leading diagonal of a symmetric or triangular matrix is all '1', and the matrix' memory locations holding it should not be referenced.
- **alpha, beta** — scalars.

3.2.3 Performance Tables

We describe here the operation performed by each implemented BLAS-2 routine, its arguments, and the speed-up achieved in its parallel implementation. Note that the resolution of the timers used is about 30 milliseconds (50 milliseconds for the times reported for serial implementations). Timing overhead is of the same magnitude (about 30 milliseconds). Memory constraints limited us to problems small enough for these times to be meaningful.

dgemv(): general matrix-vector products.

- called: `dgemv(trans,m,n,alpha,a,lda,x,incx,beta,y,incy)`
- operation:

$$y = \alpha A x + \beta y \qquad \text{for trans="n"}$$
$$y = \alpha A^T x + \beta y \qquad \text{for trans="t"}$$

- performance:

Problem size — $360 * 360$.
Serial time — 256 milliseconds.

processors	1	2	3	4	5	6
Total time	1.03	0.70	0.67	0.60	0.57	0.57
Speedup	0.25	0.37	0.38	0.43	0.45	0.45
Relative speedup	—	1.48	1.55	1.72	1.81	1.81
Computations	0.53	0.27	0.17	0.13	0.1	0.067–0.1

dsymv(): symmetric matrix-vector products.

- called: `dsymv(uplo,n,alpha,a,lda,x,incx,beta,y,incy)`
- operation:

$$y = \alpha A x + \beta y$$

$$A \text{ symmetric}$$

- performance:

Problem size — $360 * 360$.
Serial time — 600 milliseconds.

processors	1	2	3	4	5	6
Total time	1.73	1.30	1.07	1.00	0.90	0.77
Speedup	0.35	0.46	0.56	0.60	0.67	0.78
Relative Speedup	—	1.33	1.62	1.73	1.93	2.26
Computations	1.03	0.77	0.57	0.53	0.47	0.3–0.4

dtrmv(): triangular matrix-vector products.

- called: `dtrmv(uplo,trans,diag,n,alpha,a,lda,x,incx)`
- operation:

$$x = \alpha A x \qquad \text{for trans=“n”}$$

$$x = \alpha A^T x \qquad \text{for trans=“t”}$$

- performance:

Problem size — $360 * 360$.
Serial time — 283 milliseconds.

processors	1	2	3	4	5	6
Total time	0.93	0.77	0.67	0.60	0.60	0.57
Speedup	0.30	0.37	0.42	0.47	0.47	0.50
Relative speedup	—	1.22	1.40	1.56	1.56	1.64
Computations	0.27	0.20	0.17	0.13	0.10	0.067

dger(): general rank-1 updates.

- called: `dger(m,n,alpha,x,incx,y,incy,a,lda)`
- operation:

$$A = \alpha x y^T + A$$

- performance:

Problem size — $360 * 360$.
Serial time — 900 milliseconds.

processors	1	2	3	4	5	6
Total time	2.50	1.43	1.30	1.17	1.10	1.03
Speedup	0.36	0.63	0.69	0.77	0.82	0.87
Relative speedup	—	1.75	1.92	2.14	2.27	2.42
Computations	0.90	0.43	0.30	0.23	0.20	0.17

dsyr(): symmetric rank-1 updates.

- called: `dsyr(uplo,n,alpha,x,incx,a,lda)`
- operation:

$$A = \alpha x x^T + A$$

$$A \text{ symmetric}$$

- performance:

Problem size — $360 * 360$.
Serial time — 433 milliseconds.

processors	1	2	3	4	5	6
Total time	1.57	1.27	1.17	1.17	1.17	1.13
Speedup	0.28	0.34	0.37	0.37	0.37	0.38
Relative Speedup	—	1.24	1.34	1.34	1.34	1.38

dsyr2(): symmetric rank-2 updates.

- called: `dsyr2(uplo,n,alpha,x,incx,y,incy,a,lda)`
- operation:

$$A = \alpha x y^T + \alpha y x^T + A$$

$$A \text{ symmetric}$$

- performance:

Problem size — $360 * 360$.
Serial time — 467 milliseconds.

processors	1	2	3	4	5	6
Total time	1.70	1.17	1.07	1.10	1.10	1.10
Speedup	0.275	0.401	0.438	0.425	0.425	0.425
Relative speedup	—	1.458	1.595	1.545	1.545	1.545
Computations	0.54	0.37	0.23	0.17	0.17	0.13

4 Parallel Implementation of BLAS-3

4.1 BLAS-3 Routines

4.1.1 Conventions

The set of Level-3 BLAS routines is described in [Dongarra *et al.* 1990a]. Here we will briefly repeat routine names, operation, and calling sequences.

The first letter of a routine name reveals the data type it is supposed to handle: S and D stand for real numbers (single and double precision), C and Z for complex numbers. We implemented the routines for a user defined type *main_t*. Currently, *main_t* is defined as C++'s *double* and the implementation is tuned accordingly. For this reason we will use the 'D' prefix when referring to routine names. This definition can easily be changed to any predefined or user defined type (*complex*, for example).
The argument names in the calling sequences should be interpreted as follows:

- a, b, c — the matrices. Their exact dimensions depend upon the specific routine and the other arguments.
- lda, ldb, ldc — number of data elements in one row of the matrix' memory. It is the number of memory locations between the first element of one row and the first element of the next. This number can be obtained using the supplied method *mem_cols()*.
- m,n,k — matrices dimensions.
- trans — "t"/"T" means the argument matrix should be transposed.
- uplo — "l" means only the lower triangle of a symmetric or triangular matrix should be referenced. "u" — the upper triangle.
- diag — "u" means the leading diagonal of a symmetric or triangular matrix is all '1', and the matrix' memory locations holding it should not be referenced.
- side — a "l" (left) or "r" (right) values choose one of two options for the routine behavior.
- alpha, beta — scalars.

4.1.2 Routines Description

dgemm(): general matrix-matrix products

- called: `dgemm(transa,transb,m,n,k,alpha,a,lda,b,` `ldb,beta,c,ldc)`
- operation:

$$C = \alpha\mathcal{A}\mathcal{B} + \beta C$$

$$\mathcal{A} \text{ is } A \text{ or } A^T$$

$$\mathcal{B} \text{ is } B \text{ or } B^T$$

dsymm(): matrix-matrix products where one matrix is symmetric

- called: `dsymm(side,uplo,m,n,alpha,a,lda,b,ldb,beta,c,ldc)`
- operation:

$$C = \alpha AB + \beta C \qquad \text{for side=“l”}$$
$$C = \alpha BA + \beta C \qquad \text{for side=“r”}$$

dsyrk(): rank-k updates of a symmetric matrix

- called: `dsyrk(uplo,trans,n,k,alpha,a,lda,beta,c,ldc)`
- operation:

$$C = \alpha AA^T + \beta C \qquad \text{for trans=“n”}$$
$$C = \alpha A^T A + \beta C \qquad \text{for trans=“t”}$$

dsyr2k() : rank-$2k$ updates of a symmetric matrix

- called: `dsyr2k(uplo,trans,n,k,alpha,a,lda,b,ldb,beta,c,ldc)`
- operation:

$$C = \alpha AB^T + \alpha BA^T + \beta C \qquad \text{for trans=“n”}$$
$$C = \alpha A^T B + \alpha B^T A + \beta C \qquad \text{for trans=“t”}$$

dtrmm() : triangular matrix-matrix product

- called: `dtrmm(side,uplo,trans,diag,m,n,alpha,a,lda,b,ldb)`
- operation:

$$B = \alpha \mathcal{A} B \qquad \text{for side=“l”}$$
$$B = \alpha B \mathcal{A} \qquad \text{for side=“r”}$$
$$\mathcal{A} \text{ is } A \text{ or } A^T \text{ according to trans}$$

dtrsm() : solution of a triangular system of equations

- called: `dtrsm(side,uplo,trans,diag,m,n,alpha,a,lda,b,ldb)`
- operation:

$$B = \alpha \mathcal{A}^{-1} B \qquad \text{for side=“l”}$$
$$B = \alpha B \mathcal{A}^{-1} \qquad \text{for side=“r”}$$
$$\mathcal{A} \text{ is } A \text{ or } A^T \text{ according to trans}$$

Example:

```
dtrsm("right","low","no transpose","unit",
  b.rows(),b.cols(),1.5,a,a.mem_cols(),b,b.mem_cols());
```

4.2 Guide Rules for the Implementation

The following principles guide this implementation:

- Using standard BLAS-3 call conventions:
  as the C++ *matrix* contains its dimensions and memory organization data, we do not really need most of the parameters in BLAS-3 calls. The calling conventions are kept (see Fig. 4), to make porting programs to our C++ BLAS-3 implementation easier. The user is totally unaware of the parallel nature of the implementation. We do supply an alternative set of routines, with the BLAS-3 functionality, but with less arguments.

```
do_something(matrix& A, matrix& C)
//  compute C = 2.5AA' - 3C
//  A is n*k matrix
//  C is symmetric, n*n matrix
{
     //  call conventions are kept
     dsyrk("lower", "no transpose", A.rows(), A.cols(),
          2.5, A, A.mem_cols(), -3.0, C, C.mem_cols());
}
```

Figure 4: Calling BLAS-3

- Code portability:
  our main goal is to create a portable implementation of BLAS-3, one that can be ported with no pains to any environment supported by VMMP. By doing so, we avoid the use of any features of the specific hardware we used, or even any knowledge of the real nature of VMMP's shared memory objects, i.e. we never access shared memory directly — we use VMMP services.

- Efficiency:
  performance and implementation efficiency are a prime target whenever the two previous rules allow. The main result is a tendency to ignore the packaging and data hiding mechanisms provided by C++, if their use incurred a performance cost.

- Clean design:
  throughout the implementation we emphasize structured design, as long as that did not have major impact on performance. Our approach calls for the use of the tools provided by the implementation language (C++). We define auxiliary *classes* when needed (like timers for performance measurements), prefer to access matrices internal data through exported methods rather than directly, and use common building blocks (routines and macros) in the design of the kernel routines.

4.3 VMMP Usage

4.3.1 Crowd Routines

The BLAS-3 implementation uses VMMP's *crowd* services and its *shared memory objects* (section 2.2). A BLAS-3 routine starts by creating shared memory objects for the matrices passed to it as arguments. The matrices data is copied into the shared memory objects, in a compact and sometimes transposed format. Next, the routine creates a crowd of processes, each to handle part of the calculations required.

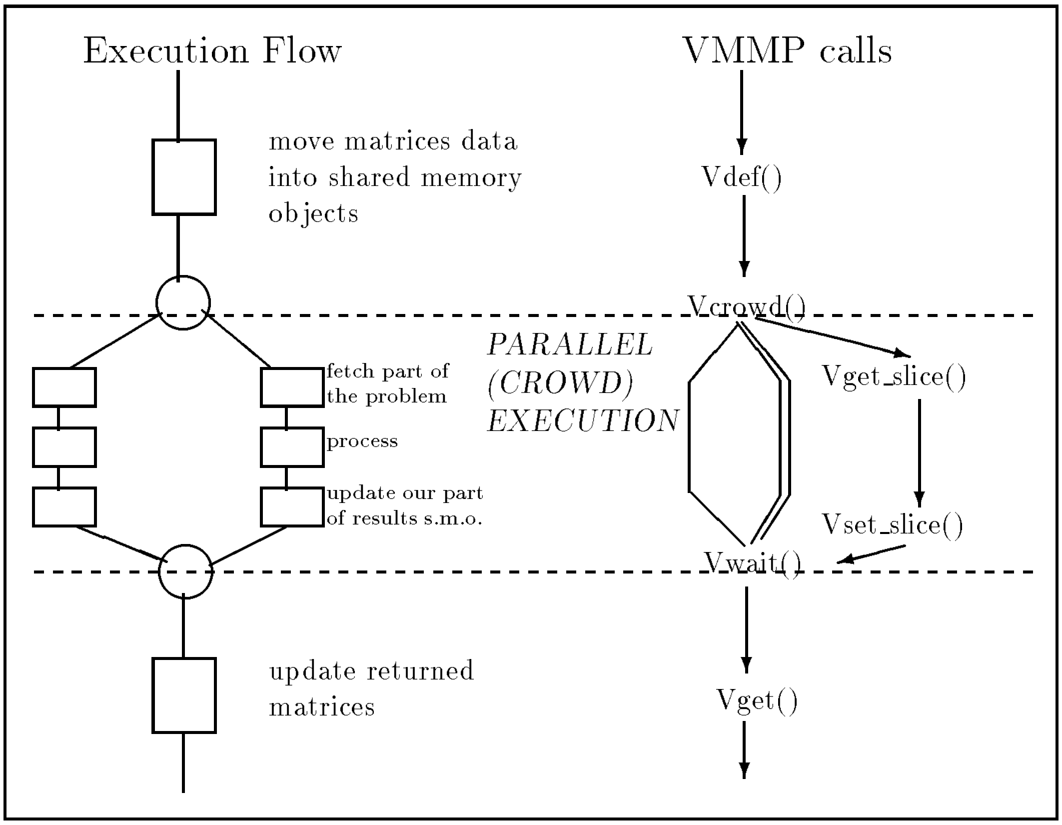

Figure 5: Routine Structure

Every crowd process fetches part of the problem matrices from shared memory objects into local store. See in section 4.3.3 how the handled part of the matrices is determined. After doing its share of the computational work, the crowd process copies its modifications back into the shared memory object, and quits. The parent routine, when all crowd processes have terminated, fetches the updated result matrix from the shared memory object, and uses that data to update one of the argument matrices.

4.3.2 Services Used

The following VMMP services are used:

- **Vcrowd()** and **Vwait()** — to create the crowd processes, and then to wait for their termination.

- **Vdef()** and **Vdefu()** — to create shared memory objects and, in the case of Vdef(), to initialize them with the matrices data. Vdefu() is used whenever we want to hold the transpose of a matrix in the shared memory, or when the original matrix' data is not contiguous in memory (like when submatrices are handled).
- **Vget()** and **Vget_slice()** — the first is used by a crowd process to get a local copy of a shared memory object. The second only brings over a part of the object (usually — a range of rows).
- **Vset_slice()** — used by the crowd process to update the result matrix in shared memory. Never do two crowd processes update the same part of the result matrix. Whenever possible, the updated locations are part of one of the original matrices — the same part used by that crowd process (it is not possible to have crowd process A update a row which is used as argument to crowd process B. We try to avoid any synchronizations apart from the Vwait() at the end of the parallel processing).

Some auxiliary and housekeeping routines are also used, like **Vcrowd_part()** that helps divide a range of numbers between the processes, or **Vproc_id()** that is used for scattered distribution of matrix rows.

4.3.3 Matrix Partitioning

Matrices that appear on the left side of the multiplication (like the A in the $C = \alpha AB + \beta C$ operation of **dsymm()**) are usually distributed between the parallel crowd processes by rows. This corresponds to the way C++ holds two dimensional arrays in memory, and is faster to implement. The range of rows allotted to each process depends upon the matrix structure. For regular matrices computations, the amount of work needed to handle a row of elements does not change with the row number. In these cases **Vcrowd_part()**, a VMMP service, is used to partition the work uniformly. When dealing with symmetric or triangular matrix (like the B in the $B = \alpha AB$ operation of **dtrmm()**) we resort to scattered allocation.

When an argument matrix appears on the right hand side of the multiplication, it is usually not partitioned at all, but rather brought whole by one Vget() operation. In **dgemm()** (that calculates $C = \alpha AB + \beta C$) for example, each crowd process handles only a range of A rows but needs to have all of B.

There is one special case: **dsyr2k()**. Here we have to calculate $AB^T + BA^T$. As the result is a symmetric matrix, we only need half of it (when dealing with symmetric matrices, only the upper or lower triangle of the matrix is referenced or updated). We try to use that fact to avoid bringing in all the elements of A and B. So, A is partitioned by rows, and B is fetched according to A's range, causing crowd processes handling higher-numbered lines in A to have an increased workload. We use a primitive routine of our own to partition the work according to a predefined array of value ranges.

4.4 Performance Optimizations

4.4.1 The VMMP Execution Model

VMMP is meant to run both on shared memory and message passing hardware. To maintain programs compatibility between the two different environments, VMMP restricts the use of the shared memory (section 2). A program that uses VMMP defines and uses shared memory objects that are copied back and forth between processors' local memory and the shared memory. VMMP handles the synchronization issues that arise when updating multiple copies of the same shared memory segment.

The effect is a dramatical increase in the effective time it takes a processor to modify a shared memory location. Thus, making the VMMP "virtual machine" an extreme example of an hierarchical memory system. The efficiency of parallel algorithms is heavily affected by this fact, and many of them become unusable. On the other hand, the VMMP virtual machine differs from message passing computers, too. Message passing algorithms are designed to take into account high cost of distributing problem elements to the processors, and the need to shorten the paths data elements should pass, transferred from one processor to another. When using VMMP the data transfers are hidden from the programmer, on shared memory and message passing systems alike. The programmer can no longer use his knowledge of the hardware architecture to create a faster implementation of an algorithm. Moreover, trying to impose message-passing-type considerations on a program will result in a poor performance on shared memory hardware. All the above can be summarized in the following design rule, that apply both to shared memory based and message passing based VMMP: *the VMMP machine has an hierarchical memory system. Data locality is the most important factor when trying to minimize VMMP overhead.*

There is one more design rule emanating from the way typical VMMP hardware (unlike vector machines, for example) is designed. It is best to take into account the existence of some hardware processor and shared memory caches of some unspecified size and line length. Which means that the library's internal program loops should access memory locations in consecutive order whenever possible.

It is important to note that the VMMP execution model is far different from vector machines: to achieve high performance on a vector machine, a program must make good use of the vector pipeline [Dongarra *et al.* 1984, Dongarra and Eisenstat 1984]. It must be built to use vector instructions on long-enough vectors. Those vectors must be arrayed in memory, and transferred into the CPU's vector registers (the exact details might differ, depending upon the specific vector machine used). After some setup delay, the mathematical operation will be performed in the full pipeline speed. The resulting vector will be used for further computations or will be written to the memory. Therefore, the main overhead to consider, even when dealing with a multi-processors vector machine, is that of the pipeline setup time. The main factor to consider is pipeline usage. This is different from the hierarchical memory concerns of VMMP.

4.4.2 Optimization Techniques Used

The following techniques were used to improve data locality and cache use:

loop sectioning: breaking a loop range into blocks. The idea it to handle submatrices small enough to fit in the cache, and to make full use of every submatrix before it is replaced in the cache. For example, when multiplying matrix A by matrix B, matrix B is referenced by blocks of columns. The number of columns in the block is tuned according to the cache parameters of the specific hardware. Figure 6 shows how loop sectioning affects dgemm() results, for two problem sizes: one (N=255) that fits into a processors memory, and another (N=360) that underlines the hierarchical nature of the memory system by introducing paging, as well. Note that the effect lessens with the increase in the number of processors, as each processor's problem size decreases.

$\alpha AB + \beta C$

	# of processors	1	2	3	4	5	6
255×255	with loop sect.	60.03	30.47	20.90	16.33	13.23	13.03
	w/o	62.53	31.90	21.93	17.20	14.00	13.80
360×360	with loop sect.	176.53	93.63	64.17	49.97	41.77	43.50
	w/o	205.63	97.53	66.70	52.70	45.83	47.27

Figure 6: Loop-sectioning effect on dgemm() results (run time in seconds)

loop ordering: computational program loops were ordered in the correct sequence to generate sequential memory references. Combined with transposing of argument matrices (next item here) it improves the use of the cache line.

transposing: as mentioned in section 4.3, we always have to copy an argument matrix from its original local memory residence into a shared memory object. As we already pay the copy price, we may use this step to transpose the matrix in the shared memory object. This is used when a matrix appears on the right side of a '$*$', and transposing the matrix guaranties sequential access to its elements. The improvement in the run time due to the consecutive memory references compensates for the extra cost of replacing one data movement request with element-by-element copy.

special cases: every BLAS-3 implementation will probably exploit some of the BLAS-3 calls' special cases (e.g. being called with $\alpha = 0$ in dgemm()) to avoid some of the computations. The model implementation (section 2.1.4) does it, and so do the other implementations we read about. Besides identifying those special cases, we also identify those cases that might save some overhead time. A good example is dgemm() with β equal to 1. In this case, only one operation is performed on an element of C. We prefer not to copy C into the shared memory objects, and handle that *add* operation in the serial part of the routine. Another example is calling dgemm() with the same matrix appearing twice in argument list (as in $C = B(C + I)$). Here again we can avoid some data transfers, and use only one shared memory copy of C.

5 Performance

We tested our implementation on shared memory hardware — the commercial Sequent Symmetry 81 [Osterhaug 1986], and Taunop, an experimental machine machine built in Tel-Aviv University.

The Sequent we run on has 24 i386 (at 16MHz) processors, of which 22 are used by VMMP. The processors are connected by a high speed synchronous bus to a global memory of 32 Megabytes. The processors has no local memory, but do have a 64KB local cache. Floating point calculations are accelerated using a Weitek WTL 3167 co-processor. The operating system is DYNIX V3.1.2.

Taunop is build of eight NS32532 processors on a VME bus. Each processors has a 4MB local memory, accessed through a 64KB cache. A 8MB memory board on the bus is shared between the processors. The cache's line length is 16 bytes. The operating system used is MOSIX, a parallel version of UNIX developed at the Hebrew University, Jerusalem [Barak and Litman 1985]. We measured the following values for update (read-modify-write) accesses on this machine:

$$\frac{\text{local memory access (cache miss) time}}{\text{cached local memory access time}} = 2.4$$

$$\frac{\text{shared memory access (cache miss, no bus contention) time}}{\text{cached local memory access time}} > 8$$

Following are the results on both machines for each BLAS-3 routine. The best speedup values (15.89 for twenty-two processors) are achieved when each crowd process has to fetch and update only a small part of the data in the shared memory objects. **Dgemm()**, for example, has a speedup of 15.12 for twenty processors (Sequent) and 6.71 for eight (Taunop). **Dsyr2k()**, that has its crowd processes access almost every element in the shared memory segment, measures a speedup of only 12.30 for twenty processors.

As explained earlier (section 4.3), each BLAS-3 routine executes the following stages:

1. copying data into the shared memory object
2. spawning a crowd of processes, all of which:
   (a) obtain some of the data from the memory objects
   (b) perform the calculations required
   (c) update their part of the shared memory objects
3. copying the updated data back from the memory object.

The first and last stages are fixed in their execution time and are executed serially. The second stage is executed in parallel by the crowd processes. Its execution time is a complex function of the actual computation time (proportional to the workload on each processor) and of the cache utilization and the memory and bus contention.

Please note the following:

- The tests were conducted on matrices of size 255×255 — a size large enough for practical applications, but small enought to fit in main memory (of both machines) without paging.

- The speed-up values are measured compared to the execution time of a serial solution to the problem, executed on the same processor.

- Some of BLAS-3 routines have arguments that change their behavior, e.g. the "side" flag to **dtrmm()**. Sometimes this change affects the way those routines are implemented and the efficiency of our implementation. In those cases, we give the results for all the different options. When the effects were minimal, we present average results.

- Speedup and efficiency values are defined as follows:

$$\text{speed-up} = \frac{\text{serial algorithm execution time}}{\text{execution time}}$$

$$\text{relative speed-up} = \frac{\text{execution time with one processor}}{\text{execution time}}$$

$$\text{efficiency} = \frac{\text{execution time with one processor}}{\#\text{ of processors} \times \text{execution time}}$$

5.0.3 dgemm() — general matrix-matrix products

dgemm() is one of the most important BLAS-3 routines. This is also where we tested various implementation ideas and trade-offs. Our tests led, for example, to the separate handling, by a dedicated internal routine, of the special case of $\beta = -1$ or $\beta = 1$. This routine performs about 2% better than the general dgemm().

$C = \alpha AB + \beta C$

Best serial time — 135.9 seconds.

processors	1	2	4	8	12	16	20	22
Total time	138.81	70.47	36.25	18.84	13.56	10.48	8.99	8.55
Speedup	0.98	1.93	3.75	7.21	10.02	12.96	15.12	15.89
Relative Speedup	—	1.97	3.83	7.37	10.24	13.24	15.44	16.23
Efficiency	—	98.5%	95.7%	92.1%	85.3%	82.7%	77.2%	73.8%

dgemm() Performance on the Sequent Symmetry

Best serial time for $\alpha AB + \beta C$ case — 72.2 seconds.
Best serial time for $\alpha AB^T + \beta C$ case — 72.00 seconds.†

processors	1	2	3	4	5	6	7	8
Total time	72.15	36.57	24.77	18.90	15.53	13.17	11.70	10.73
Speedup	1.00	1.97	2.91	3.81	4.64	5.47	6.15	6.71
Relative Speedup†	—	1.97	2.91	3.82	4.65	5.48	6.17	6.72
Efficiency	—	98.6%	97.1%	95.4%	92.9%	91.3%	88.1%	84.0%

dgemm() Performance on Taunop

5.0.4 dsymm() — matrix-matrix products where one matrix is symmetric

dsymm() calculates $\alpha AB + \beta C$ or $\alpha BA + \beta C$. In both cases we need a full line/column of A. As we are allowed to reference only half of the matrix, this full line/column will be gathered from information in part of a line and part of a column (Figure 7).

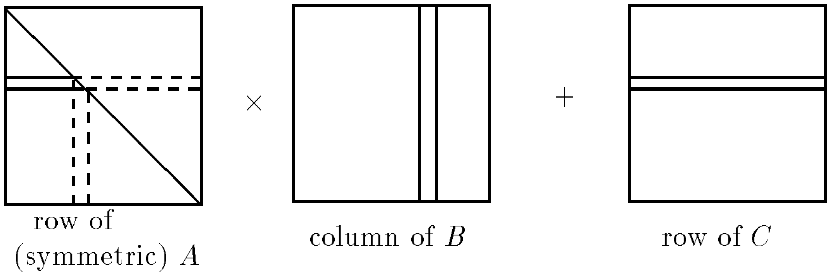

Figure 7: Symmetric Matrix in dsymm()

side flag = “right”.
Best serial time — 119.83 seconds.

processors	1	2	4	8	12	16	20	22
Total time	128.49	64.95	33.22	17.36	12.44	9.61	8.26	7.83
Speedup	0.93	1.85	3.61	6.90	9.63	12.47	14.51	15.30
Relative Speedup	—	1.98	3.87	7.40	10.33	13.37	15.56	16.41
Efficiency	—	98.9%	96.7%	92.5%	86.1%	83.5%	77.8%	74.6%

dsymm() Performance on the Sequent Symmetry

side flag = "right"
serial time - 71.9 sec.

processors	1	2	3	4	5	6	7	8
Total time	82.00	45.31	33.43	27.73	24.30	21.60	20.83	19.07
Speedup	0.88	1.59	2.15	2.59	2.96	3.33	3.45	3.77
Relative Speedup	—	1.81	2.45	2.96	3.37	3.80	3.94	4.30
Efficiency	—	90.5%	81.8%	73.9%	67.5%	63.3%	56.2%	53.7%

dsymm() Performance on the Taunop

side flag = "left"
serial time - 78.17 sec.

processors	1	2	3	4	5	6	7	8
Total time	85.27	45.57	36.23	29.13	24.97	21.87	20.77	18.57
Speedup	0.92	1.72	2.16	2.68	3.13	3.57	3.76	4.21
Relative Speedup	—	1.87	2.35	2.93	3.41	3.90	4.11	4.59
Efficiency	—	93.6%	78.5%	73.2%	68.3%	65.0%	58.6%	57.4%

dsymm() Performance on the Taunop

5.0.5 dtrmm() — triangular matrix-matrix product

dtrmm() performs one of two actions with a triangular matrix — right-side multiplication (αCT) and left-side multiplication (αTC). The implementation details of the two cases differ. On the 'left' case, C is handled by blocks of columns that are accessed column-wise (i.e. the memory references to this matrix' elements are not consecutive). In the 'right' case T is transposed in advance, when copied into the shared memory object. The timing results for the two cases differ by only about 1% ('right' is faster). Thus, we can present here only one set of measurements:

side flag = "right".
Best serial time — 63.5 seconds.

processors	1	2	4	8	12	16	20	22
Total time	66.67	34.20	17.54	9.31	6.72	5.25	4.54	4.31
Speedup	0.95	1.86	3.62	6.82	9.44	12.08	13.99	14.73
Relative Speedup	—	1.95	3.80	7.16	9.92	12.69	14.70	15.47
Efficiency	—	97.5%	95.0%	89.5%	82.7%	79.3%	73.5%	70.3%

dtrmm() Performance on the Sequent Symmetry

side flag = “right”.
serial time - 35.10 sec.

processors	1	2	3	4	5	6	7	8
Total time	36.40	18.70	12.67	9.73	7.93	6.80	6.10	5.60
Speedup	0.96	1.88	2.77	3.61	4.43	5.16	5.75	6.27
Relative Speedup	—	1.95	2.87	3.74	4.59	5.35	5.97	6.50
Efficiency	—	97.3%	95.8%	93.5%	91.8%	89.2%	85.2%	81.2%

dtrmm() Performance on the Taunop

5.0.6 dsyrk() — rank-k updates of a symmetric matrix

dsyrk() computes $C = \alpha AA^T + \beta C$ and $C = \alpha A^T A + \beta C$. If A was to be distributed by blocks of rows, than each processor has to fetch two such blocks, only to compute one small block of the resulting matrix (Fig. 8). Considering this, we have chosen to broadcast all of A elements (transposed if necessary) to all the processors. The symmetric matrix C is handled using scattered allocation.

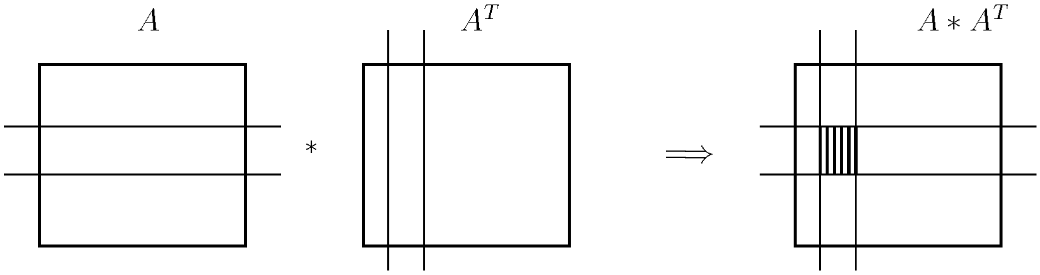

Figure 8: Distributing **dsyrk()** elements by rows

side flag = “right”.
Best serial time — 64.8 seconds.

processors	1	2	4	8	12	16	20	22
Total time	72.75	37.00	19.18	10.27	7.39	5.95	5.27	5.02
Speedup	0.89	1.75	3.38	6.31	8.77	10.89	12.30	12.90
Relative Speedup	—	1.97	3.79	7.09	9.84	12.22	13.81	14.48
Efficiency	—	98.3%	94.9%	88.6%	82.0%	76.4%	69.0%	65.8%

dsyrk() Performance on the Sequent Symmetry serial time - 35.07 sec.

processors	1	2	3	4	5	6	7	8
Total time	39.67	20.30	13.97	10.70	8.87	7.70	7.10	6.67
Speedup	0.88	1.72	2.50	3.26	3.93	4.53	4.92	5.24
Relative Speedup	—	1.95	2.84	3.71	4.47	5.15	5.59	5.95
Efficiency	—	97.7%	94.7%	92.7%	89.4%	85.9%	79.8%	74.4%

dsyrk() Performance on the Taunop

5.0.7 dsyr2k() — rank-$2k$ updates of a symmetric matrix

As mentioned in 4.3.3, **dsyr2k()** computation accesses most of A *and* B elements on each processor, increasing the data movements overhead. We are looking for a better algorithm to implement this operation.

Best serial time — 141.5 seconds.

processors	1	2	4	8	12	16	20	22
Total time	160.57	103.22	57.01	37.86	26.39	19.84	17.19	16.29
Speedup	0.88	1.37	2.48	3.74	5.36	7.13	8.23	8.68
Relative Speedup	—	1.56	2.82	4.24	6.09	8.09	9.34	9.86
Efficiency	—	77.8%	70.4%	53.0%	50.7%	50.6%	46.7%	44.8%

dsyr2k() Performance on the Sequent Symmetry

serial time — 67.13 sec.

processors	1	2	3	4	5	6
Total time	78.00	45.93	30.60	25.37	21.33	19.30
Speedup	0.86	1.46	2.19	2.65	3.15	3.48
Relative Speedup	—	1.70	2.55	3.07	3.66	4.04
Efficiency	—	84.9%	85.0%	76.9%	73.1%	67.4%

dsyr2k() Performance on the Taunop

5.0.8 dtrsm() — solution of a triangular system of equations

dtrsm() was implemented by treating the problem as an array of matrix-vector problems — $Ax = y$ (or $xA = y$). The B matrix was split by columns (rows) between the processors, each processor handling its own set of result-vectors.

This algorithm minimizes the overhead of synchronization between the processors and achieves an even distribution of work. Its data locality, though, can be improved. Block algorithms for the triangular solver, that have better memory access patterns, can be found, for example, in [Gallivan *et al.* 1990, section 3.5] or in [Bischof 1990].

The best algorithm for a specific system must be determined according to the cost of data movements between processors compared with the cost of data movements between different memory hierarchy levels (e.g. from the shared memory to the processor's cache).

side flag = "left".
Best serial time — 103.08 seconds.

processors	1	2	4	8	12	16	20	22
Total time	114.38	57.94	29.48	15.32	10.86	8.33	7.16	6.75
Speedup	0.90	1.78	3.50	6.73	9.49	12.38	14.40	15.27
Relative Speedup	—	1.97	3.88	7.47	10.53	13.73	15.98	16.95
Efficiency	—	98.7%	97.0%	93.3%	87.8%	85.8%	79.9%	77.0%

dtrsm() Performance on the Sequent Symmetry

side flag = "left".
serial time - 49.87 sec.

processors	1	2	3	4	5	6	7	8
Total time	56.70	28.83	20.70	14.87	12.73	11.03	9.53	8.60
Speedup	0.88	1.73	2.41	3.35	3.92	4.52	5.23	5.80
Relative Speedup	—	1.97	2.74	3.81	4.45	5.14	5.95	6.59
Efficiency	—	98.3%	91.3%	95.3%	89.1%	85.7%	85.0%	82.4%

dtrsm() Performance on the Taunop

5.1 Performance Optimizations

5.1.1 The VMMP Execution Model

VMMP is meant to run both on shared memory and message passing hardware. To maintain programs compatibility between the two different environments, VMMP restricts the use of the shared memory (section 2). A program that uses VMMP defines and uses shared memory objects that are copied back and forth between processors' local memory and the shared memory. VMMP handles the synchronization issues that arise when updating multiple copies of the same shared memory segment.

The effect is a dramatical increase in the effective time it takes a processor to modify a shared memory location. Thus, making the VMMP "virtual machine" an extreme example of an hierarchical memory system. The efficiency of parallel algorithms is heavily affected by this fact, and many of them become unusable. On the other hand, the VMMP virtual machine differs from message passing computers, too. Message passing algorithms are designed to take into account high cost of distributing problem elements to the processors, and the need to shorten the paths data elements should pass, transferred from one processor to another. When using VMMP the data transfers are hidden from the programmer, on shared memory and message passing systems alike. The programmer can no longer use his knowledge of the hardware architecture to create a faster implementation of an algorithm. Moreover, trying to impose message-passing-type considerations on a program will result in a poor performance on shared memory hardware. All the above can be summarized in the following design rule, that apply both to shared memory based and message passing based VMMP: *the VMMP machine has an hierarchical memory system. Data locality is the most important factor when trying to minimize VMMP overhead.*

There is one more design rule emanating from the way typical VMMP hardware (unlike vector machines, for example) is designed. It is best to take into account the existence of some hardware processor and shared memory caches of some unspecified size and line length. Which means that the library's internal program loops should access memory locations in consecutive order whenever possible.

It is important to note that the VMMP execution model is far different from vector machines: to achieve high performance on a vector machine, a program must make good use of the vector pipeline [Dongarra *et al.* 1984, Dongarra and Eisenstat 1984]. It must be built to use vector instructions on long-enough vectors. Those vectors must be arrayed in memory, and transferred into the CPU's vector registers (the exact details might differ, depending upon the specific vector machine used). After some setup delay, the mathematical operation will be performed in the full pipeline speed. The resulting vector will be used for further computations or will be written to the memory. Therefore, the main overhead to consider, even when dealing with a multi-processors vector machine, is that of the pipeline setup time. The main factor to consider is pipeline usage. This is different from the hierarchical memory concerns of VMMP.

5.1.2 Optimization Techniques Used

The following techniques were used to improve data locality and cache use:

loop sectioning: breaking a loop range into blocks. The idea it to handle submatrices small enough to fit in the cache, and to make full use of every submatrix before it is replaced in the cache. For example, when multiplying matrix A by matrix B, matrix B is referenced by blocks of columns. The number of columns in the block is tuned according to the cache parameters of the specific hardware. Figure 6 shows how loop sectioning affects dgemm() results, for two problem sizes: one (N=255) that fits into a processors memory, and another (N=360) that underlines the hierarchical nature of the memory system by introducing paging, as well. Note that the effect lessens with the increase in the number of processors, as each processor's problem size decreases.

$\alpha AB + \beta C$

	# of processors	1	2	3	4	5	6
255×255	with loop sect.	60.03	30.47	20.90	16.33	13.23	13.03
	w/o	62.53	31.90	21.93	17.20	14.00	13.80
360×360	with loop sect.	176.53	93.63	64.17	49.97	41.77	43.50
	w/o	205.63	97.53	66.70	52.70	45.83	47.27

Figure 9: Loop-sectioning effect on dgemm() results (run time in seconds)

loop ordering: computational program loops were ordered in the correct sequence to generate sequential memory references. Combined with transposing of argument matrices (next item here) it improves the use of the cache line.

transposing: as mentioned in section 4.3, we always have to copy an argument matrix from its original local memory residence into a shared memory object. As we already pay the copy price, we may use this step to transpose the matrix in the shared memory object. This is used when a matrix appears on the right side of a '$*$', and transposing the matrix guaranties sequential access to its elements. The improvement in the run time due to the consecutive memory references compensates for the extra cost of replacing one data movement request with element-by-element copy.

special cases: every BLAS-3 implementation will probably exploit some of the BLAS-3 calls' special cases (e.g. being called with $\alpha = 0$ in dgemm()) to avoid some of the computations. The model implementation (section 2.1.4) does it, and so do the other implementations we read about. Besides identifying those special cases, we also identify those cases that might save some overhead time. A good example is dgemm() with β equal to 1. In this case, only one operation is performed on an element of C. We prefer not to copy C into the shared memory objects, and handle that *add* operation in the serial part of the routine. Another example is calling dgemm() with the same matrix appearing twice in argument list (as in $C = B(C + I)$). Here again we can avoid some data transfers, and use only one shared memory copy of C.

6 Applications

6.1 Introduction

In the previous sections, we showed that it is possible to implement BLAS-3 on top of VMMP with reasonable speedups. The examples in this section (LU, QR and Cholesky decompositions) demonstrate the use of the parallelized BLAS-3 routines in practical applications. Using these routines in a program which is otherwise written and executed serially is sufficient to gain overall speedups. The tests were conducted on the same machine described in section 5. Each example includes a description of the problem and the algorithm, and the performance testing results.

6.2 Algorithms

The problems demonstrated are well researched in linear algebra, such as the LU or the QR decomposition. The standard algorithms for these problems are usually expressed in vector-vector or, in the best case, matrix-vector operations. But, as BLAS-2 (matrix-vector) routines implemented on top of VMMP are slow, we must use matrix-matrix operations. Thus, we used modified versions of the standard algorithms, e.g. [Bischof and Van Loan 1987] for the QR decomposition. Those modified algorithms still contain parts where BLAS-3 routines cannot be used. Therefore, those parts are executed serially and take considerable execution time. Where applicable, we have chosen algorithm parameter values in such a way that the serial time is minimized.

For the LU, QR or Cholesky decompositions handled here, one can consider the block version of the problem. These blocked versions are easier to implement using BLAS-3. We preferred, though, to present the non-blocked problems, due to the limited use of the blocked results.

6.3 Example 1: LU Decomposition with Partial Pivoting

In this example we decompose a given $n \times n$ matrix A, into a lower triangular matrix L and an upper triangular matrix U such that $A = LU$.

The standard Gaussian elimination method can be implemented using matrix-vector (BLAS-2) operations. We would like to perform the elimination step using BLAS-3 (matrix-matrix) operations. The modified algorithm is based on [Gallivan *et al.* 1990, p. 92] and the comments in [Dongarra and Hewitt 1986]. It involves grouping elimination vectors into a matrix, and then applying that matrix to A.

Another way to employ matrix-matrix operations is in the block generalization of the standard LU factorization. The result of this factorization is the block triangular matrices L and U. This algorithm promises better speedup but, as no pivoting is performed in the simple block LU version, it can not be used on every system.

6.3.1 The Algorithm

We handle A by blocks of size ω. In step i, the first $(i-1)w$ columns of L and rows of U are known, and we try to find the next ω rows and columns (Figure 10). A' denotes that part of A that is not yet decomposed.

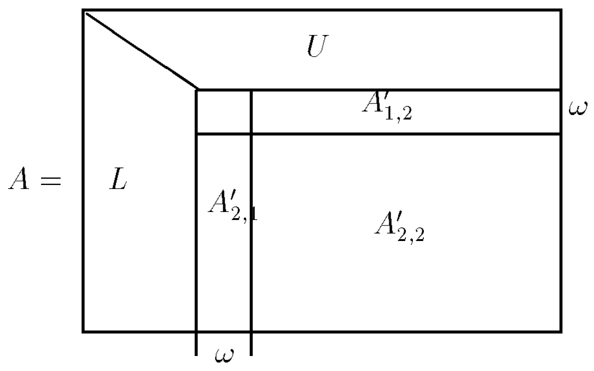

Figure 10: LU Decomposition by Blocks

The first step is to find ω pivot rows, and use them to eliminate the first columns of A' ($A'_{2,1}$) and its first rows ($A'_{1,2}$). This step is executed serially. Then, the same transformation is applied to $A'_{2,2}$.

In Fig. 11 there is a stripped down version of the main routine.

```
for s=1 to m (step = p)
        compute (serially) a transformation such that A'_{2,1}
           will be replaced by L'_{2,1}, A'_{1,2} by U'_{1,2}.
        apply (in parallel, using dgemm()) the same transformation
           to A'_{2,2}.
end s
```

Figure 11: LU Decomposition

6.3.2 Performance

The parallel part of the algorithm is the execution of **dgemm()**, updating the submatrix $A'_{2,2}$. We minimize the serial part execution time by using the smallest ω for which **dgemm()** execution does not take too long. Even so, the serial part takes up to 40% of the execution time for a problem of size 360×360 (this size is nearly the maximal we can handle in the current memory configuration of our machine).
Here is an example of the speedup achieved:

Problem size - 360×360 matrix, block size of 12.
Serial algorithm (Gaussian elimination) execution time - 85.73 seconds.

# of processors	1	2	3	4	5	6	7	8
Total time	98.63	61.23	49.30	42.67	39.03	36.33	34.83	33.27
Speedup	0.87	1.40	1.74	2.01	2.20	2.36	2.46	2.58
Relative Speedup	—	1.61	2.00	2.31	2.53	2.71	2.83	2.96
Relative efficiency	—	80.5%	66.7%	57.8%	50.5%	45.2%	40.5%	37.1%
Parallel time	82.5	45.3	33.1	26.7	22.6	20.3	18.6	17.0
Parallel part speedup	—	1.82	2.49	3.09	3.65	4.07	4.43	4.83

6.3.3 Results by Others

LU factorization is a widely used linear algebra procedure. For instance:

1. CRAY X-MP-4 implementation [Dongarra and Hewitt 1986]

   J. Dongarra and T. Hewitt implemented LU factorization on the CRAY X-MP-4 — a multi-vector machine with shared memory. Their algorithm is the basis for the one we used. Their implementation differs, as follows:

   - it is written in FORTRAN,
   - it uses assembly language for synchronization and for the process of finding the pivot rows (a process which is not multitasked),
     and —
   - it uses shared memory directly.

   Their results are better then ours — e.g. relative speedup of 3.56 using four processors (400×400 matrices). This results from the following:

   - the direct use of shared memory, compared to the extra data movements inherent in VMMP usage,
   - the CRAY provides better support for parallel accesses to the shared memory, and
   - the use of assembly language to decrease the execution time of the non-multitasked portion of the algorithm.

2. IBM 3090 implementation [Radicati *et al.* 1988]

   G. Radicati, Y. Robert and P. Sguazzero implemented the same algorithm on the IBM 3090 vector multiprocessor. Unlike the CRAY X-MP, the 3090 has a hierarchical memory organization. Following the reasoning stated earlier in this paper, their implementation uses matrix-matrix operations to better the data locality and cache usage.

   Their implementation uses a special method to minimize its sequential part. The main processor is assigned less work than the others, leaving it enough time to prepare the next transformation matrix (this method is not available to us, as we use the "atomic" BLAS-3 library routines). The relative speedup obtained is between 4.14 and 4.74 for six processors (matrices of size 600×600 to 1000×1000).

6.4 Example 2: QR Factorization

Here is another example showing the effective use of BLAS-3/VMMP routines in mathematical problems. QR factorization is used in the solution of the linear least squares problem: Given a matrix A, we are looking for an orthogonal matrix Q and an upper triangular matrix R, such that $A = QR$.

We use a variant of the Householder algorithm. A full description of the Householder algorithm can be found, for example, in [Golub and Van Loan 1989]. The variant is proposed by Bischof and Van Loan [Bischof and Van Loan 1987], and the description here is taken from their article.

The Householder algorithm is based on applying a sequence of Householder matrices on the given matrix A. An Householder matrix is orthogonal, of the form

$$P = I + uv^T,$$

where v has unit 2-norm length, and $u = -2v$. In BLAS terminology, this algorithm is rich in BLAS-2 (matrix-vector) operations. Bischof and Van Loan showed how to modify the algorithm to use matrix-matrix operations.

The algorithm is based on the fact that a product of Householder matrices

$$Q_k = P_1 \cdots P_k$$

can be written in the form

$$Q_k = I + W_k Y_k^T.$$

W and Y can be built column by column:

$$W_k = [W_{k-1}, u_k],$$

$$Y_k = [P_k Y_{k-1}, v_k]$$

by the following procedure:

```
for k=1 to n
        a_k := (I + WY^T)^T a_k (skip if k=1)
        Compute unit v such that if u = -2v then (I + uv^T)a_k is
                   zero in components k+1 to m.
        W := [W, (I + WY^T)u]    (W = [u] if k = 1)
        Y := [Y, v]              (Y = [v] if k = 1)
end k
```

This algorithm is applied to blocks of the matrix A. The W and Y that result are then used to modify A in a matrix-matrix operation, as desired:

```
for k=1 to N
              s:=(k-1)p+1
              Compute W and Y such that I + WY^T is orthogonal
                and (I + WY^T)A(s : m, s : s + p - 1) is upper
                triangular.
              A(s : m, s : n)  :=  (I + WY^T)^T A(s : m, s : n)
end k
```

In our implementation **dgemm()** is used to calculate $(I + WY^T)^T A_{(s:m,s:n)}$.

6.4.1 Performance

The algorithm consists of two phases: finding W and Y, which is executed serially, and applying $(I + WY^T)$, which uses the parallel BLAS-3 routines. The key to good speedup is minimizing the time dedicated to the serial phase by using a small block size p, without escalating the execution time of the parallel one.

Here are the results for a matrix of size $228 * 228$. We used two block sizes: 3 — for better results on 1 to 8 processors, and 2 — which, because of the decreased work in the serial phase, achieves better speedup and is thus potentially better for a larger number of processors.

228*228 matrix.
Serial algorithm execution time:
block size 2 — 1,637.9 seconds;
block size 3 — 1,212.8 seconds.

Block	# of processors	1	2	3	4	5	6	7	8
2	Total time (seconds)	1662.2	927.4	695.7	572.3	493.3	450.2	424.9	404.3
	Speedup	0.99	1.77	2.35	2.86	3.32	3.64	3.85	4.05
	Relative speedup	—	1.79	2.39	2.90	3.37	3.69	3.91	4.11
	Relative efficiency	—	89.6%	79.6%	72.6%	67.4%	61.5%	55.9%	51.4%
3	Total time	1234.0	747.1	583.1	506.2	453.7	424.1	402.7	387.8
	Speedup	0.98	1.62	2.08	2.40	2.67	2.86	3.01	3.13
	Relative Speedup	—	1.65	2.12	2.44	2.72	2.91	3.06	3.18
	Relative efficiency	—	82.6%	70.5%	60.9%	54.4%	48.5%	43.8%	39.8%

6.4.2 Results by Others

1. M. Berry *et al.* implemented this (Bischof and Van Loan) algorithm on the eight-processors Alliant FX/8 in Fortran-90 (not using BLAS-3) [Berry *et al.* 1986]. They do not report the efficiency of the parallel implementation, but do supply other interesting results. Their results were compared with:

   (a) a serial implementation of the classical Housholder factorization method;

   (b) the same (i.e. Bischof and Van Loan) algorithm reduced to matrix-vector operations (by setting the block width to be 1).

   The speedups over (a) range from 8.83 for a matrix of order 100, to 16.30 for a matrix of order 1000. Note that these numbers reflect both the parallel use of eight processors and the matrix-matrix nature the algorithm. The speedups over (b) range from 1.83 to 3.11.

6.5 Example 3: Cholesky Factorization

Given a positive-definite symmetric matrix A, we are looking for a lower-triangular matrix L such that $A = L * L^T$.

The algorithm computes L by blocks of columns [Dongarra *et al.* 1990a]. It is designed to keep the submatrices passed to BLAS-3 routines as large as possible, minimizing the initialization time required by those routines.

In the following description, we assume A is partitioned into blocks A_{ij} of size $nb * nb$ — nb being a parameter to the algorithm. The notation $A_{a:b,c:d}$ refers to a submatrix of A containing rows a to b, columns c to d.

for j=1 to $\lceil n/nb \rceil$

$A_{jj} \leftarrow A_{jj} - A_{j,1:j-1} A^T{}_{j,1:j-1}$	using *dsyrk()*
factorize A_{jj}	unblocked algorithm
$A_{j+1:p,j} \leftarrow A_{j+1:p,j} - A_{j+1:p,1:j-1} A^T_{j,1:j-1}$	using *dgemm()*
$A_{j+1:p,j} \leftarrow A_{j+1:p,j} (A^T_{jj})^{-1}$	using *dtrsm()*

end

6.5.1 BLAS-3 Usage

dsyrk(), a BLAS-3 routine for symmetric matrix multiplication, is used to update the diagonal block (before the unblocked Cholesky algorithm can be applied to it). **dgemm()** and **dtrsm()** are used to update the sub-diagonal block of A, and to compute the corresponding block of L.

6.5.2 Performance

The algorithm consists of two phases: the sequential factorization of a diagonal block, and the parallel updating of the rest of the matrix. Thus, there is a trade-off between the total execution speed, and the speed-up values achievable. The higher the block size (nb) is, the longer the sequential routine takes to factor the diagonal blocks — degrading speed-ups. For very small nb values the sequential time is minimal, but the BLAS-3 calls' overhead becomes the major factor.

Problem size - 320*320 with blocks of size 40.

Serial algorithm execution time - 27.40 seconds.

# of processors	1	2	3	4	5	6	7	8
Total time (sec.)	30.50	19.43	11.00	8.83	7.77	7.23	6.57	6.53
Speedup	0.90	1.41	2.49	3.10	3.53	3.79	4.17	4.19
Relative Speedup	—	1.57	2.77	3.45	3.93	4.22	4.64	4.67
Efficiency	—	78.5%	92.4%	86.3%	78.5%	70.3%	66.3%	58.4%

6.5.3 Results by Others

C. Bischof [Bischof 1990] discusses the three possible variants of blocked Cholesky algorithm and presents a summary of the memory access cost of the different variants. The same paper also present MFLOPS results for the blocked Cholesky factorization on the four-processor CRAY-2 and the eight-processor CRAY Y-MP. The CRAY-2 performs at about 400 MFLOPS for a 300*300 matrix. The CRAY Y-MP — 700 up to 900 MFLOPS, according to the variant used.

7 Example 4: Reducing the General Symmetric Eigenvalue Problem Into the Standard One

7.1 Problem Description

Let B be a symmetric matrix. Then, the general symmetric eigenvalue problem is solving $Ax = \lambda Bx$. The common way of dealing with this generalized problem, is reducing it to the standard one — $Ax = \lambda x$. This can be achieved in various ways. Here we show one possible implementation of such algorithm, using BLAS-3 calls [Gallivan *et al.* 1990].

7.2 The Algorithm

It is said (again, in [Gallivan *et al.* 1990]) that B is a well-conditioned positive definite matrix in most applications. Thus, I can use the Cholesky factorization routine (example 3) to find L, such that $B = L * L^t$. The original problem is then transformed into $(L^{-1}\,A\,L^{-t}) = \lambda x$. A stripped down version of a program that factors B and transforms A looks like:

```
        //
        // read in the matrix B
        ....
        // Cholesky factorization
        dlltb(B, size, block_size);

        // L is stored in B
        // invert L
(*)     dtrsm("right", "lower", "no transpose", "non-unit", 1.0, B,
               B.mem_cols(), inv_L, inv_L.mem_cols());

        //     // find L^{-1} * A * L^{-t}
        dtrmm("left", "lower", "no transpose", "non-unit", size, size,
               1.0, inv_L, inv_L.mem_cols(), A, A.mem_cols());

        dtrmm("right", "lower", "transpose", "non-unit", size, size,
               1.0, inv_L, inv_L.mem_cols(), A, A.mem_cols());
```

Actually, when inverting L (step (*)), the call to dtrsm() for a full sized L was replaced by an algorithm which handled L by blocks. The improved data locality decreased the execution time by about 10%.

7.3 BLAS-3 Usage

The following BLAS-3 routines are used in this example:

dsyrk() used by **dlltb()** to factor B into $L * L^t$

dgemm() again — in dlltb()

dtrsm() to find L^{-1}

dtrmm() to compute $L^{-1}AL^{-t}$

7.4 Performance

The following table summarizes execution times for 240*240 matrices, on one to six processors.

# of processors	serial	2	3	4	5	6
Total time	90.10	48.63	32.90	25.63	21.40	18.67
Speedup	—	1.85	2.74	3.52	4.21	4.83
Efficiency	—	92.6%	91.3%	87.9%	84.2%	80.4%
milliseconds for BLAS-3 routines						
dsyrk()	2001	1134	900	767	801	800
dgemm()	7001	3768	2667	2234	1832	1700
dtrmm()	62767	33700	21967	16999	13900	11999
dtrsm()	17899	9766	7032	5366	4567	3900

8 Conclusion

8.1 Summary

This paper describes a specific implementation of a concept that has, we believe, wide applicability. The BLAS-3/VMMP is based upon the combination of two successful concepts — MIMD parallel processing (with its various architectures), and the portable use of standard libraries.

We showed how the standard libraries — widely used in numerical software, but applicable in other areas as well — can be made, using VMMP, portable to a new degree. We demonstrated speedup results of up to 15.89 (on twenty-two processors) for the same BLAS-3 code running on two multiprocessor machines.

8.2 Related Work

8.2.1 SLAP — A Parallel Implementation of BLAS-3

I. Gutheil, W. Roensch and H. Strauss [Gutheil *et al.* 1989, Trottenberg 1986] of SUPRENUM (Bonn) created a parallel Linear Algebra package, named SLAP, for the Suprenum computer. This package covers single-precision BLAS-3 and some of the single-precision routines of BLAS-2.

The SUPRENUM implementation was allowed more freedom than we allowed ourselves in our implementation:

- The user is responsible for distributing the matrices data between the processors. The SLAP library supplies a set of routines to accomplish distribution and gathering.
- Argument lists were modified, to include temporary storage areas and some parallelism management details. (in FORTRAN there are no facilities for dynamic storage allocation. We, using C++, avoided this problem.)
- The BLAS-3 routine names were changed, to underline their parallel behavior.

The first point potentially allows SLAP a performance advantage over the VMMP implementation which distributes the data and then gathers it back within each routine and for each call. The specific performance results are not known to us. The price is portability — the main goal of BLAS-3/VMMP. SLAP uses send and receive commands specific to the SUPRENUM message passing operating system.

8.2.2 LAPACK

Argonne National Laboratory, the Courant Institute for Mathematical Sciences, and the Numerical Algorithms Group, Ltd., are developing LAPACK — a portable linear algebra library[Bischof and Dongarra 1989, Bischof 1990]. LAPACK ("Linear Algebra Package") will provide routines for solving systems of linear equations, least-squares solutions of overdetermined systems of equations, and eigenvalue problems.

Meant to succeed the successful EISPACK and LINPACK libraries, LAPACK is designed to be more appropriate for newer high-performance computers. It was said earlier, when we considered the motivation of the BLAS-3 definition, that efficient use of parallel or hierarchical-memory machines requires codes to maximally reuse data. Otherwise, the cost of data movements between the upper and lower levels of the memory hierarchy will severly limit the machine performance. LAPACK algorithms are designed to use block versions (based on matrix-matrix operations) of its algorithms whenever applicable.

To ensure LAPACK portability, its code relies upon the different levels of BLAS — mostly on the matrix-matrix based BLAS-3. It is assumed by the LAPACK developers that BLAS routines will be efficiently ported to each target machine. However, they do plan to investigate the option of doing gross-grained multiprocessing, outside the scope of the BLAS routines. They plan to use the portable parallel language SCHEDULE [Dongarra *et al.* 1988c].

We will test LAPACK routines on our BLAS implementation once LAPACK code will be publically available.

8.2.3 Other BLAS-3 Implementations

BLAS-3 is implemented on many high-performance machines. On some (at least — CRAY, Convex and Alliant) it is included in the basic scientific library. The following table shows the performance of a Convex C210 implementation of BLAS-3 [Bischof 1989]. Note the difference between the performance of the general-matrix handler **sgemm()**, and the symmetric and triangular matrices routines.

	Mflops problem size				
Name	32	64	128	256	512
sgemm()	38	44	47	47	47
ssymm()	6	10	14	18	21
strmm()	5	8	12	15	17
strsm()	4	7	11	14	16
ssyrk()	6	9	13	16	17

8.3 Future Work

Our work can be extended in the following directions:

MIMD architectures — currently, BLAS-3/VMMP is implemented on a shared memory multiprocessor. Implementation on a message passing (transputer-based) machine is in its first stages. The results achieved thus far make us optimistic about the attainable BLAS-3/VMMP performance on that implementation.

Parallel machines — BLAS-3/VMMP can be ported to other shared memory machines — those on which VMMP is currently implemented, and new ones. In order to easily create an efficient implementation on machines with different low-level details (in particular — different cache sizes), we use some architecture dependent constants that govern index ranges in tight internal loops. An implementation on a different machine will enable us to test the way those constants are determined and used. Of a special interest would be multi-vector implementations, in which the factors that influence program performance are quite different (see section 5.1.1).

Performance improvements — the performance of this specific implementation can probably still be improved. One possible way is to make better use of special cases (e.g., problems with "degraded" (vector-like) matrices).

Acknowledgement

The authors are grateful to Argonne National Laboratory for providing use of the Advanced Computing Research Facility in the Mathematics and Computer Science Division.

References

[Anderson *et al.* 1990] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D. 1990. *LAPACK: A Portable Linear Algebra Library for High-Performance Computers. LAPACK Working Note 20.* CS-90-105 (May), Computer Science Department, Univ. of Tennessee.

[Barak and Litman 1985] Barak, A., and Litman, A. 1985. *MOS: A Multicomputer Distributed Operating System.* Software — Practice and Experience, Vol. 15, No. 8 (Aug.), 725-737.

[Berry *et al.* 1986] Berry, M., Gallivan, K., Harrod, W., Jalby, W., Lo, S., Meier, U., Philippe, B., and Sameh, A. 1986. *Parallel Algorithms on the CEDAR System.* CONPAR 86. Lecture Notes in Computer Science #237, Springer-Veriag, pp. 25-39.

[Bischof 1989] Bischof, C. H. 1989. *LAPACK — A Linear Algebra Library for High-Performance Computers.* Notes from a lecture given at Argonne National Laboratory (Dec.).

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

[Bischof and Dongarra 1989] Bischof, C. H., and Dongarra, J. 1989. *A Linear Algebra Library for High-Performance Computers.* Parallel Supercomputers: Methods, Algorithms and Applications, Edited by Graham F. Carey. John Wiley & Sons Ltd., pp. 45-55.

[Bischof and Van Loan 1987] Bischof, C., and Van Loan, C. 1987. *The WY Representation for Products of Householder Matrices.* SIAM J. Sci. Stat. Comput., Vol. 8, No. 1 (Jan.), s2-s13.

[Dongarra and Eisenstat 1984] Dongarra, J., and Eisenstat, S. 1984. *Squeezing the Most out of an Algorithm in CRAY FORTRAN.* ACM. Trans. Math. Soft. , Vol. 10, No. 3 (Sep.), pp. 219-230.

[Dongarra and Hewitt 1986] Dongarra, J., and Hewitt, T. 1986. *Implementing Dense Linear Algebra Algorithms Using Multitasking on the CRAY X-MP-4.* SIAM J. Sci. Stat. Comput., 7, 1 (Jan.), pp. 347-350.

[Dongarra *et al.* 1984] Dongarra, J., Gustavson, F., and Karp, A. 1984. *Implementing Linear Agebra Algorithms for Dense Matrices on a Vector Pipeline Machine.* SIAM Rev. , 26, 1 (Jan.), pp. 91-112.

[Dongarra *et al.* 1990a] Dongarra, J., Du Croz, J., Duff, I., and Hammarling, S. 1990. *A Set of Level 3 Basic Linear Algebra Subprograms.* ACM Tran. Math. Soft., Vol. 16, No. 1 (Mar.), pp. 1-17.

[Dongarra *et al.* 1990b] Dongarra, J., Du Croz, J., Duff, I., and Hammarling, S. 1990. *A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs.* ACM Tran. Math. Soft., Vol. 16, No. 1 (Mar.), pp. 18-28.

[Dongarra *et al.* 1988a] Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. 1988. *An Extended Set of FORTRAN Basic Linear Algebra Subprograms.* ACM Tran. Math. Soft., Vol. 14, No. 1 (Mar.), pp. 1-17.

[Dongarra *et al.* 1988b] Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. 1988. *An Extended Set of FORTRAN Basic Linear Algebra Subprograms: Model Implementation and Test Programs.* ACM Tran. Math. Soft., Vol. 14, No. 1 (Mar.), pp. 18-32.

[Dongarra *et al.* 1988c] Dongarra, J., Sorensen, D., Connolly, K., and Patterson, J. 1988. *Programming methodology and performance issues for advanced computer architectures.* Parallel Computing 8, pp. 41-58.

[Eckel 1989] Eckel, B. 1989. *Using C++.* McGrow-Hill.

[Gabber 1988] Gabber, E. 1988. *Parallel Programming Using the MMX Operating System and its Processor.* Proc. 3rd Israel Conf. Comput. Syst. Software Eng., Tel-Aviv, Israel, June, pp. 122-132. *Also appeared as Tech. Rep. 107/88, Eskanasy Inst. Comput. Sci., Tel-Aviv Univ., Israel.*

[Gabber 1990] Gabber, E. 1990. *VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors.* IEEE Tran. Par. Dist. Sys., Vol. 1, No. 3 (July), pp. 304-317.

[Gallivan *et al.* 1987] Gallivan, K., Jalby, W., and Meier, U. 1987. *The Use of BLAS3 in Linear Algebra on a Parallel Processor With a Hierarchical Memory.* SIAM J. Sci. Stat. Comput., Vol. 8, No. 6 (Nov.), pp. 1079-1084.

[Gallivan *et al.* 1990] Gallivan, K., Plemmons, R., and Sameh, A. 1990. *Parallel Dense Linear Algebra Algorithms.* SIAM Review, Vol. 32, No. 1 (Mar.), pp. 54-135.

[Golub and Van Loan 1989] Golub, G., Van Loan, C. 1989. *Matrix Computations.* 2nd edition, the John Hopkins University press, Baltimore, Maryland.

[Gutheil *et al.* 1989] Gutheil, I., Roensch, W., and Strauss, H. 1989. *SLAP — Version 1.0 preliminary.* SUPRENUM, Bonn.

[Lawson *et al.* 1979] Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. 1979. *Basic Linear Algebra Subprograms for Fortran Usage.* ACM Tran. Math. Soft., Vol. 5, No. 3 (Sep.), pp. 308-323.

[Louter-Nool 1989] Louter-Nool, M. 1989. *LINPACK Routines Based on Level-2 BLAS.* J. of Supercomputing, 3, 4 (Dec.), 331-349.

[Osterhaug 1986] Osterhaug, A. 19* *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 1986.

[Radicati *et al.* 1988] Radicati, G., Robert, Y., and Sguazzero, P. 1988. *Dense Linear Systems FORTRAN Solvers on the IBM 3090 Vector Multiprocessor.* Parallel Computing 8, 3, pp. 377-384.

[Stroustrup 1986] Stroustrup, B. 1986. *The C++ programming language.* Addison-Wesley.

[Trottenberg 1986] Trottenberg, U. 1986. *SUPRENUM — an MIMD multiprocessor System for Multi-level Scientific Computing.* CONPAR 86. Lecture Notes in Computer Science #237, Springer-Veriag, pp. 49-52.