# Language and Compiler Issues in Scalable High Performance Scientific Libraries

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

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:25680329">http://nrs.harvard.edu/urn-3:HUL.InstRepos:25680329</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>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 <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Language and Compiler Issues in Scalable
High Performance Scientific Libraries

S. Lennart Johnsson

TR-18-92
September 1992

Parallel Computing Research Group
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

This work has in part been supported by the Air Force Office of Scientific Research under contract AFOSR-89-0382, and in part by NSF and DARPA under contract CCR-8908285.

Language and Compiler Issues in Scalable High Performance Scientific Libraries

S. Lennart Johnsson
Harvard University
and
Thinking Machines Corp.

Abstract

Library functions for scalable architectures must be designed to correctly and efficiently support any distributed data structure that can be created with the supported languages and associated compiler directives. Libraries must be designed also to support concurrency in each function evaluation, as well as the concurrent application of the functions to disjoint array segments, known as multiple-instance computation. Control over the data distribution is often critical for locality of reference, and so is the control over the interprocessor data motion. Scalability, while preserving efficiency, implies that the data distribution, the data motion, and the scheduling is adapted to the object shapes, the machine configuration, and the size of the objects relative to the machine size.

The Connection Machine Scientific Software Library is a scalable library for distributed data structures. The library is designed for languages with an array syntax. It is accessible from all supported languages (*Lisp, C++, CM-Fortran, and Paris (PARallel Instruction Set) in combination with Lisp, C, and Fortran 77). Single library calls can manage both concurrent application of a function to disjoint array segments, as well as concurrency in each application of a function. The control of the concurrency is independent of the control constructs provided in the high-level languages. Library functions operate efficiently on any distributed data structure that can be defined in the high-level languages and associated directives. Routines may use their own internal data distribution for efficiency reasons. The algorithm invoked by a call to a library function depends upon the shapes of the objects involved, their sizes and distribution, and upon the machine shape and size.

1 Introduction

This paper addresses some of the data management issues that are critical in achieving high efficiency in distributed memory hierarchies. In conventional memory hierarchies, loop partitioning, loop reordering, skewing, and unrolling are techniques used to manage temporal and spatial locality of reference. These techniques attempt to minimize the number of memory references, the number of cache misses in cache based architectures,
and the number of page faults in virtual memory systems. In order to maximize performance, interprocedural analysis is often required. These techniques are also applicable in each node in a distributed memory architecture. However, the memory hierarchy in such systems is more complex, and the data access time is typically not uniform across memory units. In addition to the issues for managing data references in conventional memory hierarchies, it is necessary to effectively manage the data distribution across the network interconnecting the processing nodes, path selection, and scheduling of data motion in the network. In this paper a node refers to a processor, its local memory, and associated communication circuitry.

The emphasis in this paper is on techniques to avoid unnecessary data motion in the context of calls to generic subroutines forming a scientific software library. In the absence of powerful interprocedural analysis and associated optimization techniques with respect to data motion in distributed memory architectures, we have devised a set of techniques that attempt to minimize the data motion in distributed memory architectures. The techniques are key elements of the Connection Machine Scientific Software Library [68], CMSSL. In this paper we illustrate the essence of our techniques through a few examples. The techniques focus on:

1. Avoiding unnecessary data motion in emulating the virtual machine defined by the programming languages on the physical machine.

2. Choosing the data distribution for each function such that communication is minimized and load–balance is maximized with the given distributions of the input and output objects.

3. Avoiding the use of temporary variables representing arrays or segments thereof, whenever possible. Avoiding temporary variables not only reduces the storage requirements, but also eliminates potential data redistribution of selected array elements.

4. Avoiding unnecessary data motion while maximizing concurrency through multiple-instance computation in a single call to a library function.

5. Choosing algorithms for a given library call depending upon the distribution of the objects, such that the data motion is minimized and the load–balance is maximized.

6. Utilizing high–level communication functions providing optimal path selection and schedules for common operations requiring data motion between nodes. The communication functions form a subset of CMSSL, or are included as part of the Connection Machine Run–Time System.

The distinction between the virtual machine defined by the programming languages and the physical machine is very important in managing the data motion. For instance, in pure data motion operations, such as CSHIFT or EOSSHIFT in Fortran 90 [54], excess data motion in the memory of each node occurs in simple implementations of the virtual machine model. The time for the excess local data motion may, indeed, dominate the
data motion between nodes. Reducing, or eliminating it, may have a significant impact on performance. In Section 7 we discuss this issue and show how the local data motion can be reduced, and even eliminated, through suitable program structuring. The notion of local functions in the proposed High Performance Fortran standard encapsulates this idea.

The data distribution that minimizes the communication requirements depends both on the algorithm chosen, and the shapes of the objects involved. For many computations, minimizing the surface area for a given size of the data set per node is a good strategy. Matrix–vector multiplication, matrix–matrix multiplication, LU factorization and triangular system solution belong to this category. We discuss this issue in some detail in Section 10.

The data distribution has a significant impact on performance. Different functions have different ideal data distributions on different networks. Locality of reference is not only a function of the computations to be performed, but also a function of the topology of the network interconnecting the nodes. Moreover, in distributed memory systems some computations may also exhibit a trade-off between locality of reference and load-balance. Traditionally, the address space is treated as uniform in access time, and the index space of multidimensional arrays is linearized. A linearized address space is often not efficient with respect to data motion in distributed memory architectures. Traditional programming languages do not have any constructs for managing the data distribution across memory modules. In Connection Machine Fortran [67], a dialect of Fortran 90, a multidimensional address space is used, and a few compiler directives allow for a limited control of the data distribution. Additional control is available in the Connection Machine Run–Time System. Recent language research efforts, such as Fortran D [12] and Vienna Fortran [7, 75], and the emerging High Performance Fortran standard, places a heavy emphasis on directives for data distribution, also referred to as data layout. Software systems must treat the data distribution as an entity known first at run-time. This need arises both due to the need to use different data distributions for different functions, and due to the need to execute a program on different numbers of nodes at different times without recompilation. We discuss data distribution issues in Section 4.

Particularly important in multiple-instance computation is the need to avoid temporary variables with a different data distribution than the original array when operations are performed on subsets of data, or even entire arrays. Multiple-instance computation consists of computations on disjoint subsets of array data. With proper attention to the data reference patterns, load–balance, and concurrency, many operations can be performed in-place with maximum concurrency and minimal data motion. We discuss managing temporary storage requirements in the context of multiple-instance computation in Section 8.

For some computations there exists a duality between data distribution and scheduling of operations in that a given efficiency can be achieved by matching a given data distribution strategy with a suitable scheduling strategy, whereas, in others the scheduling strategy may be completely determined by the algorithm and independent of the data distribution. LU factorization and triangular system solution belong to the first category [23, 24, 50,
the Fast Fourier Transform (FFT) belong to the second [39, 69]. For computations
that involve more than one object, such as matrix multiplication, the shapes, sizes, and
distributions of the different objects have a significant impact on what algorithms should
be chosen for optimal performance [27, 29, 43, 44, 53]. Partitioning, replication, and
nodal array shape may vary widely. In the CMSSL, the decision of which algorithm to
use for a given function is made at run-time. Some of the issues in choosing an algorithm,
depending upon data distribution and object shapes and sizes, are indirectly addressed
in Section 8, while Section 9 discusses the issues directly.

For a given function, the routing of data through the network often has a significant
impact on performance. The communication system is the most critical resource for
performance in most distributed memory systems. On the Connection Machine systems,
a communications library is used for a number of data motion patterns for which the
general router yields unsatisfactory performance. The library routines accepts any data
distribution that can be specified in the supported languages, and provides optimal routing
for the desired operation. Some of the functions in the communications library are briefly
discussed in Section 11.

The outline of this paper is as follows. We first review some of the essential characteristics
of the technology used for massively parallel, distributed memory architectures. Then,
we review the benefits of preserving locality of reference in a few typical scientific subroutine
library functions. In Section 4, we discuss data distribution in distributed memory
systems. Section 5 presents the essential characteristics of the Connection Machine sys-
tem CM-200, and Section 6 discusses multiple-instance computation. Section 7 discusses
the relationship between the virtual machine defined by the programming languages, and
two techniques to avoid unnecessary local data motion. Section 8 focuses on avoiding
unnecessary data motion in multiple-instance computation in the context of subroutine
libraries. Section 9 gives an example of how scheduling of computations can be matched
to the data distribution to achieve load-balance, as well as an example of where this is
not possible. Section 11 discusses some of the communication functions that support the
CMSSL. Section 12 discusses how run-time local loop partitioning and loop ordering are
made in one of the functions in the CMSSL. Section 13 provides a summary of the issues
addressed in this paper.

2 Scalable architectures

Scalable architectures are multiprocessor, or multicompiler architectures, that are exten-
sible to extreme performance with preserved efficiency and without change in the basic
architecture. A central design issue in architectures for extreme performance is memory
bandwidth. Preservation of locality of reference is a key issue in the design of algorithms,
compilers, and run-time systems. The clock rate at which processors are operated have
been higher than the rate at which memory units are operated throughout most of the
history of high performance computers. In the Atlas, the first virtual memory machine,
the difference was due to the use of different technologies for memory and processors. The
use of different technologies is also the reason for the large difference in speed between processors and memory in the Cray-2. In computers like the Cray YMP (bipolar technology), and in personal computers and workstations (MOS technology), the difference is due to the characteristics of devices and interconnections in silicon technologies. The difference in the rate by which a processor can accept and process data, and the rate at which a single memory unit can deliver or accept data, has resulted in computer systems with more memory units, or banks, than processors. Some designs have as many as 256 memory banks per processor. The total number of memory banks in traditional supercomputers has been in the range 32 – 1024.

The dominating memory technology today is MOS technology. The speed of individual chips in this technology is expected to increase slowly over the next few years. Therefore, tens of thousands of MOS memory units will be required for the next generation supercomputers with a capacity of a few trillion operations per second. Such supercomputers will have the processors distributed among the memory units. Processors with local memories will be interconnected by a network. A processing unit with its local memory and communication circuitry is referred to as a node. The network capacity is expected to be lower than the aggregate memory bandwidth represented by all the nodes, or the aggregate processing capacity. Exploiting locality of reference reduces the need for communication capacity, and is necessary for sustained high performance.

Whereas the speed of memory chips is expected to increase only slowly, the speed of processors is expected to increase dramatically over the next few years. Currently, the difference in memory and processor speed in MOS technologies is about a factor of two, with memories operating at 15 - 20 MHz, and processors operating at 30 – 40 MHz. The clock rate for MOS processors are expected to increase by a factor of 10, while the speed of memory chips is expect to grow only modestly during the next 3 – 4 years. With such a large difference in speed between processor and memory chips, a balanced design will require a careful design of the memory system in each node. Whether a banked local memory system, as in traditional supercomputers, or a combination of caches and interleaved memories will be preferable, is an open issue.

The bandwidth requirement on the communications network is substantial in a computer system operating at a few trillion operations per second, even if locality of reference is exploited properly. For instance, consider first the case with no locality of reference for a three operand instruction, such as $a = b + c$. In 64-bit precision, each instruction requires 24 bytes of data. Furthermore, each operand may require a 6-byte address. With an 8-byte operations code, a total of 50 bytes are required per instruction. At a total rate of two trillion operations per second, the memory system must deliver 100 Tbytes/sec. With good locality of reference and address generation through fixed increments (strides), the demand on the memory system may be reduced to 1 Tbyte/s or less for many computations. A few examples are given in the next section.

Much of the cause of a limited network bandwidth compared to the aggregate nodal memory bandwidth, is due to limitations of current packaging technologies. These technologies introduce a hierarchy in computer system design by limiting the communication rates between parts of the system. The rate at which data can be moved on a chip is
about two orders of magnitude higher than the rate at which a chip can communicate with other parts of the system. On-chip communication can fully use the surface area, often in several layers, while external interconnections are limited to the perimeter. Moreover, the physical extent of the interconnections are often considerably larger than the interconnections on the chip for mechanical reasons, further decreasing the ratio between the off-chip and on-chip communication capacities. Similarly, the rate at which data can be moved on a printed circuit board is one to two orders of magnitude higher than the rate at which data can be exchanged between boards.

In summary, the next generation supercomputers will consist of thousands to tens of thousands of processing nodes. The on-chip data motion capacity is about two orders of magnitude higher than the rate by which information can be exchanged with the environment. The same ratio holds for the data motion rates on circuit boards and between boards. Each node will have its own memory system, consisting of registers, cache, and a few memory banks. Locality of reference must be exploited for sustained high performance, both with respect to the local memory system, and, in particular, with respect to the communications network between nodes. With each communication is associated both a latency and a data transfer time. The latter is determined by the communications and memory bandwidths. Though latency is significant in some existing systems, excess parallelism and pipelining can be used to substantially reduce or eliminate its impact. Reducing the need for communication bandwidth requires a different, less conventional analysis, and different solutions.

3 Benefits of locality of reference

The potential benefits of exploiting locality of reference for three typical computations are quantified in Tables 1 and 2. Table 1 shows how the optimum ratio of local memory references to remote references depend on the local memory size $M$, $\alpha$ and $\beta$ are problem dependent constants, while $d$ denotes the dimensionality of the problem. Block algorithms for matrix multiplication yield the optimum ratio of local to remote references for square matrix blocks of maximum size with respect to local memory. For the FFT, the radix of the algorithm should be the maximum possible given the size of the local memory [14, 22]. Sorting behaves as the FFT with respect to the benefits of exploiting locality of reference.

Table 2 [26] gives the number of operations (and local references) per remote reference per processor for a few local memory sizes, assuming optimal locality of reference. For matrix multiplication and a block of size $100 \times 100$ (local storage 32k words), the reduction in the need for memory or communications bandwidth is a factor of 100 compared to no locality of reference. For the FFT, the reduction in required interprocessor communication bandwidth is a factor of 14 for a radix-16k algorithm. The value of locality of reference is clear. Table 2 also illustrates the fact that data motion is a much more significant issue for computationally efficient algorithms, such as the FFT. Computationally efficient algorithms require relatively few operations per data point.

In a network of nodes, the distance from a node to other nodes is typically not uniform.
<table>
<thead>
<tr>
<th>Scaling</th>
<th>Operation</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta(M/\alpha)^d$</td>
<td>Dense Matrix-Vector Mpy</td>
<td>$d = 2$</td>
</tr>
<tr>
<td></td>
<td>Dense Matrix-Matrix Mpy</td>
<td>$d = 2$</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iterative/Relaxation Methods</td>
<td>$d = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d = 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d = 3$</td>
</tr>
<tr>
<td>$\beta \log(M/\alpha)$</td>
<td>FFT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sorting</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Local references/ remote references with optimal locality. $M =$ Local storage. $\alpha =$ Variables per object. $\beta =$ Operations per object.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Local storage (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processor</td>
</tr>
<tr>
<td></td>
<td>Reg.</td>
</tr>
<tr>
<td>Matrix Mult.</td>
<td>1</td>
</tr>
<tr>
<td>3D-Relaxation</td>
<td>2</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Number of local operations per remote reference. 3D-Relaxation: 7-point stencil, vector-length 8 ($\alpha = 8$, $\beta = 96$).
For instance, in a two-dimensional mesh with wraparound (a torus), there are four nodes at distance one, eight nodes at distance two, etc. In the binary \( n \)-cube network there are \( n \) nodes at distance one, and \( \binom{n}{i} \) nodes at distance \( i \). Thus, for locality of reference in networks, the network topology must be taken into account in distributing data to nodes, as well as in scheduling arithmetic, logic, and data motion operations.

Exploiting locality of reference affects both the latency, and the effective bandwidth of communications. For most circuit-switched systems, the variance in latency as a function of locality is typically insignificant, while in some packet-switched communication systems, the latency may depend upon locality of reference across the network in a significant way. Except for extremely fine-grain systems, there is often sufficient excess parallelism to hide the latency through pipelining and interleaving of computations and communications. Thus, the network latency due to lack of locality of reference may not degrade performance significantly. However, lack of locality of reference increases the requirement on network bandwidth, and contention may occur. Pipelining cannot be used to reduce the bandwidth requirement. A flexible scheduling of processing tasks with different communication needs may reduce the contention, and increase the throughput. The bandwidth limitations are often significantly more difficult to overcome than a relatively high latency.

4 Data distribution

Data distribution and algorithms (the scheduling of operations) must be viewed together for optimal performance. This fact is even more important in distributed memory architectures than in exploiting a local memory hierarchy. Spatial locality of reference is important both in virtual memory systems and in DRAM based physical memories. DRAMs are often operated in page mode, in which the difference between accessing different data items within a page and in different pages may amount to several cycles. Finding a good loop ordering, partitioning and skewing with respect to both temporal and spatial locality of reference and use of pipelines is a hard problem, and a subject of current research in compiler technology [73, 74]. The problem is further complicated by the fact that the data distribution may not be known until run-time, which is the case with the Connection Machine systems.

In a distributed memory system, determining a good schedule must not only take spatial and temporal locality of reference into account, but also network contention and load-balance. The goal of distributing the data across the memory units is to maximize concurrency, and to minimize the contention for memory accesses. The distribution of data elements to memory units, together with the scheduling of operations, affects the load-balance, the network contention, local memory accesses, vector length, and block sizes.

Consecutive (block) and cyclic distribution [24], or combinations thereof [23], are common schemes for allocating one-dimensional data arrays to memory modules in distributed memory architectures. In the consecutive distribution scheme, an array of \( M \) elements is
allocated to $N$ memory units by allocating $\left\lceil \frac{M}{N} \right\rceil \left\lfloor \frac{M}{N} \right\rfloor$ successive elements to the same memory unit. In cyclic distribution, element $i = M \mod N$ is allocated to memory unit $i$. Cyclic distribution is traditionally used in banked memory systems [18]. With the stride being a multiple of the number of memory units, the memory system operates at the speed of a single bank. This phenomenon is known as bank conflicts. The consecutive distribution scheme is currently used on the Connection Machine systems. Cyclic as well as consecutive distribution, and combinations thereof, are included in Fortran D [12], Vienna Fortran [7, 75], and the proposed High Performance Fortran.

The distribution of two-dimensional arrays to memory units for contention-free access was studied carefully in the context of the Burroughs Scientific Processor (BSP) [5, 46, 47, 48]. In all distribution schemes considered for the BSP, a two-dimensional index space was converted to a one-dimensional (linear) index space through row or column skewing in combination with the standard row and column major orderings. The resulting one-dimensional array was in all cases mapped to the memory units by the cyclic distribution scheme. It was shown that rows, columns, diagonals, and blocks can only be accessed concurrently when the number of memory units is a prime number. The resulting design, which includes algorithms for address generation and routing, is known as the *Prime Memory System* [48].

One important assumption in the Prime Memory System was that the interconnection network can support the communication of one word per memory unit every clock cycle to each processor, i.e., the network has the capacity of a full crossbar between processors and memory units. In current distributed memory architectures, this assumption is not valid. The network capacity may be up to two orders of magnitude less than the aggregate peak capacity of all memory units. Therefore, with a processor associated with each memory unit, or groups of several memory units, it is important to exploit locality of reference. The use of a *multidimensional address space* is necessary, in general, to achieve the desired level of locality of reference.

For instance, a Cartesian address space of the same dimensionality as the problem space, yields the desired locality of reference for relaxation methods and convolution, when combined with grouping of data by the consecutive scheme. A three-dimensional address space and consecutive allocation reduces the communication requirements for explicit methods for the solution of partial differential equations by up to two orders of magnitude, see Table 2. A cyclic data distribution, though effective in achieving load-balance for some computations, adversely affects locality of reference in others, such as relaxation. For the FFT a binary cube address space is ideal, and the address space shall have $\log_2 N$ dimensions.

The shape of a multidimensional address space, i.e., the shape of the local subarrays, has a significant impact on the performance for many computations. For a $d$ dimensional data array with equally frequent data references along each of the axes, and the same reference pattern for each array element, the number of remote references is minimized when the lengths of the segments of all axes mapped to a memory unit are the same [26]. Thus, for two-dimensional arrays, the local subarray shall be a square for minimum communication needs, and for three-dimensional arrays the local data set shall be a cube, etc. On the
Connection Machine systems the default data distribution strategy attempts to make the lengths of all segments mapped to a memory unit as equal as possible. This mapping is ideal for the explicit solution of partial differential equations, when the difference “stencil”, or “molecule”, is symmetric with respect to all axes. The mapping is also ideal for matrix multiplication [20, 30] and for LU factorization and triangular system solution [50].

The choice between consecutive distribution, cyclic distribution, or a combination thereof, or some other distribution such as spectral decomposition [56], or random allocation [58, 57], is a trade-off between load-balance, locality of reference, and communication requirements. The consecutive distribution minimizes the number of remote references for stencil-like reference patterns. It also offers the possibility to improve the efficiency of the operations in each node by increasing the chance for good cache behavior through optimal blocking, and through long vectors for pipelined processors. However, it may yield poor load-balance in computations that use the data set nonuniformly, such as LU and QR factorization, and the forward and backward solution of triangular systems of equations. For such computations, either a (block) cyclic distribution can be used to improve the load-balance [23, 24, 72], or a block cyclic elimination order used in combination with the consecutive distribution [50]. In other computations, such as the FFT, the consecutive data distribution results in higher communication requirements than necessary. A cyclic distribution yields up to a factor of two lower communication requirements [39, 69]. A factor of two is very significant for computations such as the FFT.

The consecutive and cyclic distribution schemes, together with the dimensionality of the address space, determines which elements are grouped together for a node, and the communication needs of each node. The total demand on the communication system is also affected by how the groups of data are assigned to nodes. Ideally, the groups of data are allocated to the nodes such that the contention is minimized and the concurrency maximized. The distribution of groups of data must take both the data reference pattern and the network topology into account, as well as the routing scheme.

For nearest neighbor references along data array axes, such as for typical difference stencils, proximity preserving data array embeddings are ideal. If such embeddings exist. Such embeddings do exist in nodal arrays of a dimensionality that is at least as high as that of the data array [60]. Most binary cube based machines has a network of sufficient dimensionality to allow nearest neighbor embeddings of typical data arrays. On the Connection Machine system CM-200 [65], the default encoding of each data array axis preserves adjacency. Preserving adjacency in distributing data arrays can also be enforced through the compiler directive NEWS. Proximity preserving data array embeddings do not exist when the nodal array is of lower dimensionality than the data array that shall be embedded [60]. Similarly, proximity preserving embeddings of multidimensional data arrays do not exist in many other networks, such as binary trees [61] and butterfly networks.

For irregular data structures, preservation of locality of reference is a much more difficult problem. As in the structured case, there are two parts to the problem: determining which data elements shall be mapped into the same memory unit, and the placement of blocks of data to different memory units. The same techniques can also be used for structured problems when a proximity preserving embedding is either not known, or hard
to find. Recently, spectral decomposition techniques [56] have been used successfully to partition unstructured grids into blocks of data which exhibit a good surface to volume ratio with respect to data references. Thus, the techniques measurably reduce the load on the communication system compared to a simple linearized address space. Randomized placement of (blocks of) data [58, 57] reduces the risk for bottlenecks in the routing system. The randomized placement of data achieves the same communication load characteristics in a single (deterministic) routing phase as randomized routing achieves in two phases [70, 71]. Randomized placement is an option in some CMSSL routines [68]. Spectral decomposition is currently being included in the CMSSL.

In summary, the assignment of data to memory units affects load–balance, communication requirements, network contention, and the performance of each node by affecting the ability to carry out local blocking and pipelining of operations. Consecutive distribution preserves locality of reference along data array axes, while cyclic distribution may improve the load–balance for nonuniform data use (e.g., LU factorization), or yield reduced communication needs (e.g., FFT). Combinations of consecutive and cyclic distribution may offer a good trade–off between load–balance and the ability to perform local blocking [25]. For operations on irregular data structures, spectral decomposition [56] has proven effective in blocking data with respect to locality of reference. The placement of data grouped together among processors is also of great significance. For irregular computations, and when proximity preserving embeddings may not be possible, minimizing the contention through randomized distribution [58, 57], or randomized routing [70, 71], may be desirable.

An optimum distribution of data requires not only a data dependence analysis, but also an understanding of optimum embeddings and routing algorithms for the network at hand. This optimization problem is the topic of current research [8].

Finally, we remark that data distributions must be treated as defined first at run–time. Library routines must yield correct result, and as good a performance as possible, given the specified data distributions of the input and output objects. The decisions of what algorithm to choose for a given function, and whether or not a redistribution shall be performed before and after executing the function, is made at run–time.

5 The Connection Machine system architecture

All languages for the Connection Machine systems support a global address space, and distribute data evenly among the nodes, using the consecutive distribution scheme. For multidimensional arrays, the Connection Machine Run–Time System geometry manager creates a canonical layout based on the shape of the data array and the total number of nodes. The geometry manager attempts to configure the nodes such that the surface area in each node is minimized for a data subarray of fixed size. Thus, a multidimensional address space is used. The strategy used by the geometry manager in assigning nodes to the axes of the address space maximizes the average number of local references for each remote reference, when the data references are equally frequent along all array axes. For
multidimensional arrays, alternate data distributions (layouts) can be obtained through compiler directives. A data array axis can be made local to a node through the directive SERIAL. The length of the local axis segment can also be controlled through axes’ weights. A relatively high weight for an axis increases the length of the local segment of that axis at the expense of the length of the segments of the other axes. The total size of the data subarray is independent of the assignment of weights (for sufficiently large data subarrays). Only the shape of the data subarray changes. Thus, in order to optimize the performance by reducing the communication requirements, a high weight may be used for axes with frequent references compared to other axes.

As an example, consider a $64 \times 128$ array assigned to 64 nodes. Each node receives 128 elements. With the nodes configured as an $8 \times 8$ nodal array, the data subarray assigned to each node is of shape $8 \times 16$. Since a consecutive distribution scheme is used, the elements of the data subarrays are selected as 8 and 16 successive elements along each axes, respectively. With the nodes configured as a one-dimensional array through compiler directives, each node is assigned a data subarray of shape $64 \times 2$ when the first axis is local to a node, and of shape $1 \times 128$ when the second axis is local.

The shape of the nodal array neither affects the number of arithmetic operations to be performed, nor the arithmetic load-balance, as long as all nodes are assigned data subarrays of the same size. But, the shape of the nodal array may affect the efficiency in utilizing the arithmetic units, for example, by affecting vector lengths and blocking. The shape of the nodal array also affects the communication requirements as explained in Section 4.

The elements of the Connection Machine system CM–200 are shown in Figure 1. The nodes in this architecture are interconnected as a binary cube with two channels between
each pair of nodes. The number of cube dimensions ranges from 7 to 11, depending upon system size. The number of nodes varies from 128 to 2048, each of which is equipped with 4 Mbytes of local memory. The floating-point processor in each node has one multiplier, one adder, 32 registers, 64-bit wide internal data paths, but only one 32-bit wide data path to local memory. Figure 2 illustrates the local node architecture. The clock frequency is 10 MHz. The peak floating-point rate for each processor is 20 Mflop/s, both in 32-bit and 64-bit precision. But, for operations on 64-bit data types, the peak rate is limited to 10 Mflop/s for any operation that requires one operand to be loaded from memory. The arithmetic units are pipelined, with a pipeline length of three cycles. There is a one-cycle delay for memory operations. In the Connection Machine Instruction Set (CMIS), each vector operation requires a minimum of six cycles. The main memory is made up of DRAM, operated in page mode. Successive loads of data elements within the same page require a single cycle per 32-bit element, while loading elements stored in different pages incurs a page fault. A page fault adds one cycle to a load. Storing data in memory requires about two cycles per item.

The default distribution of data arrays preserves adjacency by using a binary–reflected Gray code \[59\] encoding of each axis. With the number of nodes assigned to an axis being a power of two, the binary–reflected Gray code is efficient both in preserving adjacency and in processor utilization, when the set of nodes can be partitioned among the axes such that the length of each data array axis is a multiple of the number of nodes assigned to that axis \[17\].
For arbitrary data array axes lengths, the Gray code may be combined with other techniques to generate efficient embeddings [6, 19]. However, these techniques are currently not used on the Connection Machine systems. Instead, the size of an array is extended such that its size is a multiple of the number of nodes [66], a garbage mask being used to mark the added array elements. This strategy may have significant performance implications, in that, on the Connection Machine system CM–200, conditional operations are performed on all array elements; the store operations are conditional. Thus, both the condition evaluation and the increased array size adversely affects the performance on arrays with a garbage mask. Some routines in the CMSSL avoid this penalty by operating on valid elements only, at the expense of more complex control.

Though many algorithms may use local references in a Cartesian index space, many divide-and-conquer algorithms, such as the FFT, for instance, reference data that are adjacent in a binary decomposition of the index space. For such computations, the standard binary encoding of the index space is preferable over a binary–reflected Gray code encoding. In the particular case of the FFT, if the data is allocated with binary–reflected Gray code, the conversion to binary code can be integrated into the FFT computation at no additional cost in communication or computation over a binary encoding of the data set [37], but at added code complexity. Avoiding data redistribution through integrated code conversion may not always be possible for computations requiring nearest neighbor references in a binary encoding of the index space. Therefore, the Connection Machine system CM–200 also supports the standard binary encoding of array axes (in combination with the consecutive distribution scheme). This encoding is obtained through the compiler directive SEND. Data redistribution from NEWS to SEND, or vice versa, is supported through optimized communication routines.

The distribution of one array with respect to another is important in computations referencing more than one array. For instance, matrix multiplication, \( C \leftarrow A \times B \), has three operands of shape \( P \times R, P \times Q \) and \( Q \times R \), respectively. With the canonical layout, the set of nodes may have three different configurations, one for each operand. Efficiency in the evaluation of the matrix product may dictate that a shared nodal array configuration be established. On the Connection Machine systems such a shared nodal array shape is achieved through the directive ALIGN.

In summary, all languages used to program the Connection Machine systems use a global address space. The address space is partitioned by default into as many axes as there are axes in a data array. A consecutive distribution scheme is used for each axis. The default shape of each local subarray has local axis segments of equal length for all axes. The default distribution of subarrays preserves adjacency along each of the axis (NEWS). Together, the default layout options constitute the canonical layout. A noncanonical layout can be achieved through the compiler directives SERIAL and ALIGN and through axis weights, which affects the shape of the data subarrays (and the nodal array), and through the directives SEND and ALIGN. Different data array axes may have different layout directives. Layout directives similar to the ones currently available on the Connection Machine systems are included in Fortran D, Vienna Fortran, and High Performance Fortran.

On the Connection Machine systems the actual physical distribution is not known until
run-time. Clearly, correctness of most operations requires that the actual layout of the data be known. An array descriptor associated with each array keeps all the information about the array data type, the length of each of the array axis, the encoding of each of the array axis (SEND or NEWS), the distribution of the array among the nodes in the form of subarray axes lengths and range of nodal addresses for each axis, and the subarray distribution in the form of strides for each of the subarray axes. Library routines make use of this information for correctness and efficiency.

6 Multiple-instance computation

Many large scale applications require that several computations of the same type be performed. Often, these computations can be performed concurrently. For instance, the computations for all elements in a finite element application can be performed concurrently [42], and so can the computations for the different lattice sites in Quantum ChromoDynamics (QCD) computations [1], and the computations involving the shear matrices in each grid point in finite difference methods for Navier–Stokes solvers [55]. We refer to the concurrent application of the same function to different segments of arrays as multiple-instance computation. The shear matrix in a grid point represents one instance of an array of shear matrices. There may be hundreds of thousands to millions of shear matrices in a large scale application. Similarly, there may be millions of so called SU(3) matrices in a QCD application.

It is convenient to represent multiple-instances of an object in a single, multidimensional array. Thus, in QCD computations, the SU(3) matrices are represented by six-dimensional arrays (two axes for a matrix, and four axes for the lattice). In three-dimensional finite difference Navier-Stokes solvers, five-dimensional arrays are used for the shear matrices, and four-dimensional arrays for the flux vectors.

In the case of the SU(3) matrices and the shear matrices in the Navier-Stokes equations, each matrix is very small; it is desirable to make each instance local to a node. For finite element methods, the elemental stiffness matrices are substantially larger. Some solution methods, such as nested dissection, recursively creates larger and larger matrices. Thus, even though the initial matrices may be allocated to a node, some of the matrices created in the recursive process will no longer be local to a node but distributed over several, or possibly, all nodes. Thus, in multiple-instance computation, each instance may be confined to a single node, or maybe distributed over a few or several nodes. The number of instances and their size may vary widely. Typically, there is a very large number of instances when each instance is small, but few instances when each instance is large.

Performing multiple-instance computations efficiently is critical for the success of massively parallel computing. With respect to library routines, multiple-instance computations pose the problem of whether the multiple-instances are handled by a single call, or of whether multiple calls are made to routines each of which handles a single-instance. In the latter case, the call to the library routine would be inside a loop nest, or a FORALL statement. Performing the multiple-instance computation in a single call puts the burden
on managing concurrency on the writer of the library routine. Using routines for single-instance computation puts the burden on managing the concurrency on the compiler and the run-time system. In the CMSSL, the former approach was adopted. A single call to a library routine suffice for computation on multiple-instances of objects.

A critical issue in multiple-instance computation is data (re)distribution. In the next section we use the FFT and matrix-vector multiplication as model cases in discussing this issue. In particular, the use of canonical data layouts is discussed in the context of subroutine libraries and multiple-instance computation.

7 Programming for a physical machine in the virtual machine model

All programming languages on the Connection Machine systems support a global address space in which the distributed nature of physical memory is transparent to the programmer. However, for efficiency in data motion operations, it is important to recognize that the physical memory is distributed, and to minimize both data motion between nodes, and unnecessary local memory moves.

On the Connection Machine systems, the implementation of the virtual machine on the physical machine is fairly simple. Operations such as CSHIFT and EOSHIFT applied to an array implies that all elements move as specified. Thus, a one step circular shift on a one-dimensional array implies that every element is moved one location. With $K$ elements per node, $K - 1$ elements are moved in local memory while a single element is moved between nodes. Eliminating the local data motion by separating the set of data that must move between nodes from the data that stays within local memory may yield a significant performance improvement. Eliminating the local data motion requires that a proper set of pointers be updated, such that subsequent accesses to the data array is consistent with the data motion specified in operations such as CSHIFT and EOSHIFT. This approach is used in the so called “Stencil Compiler” for the Connection Machine systems [2].

The notion of local functions in the proposed High Performance Fortran standard will allow operations on local data to be treated differently from the data that must be moved between nodes. On the Connection Machine systems, a separation between the internode and intranode data motion can be made at the Run–Time System level, where the distinction between the virtual and physical machines can be made visible. In some cases, it is also possible to manage the local and internode data motion in Connection Machine Fortran through the partitioning of array axes. We illustrate the effect of partitioning an array axis on performance by considering all-to-all broadcast.

Consider the computation [3]

$$z_i = \sum_{j=1}^{M} F(y_i, x_j) \quad i = 1, \ldots, M$$

(1)
which can be expressed as follows using all-to-all broadcast. The program is entirely independent of the machine size. Local data motion is required when $M$ is larger than the number of nodes $N$.

**code 1**

```fortran
array :: x(M), y(M), z(M)
do i = 1, M
    all-to-all-broadcast(i, x(:), axis=1)
    z(:) = z(:) + F(y(:), x(:))
endo
do i = 1, M
    all-to-all-broadcast(i, x(:, :), axis=1)
endo
do j = 1, M/P
do k = 1, M/P
    z(j, :) = z(j, :) + F(y(j, :), x(k, :))
endo
do j = 1, M/P
do k = 1, M/P
    z(j, :) = z(j, :) + F(y(j, :), x(k, :))
endo
do j = 1, M/P
do k = 1, M/P
    z(j, :) = z(j, :) + F(y(j, :), x(k, :))
endo
do j = 1, M/P
do k = 1, M/P
    z(j, :) = z(j, :) + F(y(j, :), x(k, :))
endo
do j = 1, M/P
do k = 1, M/P
    z(j, :) = z(j, :) + F(y(j, :), x(k, :))
endo
do j = 1, M/P
do k = 1, M/P
endo
do j = 1, M/P
do k = 1, M/P
endo
do j = 1, M/P
do k = 1, M/P
endo
```

The instruction `all-to-all-broadcast(i, x(:, :), axis=1)` performs the $i^{th}$ all-to-all broadcast step along the first axis of array $x$.

In the program below the axis of extent $M$ is factored into two axes of lengths $P$ and $M/P$, respectively. All-to-all broadcast is performed along the axis of extent $P$. The program is again independent of the machine size, and yields correct result for any machine size (for all values of $P$ in the range $[1,M]$ that divides $M$). But, there is no local data motion when $P$ is equal to the number of nodes $N$, and the data motion is significantly reduced when $P \geq N$ is close to $N$. In the first use of the all-to-all broadcast function on the Connection Machine systems [4], $P$ was chosen as $N$.

**code 2**

```fortran
array :: x(M/P, P), y(M/P, P), z(M/P, P)
do i = 1, P
    all-to-all-broadcast(i, x(:, :), axis=2)
do j = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
do k = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
endo
do j = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
do k = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
endo
do j = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
do k = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
endo
do j = 1, M/P
    all-to-all-broadcast(i, x(:, :), axis=2)
do k = 1, M/P
endo
do j = 1, M/P
endo
do j = 1, M/P
endo
do j = 1, M/P
endo
```

For a data set of size $M = 2^{15} = 32,768$ and $N = 256$ processors, the broadcast time between nodes and the total broadcast time (including local broadcast) are shown in Figure 3 for various array shapes. Reshaping the array from a machine axis of length 32,768 and a local axis of length 1 to a machine axis of length 1,024 and a local axis of length 32 reduce the communication time seven-fold.

### 8 Canonical layouts and subroutine libraries

Three critical issues in designing subroutine libraries are:
Figure 3: Connection Machine system CM-200 all-to-all broadcast along the second axis of a two-dimensional array $A(M/P, P)$ with $M = 2^{18}$ and $N = 256$ nodes. A data element is four bytes wide.

- Data layouts for optimal performance of each function.
- Avoidance of unnecessary data motion in the repeated application of a function to disjoint data sets, as in multiple-instance computation.
- Avoidance of unnecessary data motion upon entering and exiting a subroutine.

Each function, such as the FFT, or matrix-vector multiplication, has its own ideal data layout which depends on the network topology and communication system. However, the layout of the array in the calling program may be determined by other considerations. Changing the layout may incur a significant performance penalty. On the other hand, preserving the layout of the array in the calling program may also incur a severe performance penalty, either through a high communication cost, or through poor load–balance. Communicating layout information between a calling program and a called program, and deciding when to change the layout of arrays used in the calling program, are key issues in subroutine library design. We illustrate these issues in this section by considering the FFT and matrix-vector multiplication. The FFT involves a single array, while the matrix-vector product involves three operands all of which may have different layouts and require an alignment to assure that each node has matching indices of the three operands.

### 8.1 The Fast Fourier Transform.

We consider the computation of the FFT along one axis of a two-dimensional data array of shape $P \times Q$. With a canonical data layout, as defined in Section 4, the set of nodes are configured as a two-dimensional array. Figure 4 illustrates the layout of a two-dimensional data array in row and column major order on a $2 \times 4$ nodal array. In general, a two-dimensional nodal array is of shape $N_r \times N_c$, where $N_rN_c = N$. 

18
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. Maximization of the concurrency for each FFT through the use of a canonical data layout for one-dimensional arrays of size $P$.
2. Computation of each FFT without data relocation.
3. Concurrent computation of all $Q$ FFT, each of size $P$, through multiple-instance routines.

Alternative 1 can be captured by 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
```

For Alternative 1, 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) [31] within processing node rows for the row major
Figure 5: Data redistribution for load-balanced column processing on a two-dimensional nodal array labeled in row major order.

ordering. In one–to–all personalized communication, a node sends a unique piece of data to 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 5. The redistribution corresponds to a change in data allocation from $A(:, :) \rightarrow A(:, \text{SERIAL})$ and back to the original allocation, one column at a time. 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 all–to–all personalized communication as well as after it. The skewing step is shown in Figure 6.

Alternative 2 can be captured by the following code fragments:

```fortran
    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
```

In Alternative 2, 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 can be illustrated by the code fragment:
Figure 6: The skewing step in data redistribution for load-balanced processing of columns in a column major labeling of a two-dimensional nodal array.

\[ \text{FORALL } J \text{ DO} \]
\[ \text{CALL FFT2}(A(:,J)) \]
\[ \text{ENDFOR} \]

Alternative 3 is used in the CMSSL. All different instances of the FFT represented by the different columns are treated concurrently in-place. However, a \texttt{FORALL} statement is not used, since its use in Connection Machine Fortran [67] is limited to a single assignment. The concurrency and data layout issues are managed inside the FFT routine, below the Fortran level of programming. The call to the multiple-instance library routine is consistent with a language with array syntax, and of the form

\[ \text{CALL FFT}(A, \text{DIM} = 1) \]

The argument \texttt{DIM} specifies the axis of the array \( A \) to which the function shall be applied. The actual CMSSL call to the FFT routine 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 [68].

The three outlined alternatives have the following performance characteristics.
Table 3: Optimal communication times for FFT computation on the Connection Machine system CM–200.

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 \( \min(P, N_r) \).


The arithmetic speedup is proportional to \( \min(N, PQ) \).

The third choice is clearly preferable both with respect to communication and computation. The communication time in Alternative 1 is determined by the time for one–to–all and all–to–one personalized communication. For a \( P \times Q \) data array allocated to a \( N_r \times N_c \) nodal array, the communication time in a node limited communication system is proportional to \( 2Q \frac{P}{N_r} \). In a node limited communication system, the communication time is determined by the number of data elements entering and leaving a node. In a communications channel limited system, such as the Connection Machine system CM–200, the optimal communication time is determined by the number of data elements that must pass through a channel. The one–to–all and all–to–one personalized communication operations, requires a time of \( 2Q \frac{P}{N_r \log_2 N_c} \) [31]. The time required for the FFT computation itself is proportional to \( \frac{PQ}{N_r N_c} \) in the node limited model, and to \( \frac{PQ}{2N_r N_c} \) in the channel limited model [38, 40, 41, 64, 69]. Thus, the data redistribution time in Alternative 1 far exceeds the communication time required for the FFT computation. The communication times are summarized in Table 3, and the arithmetic speedups in Table 4.

Note that with a single–instance library routine and canonical layouts, Alternative 1 would be realized on a Connection Machine system.

The ideal encoding of the data array axes for the FFT is the binary encoding, i.e., SEND order. But, the default encoding is NEWS. The actual axes encoding and other layout
Table 4: Arithmetic speedup for FFT computation on the Connection Machine system CM–200.

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Alternative 1</th>
<th>Alternative 2</th>
<th>Alternative 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(N, P)</td>
<td>min(N, P)</td>
<td>min(N, PQ)</td>
<td></td>
</tr>
</tbody>
</table>

characteristics are known to the FFT routine through the array descriptor. If the encoding of the axis to be transformed is NEWS, then, in the CMSSL, currently a data redistribution from NEWS to SEND is performed before the FFT computation, and a SEND to NEWS reordering performed after it. Even though optimized routines are used for this redistribution, each redistribution step requires a time proportional to \( \frac{PQ}{N_{\text{NE }}} \) [24, 36], i.e., a time comparable to the time for the FFT computation [34]. It is possible to perform the FFT directly on NEWS ordered data [37], but such an algorithm is currently not implemented on the Connection Machine system CM–200.

The FFT algorithm bit-reverses the indices. An ordered transform produces the result in the same order as the order of the input. The CMSSL FFT routine has an explicit reordering stage when an ordered transform is specified. The reordering is of the form all-to-all personalized communication [31], for which optimal routines have been implemented on the Connection Machine system CM–200 [10]. The communication time for the bit-reversal phase, if requested, is comparable to the time for the communication in the FFT computation itself.

Finally, a cyclic distribution may have up to a factor of two lower communication requirements than a consecutive distribution [38, 69]. Redistribution from consecutive to cyclic distribution constitutes a shuffle operation, which, with optimal algorithms on binary cube networks, requires as much time as the potential reduction in communication time for the FFT computation.

In summary, the communication requirements for an FFT on a distributed multidimensional data array with a canonical layout are minimized when an ordered transform is computed in-place on data allocated cyclicly in SEND order. For NEWS ordered axes, and for ordered transforms, optimized communication routines are called by the CMSSL FFT. The communication routines used by the CMSSL FFT are:

- Butterfly network emulation.
- Binary code to binary-reflected Gray code conversion.
- Binary-reflected Gray code to binary code conversion.
- Bit-reversal.

The communication complexities are summarized in Table 3. For performance measurements see [40, 41].
The CMSSL real-to-complex and complex-to-real FFT is based on a complex-to-complex FFT as described above. In addition to the communication routines mentioned above, optimized index-reversal routines of the form \( k \leftarrow N_r - k \) are also used.

### 8.2 Matrix–vector multiplication.

The index space for matrix operations of the form \( C \leftarrow A \times B \) is best viewed in a three-dimensional Cartesian coordinate system, as shown in Figure 7. When two axes have their extents as one, then the resulting function is a level-1 BLAS function [49]. For example, the line \( P = R = 1 \) defines an inner product. Level-2 BLAS functions correspond to one axis having extent one. In Figure 7, matrix–vector multiplication corresponds to the plane \( R = 1 \). Similarly, the plane \( P = 1 \) defines a vector–matrix multiplication, and the plane \( Q = 1 \) defines a rank-1 update (outer product). The extent of all three axes for matrix multiplication (a level-3 BLAS) is greater than one.

The level-1 BLAS were developed for the early vector computers. Inner product and _AXPY_ routines form part of the level-1 BLAS, and can be used for matrix–vector multiplication, \( y \leftarrow Ax \). However, it was found later that the level-1 BLAS functions required an excessive amount of memory references for operations such as matrix–vector multiplication, and level-2 BLAS were defined [9]. The level-2 BLAS provide an interface for which both required loops for a single-instance computation are inside the subroutine call, whereas the use of level-1 BLAS put one of the loops inside the call, and the subroutine call inside the other loop. Thus, the level-2 BLAS provide additional optimization opportunities for the implementer of the library function, and reduces the demands upon

![Figure 7: The index space for matrix multiplication.](image)
the compiler.

Below, we discuss the use of _DOT and _AXPY routines in performing matrix–vector multiplication on distributed data. We again focus on the data motion, and demonstrate the consequences of various approaches, and of canonical data layouts. Effective conservation and management of data motion is not possible with a canonical layout of data arrays. In the CMSSL, three levels of a Distributed BLAS [43, 44], DBLAS, are used for the implementation of matrix operations.

8.2.1 Using level–1 _DOT Dblas for matrix–vector multiplication.

Using a _DOT routine to perform matrix–vector multiplication implies that the product is evaluated as a sequence of operations of the form: \( y(I) \leftarrow A(I,:) \times x(:,1) \). We first consider the computation of a single matrix–vector product, then discuss multiple–instance computations.

Single–instance matrix–vector multiplication

Each call for a single inner product involves a scalar, \( y(I) \), and two vectors. In a single–instance matrix–vector multiplication \( x \) and \( y \) are represented in one–dimensional arrays, while \( A \) is represented by a two–dimensional array of shape \( P \times Q \). As in the case of the FFT, we consider three alternatives in computing the matrix–vector product:

1. Maximize the concurrency for each inner product through a redistribution of row \( A(I,:) \).
2. Compute each inner product without redistribution of \( A(I,:) \).
3. Compute concurrently all inner products through multiple–instance routines.

Alternative 1 can be represented by the code fragments

```
FOR I = 1 TO P DO
    TEMP = A(I, :)
    CALL _DOT1(S,TEMP,X,Q)
    Y(I) = S
ENDFOR

SUBROUTINE _DOT1(T,B,C,N)
ARRAY B(N),C(N)

Single–instance _DOT computation
END _DOT1
```

In Alternative 1, \( A(I,:) \) is redistributed over all nodes to maximize the concurrency in the evaluation of each inner product. Since \( x \) is a one–dimensional array, \( A(I,:) \) and \( x \) are automatically *aligned* after the redistribution, for canonical layouts, i.e., elements \( A(I,J) \)
and \(x(J)\) are assigned to the same node. Each node computes a local inner product, followed by a global sum. The redistribution of \(A(I,:)\) in a column major ordering implies a one-to-all personalized communication within nodal columns. In row major ordering the communication corresponds to a reordering from row major to column major order, followed by the one-to-all communication within nodal columns. In effect, the matrix \(A\) is reallocated one row at a time from the layout \(A(:,:,:)\) to the layout \(A(:,:,SERIAL,:,:)\), since all elements in a column are allocated to the same node. After the computation of each inner product, the element \(y(I)\) must be sent to its final destination. Thus, Alternative 1 using canonical layouts, results in a redistribution of the entire matrix \(A\) in a load-imbalanced way with respect to communication. The arithmetic speedup is proportional to \(\min(N, Q)\).

Alternative 2 can be represented by the code fragments

\[
\text{FOR } I = 1 \text{ TO } P \text{ DO} \\
\quad \text{CALL } _2\text{DOT}(S,A,P,Q,X,I) \\
\quad Y(I) = S \\
\text{ENDDO}
\]

\[
\text{SUBROUTINE } _2\text{DOT}(T,B,M,N,C,K) \\
\quad \text{ARRAY } B(M,N), C(N) \\
\quad \text{ARRAY } \text{TEMP}(M,N) \\
\quad \text{TEMP}(K,:) = C \\
\quad \text{Single-instance, in-place } _2\text{DOT} \text{ computation on } B(K,:) \text{ and } \text{TEMP}(K,:) \\
\text{END } _2\text{DOT2}
\]

Alternative 2 avoids the redistribution of \(A\). But, since the indices of \(x\) and \(A(I,:)\) must be aligned, the vector \(x\) must now be reallocated from being distributed over all nodes to being distributed over the same nodal row as row \(A(I,:)\). Thus, Alternative 2 has as extensive communication requirements as Alternative 1, and, in addition, the arithmetic is poorly load-balanced. The arithmetic speedup is proportional to \(\min(N_c, Q)\).

The discussion of Alternative 1 suggests that a canonical layout of \(x\), and the noncanonical layout \(A(:,:,SERIAL,:)\) of \(A\) should be used. This layout only requires an all-to-all reduction within rows, which is performed as a global reduction yielding one element at a time, followed by a one-to-all personalized communication of the result to comply with a canonical layout of the vector \(y\). This redistribution is performed one element at a time, when a single-instance \_DOT routine is used. The merits of noncanonical layouts are discussed further in Section 8.2.4.

Thus, the use of either Alternatives 1 or 2 for canonical layouts results in excessive communication. Noncanonical data layouts can significantly reduce the need for data motion, but since most arrays are used in many computations, it is necessary that a library routine is efficient for a variety of layouts, including canonical data layouts. Alternative 3 has these desirable characteristics.

In the calling program, Alternative 3 corresponds to a code fragment of the form
FORALL I DO 
  CALL _DOT(Y,A(I,:),X)  
ENDFOR

The actual evaluation of such a FORALL statement requires an alignment of A, X, and Y. It also requires a replication of X for a concurrent evaluation of Y. The CMSSL does not rely on a FORALL statement as indicated above, but performs a fully concurrent evaluation of the matrix–vector product as described in Section 8.2.4. The call is of the form

CALL _DOT(Y,A,DIM=2,X)

Multiple–instance matrix–vector multiplication

In a multiple–instance computation, x and y are multidimensional arrays of the same rank. The rank of the array for A is one higher than the rank of the arrays for x and y. In addition to the alignment and data motion issues mentioned for the single–instance case, the alignment of instances also becomes an important issue. The shape of each instance of an object is defined by the problem axes (axis), while the instances are defined by the instance axes (axis). Arrays have conforming shapes when the shapes of the subarrays defined by the instance axes (axis) are the same, i.e., the shapes of the arrays are identical if the problem axes are removed. The call below illustrates these issues.

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).

In the example, 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 x and y, 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 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 a $M \times L$ matrix by a vector of length L. There are $N \times K$ such instances. The call is independent of the distribution of the arrays.

Restricting the arrays in an operation to have conforming shapes allows for an implicit ordering of instances corresponding to the ordering of the axes in the arrays.

8.2.2 Using level–1 _AXPY DBLAS for matrix–vector multiplication.

Using an _AXPY function for a single–instance matrix–vector (or vector–matrix) multiplication implies that the matrix–vector product is computed through a sequence of operations $y(\cdot) \leftarrow A(:, J)x(J)$. Again, three obvious alternatives are:
1. Maximize the concurrency for each \_AXPY operation through a redistribution of column \( A(:, J) \).

2. Compute each \_AXPY operation without redistribution of \( A(:, J) \).

3. Compute concurrently all \_AXPY operations through multiple-instance routines.

Alternative 1 can be represented by the code fragments

```
FOR \( I = 1 \) TO \( Q \) DO
    TEMP1 = A(:, I)
    S = X(J)
    CALL \_AXPY1(Y, TEMP1, S, P)
ENDFOR
```

```
SUBROUTINE \_AXPY1(A,B,T,N)
ARRAY A(N),B(N)
ARRAY TEMP2(N)

    TEMP2 = T
    A = A + TEMP2*B
ENDDAXPY1
```

In Alternative 1, each column is extracted from the matrix and reallocated over all nodes. With a canonical layout of the result vector \( y \) and the extracted column, the two vectors in the \_AXPY operation are aligned. In order to perform the computation, a broadcast of \( x(J) \) is also required. The total effect is that the matrix \( A \) is reallocated one column at a time to a layout corresponding to the layout \( A(:, SERIAL) \). This reallocation is identical to the reallocation for the FFT computation illustrated in Figures 5 and 6. Each element of \( x \) is broadcast to every node, i.e., an all-to-all broadcast [3, 31] is performed one element at a time. Thus, the communication requirements are as extensive as in Alternative 1 for the \_DOT alternative. The arithmetic speedup is proportional to \( \min(N, P) \).

Alternative 2 implies that \( y \) is aligned with each column of \( A \), one column at a time. In addition, elements of \( x \) must be aligned with the corresponding column of \( A \), and broadcast to all nodes over which that column is distributed. Moreover, the arithmetic is poorly load-balanced. Thus, with respect to communication, Alternative 2 is more demanding than Alternative 1.

For Alternative 3 the same comments apply to the \_AXPY alternative as to the \_DOT alternative.

8.2.3 Discussion.

By using either a \_DOT or an \_AXPY routine for matrix-vector multiplication, the entire matrix is moved in Alternative 1. The difference is whether it is moved a row at a time,
or a column at a time. The communication is a sequence of one-to-all personalized communications for the matrix $A$. Though maximum concurrency in computation is achieved (ignoring a certain inefficiency in the reduction operation between nodes in the single-instance case), clearly much more data is moved than if only the two vectors are moved. In addition, moving a row or a column at a time does not yield good load–balance in the communication system for most networks.

Avoiding the canonical layout for the rows in the _DOT function, and the columns in the _AXPY function, by aligning the computations with a row (_DOT) or a column (_AXPY) of the matrix, causes a copy of $x$ and $y$ to be aligned with each row and column of the matrix, respectively. The data motion is the same as if $A$ is aligned with $x$ and $y$, respectively. In addition, aligning the computations with $A$ results in poor arithmetic load–balance.

Thus, we conclude that for matrix–vector multiplication using single-instance _DOT routines

- causes a redistribution of the matrix from the layout $A(:, :) \rightarrow A(:, SERIAL :)$ one row at a time, when $A$ is aligned with $x$. The arithmetic speedup is proportional to $\min(N, Q)$.

- causes a redistribution of $x$ from the layout $x(,:) \rightarrow A(K, :) \rightarrow A(K, SERIAL)$ for all $P$ values of $K$ when $x$ is aligned with $A(:, :).$ The arithmetic speedup is proportional to $\min(N, Q)$.

and using single-instance _AXPY routines

- causes a redistribution of the matrix from the layout $A(:, :) \rightarrow A(:, SERIAL :)$ one column at a time when $A$ is aligned with $y$. The arithmetic speedup is proportional to $\min(N, P)$.

- causes a redistribution of $y$ from the layout $x( :) \rightarrow A(,:, K)$ for all $Q$ values of $K$ when $y$ is aligned with $A(:, :).$ The arithmetic speedup is proportional to $\min(N, Q)$.

The communication and arithmetic requirements for Alternative 1 of the _DOT and the _AXPY alternatives are summarized in Tables 5 and 6. To minimize the data motion, and yet achieve good load–balance for canonical layouts, algorithms that keep the matrix stationary and perform multiple _DOT or _AXPY computations concurrently are required. Thus, multiple-instance computation must be considered also for BLAS operations on distributed data structures. In the absence of powerful analysis tools, level–2 DBLAS are required both for efficiency in data motion between nodes, and for efficiency in utilizing the memory hierarchy and pipelines in each node. Also, up to $PQ$ nodes may be used effectively for the computation rather than $P$ or $Q$ nodes.

**Remark.** If the number of processors is smaller than the maximum of the number of rows and columns, then full parallelism can be achieved with a noncanonical layout in
which either complete rows or columns are allocated to each node. Communication of either the input vector, or the output vector may suffice in such a layout. Changing a canonical matrix layout to correspond to a one-dimensional array of processing nodes is equivalent to (block) matrix transposition. The transposition can be made more efficiently than the communication required for Alternative 1 for _DOT and _AXPY routines, since the transposition of the entire matrix at once, offer the opportunity for load-balanced communication.

8.2.4 A level-2 DBLAS for matrix-vector multiplication.

In this section we consider the required data motion when both loops in single-instance matrix-vector multiplication are performed concurrently. We first consider a one-dimensional partitioning of the matrix by rows, then by columns, followed by a canonical layout of both vectors and the matrix.

With a one-dimensional partitioning of the matrix by rows, as shown in Figure 8, and the input vector distributed evenly over all nodes, an all-to-all broadcast of the input vector is required in order to carry out the matrix-vector product. No communication is required for the result vector. The arithmetic speedup is proportional to \( \min(N, P) \).

If, instead, the matrix is distributed by columns as shown in Figure 9, and the input and output vectors are evenly distributed over the processing nodes, then no communication is required for the input vector, but an all-to-all reduction is required for the result vector. The arithmetic speedup is proportional to \( \min(N, Q) \).

Thus, in a linear array with the matrix partitioned by rows, the matrix-vector multiplication can be expressed as

\[
\text{All-to-all broadcast of the input vector} \\
\text{Local matrix-vector multiplication.}
\]

In a linear array configuration with the matrix partitioned by columns, the matrix-vector multiplication can be expressed as

\[
\text{Local matrix-vector multiplication} \\
\text{All-to-all reduction for the output vector.}
\]
For a canonical layout the nodes are configured as a two-dimensional array for the matrix, and as a one-dimensional array for the input and output vectors. For such a layout, both all-to-all broadcast and all-to-all reduction is required. Figure 10 illustrates the data distribution for both row major and column major ordering.

For the matrix–vector multiplication, the vector must be aligned with a row and spread along a column. The alignment is an all-to-one personalized communication (gather) within columns in the column major ordering. The spread is a one-to-all broadcast within columns. The net effect of these two operations is an all-to-all broadcast \[13, 31, 62, 63\] within processing node columns for any length of the processing node axes, and any length of the inner axis \(Q\) of a \(P \times Q\) matrix \(A\). The all-to-all broadcast requires the same time as each of the all-to-one personalized communication and the one-to-all broadcast [31]. Thus, it is clearly preferable to perform the alignment and the spread as a single all-to-all broadcast.

After the all-to-all broadcast, each node performs a local matrix–vector multiplication. The result of this operation is that each node contains a segment of the result vector \(y\).
The nodes in a processor row contain partial contributions to the same segment of $y$, while different rows of nodes contain contributions to different segments of $y$. No communication between rows is required for the computation of $y$. Communication within rows suffices.

The different segments of $y$ can be computed by all-to-all reduction within processor rows. This reduction yields a row major ordering. But, the assumed node labeling for the previous discussion is column major order, and a reordering from row to column major ordering is required in order to establish the final distribution of $y$. Thus, for a column major ordering of the matrix elements to nodes, the operations are

- All-to-all broadcast of the input vector within processing node columns
- Local matrix-vector multiplication
- All-to-all reduction within processing node rows to accumulate partial contributions to the result vector
- Reordering of the result vector from row major order to column major order.

The reordering from row major ordering to column major ordering is equivalent to a shuffle, or matrix transposition.

If the elements of the matrix $A$ had been allocated to processing nodes in row major order instead of column major order, then a reordering from row major order to column major order must be performed prior to the all-to-all broadcast of the input vector. No reordering is required for $y$. Thus, for row major ordering of matrix elements to nodes, the operations are

- Reordering of the input vector from row major order to column major order
- All-to-all broadcast of the input vector within processing node columns
- Local matrix-vector multiplication
- All-to-all reduction within processing node rows to accumulate partial contributions to the result vector.

With the matrix uniformly distributed across all nodes, the arithmetic is load-balanced. The speedup is proportional to $\min(N, PQ)$. The operations in each node can be performed using a traditional BLAS.

In summary, the level-2 DBLAS routine described above computes the matrix-vector product with the matrix in-place. The vector $x$ is replicated, but the required communication time is a factor $P$ less than in Alternative 2 of using a single-instance _DOT routine. Similarly, the all-to-all reduction of $y$ is a factor $Q$ faster than in Alternative 2 of the single-instance _AXPY approach. The communication requirements are:

- All-to-all broadcast of $x$ requiring a time proportional to $\frac{Q}{N_c}$ in a node limited communication model, and proportional to $\frac{Q}{N_c \log_2 N_r}$ on the CM-200 (link limited).
- All-to-all reduction on $y$ requiring a time proportional to $\frac{P}{N_c}$ in a node limited communication model, and proportional to $\frac{P}{N_c \log_2 N_r}$ on the CM-200.
- Transposition requiring a time proportional to either $\frac{Q}{2N}$ for $x$, or $\frac{P}{2N}$ for $y$ in a node limited communication model, and to $\frac{Q}{2N}$ and $\frac{P}{2N}$ on the CM-200, respectively.
Table 5: Communication times for level-1 and level-2 DBLAS for matrix–vector multiplication on the CM–200.

<table>
<thead>
<tr>
<th>Operation</th>
<th>_DOT (Alt. 1)</th>
<th>_AXPY (Alt. 1)</th>
<th>Level-2 DBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment of A</td>
<td>$\frac{Q}{N_c \log_2 N_r}$</td>
<td>$\frac{P}{Q(1 + \log_2 N_r N_c)}$</td>
<td>$-$</td>
</tr>
<tr>
<td>Broadcast of $x$</td>
<td>$-$</td>
<td>$-$</td>
<td>$\frac{Q}{N_c \log_2 N_r}$</td>
</tr>
<tr>
<td>Reduction on $y$</td>
<td>$P(1 + \log_2 N_r N_c)$</td>
<td>$Q(1 + \log_2 N_c N_r)$</td>
<td>$\frac{Q}{N_c \log_2 N_r}$ + $\frac{P}{N_r N_c}$</td>
</tr>
<tr>
<td>Total</td>
<td>$P(\frac{Q}{N_c \log_2 N_r} + \log_2 N_r N_c + 1)$</td>
<td>$Q(\frac{P}{N_c \log_2 N_r} + \log_2 N_c N_r + 1)$</td>
<td>$\frac{Q}{N_c \log_2 N_r}$ + $\frac{P}{N_r N_c}$ + $\frac{P}{N_r N_c}$</td>
</tr>
</tbody>
</table>

Table 6: Arithmetic speedup for level-1 and level-2 DBLAS for matrix–vector multiplication on the CM–200.

<table>
<thead>
<tr>
<th>Operation</th>
<th>_DOT (Alt. 1)</th>
<th>_AXPY (Alt. 1)</th>
<th>Level-2 DBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>min($N, Q$)</td>
<td>min($N, Q$)</td>
<td>min($N, PQ$)</td>
</tr>
</tbody>
</table>

Table 5 summarizes the communication requirements for the level-2 DBLAS routine described above in column major order, and Alternative 1 of the _DOT based matrix–vector multiplication also for column major order, and Alternative 1 of the _AXPY based matrix–vector multiplication for row major order. Table 6 gives the arithmetic speedup. The term $Q$ in the broadcast operation in the _AXPY alternative is due to the assumption that the broadcast of any element must be performed from a fixed node. Likewise, the term $P$ in the reduction on $y$ in the _DOT alternative is due to the assumption that all reduction operations leave the result in a fixed node. One additional communication is required to move the result of each reduction to its final destination.

The fact that matrix transposition from a performance point of view is much less significant than the other communication operations has been verified on the Connection Machine system CM–200 [52]. The all-to-all broadcast and all-to-all reduction are performed within columns of nodes, and rows of nodes, respectively. The different broadcast operations are completely independent of each other. Similarly, the different reduction operations are independent of each other.

Remark: The above algorithm for matrix–vector multiplication degenerates to the first two algorithms discussed in this section when the matrix is partitioned either by rows, or by columns.

For multiple-instance matrix–vector multiplication, the comments made in Section 8.2.1 still apply. However, with a canonical layout of all arrays, an explicit alignment of $x$ with $A$, and of the result with $y$ may be required. But, performing this alignment as a shuffle operation followed by an all-to-all broadcast may still prove advantageous, since the contention is minimized.

The level-2 DBLAS matrix–vector multiplication, as outlined above, is part of the CMSSL.
Figure 11: Consecutive and cyclic allocation of an $8 \times 8$ data array on a $4 \times 4$ nodal array.

9 Data distributions and scheduling of operations

9.1 Load–balanced Gaussian elimination.

Gaussian elimination [16] on a $P \times P$ matrix progresses in $P$ steps on a diminishing set of rows and columns. In the case pivoting is made on the diagonal (“no pivoting”), rows and columns 1 through $P - 1$ are updated in the first elimination step with rows and columns labeled from 0. In the second elimination, step rows and columns 2 through $P - 1$ are updated, etc. With a square nodal array of shape $\sqrt{N} \times \sqrt{N}$, the first nodal row and column become idle after $\frac{P}{\sqrt{N}}$ elimination steps. After yet another $\frac{P}{\sqrt{N}}$ elimination steps yet another nodal row and column become idle. In a cyclic allocation, as shown in Figure 11, it is clear that all nodes participate until the last $\sqrt{N}$ variables. The gain in arithmetic efficiency is a factor of 3 compared to the consecutive allocation and consecutive elimination [24].

But, by changing the pivoting order from consecutive to cyclic for a consecutive allocation, such that in elimination step $i$, row and column $i \frac{P}{\sqrt{N}}$ are selected as pivot row and column instead of row and column $i$, a load–balanced elimination is achieved also for consecutive allocation. Thus, a duality exists with respect to data distribution and scheduling of operations for load–balanced Gaussian elimination: consecutive allocation and cyclic elimination order, and cyclic allocation and consecutive allocation yield the same load–balance. Assuring pivoting on the diagonal for rectangular nodal arrays using cyclic elimination order and consecutive allocation requires explicit permutations. Such permutations are not necessary, however, when partial pivoting is used. For details of the block–cyclic elimination scheme, with and without partial pivoting, see [50].
9.2 Data distribution and communication in the FFT.

The computations in the FFT always progress from the most significant bit in the encoding of the index space to the least significant. Whether the computations proceed from the most significant address bit to the least significant, or from the least to the most significant address bit, depend upon whether normal or bit-reversed input order is used. With normal order input, the least significant bits of the index space are mapped to the local memory address, while the most significant bits are mapped to the processor address field. Thus, in computing an FFT through multisectioning [38, 40, 64, 69], it is necessary to carry out a permutation, or a sequence of permutations, such that data corresponding to complement values of the bits in the processor address field are reallocated to the same memory unit. However, if memory is conserved, this permutation implies that bits in the index space that originally were allocated to a memory unit become allocated to the processor address field. Thus, yet another permutation is necessary to bring these index bits back into local memory.

The following example illustrates this point. In the example, an FFT on 64 data points is performed on 16 nodes, each of which is assigned four elements in a consecutive allocation scheme. The address is represented as a sequence of numbers denoting the order of the bits in the encoding of the data index, and the address. With data in normal order and a consecutive allocation scheme, the data index and the address coincide. Below, \( \text{paddr} \) denotes the processor address field, and \( \text{lmaddr} \) denotes the local memory address field:

\[
\begin{align*}
(5 & | 4 & 3 & 2 & 10), \\
& \text{paddr} & \text{lmaddr}
\end{align*}
\]

Now, in order to perform the FFT computations, butterfly computations are carried out starting with bit 5, and progressing in descending order to bit 0. With computations performed locally, bits 5 and 4 can be moved to local memory through a 4-sectioning step, resulting in the data allocation:

\[
\begin{align*}
(0 & | 1 & 3 & 2 & 54), \\
& \text{paddr} & \text{lmaddr}
\end{align*}
\]

After the first two ranks of butterfly stages are performed on bits 5 and 4, a new 4-sectioning brings bits 3 and 2 into local memory:

\[
\begin{align*}
(0 & | 1 & 5 & 4 & 32), \\
& \text{paddr} & \text{lmaddr}
\end{align*}
\]

Two new ranks of butterfly computations can now be performed on bits 3 and 2 respectively. In order to complete the FFT, bits 1 and 0 must be brought back into local memory:
The net effect is that a right cyclic shift by one digit (2 bits in this case) has been performed on the processor address field. In addition, the FFT computation bit-reverses the bits. For a detailed discussion of ordering and communication issues see, for instance, [40, 64].

In a cyclic allocation scheme the initial assignment of index bits to memory address is

\[
\begin{pmatrix}
3254 \\ \text{padr}
\end{pmatrix} \begin{pmatrix}
10 \\ \text{lmaddr}
\end{pmatrix}.
\]

and it is clear that no communication is required for the first two ranks of butterfly computation. Two 4-sections suffice to complete the computations of an unordered transform. No permutation needs to be repeated.

From the example, it should be clear that if the axis segment assigned to a node is at least as long as the segment assigned to the processor address field, then the consecutive allocation scheme results in a communication need that is twice that of the cyclic allocation for an unordered transform. For details, see [38, 40, 69].

### 9.3 Matrix multiplication.

In Section 8 we described an efficient level-2 DBLAS algorithm for matrix–vector multiplication. The algorithm we presented for two-dimensional nodal arrays degenerates to well-known algorithms for one-dimensional arrays, should indeed the nodal array be one-dimensional. Thus, in this particular case, the generic algorithm for a two-dimensional nodal array can be viewed as a *polymorphic algorithm*. Its behavior, i.e., the schedule of operations depends upon the data distribution. In other cases, such as for matrix–matrix multiplication, the variety of schedules to be considered is much greater, and a library routine may need to employ one of a (large) set of algorithms coded explicitly. The matrix–matrix multiplication routine in the CMSSL exhibits such a polyalgorithmic behavior.

The matrix–matrix multiplication routine in the CMSSL uses an inner product routine if \( P = R = 1, \ Q > 1 \), where in the operation \( C \leftarrow A \times B + C \) \( A \) is of shape \( P \times Q \) and \( B \) is of shape \( Q \times R \). An outer product routine is called if \( Q = 1 \) and \( P, R > 1 \), and a matrix–vector routine is called if \( P, Q > 1 \) and \( R = 1 \), or \( Q, R > 1 \) and \( P = 1 \) (vector–matrix multiplication). For \( P, Q, R > 1 \) two nontrivial matrices are indeed multiplied. Several different algorithms must be employed in order to assure a good performance for any combination of matrix shapes. When all matrices are large relative to the number of nodes, an algorithm that keeps the largest matrix stationary and moves the other two as
required, shall be the one used [53, 43, 44]. Moreover, a three–dimensional nodal array configuration may yield the best performance [29, 27, 51]. For matrices of extreme shapes, load–balance may be a serious problem, and replication of one, two, or all three operands may be required for efficiency [28, 44].

For the multiplication of a pair of matrices that are large relative to the number of nodes, it has been shown [29, 27] that depending upon the shape of the matrices, the optimal shape of the nodal array may be one–dimensional, two–dimensional, or three–dimensional. Clearly, the schedule of operations are different, and, indeed, corresponds to different forms of loop partitioning and skewing. It is somewhat arbitrary whether or not these differences are considered as corresponding to different algorithms, or whether they constitute the same algorithm. However, different codes are indeed required; the selection must be made at run–time. For matrices of extreme shape, additional choices that involve replication must be considered [27, 53].

10 Optimal shape of the nodal array

In Section 4 we raised the issue of the significance of the shape of the address space with respect to communication performance. We now discuss this issue in the context of matrix–vector and matrix–matrix multiplication, LU factorization and triangular system solution, and the FFT.

For the matrix–vector multiplication algorithm described in Section 8.2.4, the communication consists of all–to–all broadcast, all–to–all reduction, and reordering between row and column major order. The latter is independent of the shape of the nodal array [31, 52]. The all–to–all broadcast time is proportional to $\frac{P}{N_c}$, while the all–to–all reduction time is proportional to $\frac{P}{N_r}$ [31]. With $N_r \times N_c = N$, the total communication time is minimized when $N_r = \sqrt{\frac{PN}{Q}}$ and $N_c = \sqrt{\frac{QN}{P}}$, or the submatrices local to a node is of shape $\sqrt{\frac{PN}{Q}} \times \sqrt{\frac{PN}{Q}}$. Note, that this is the default nodal array shape for the matrix in the Connection Machine systems. It follows that for matrices of extreme shapes a one–dimensional nodal array is optimal, since in such a case the transposition may be avoided, and the savings in communication time outweighs a minor increase in either the broadcast or reduction time due to a nonoptimal shape [27, 44].

In the case of matrix–matrix multiplication with the product matrix $C$ stationary, an all–to–all broadcast of $A$ is performed within nodal rows, and an all–to–all broadcast of $B$ within nodal columns. The communication times are proportional to $\frac{PQ}{N_r}$ and $\frac{QR}{N_c}$, respectively. The optimal values of $N_r$ and $N_c$ are $\sqrt{\frac{PN}{Q}}$ and $\sqrt{\frac{QN}{P}}$, respectively. The shape of the local submatrix of $C$ is $\sqrt{\frac{PN}{Q}} \times \sqrt{\frac{PN}{Q}}$, which is indeed the default shape on the Connection Machine systems. A similar analysis, when an algorithm with $B$ stationary is used, shows that the local submatrix of $B$ shall be square. With $A$ stationary, the local submatrix of $A$ shall be square [53]. For matrices of extreme shapes, one–dimensional nodal array shape may be advantageous, and for large nodal arrays a three–dimensional
shape may be optimal [29, 27].

In LU factorization and triangular system solution, the optimal nodal array shape is also determined by the time required for broadcast operations. For the factorization of a square matrix the nodal array shall be square, while for the forward and backward substitution on multiple right hand sides, the shape of the array of right hand sides determines the optimal shape of the nodal array. Square submatrices are again ideal [50].

Finally, for the FFT as it is implemented on the CM–200, once an axis subject to transformation is not entirely local, the distribution of data across nodes does not appreciably affect performance. Pipelining of communications across different cube dimensions makes the communication time depend only upon the number of local data elements [41].

11 Communication primitives

All existing distributed memory systems are equipped with a general purpose router, which, given a destination address, sends the data to that address, or fetches the data from a given source address. Most existing routers are nonadaptive, i.e., the routing strategy for a message is independent of the routing strategy for other messages. The nonadaptive routing strategy may be identical to an optimal strategy in some cases, but it may be also considerably worse. The purpose of communications libraries is to provide high efficiency in data motion. As is the case for arithmetic functions, higher level functions offers a potential for many more optimizations than low-level functions. High-level communication functions offer the potential to make the communication system transparent to the user. The communication functions included in the CMSSL provides optimal schedules for a number of high-level functions. Each such function takes as input an array with any data distribution that can be specified in any supported language, and the desired operation thereupon, such as reduction along a certain axis, or subset of axes, and a specification of how the result shall be allocated. It is the task of the designer and implementor of the communications function to determine from this information, and from the array descriptors and knowledge of the communication system, what algorithm to choose for the desired function.

On the Connection Machine system CM–200 the following communication functions have optimal implementations

- Global and segmented one-to-all broadcast.
- Global and segmented all-to-all broadcast.
- Binary code to binary-reflected Gray code redistribution.
- Binary-reflected Gray code to binary code redistribution.
- Circular shift.
- Polyshift.
### Figure 12: Achieved peak bandwidths per node in a 2048 node CM–200.

- Butterfly network emulation.
- Bit-reversal.
- Index-reversal.

All functions derive routing and scheduling information based on the actual layout of the arrays from the array descriptor. Different elements of the array are routed along different minimum length paths to minimize the contention, and the time to completion for a given operation. For some routing patterns, such as bit-reversal, the Connection Machine system router performs considerably worse than the optimized routine (by more than a factor of 10). The peak achieved bandwidth per node of a CM–200 is shown in Figure 12 for a few functions. For details of algorithms and implementations of one-to-all and all-to-all communication, see [31, 3, 52]. For binary code to binary-reflected Gray code conversion, see [24, 26, 32, 36, 21, 11]. Our polyshift implementation is presented in [15], while all-to-all communication is discussed in [10, 31, 33, 35].

### 12 Managing DRAM page faults

Though the local memory hierarchy in each node of a distributed memory architecture is conventional, the CMSSL has some unique features in scheduling for spatial locality. The
local memory hierarchy in the Connection Machine system CM–200 consists of a register set and DRAM operated in page mode. Page faults may reduce the performance by a factor of up to two, which is unacceptable for critical applications. The floating-point units are pipelined, and the vector length has a significant impact on the performance. This fact must also be accounted for in determining the schedule of operations. The multiple-instance capability of the library implies that none of the axes may have a stride of one, and that the strides may not be directly related. Anyone of the axes may have the smaller stride.

The physical data distribution on a Connection Machine system CM–200 is not known until run-time. Therefore, optimization of loop partitioning and loop ordering must have a run-time component. In the Local BLAS [45], LBLAS, on the Connection Machine systems, one out of a predetermined set of loop partitionings and loop orderings is selected based on an estimate of the

- number of page faults,
- looping overhead,
- pipeline overhead, and
- time for parameter passing from the front-end to the instruction sequencer.

We illustrate the issue with loop partitioning and loop order for matrix–vector multiplication, as seen in Figure 13.

There are 32 registers in each floating-point unit. A matrix–vector multiplication typically turns into four nested loops. Loop 0 is the innermost loop, which is performed as a vector operation. Iterations in loop 1 can be chained, but the number of vectors being chained is predetermined to 1 or 8. Loops 0 and 1 form a tile that is iterated to cover the whole matrix. The number of vector operations chained together is fixed because of architectural limitations. Limiting the number of different tiles to two is a trade-off.
between the potential performance gain, and the added time (complexity) for dispatch of tiles. With 32 registers the vector length (loop 0) is limited to 22 elements. Two registers are used for temporary variables.

The tile corresponds to the vector operation $u \leftarrow \alpha u + v$, with the accumulation vector $u$ and the coefficient $\alpha$ kept in registers between the iterations of loop 1. For each iteration in loop two, a set (1 or 8) of coefficients $\alpha$ is loaded before a tile is invoked. The vector $v$ is always fetched from memory. The peak efficiency in 32-bit precision is 90% and 92% in 64-bit precision. The vector length (number of iterations in loop 0) has a very strong influence on performance, as seen in Figure 14. For four columns, the performance increases by a factor of about 5.1 when the vector length increases from 2 to 22, while for 500 columns, the increase in performance is approximately a factor of 1.8 for the same range of vector lengths.

With the stride within columns being smaller than the stride within rows, DRAM page faults will first enter into loop 1. For the single-instance case and column major order, or in the multiple-instance case with the problem axes being the least significant, the number of page faults in loop 1 increases with the number of rows (in an instance), until a page fault occurs for every vector chained together in loop 1. The effect of page faults for matrix–vector multiplication is only a few percent for column major order. However, for a row major order, or for vector–matrix multiplication in column major order, loop 0 has the larger stride, and page faults will first enter into the innermost loop. The effect of the page faults is very apparent for $Q = 512$ in Figure 15. The performance losses
CM-200 Performance of DGEMV for Vector-Matrix Multiplication

Figure 15: The aggregate performance for vector–matrix multiplication in 64-bit precision on a 2048 processor Connection Machine system CM-200. The matrix shape is $Q \times R$.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Matrix–vector multiplication</th>
<th>Vector–matrix multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>38%</td>
<td>42%</td>
</tr>
<tr>
<td>64-bit</td>
<td>23%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 7: Possible peak performance loss due to DRAM page faults.

due to page faults are summarized in Table 7. In case there is a large number of page faults, interchanging loop 0 and 1 may be necessary. However, such a change implies that the _AXPY–like_ schedule changes to a _DOT–like_ schedule, and the efficiency of inner product computation becomes critical [45].

For matrix–matrix multiplication, one additional loop level is introduced. The multiplication can be performed as a sequence of either matrix–vector or vector–matrix multiplications. Selecting one or the other loop order for best performance must take the vector length, various overheads, and DRAM page faults into account. Table 8 [45] gives the expressions used in choosing the loop order. $A$ is a $P \times Q$ matrix with strides $A_0$ and $A_1$, $B$ a $Q \times R$ matrix with strides $B_0$ and $B_1$, and $C$ a $P \times R$ matrix with strides $C_0$ and $C_1$. Because of the multiple–instance capability, the strides of the $P$–axis of $A$ and $C$ need not be the same, neither need the strides along the $Q$–axis of $A$ and $B$ be the same, or the strides of the $R$–axis of $A$ and $C$. The number of 32-bit memory words loaded from
and stored to memory is accounted for explicitly by the factor $S$, where $S = 1$ for 32-bit precision, $S = 2$ for 64-bit precision. The sum of the pipeline start-up and shutdown cost is denoted $\tau$. In addition to the entries in Table 8, there is also a one time cost $\sigma$ for the function call. The value of $\tau$ and $\sigma$ amounts to about 6 and 10 cycles, respectively.

For vector-matrix multiplication the roles of $A$ and $B$ are interchanged. More precisely, $A$ has taken the role of $B$ with the stride $A_1$ replacing that of $B_0$; $B$ has taken the role of $A$ with $B_1$ replacing $A_0$ and $B_0$ replacing $A_1$. $C$ is accessed by rows instead of columns, and $C_1$ replaces $C_0$.

The losses due to failure by the above formula to select a nonoptimal loop order is illustrated in Figure 16.

### 13 Summary

In summary, we have shown that

- A multidimensional address space is required for many computations in order to preserve locality of reference in distributed memory architectures. A linearized address space is often not appropriate.

- Data distribution must be treated as a run–time entity, both because the size and shape of arrays may vary during execution, and because of a need to execute a given program on a different number of nodes without recompilation.

- A data distribution that minimizes the surface area for a given local data set size is ideal for several common computations.

- Communication libraries at high level allows user programs to treat the distributed memory as a shared address space without loss of efficiency. Scheduling of communications is an even more complex task than scheduling the execution for a single memory hierarchy.
Figure 16: The percentage performance loss due to nonoptimal choice of loop ordering in matrix–matrix multiplication local to each Connection Machine system CM–200 processor.

- Multiple-instance computation is one of the most important characteristics of libraries for distributed memory architectures.

- For certain computations, the scheduling can be adapted to the data distribution such that load–balance is achieved for both cyclic and consecutive data distribution. For others, the scheduling may be fixed and different data distributions may result in a significant performance difference.

References


47


