View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Harvard University - DASH

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD

DASH.HARVARD.EDU

HARVARD LIBRARY

Office for Scholarly Communication

CMSSL: A acalable scientific software library

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

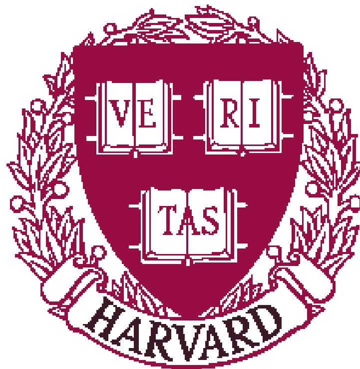
Citation	Johnsson, S. Lennart. 1993. CMSSL: A scalable scientific software library. Harvard Computer Science Group Technical Report TR-23-93.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:35059720
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

CMSSL: A Scalable Scientific Software Library

S. Lennart Johnsson

TR-23-93

December 1993



Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in *Scalable Parallel Libraries*, IEEE Computer Society.

CMSSL: A Scalable Scientific Software Library

S. Lennart Johnsson
Thinking Machines Corp.
and
Harvard University
Cambridge, MA

Abstract

Massively parallel processors introduces new demands on software systems with respect to performance, scalability, robustness and portability. The increased complexity of the memory systems and the increased range of problem sizes for which a given piece of software is used poses serious challenges for software developers. The Connection Machine Scientific Software Library, CMSSL, uses several novel techniques to meet these challenges. The CMSSL contains routines for managing the data distribution and provides data distribution independent functionality. High performance is achieved through careful scheduling of operations and data motion, and through the automatic selection of algorithms at run–time. We discuss some of the techniques used, and provide evidence that CMSSL has reached the goals of performance and scalability for an important set of applications.

1 Introduction

The main reason for large scale parallelism is performance. Most scalable architectures are constructed out of mass produced, state–of–the–art components available at a fraction of the cost of the custom made, low integration–level parts used in conventional supercomputers. In order for scalable architectures, in the form of massively parallel processors, MPPs, to deliver on the promise of extreme performance compared to conventional supercomputer architectures, a comparable level of efficiency in resource use is necessary.

Scalable architectures are available in sizes from a few processors to several thousand processors. Programs for production use on a large number of nodes may be developed on few nodes, or even on a single node. Software must be designed to operate on systems that may vary in size by as much as four orders of magnitude. This level of scalability must be accomplished transparently to the user, i.e., without change the same program must execute not only correctly but also efficiently over this range in processing capacity and corresponding range in problem size. Moreover, programs should not have to be recompiled for various system sizes. This requirement will be even more important in the future, since over time the assignment of processing nodes to tasks is expected to become much more dynamic than today.

Robustness of software both with respect to performance and numerical properties are becoming increasingly important. Todays high performance microprocessors used in MPPs have a processing capacity that exceeds the ability of MOS memories to deliver and accept data. By 1995, the speed of a high performance microprocessor may exceed that of DRAM (Dynamic Random Access Memory) chips by a factor of 10 or more. The memory system in each node will become more complex. In addition, the distributed nature of the total memory compounds the complexity of the memory system. It is imperative that software systems deliver a large fraction of the available performance over a wide range of problem sizes transparently to the user. Small changes in array sizes should not impact performance in a significant way. Robustness with respect to performance in this sense is more demanding on the software systems for MPPs than on conventional architectures. Much of these demands must be resolved at run–time.

Robustness with respect to numerical properties is also becoming increasingly important. The same software may be used for problem sizes over a very wide range. Condition numbers for many numerical methods are significantly worse for large problems than for small problems. As a minimum, condition estimators must be provided to allow users to assess the numerical quality of the results. It will also be increasingly necessary to furnish software for ill–conditioned problems, and whenever possible, automatically choose an appropriate numerical method. Some parallel methods do not have as good a numerical behavior as sequential methods, and this disadvantage is often increasing

with the degree of parallelism. Much research is needed before the choice of algorithm with respect to numerical properties and performance can be automated.

The Connection Machine Scientific Software Library, the CMSSL, today has about 250 user callable functions covering a wide range of common operations in scientific and engineering computation. The library is designed for languages with an array syntax, which allows a richer functionality for each routine (through overloading) than in a conventional Fortran 77 (F–77) library. For instance, whereas one routine is required for each data type in libraries such as the BLAS or LApack, a single routine suffices in CMSSL. Hence, the approximately 250 CMSSL routines are equivalent to about 1,000 F–77 routines (for floating–point computations).

In the next section we state the major design goals for the CMSSL. The remaining sections discusses in some detail the techniques used to achieve the goals. Section 3 briefly introduces the software architecture followed by a discussion of language issues in Section 4. A single call to a CMSSL routine suffices to specify identical high–level operations on a collection of operands. This multiple–instance capability provides the basis for high performance without reliance on sophisticated interprocedural data dependence analysis. The multiple–instance feature is discussed in Section 5. Scalability and robustness with respect to performance both depend heavily on the ability to automatically select appropriate schedules for operations and data motion, and proper algorithms. These issues are discussed in Section 6. The data distribution independent functionality of CMSSL is discussed briefly in Section 7. A summary is given in Section 8.

2 Design goals of CMSSL

The ultimate goal for the CMSSL is to provide high level support for most numerical methods for the solution of partial differential equations and for optimization. CMSSL intends to support traditional numerical methods, hierarchical and multi–scale methods, and multipole and other fast N–body algorithms. The CMSSL support for these methods means functions at a sufficiently high level that architectural characteristics are essentially transparent to the user, yet that a high performance can be achieved. Specific design goals for the CMSSL, established about four years ago, were

1. Scalability across system and problem sizes.
   - Multiple–instance capability, i.e., operation on whole arrays in a way analogous to the way the language intrinsics operate on array data.
   - Data distribution independent functionality.
2. Consistency with languages with an array syntax, such as Fortran 90, Connection Machine Fortran (CMF) and C*.
3. Functionality supporting traditional numerical methods used in scientific and engineering computation.
4. High Performance.
5. Robustness.
6. Portability.
7. Support for all four conventional floating–point data types.

Operation	Mflop/s per node	Efficiency %
Local		
ℓ_2–norm	126	98
Matrix–vector	115	90
Matrix–matrix	115	90
Global		
ℓ_2–norm	126	98
Matrix–vector	80	63
Matrix–matrix	83	65
LU–factorization	61	48
Unstructured grid	37	29

Table 1: Peak local and global performance per node and efficiencies achieved for a few different types of computations on the CM–5. 64–bit precision.

All of the goals enumerated above had an impact on the architecture of the CMSSL. The requirements for the multiple–instance capability and data distribution independent functionality are critical for scalability in a real sense, i.e., system and problem size independent codes that execute at acceptable efficiency for a wide range of system and problem sizes. The requirements of scalability and consistency with languages with an array syntax impacted the user interfaces. Today, the library exists on the Connection Machine systems CM–2, CM–200, and CM–5. The CM–5 version consists of about 0.5 million lines of code, and so does the CM–2 and CM–200 version. Each version has about 250 user callable functions.

Table 1 gives a few examples of how the goal of high performance is met by the CMSSL. The table entry for unstructured grid computations actually represents a complete application [14], while the other entries represent library functions by themselves.

Tables 2 and 3 provide excellent examples of how the goal of scalability is met by the CMSSL, as well as the CM–5 architecture over a range of a factor of a thousand in system size. To first order, the performance per node is independent of the system size, thus

Number of nodes	Dense matrix operations			
	ℓ_2–norm	MV	MM	LU-fact
1	126	83	62	68
32	126	80	71	61
64	125	74	72	60
128	125	76	78	60
256	125	68	77	59
512	125	68	83	59
1024				58

Table 2: Performance in Mflop/s per node over a range of CM–5 system sizes. 64–bit precision.

Number of nodes	Unstructured grid computations		
	ENSA[1]	TeraFrac[2]	MicMac[3]
32	25	26	30
64	25	26	31
128	26	24	29
256	24	25	32
512	24	25	32
1024		26	

Table 3: Performance in Mflop/s per node over a range of CM–5 system sizes. 64–bit precision.

demonstrating excellent scalability. For some computations, like matrix multiplication, the efficiency actually increases as a function of system size. For the unstructured grid computations the performance decreases by about 5%, an insignificant amount.

3 Software Architecture

For scientific and engineering computations, the architectural dependence of user codes with respect to performance is traditionally captured in the BLAS (Basic Linear Algebra Subprograms) [2, 3, 10]. Efficient implementations of this set of routines are architecture dependent, and for most architectures written in assembly code.

On distributed memory architectures, a distributed BLAS [6, 8, 12] (DBLAS) is required in addition to a local BLAS (LBLAS) in each node [9]. Moreover, a set of communication routines are required for data motion between nodes. But, not all (high–level) algorithms parallelizes well, and there is an algorithmic architectural dependence. Architectural independence of application programs requires higher level functions than the LBLAS, DBLAS, and communication routines. Hence, the CMSSL includes a subset of functions corresponding to traditional libraries, such as Linpack, Eispack, LApack, FFTpack and ITpack to mention a few.

3.1 External architecture

The externally visible architecture of the CMSSL is similar to that of conventional libraries, as seen from the following list of categories of routines

- Distributed and local BLAS (DBLAS and LBLAS)
  - Level–1
  - Level–2
  - Level–3
- Sparse DBLAS
  - for regular grids
  - for irregular grids
- Banded direct equation solvers
- Dense direct equation solvers
- Iterative solvers
- Eigenanalysis
- Fast Fourier Transforms
- Ordinary differential equation solvers
- Statistical routines
- Communication functions
  - for regular grids
  - irregular grids
  - global operations
- Stencil/convolution compiler
- Compiled routing

The communication routines are unique to distributed memory machines. The CMSSL also contains tools in the form of two special compilers; a stencil compiler and a communications compiler.

Novel ideas in the CMSSL can be found at all levels: in the internal architecture, in the algorithms used, in the automatic selection of algorithms at run–time, and in the local operations in each node.

3.2 Internal architecture

The CMSSL supports a global shared address space as well as node level programming. Used in the global mode, CMSSL accepts distributed data structures. Internally, the CMSSL consists of a set of library routines executing in each node, a set of communication functions, and code that implements operations on distributed data using the local functions and the communication routines. The communication functions are

[1] ENSA is an Euler and Navier–Stokes finite element code [4] developed at the Division of Applied Mechanics, Stanford University

[2] TeraFrac is a solid mechanics code developed at the Division of Engineering, Brown University, Technical University of Denmark, Lyngby, and Thinking Machines Corp. [13].

[3] MicMac is a solid mechanics code developed at the Department of Mechanical Engineering, Cornell University and Thinking Machines Corp. [1].

either part of the Connection Machine Run–Time System, CMRTS, or part of the CMSSL. All communication functions that are part of the CMSSL are directly user accessible, and so are the functions in each node. For the global programming model the distributed nature of the data structures is transparent to the user. CMSSL calls are consistent with this model. The internal structure of the CMSSL has the following operational characteristics

- Extraction of data distribution information.
- Algorithm selection.
- Execution through calls to
  - Local routines
  - Communication routines

It follows from the internal architecture of the CMSSL that it also has the ability to serve as a nodal library. In fact, through the multiple–instance capability, with each instance constrained to a node, only the nodal portion of the library is invoked. In a separately compiled nodal version of the CMSSL, the global data structure information is not required.

4 Languages with array syntax

Library routines operate on data structures defined in a high–level language in a calling program, whether used for input to, or output from, a routine. The most essential hardware characteristics with respect to performance is the memory architecture and the existence of pipelines. However, most high level languages abstract away the memory hierarchy. Memory is represented as a linearized address space with presumed uniform access time. This feature is a major drawback in programming for performance, since compilers and run–time systems often are not able to resolve efficiently the difference between this model and real memory systems.

In languages with an array syntax, such as Fortran 90, many array operations are made primitive operations through operator overloading. For instance, the addition of two arrays is simply expressed as $A + B$, irrespective of the rank and shape of the two arrays. No explicit enumeration of array elements is required. The array shapes are known from the declaration of the arrays. Similarly, the array type is also known from the array declaration.

In compliance with this property of array languages, arrays are passed to CMSSL routines by reference to an *array descriptor* that contains the information about shape and data type as well as about data distribution. Thus, neither is the shape information passed explicitly in the form of arguments, nor is the type passed explicitly through routine names or arguments. This form of overloading considerably simplifies CMSSL interfaces compared to conventional libraries, such as, the BLAS, Linpack, Eispack, and LApack, at the same time as it reduces the number of required interfaces. An example of a CMSSL interface is given below.

real, array:: y(N,M,K), x(N,K,L), A(M,L,N,K)

gen_matrix_vect_mult(y, A, x, 2, 1, 2, 3, ier)

The array type does not appear in the routine name, and there is no array shape information in the call. The arguments 2, 1, 2, and 3 are due to the multiple–instance capability of the CMSSL.

In the example above, y and x represent either single vectors, or (multidimensional) arrays of vectors, and A represents a matrix, or a (multidimensional) array of matrices. The rank of the array A must be one higher than the ranks of the arrays y and x, which are of the same rank. The number 2 succeeding x states that the *problem axis* for y is the second axes of the array y, i.e., the elements of one instance of the vector y lays along the axis of extent M. Similarly, the number 1 states that the *problem row axis* for A is axis 1 of the array A, and the *problem column axis* is axis 2. The shape of each instance of A is $M \times L$. The *problem axis* for x is axis 3 of the array x. Thus, the above call defines multiple matrix–vector multiplications. Each instance consists of the multiplication of an $M \times L$ matrix by a vector of length L. There are $N \times K$ such instances. The call is independent of the distribution of the arrays. Axes not labeled as problem axes are called *instances axes*.

In our implementation of the multiple–instance capability, arrays are required to have *conforming shapes* with respect to the instance axes. Thus, disregarding the problem axes of the different arrays involved in an operation, the shape of the resulting arrays must be identical. This restriction allows for an implicit ordering of instances corresponding to the ordering of the axes in the arrays.

The use of higher dimensional arrays for a collection of vectors and matrices is often the preferred data representation in many applications. For instance, in Quantum Chromodynamics (QCD), computations are performed on a four–dimensional regular lattice, where in each lattice point the state includes small matrices and vectors. It is natural to represent the collection of matrices as six–dimensional arrays, and the vectors as five–dimensional arrays. Similarly, in finite difference methods for the solution of Navier–Stokes equa-

tions, a three–dimensional grid may be used for the spatial discretization, with the state in each grid point represented by vectors and matrices. It is natural to represent the collection of matrices in all grid points as five–dimensional arrays and the collection of grid point vectors as four–dimensional arrays.

Finally, we remark that the CMSSL is a generic library for languages with an array syntax. The same library indeed supports applications written in either CMF or C*.

5 Multiple–instance computation

The multiple–instance capability of the CMSSL is consistent with the idea of collective computation inherent in languages with an array syntax. Library routines are designed to carry out a collection of high level computations on independent sets of operands in a single call, in the same way addition of arrays are carried out through a single statement. To accomplish the same task in an F–77 or C library, the call to a library routine would be embedded in a set of nested loops. The multiple–instance capability not only eliminates loop nests, but also allows for parallelization and optimization without a sophisticated interprocedural data dependence analysis.

The multiple–instance feature for parallel computation is introduced for reasons analogous to the reasons for introducing the level–3 BLAS for uniprocessors. The level–3 BLAS provides a sufficient degree of freedom compared to the level–1 and level–2 BLAS, to allow for a desired level of optimization for cache based architectures.

We discuss the significance of the multiple–instance capability with respect to performance and simplicity of user code by considering the computation of the FFT along one of the axes of a two–dimensional array of shape $P \times Q$. We assume a *canonical data layout* in which the set of processing nodes are configured as an array of the same rank as the data array and of a shape making the local subarrays approximately square. The nodal array shape is $N_r \times N_c$. Figure 1 illustrates the layout of a two–dimensional data array in row and column major order on a 2×4 nodal array. The CMRTS by default creates such canonical layouts [15].

With the FFT performed along the P–axis, the computations on the two–dimensional array consist of Q independent FFT computations, each on P data elements. We consider three different alternatives for the computation:

1. Maximize the concurrency for each FFT through the use of a canonical data layout for one–dimensional arrays of size P.
2. Compute each FFT without data relocation.
3. Compute all Q FFTs concurrently through multiple–instance routines.

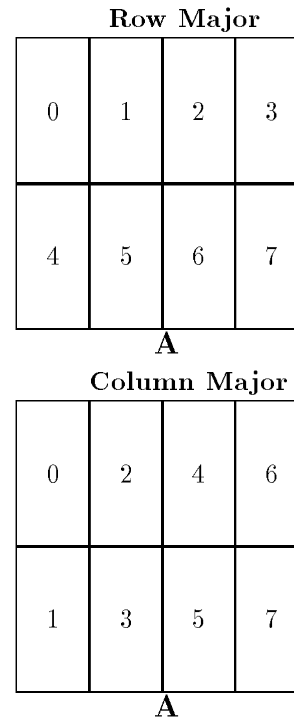


Figure 1: Data distribution on a rectangular nodal array.

Alternative 1 corresponds to the following code fragments:

```
FOR J = 1 TO Q DO
    TEMP = A(:,J)
    CALL FFT1(TEMP,P)
    A(:,J) = TEMP
ENDFOR

SUBROUTINE FFT1(B,N)
ARRAY B(N)

FFT on a one–dimensional array
END FFT1
```

A temporary one–dimensional array with a canonical layout is created for each column $A(:,J)$. The concurrency in the computation of the FFT is maximized. The data motion prior to the computation of the FFT on a column is a *one–to–all personalized communication* (scatter) [7] within processing node rows for the row major ordering. In one–to–all personalized communication, a node sends a unique piece of data to

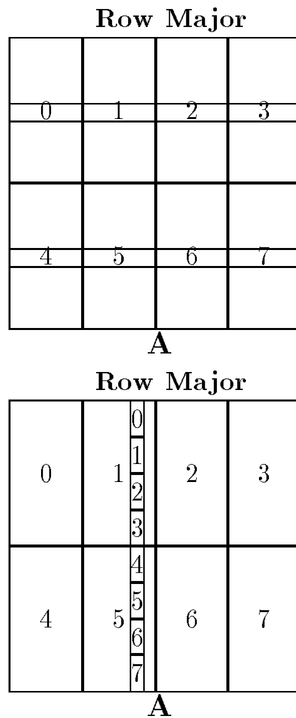


Figure 2: Data redistribution for load–balanced column processing. Nodes labeled in row major order.

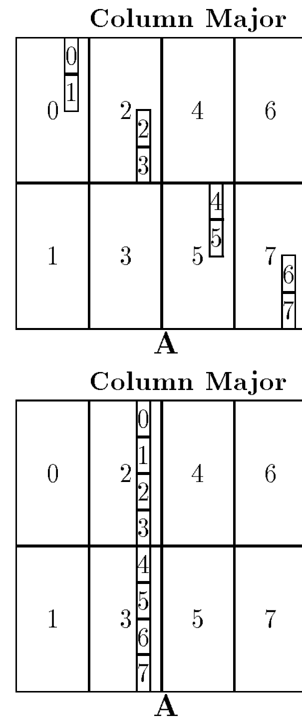


Figure 3: The skewing step in data redistribution for load–balanced processing of columns in a column major labeling of a two–dimensional nodal array.

all other nodes. Upon completion, an all–to–one personalized communication (gather) is required within processing node rows. The data redistribution is illustrated in Figure 2. The redistribution corresponds to a change in data allocation from A(:,:) to A(:, :SERIAL) and back to the original allocation, one column at a time. The compiler directive SERIAL implies that the axis is assigned to the memory of a single node. The arithmetic speedup is limited to $\min(N,P)$ for transforms on the P–axis.

In the column major ordering, a skewing is required prior to the one–to–all personalized communication within columns, as well as after the all–to–one personalized communication within columns that follows the FFT computation. The skewing step is shown in Figure 3.

Alternative 2 corresponds to the following code fragments:

```
FOR J = 1 TO Q DO
    CALL FFT2(A,P,Q,J)
ENDFOR

SUBROUTINE FFT2(B,N,M,K)
ARRAY B(N,M)

In–place FFT on column K of array B
END FFT2
```

The data redistribution is avoided by computing each instance in–place. An obvious disadvantage with this approach is the poor load–balance. The speedup of the arithmetic is proportional to $\min(N_r, P)$ for a transform along the P–axis.

Alternative 3 corresponds to the code fragment:

```
FORALL J DO
    CALL FFT2(A(:,J))
ENDFOR
```

Using the CMSSL FFT corresponds to Alternative 3. All different instances of the FFT represented by the Q columns are treated in–place in a single call. The concurrency and data layout issues are managed inside the FFT routine. The CMSSL call is of the form

CALL FFT(A, DIM = 1),

where DIM specifies the axis of the array A subject to transformation. The actual CMSSL call has additional parameters allowing the calling program to define the subset of axes for which forward transforms are desired, for which axes inverse transforms are desired, and for which axes ordered transforms are desired [16].

In summary, the qualitative features of the three alternatives are:

Alternative 1. Q one–to–all and all–to–one personalized communications within rows for a row major ordering. These communications correspond to the data redistribution A(:,:) to A(:, :SERIAL) and back to A(:,:), one column at a time. For column major ordering, a skew operation is required in addition to the personalized communication. With N nodes, the arithmetic speedup is proportional to $\min(N, P)$ for a transform along the P–axis.

Alternative 2. In–place, single–instance computation. No excess data motion. With N_r nodes along the P–axis, the arithmetic speedup is proportional to $\min(P, N_r)$.

Alternative 3. Multiple–instance, in–place computation. No excess data motion. The arithmetic speedup is proportional to $\min(N, PQ)$.

The third choice is clearly preferable both with respect to communication and arithmetic load–balance. Note that with a single–instance library routine and canonical layouts, Alternative 1 would be realized.

For particular situations, a noncanonical layout will alleviate the communication problem, but in many cases the communication appears somewhere else in the application code. Thus, we claim that our discussion based on canonical layouts reflects the situation in typical computations.

6 Automatic algorithm selection

One of the novel features of the CMSSL is the automatic selection of algorithm. An automatic selection is made both at the local level and at the global level for many functions. The purpose of the selection is to maximize performance by preserving locality of reference. We discuss this feature for matrix operations, both at the local and global level.

6.1 Local algorithm selection

We use matrix–vector multiplication to illustrate run–time loop partitioning and loop reordering in CMSSL. The need for these features arises from the memory hierarchy in each node. On the CM–2, CM–200, and CM–5, there is a single data path between each memory unit and the floating–point unit with which it is associated. Most other multiprocessors based on standard microprocessors share this characteristic. The register set forms the first level in the memory hierarchy. The next level for the Connection Machine systems is DRAM pages. DRAM is operated in page mode, which allows one memory access per processor cycle for accesses within a page. Access to a different page results in a page fault, which results in a two cycle access time for the CM–2 and CM–200, and up to a five cycle access time on the CM–5. A third level of the local memory hierarchy on the CM–5 is introduced through translation lookahead buffers, TLB. Thus, the object of the loop partitioning and loop reordering on the CM–5 is to

- maximize the use of data while in registers,
- minimize the number of DRAM page faults,
- minimize the number of TLB replacements.

Figure 4 shows the impact of DRAM page faults on the local matrix–vector multiplication performance on the CM–5. The matrix shape is $P \times Q$. The multiplication is performed by treating tiles of shape $\hat{P} \times \hat{Q}$ that fits in the register file. For the top curve, the data array layout is such that the innermost loop of length $\hat{P}$ has stride one, while the second innermost loop is of length $\hat{Q}$ with a stride of P. For the top curve, tiles are treated vertically before they are treated horizontally. Thus, the matrix is scanned by vertical panels of width $\hat{Q}$. Changing the data array layout such that the stride along the Q–axis is one for $Q = 1024$ yields a stride of 1024 along the P–axis. The performance decreases by about a factor of 3.5. Changing the loop order for the tile such that the loop on $\hat{Q}$ is innermost yields a stride of one in the inner loop. But, this change in loop order implies a change to an inner–product like algorithm instead of an AXPY like algorithm. The performance is substantially improved compared to the AXPY like algorithm with a large stride. However, since the inner–product is not a particularly efficient operation on the CM–5, the performance is not quite as good as for stride one on the P–axis and an AXPY like algorithm.

In the CMSSL, the best loop order/algorithm within the tile, as well as the best order to loop over tiles, is derived automatically from the array descriptor at run–time. The ideal shape of the tile is also determined automatically. For the example in Figure 4, the performance degradation is limited to about 30% instead of 75%.

Another effect that is visible in the top curve (labeled "pqpq_p_stride_1") in Figure 4, although small, is the effect of TLB thrashing. This is the reason for the performance degradation for large values of P. Figure 5 gives an example where the effect of TLB thrashing is much more severe. The performance is reduced by almost a factor of two. The TLB thrashing can be

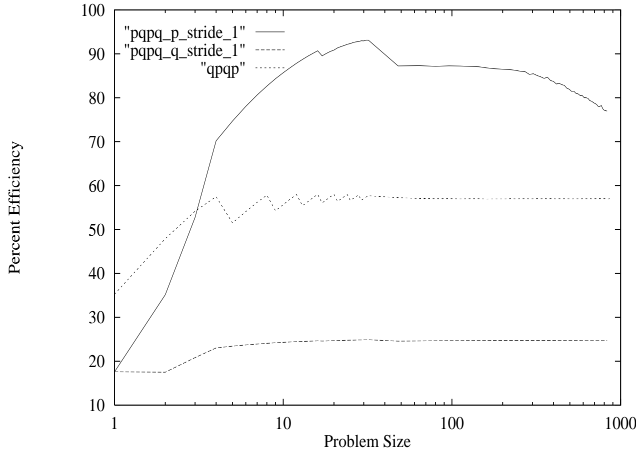


Figure 4: The efficiency of matrix–vector multiplication in 64–bit precision in each vector unit for a CM–5. The matrix shape is $P \times Q$.

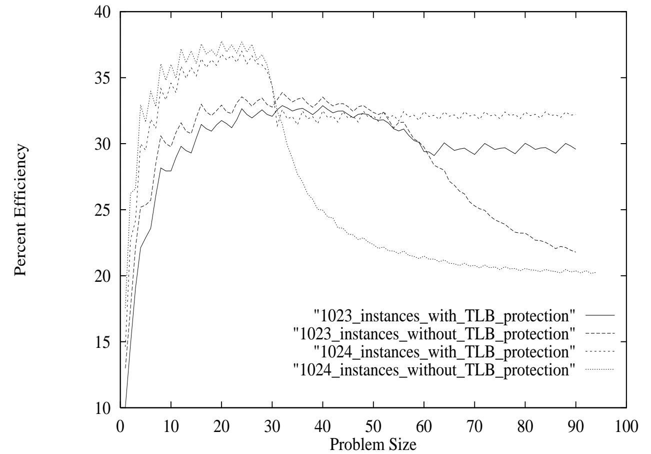


Figure 5: The efficiency of DAXPY in each vector unit for a CM–5. The vector length is P.

reduced by introducing yet another level of loop partitioning/blocking. As seen in Figure 5 the performance can be restored to close to peak performance at a very small expense.

In general, each call to a CMSSL routine handles multiple matrix–vector multiplications, and an instance loop is included in determining tile shape and looping order. Thus, in the case of matrix–vector multiplication, the tile is a three–dimensional box of a size that fits in the register file. The shape of the box is a function of P, Q, and the number of instances, desired vector length, looping overhead, and strides along the different axis. The looping over boxes is determined so as to minimize DRAM page faults and TLB thrashing, as illustrated in Figure 6.

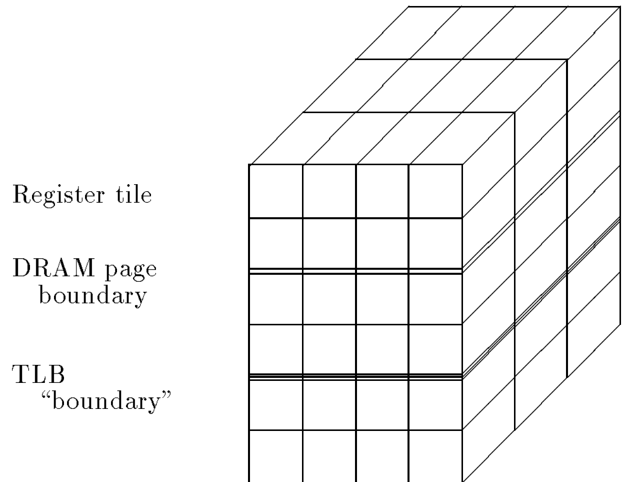


Figure 6: Tiling of the index space for optimum locality of reference.

6.2 Global algorithm selection

We again use matrix operations for illustration. The idea that the operand with the largest number of elements should be kept stationary is very plausible for matrix–vector multiplication. An obvious algorithm is:

.
Align the input vector with a row of the matrix
Broadcast the input vector along columns
Perform local matrix–vector multiplication
Reduce along rows
Align the result with the allocation of the output vector.

Depending upon the relative layouts of the matrix and the vectors, no alignment may be required. How-

ever, with the canonical layout an alignment is required for both vectors, since a one–dimensional nodal array shape is used by default for vectors, while a two–dimensional nodal array shape is used for the matrix. (Except for matrices of extreme shape, this is indeed optimal [5].) For a one–dimensional nodal array shape also for the matrix, either the broadcast or the communication for the reduction is unnecessary. For the ideal (also the default) layout, the matrix–vector multiplication in the CMSSL is implemented using all–to–all communication [11].

For matrix–matrix multiplication, it is intuitive that if one of the operands, say the multiplier, has many rows and columns, while the multiplicand only has two columns, an algorithm very similar to that used for matrix–vector multiplication should be used. Thus, for the computation $C \leftarrow A \times B$, where C and B are vector–like, B should be aligned with A and broadcast and partial products accumulated spatially and aligned with C. Similarly, if C and A are vector–like, but B has a moderate aspect ratio, B should be stationary and A be aligned and broadcast. And, if both A and B are vector–like, but C has a moderate aspect ratio, then C should be stationary and A and B aligned and broadcast [12].

In the CMSSL, a choice of algorithm as indicated above is made automatically in the matrix multiplication routine. The user need not be concerned with specifying what particular algorithm to choose for what matrix shapes and what machine size. For the CM–200, the result is shown in Figure 7. In the bottom part of the plot, where the performance is relatively flat, a matrix–vector type algorithm is used, while an algorithm with the matrix C stationary is used where the performance increases rapidly as a function of the matrix size [12].

7 Data distribution independent functionality

The data distribution independent functionality of the CMSSL is accomplished without any information being passed explicitly in a call to a library routine, as shown in the following example. Both calls to gen_matrix_vector_mult produce the same answers, but the performance will differ.

```
DIMENSION A(81,81,4096), x(81,4096), y(81,4096)
   ⋮
CALL GEN_MATRIX_VECTOR_MULT(y, A, x, 1, 1, 2, 1, ier)
```

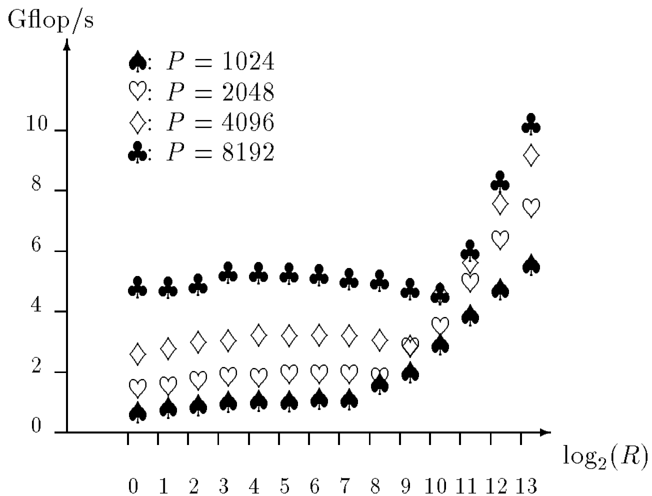


Figure 7: Performance of the matrix multiplication function in the Connection Machine Scientific Software Library for the multiplication of a $P \times P$ matrix by a $P \times R$ matrix on Connection Machine system CM–200, 64–bit precision.

```
CMF$LAYOUT A(:SERIAL,:SERIAL,),x(:SERIAL,),y(:SERIAL,)
DIMENSION A(81,81,4096), x(81,4096), y(81,4096)
   ⋮
CALL GEN_MATRIX_VECTOR_MULT(y, A, x, 1, 1, 2, 1, ier)
```

Whenever the data layout is required either for correctness or performance, the library routines retrieve this information from the array descriptor.

8 Summary

The CMSSL has been designed for performance, scalability, robustness and portability. The architecture with respect to functionality follows the approach in scientific libraries for sequential architectures. Internally, the CMSSL consists of a nodal library and a set of communication and data distribution functions. CMSSL provides data distribution independent functionality and has logic for automatic algorithm selection based on the data distribution for input and output arrays and a collection of algorithms together with performance models.

The goals of scalability and performance have largely been achieved as shown in Tables 1, 2 and 3. Particular emphasis has been placed on reducing the problem sizes offering half of peak performance. Table 4 shows how this goal has been met for a few level–1 LBLAS. Robustness with respect to performance is achieved through the automatic selection of algorithm as a function of data distribution for both low level and high level functions.

Function	Number of instances								
	1	2	3	4	5	7	10	16	32
DSCAL	16	10	8	5	4	3	3	2	1
DAXPY	11	7	6	3	3	2	2	1	1
DDOT	35	27	10	6	4	3	3	2	2
DNORM2	44	32	13	7	6	5	5	4	3

Table 4: Problem size for half of peak performance for BLAS functions local to a CM–5 node.

Acknowledgment

We would like to acknowledge the contributions of Susanne Balle of UNI-C and the Danish Institute of Technology, Lyngby, Paul Bay, Jean-Philippe Brunet, Steven Daly, Zdenek Johan, David Kramer, Robert L. Krawitz, Woody Lichtenstein, Doug MacDonald, Palle Pedersen, and Leo Unger all of Thinking Machines Corp., and Ralph Brickner and William George of Los Alamos National Laboratories, Yu Hu of Harvard University, Michel Jacquemin of Yale University, Lars Malinowsky of the Royal Institute of Technology, Stockholm, Danny Sorensen of Rice University and Deborah Wallach of MIT.

The communications functions and some of the numerical routines in the CMSSL relies heavily on algorithms developed under support of the ONR to Yale University under contracts N00014-84-K-0043, N00014-86-K-0564, the AFOSR under contract AFOSR-89-0382 to Yale and Harvard Universities, and the NSF and DARPA under contract CCR-8908285 to Yale and Harvard Universities. Support for the CMSSL has also been provided by ARPA under a contract to Yale University and Thinking Machines Corp.

References

[1] A. J. Beaudoin, P. R. Dawson, K. K. Mathur, U.F. Kocks, and D. A. Korzekwa. Application of polycrystal plasticity to sheet forming. *Computer Methods in Applied Mechanics and Engineering*, in press, 1993.

[2] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[4] Zdenek Johan, Thomas J.R. Hughes, Kapil K. Mathur, and S. Lennart Johnsson. A data parallel finite element method for computational fluid dynamics on the Connection Machine system. *Computer Methods in Applied Mechanics and Engineering*, 99(1):113–134, August 1992.

[5] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19(11):1235–1257, 1993.

[6] S. Lennart Johnsson. *Parallel Architectures and their Efficient Use*, chapter *Massively Parallel Computing: Data distribution and communication*, pages 68–92. Springer Verlag, 1993.

[7] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[8] S. Lennart Johnsson and Kapil K. Mathur. Distributed level 1 and level 2 BLAS. Technical report, Thinking Machines Corp., 1992. In preparation.

[9] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.

[10] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.

[11] Kapil K. Mathur and S. Lennart Johnsson. All–to–all communication. Technical Report 243, Thinking Machines Corp., December 1992.

[12] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.

[13] Kapil K. Mathur, Alan Needleman, and V. Tvergaard. Ductile failure analyses on massively parallel computers. *Computer Methods in Applied Mechanics and Engineering*, in press, 1993.

[14] Tayfun Tezduyar. Private communication, 1993.

[15] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.

[16] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.