LINPACK Benchmark Optimizations on a Virtual Processor Grid

Edward Anderson, Maynard Brandt and Chao Yang, Cray Inc.

ABSTRACT: The "massively parallel" LINPACK benchmark is a familiar measure of high-performance computing performance and a useful check of the scalability of multi-processor computing systems. The CRAY X1 performs well on this benchmark, achieving in excess of 90% of peak performance per processor and near linear scalability. This paper will outline Cray's implementation of the LINPACK benchmark code and show how it leads to computational kernels that are easier to optimize than those of ScaLAPACK or the High Performance LINPACK code, HPL. Specific areas in which the algorithm has been tuned for the CRAY X1 are in communicating across rows or down columns, in interchanging rows to implement partial pivoting, and in multiplying dense matrices.

The Cray LINPACK benchmark code has also been adapted to run on a virtual processor grid, that is, a *p*-by-*q* grid where *p·q* is a multiple of the number of processors. This feature is especially pertinent when the number of processors does not factorize neatly, but it can be used in any situation in which it is desired to assign more processors to row or column operations. The virtual processor grid concept is not specific to the LINPACK benchmark and could be applied to any application that uses a 2-D grid decomposition.

1. Introduction

The LINPACK benchmark attracts a lot of attention from performance analysts because it is the metric used to rank systems for the biannual Top 500 list (www.top500.org). It is also a good predictor of performance for certain applications involving the solution of large dense linear systems or eigenvalue problems. The rules for the “highly parallel” LINPACK benchmark (Table 3 in [5]) allow any problem size and any algorithm to be used as long as it solves a dense 64-bit real linear system, $Ax = b$. In practice, the algorithm is always some form of Gaussian elimination with partial pivoting, and for maximum efficiency, the problem size is usually the largest problem that will fit in memory. An excellent implementation of the LINPACK benchmark for distributed memory systems is High Performance LINPACK (HPL) [6], a C code loosely based on ScaLAPACK [3].

In this paper we describe an alternative implementation of the LINPACK benchmark developed at Cray Inc. that has been used on CRAY T3E and CRAY X1 systems. We will refer to this implementation as the *Cray LINPACK Benchmark code.* The Cray LINPACK Benchmark code is written in a mix of C and Fortran and uses Cray’s one-sided message-passing library SHMEM [4] for communication. Like HPL, it assumes a 2-D block cyclic distribution of the matrix A and solves an augmented system using a block right-looking algorithm. Unlike HPL, it stores the distribution blocks by rows, instead of by columns. This storage order optimizes the memory access pattern of the dominant matrix multiply kernel for the highest levels of the memory hierarchy and preserves the logical structure of the distribution blocks. It is also a natural fit for a generalization of the usual 2-D processor grid into a virtual processor grid, that is, a *p*-by-*q* grid where *p·q* is a multiple of the number of processors.

2. Solving linear systems on distributed memory computers

The computation measured in the LINPACK benchmark is the solution of a dense linear system $Ax = b$, where A is an n-by-n 64-bit real matrix of full rank and b is a vector of length n. The standard technique for this problem uses Gaussian elimination with partial pivoting to compute a factorization $A = LU$, where L is a product of permutation and unit lower triangular matrices and U is upper triangular, from which the solution can be obtained by solving $Ly = b$ for y and $Ux = y$ for x. When there is only one right-hand side vector b, it is advantageous to solve an augmented system, in which b is stored as the $(n+1)^{st}$ column of the array A. Then the factorization applies L^{-1} to b to form y, and the solution can be determined in one additional step by solving the triangular system $Ux = y$ by back substitution.

To solve $Ax = b$ on a distributed memory computer, the matrix A is partitioned among the processors using a block-cyclic distribution on a 2-D processor grid. One possible assignment of blocks to processors on a 2-by-3 processor grid is shown in Figure 1. The tiling pattern repeats as many times as is necessary, and the last row or column of blocks may be empty on some processors. The distribution block size and the aspect ratio of the processor grid are tunable parameters, although choices for the *p*-by-*q* grid are limited by the number of ways to factor the number of processors, N_p.

0	2	4	0	2	4
1	3	5	1	3	5
0	2	4	0	2	4
1	3	5	1	3	5
0	2	4	0	2	4
1	3	5	1	3	5

Figure 1: Block-cyclic distribution on a 2×3 processor grid

The "right-looking" algorithm to compute an *LU* factorization of a matrix *A* stored as in Figure 1 is described as follows. First, a panel of column blocks is factored using Gaussian elimination with partial pivoting. Next, information about the sequence of row interchanges that was used is broadcast to the other processors, and the row interchanges are applied to the rest of the matrix. The current block row is then updated by applying the transformations contained in the diagonal block; this operation consists of a triangular solve with multiple right hand sides and is performed locally by the Level 3 BLAS routine xTRSM. Finally the block column below the diagonal block is communicated across rows and the block row to the right of the diagonal block is communicated down columns, and the rest of the matrix is updated by a rank-*nb* update, performed locally by the Level 3 BLAS routine xGEMM.

Block algorithms were first developed for shared memory computers in order to increase the re-use of data at the highest levels of the memory hierarchy and thereby improve performance. In the numerical linear algebra package LAPACK [1], the block size is an internal parameter, used only for tuning; it does not affect how the 2-D matrix is stored. In ScaLAPACK [3], an extension of LAPACK for distributed-memory computers, the distribution block size is analogous to the algorithm block size in LAPACK, and distributed BLAS were developed analogous to the standard BLAS so that much of the algorithmic structure of LAPACK could be preserved. Since Fortran stores arrays in column-major order, ScaLAPACK stored the distribution blocks column-wise on each processor, making the local portion of the distributed matrix a 2-D matrix as in the shared-memory case.

The High Performance LINPACK code (HPL) [6] is a specialized version of the ScaLAPACK routine to solve a dense real linear system, put into a LINPACK Benchmark context with options for tuning. Like ScaLAPACK, HPL stores the distribution blocks column-wise on each processor, but it provides an option for storing the transpose of the matrix for better cache use on some architectures.

3. Row-wise storage of distribution blocks

While the column-wise storage of local distribution blocks is central to the ScaLAPACK design, there are both notational and performance disadvantages to storing these blocks in a 2-D array. First, the blocking factors used in the distribution are lost in this data structure and must be carried along in an array descriptor, separate from the array *A*. Although a distribution block is a natural piece for the algorithm designer to work with, it exists only as a sub-block of the 2-D array. Elements in the same column of a block of *A* are accessed with unit stride, but elements in the same row are accessed with a stride that grows with the matrix size. For very large problems, such as those solved in the LINPACK benchmark, even a small sub-block may span several memory pages, increasing the latency to load the sub-block into cache. Also, successively larger row strides will each have a different footprint in the cache, making it difficult to predict the performance of the algorithm for larger sizes based on the performance for smaller problems.

In the Cray LINPACK Benchmark code, the distribution blocks are stored row-wise on the local processors. This organization of blocks is most conveniently described by a 4-D array

A(nb, nb, ncblks, nrblks)

where the first two indices describe an *nb*-by-*nb* distribution block and the last two indices describe the local column index and row index of each block, respectively. For example, the first element of the last block stored on processor 5 in Figure 1 would be A(1, 1, 2, 3), since it is the (1,1) element in the second column of blocks and the third row of blocks on processor 5.[1] Assuming the blocks are stored one after the other, this data structure can be regarded as *nrblks* block rows of size *nb*-by-*nb·ncblks* where it is convenient to do so.

[1] In Co-array Fortran notation, we could be even more complete and call the element A(1, 1, 2, 3; 5).

Row-wise storage of blocks maintains the natural contiguity of the distribution blocks and facilitates their movement to the highest levels of the memory hierarchy. Because a block is contiguous in memory, its mapping to an associative cache is known *a priori* and its size can be chosen to fit in a given segment of cache. Also, a block row of local distribution blocks is contiguous, so a rank-*nb* update as in the LINPACK benchmark will result in a unit stride access pattern for the entire block row, as shown in Figure 2. This access pattern can take advantage of stream buffers or hardware prefetch instructions if they are available.

A disadvantage of storing the distribution blocks row-wise is that it requires additional indexing of the local blocks. It also may not be any better than the standard technique unless the library routines have been optimized for it. But it has the potential to be a higher-performing alternative.

4. Optimization of the matrix multiply kernel

In the Cray LINPACK Benchmark code, the main matrix multiply kernel is an *nb*-by-*nb* matrix *A* times an *nb*-by-*n* matrix *B* added to an *nb*-by-*n* matrix *C* to produce an *nb*-by-*n* result. The memory access pattern for this operation is shown in Figure 2. On the CRAY T3E, the innermost kernel was a matrix-vector multiply. The first time the block of *A* was used, it would be loaded from memory, but subsequently it would likely be found in the on-chip secondary cache (Scache) of the CRAY T3E processor. *A* would multiply a column of *B* loaded from memory, and the result vector would be accumulated in registers before being written to *C*. If the leading dimension of the *B* array matched the block size *nb*, then the columns of *B* would be accessed in one continuous stream, activating the stream buffers of the CRAY T3E processor. The matrix *C* was also accessed as one continuous stream. Thus the optimization of the matrix-vector multiply kernel for the CRAY T3E consisted of loading elements of *A* early enough to hide the Scache latency, spacing the operations far enough apart to hide the floating-point unit latency, and prefetching elements of the *B* vector to hide its latency from the stream buffers.

Constraints in the CRAY T3E processor design limited the matrix multiply performance to about 75% of peak. First, there were only 31 scalar floating-point registers, which were needed for accumulating the result vector as well as for temporarily holding elements of *A* and *B*. The latency of approximately 10 clock periods (CP) to Scache and 4 CP for the floating-point unit made instruction scheduling for cached data a challenge. The compiler and most of the scientific library unrolled loops by 4 or 8; for LINPACK, a special-purpose kernel was created that was unrolled by 12. This design meant that the optimal distribution block size *nb* was a multiple of 12. The *nb*-by-*nb* block had to fit in the Scache of the T3E, which was 96 KB or 12 KW, but because of the random replacement policy of the 3-way set associative Scache, it was better to size the block for only one 4096 word set of the Scache. Thus the biggest possible block of *A* was 64-by-64, and coupled with the multiple of 12 requirement and the 8-word Scache line size, the optimal block size for the CRAY T3E was 48. Since the innermost kernel was a 12-by-48 matrix-vector multiply, each vector of *B* was used four times as it was multiplied by each of the four 12-by-48 sub-blocks of *A*. The first time, *B* would be loaded from memory via the stream buffers, but the subsequent uses would hope to find *B* in the Scache. However, if the vector of *B* had the same relative cache offset as part of *A*, the random replacement policy meant that one of them might need to be reloaded from memory.

The CRAY X1 processor design directly addresses many of the limitations in the CRAY T3E memory hierarchy. The CRAY X1 multi-streaming processor (MSP) design is shown in Figure 3. At the highest level of the hierarchy, there are 32 vector registers, each of length 64, on each of the four single-streaming processors (SSPs) of an MSP. Vector registers are used in the CRAY X1 matrix multiply kernel to hold elements of *A* and to accumulate results before writing back to *C*. In fact, there are so many registers available that we can improve the re-use of the *A* and *B* elements and keep all the vector functional units busy by computing four columns at a time, instead of just one. Also, the on-chip cache is 2 MB or 256 KW, allowing the block of *A* to be up to 512-by-512. Finally, the four SSPs on an MSP can perform the computations with the four sub-blocks of *A* concurrently, accessing elements of *B* through their shared cache. The matrix multiply kernel on a CRAY X1 processor is shown in Figure 4. Sustained performance of the matrix multiply kernel in the context of the LINPACK benchmark is approximately 95% of peak.

The matrix-multiply routine xGEMM and the other Level 3 BLAS kernel, xTRSM, are the only parts of the Cray LINPACK benchmark code that are optimized at the SSP level. A prototype for the matrix multiply kernel showing the parallelization across SSPs is shown in Figure 5. This subroutine is compiled with

```
ftn -Oaggress -O3 -s default64 –c sgemmnn.f
```

The SSP code, named DMMNN in this prototype, is implemented in assembly language to achieve the best possible instruction scheduling for elements of *A* that are expected to be found in the cache.

```
      subroutine sgemmnn( m, n, k, alpha, a, inra,
     &                    b, inrb, beta, c, inrc )
      integer m, n, k, inra, inrb, inrc
      real alpha, beta
      real a(inra,*), b(inrb,*), c(inrc,*)
      integer i, js, m4
cdir$ SSP_PRIVATE dmmnn
```

```
      m4 = (m+3)/4
      do i = 0, 3
         js = min( m4, max( m-i*m4, 0 ) )
         call dmmnn( js, n, k, alpha, a(1+i*m4,1),
     &               inra, b, inrb, beta,
     &               c(1+i*m4,1), inrc )
      end do
      return
      end
```

Figure 5: MSP code for matrix multiply, calling SSP subroutine dmmnn

5. Optimization of the row interchanges

The row exchange is a target for optimization because it involves moving data with non-unit stride, all processors participate, and it is all overhead – no floating point computations are performed. The sequence of row interchanges from the panel factorization is described by an index vector IPIV, where IPIV(i) = j indicates that row i was exchanged with row j. The indices are not unique, because a row that is exchanged out of the pivot row block early in the panel factorization may be copied back into the pivot row block in an exchange with a later row.[2] The challenge is to turn the description of a sequential process – the row exchanges of the panel factorization – into a block update that can be applied to the distributed matrix as a whole.

We do this by translating the IPIV vector into index vectors ISCAT and IGATH, where ISCAT contains the destination row indices for the rows of the pivot block row and IGATH contains the indices of rows of the global array that are gathered into the rows of the pivot block row. Row indices in each of these vectors are unique, so the gather or scatter operation can be done as a block.

Since rows of the local array structure are non-unit stride, it is more efficient to copy them to a contiguous buffer before sending them. On processors in the pivot block row, the ISCAT vector is used to copy any rows to be transferred to a remote processor to a send buffer. On the remote processors, the IGATH vector is used to identify local rows involved in the row exchange and copy them to a buffer as well. Each processor participating in the row exchange then synchronizes with the pivot block row, after which each can get their data from the other's buffer. A final synchronization when the transfer is complete is required to indicate that the buffers can be re-used.

In the CRAY LINPACK benchmark code, the local data copy of selected rows to a contiguous buffer is called *transposeAItoB*, and the corresponding routine to copy from columns of the contiguous buffer back to rows of the local array structure is called *transposeBtoAI*. On the CRAY T3E, these operations were optimized using E-registers, like other transpose operations [2]. On the CRAY X1, the vectorizing compiler handles the indexed transpose without any special effort.

6. The virtual processor grid

One shortcoming of mapping a distributed matrix to a 2-D processor grid is the 2-D grid itself. If the number of processors N_p is a perfect square, then any operations on a block column or block row (such as the panel factorization in the LINPACK benchmark) will involve only sqrt(N_p) processors. Moreover, not every processor count factors neatly into $p\cdot q$. For example, early CRAY X1 systems were shipped with 32 four-processor nodes, with 31 nodes configured for application use. But the only factorization of 124 is 4×31 (or 31×4), and 123 = 3×41 and 122 = 2×61 are no better. Rather than leave 3 processors idle and solve the system on 121 processors, we conceived of using a 32×31 virtual processor grid, in which each physical processor would handle more than 1 block in the virtual processor grid.

More generally, one could solve any 2-D problem using N_p processors on a $p\times q$ grid such that $p\times q = k\times N_p$, where $k = 1$, $p = N_p$, and lcm(p, N_p) = $k\times N_p$. The restriction $p = N_p$ is necessary to prevent assigning more than one block in a column of the virtual processor grid to the same processor, and the constraint lcm(p, N_p) = $k\times N_p$ simply guarantees that the virtual processor grid is not a multiple of another smaller virtual processor grid. Using a virtual processor grid is not the same as multithreading, because we create only one process per processor, and the blocks of the virtual processor grid must sometimes be dealt with in a prescribed order, not always from 1 to k.

An example using 6 processors on a 4×3 virtual processor grid is shown in Figure 6. This tiling pattern has two times the parallelism of a 2×3 grid for column operations. The only drawback to the virtual processor grid, other than algorithmic complexity, is a potentially longer synchronization time when one processor must deal with more than one of its blocks at a time. For instance, if column 1 and column 2 of the virtual procesor grid in Figure 6 exchanged data, then processor 0 would have to exchange with 4 using its first block before exchanging with 2 using its second block. However, this effect is lessened on larger virtual grids, such as a 32×31 grid on 124

[2] The most extreme example of row exchanges using partial pivoting is the matrix

$$\begin{vmatrix} 0 & 1 & & & \\ & & \cdots & & \\ & & & \cdots & \\ & & & 0 & 1 \\ 1 & & & & 0 \end{vmatrix}$$

for which IPIV(1) = … = IPIV(N) = N, i.e., every row is exchanged with row N.

processors, where the blocks on the same processor are farther apart.

0	4	2
1	5	3
2	0	4
3	1	5

Figure 6: Six processors in a 4×3 virtual processor grid

In the Cray LINPACK benchmark code, the virtual processor indexing is handled by adding a 5th dimension to the local array structure A, which becomes

A(nb, nb, ncblks, nrblks, nvpi)

For some operations, such as the rank-*nb* update in the LINPACK benchmark code, the order of processing the distribution blocks is not important, so we can update all the blocks with the same virtual processor index together. In other cases, particularly communication operations, each processor must process all its blocks in the virtual processor grid tile before going on to the next tile. Furthermore, the order of processing the blocks in a tile may change as the algorithm progresses, so it is necessary to maintain the current starting point *istart*, and process the block numbered 1 + mod(*istart* + *i* – 1, *nvpi*) in a loop from 1 to *nvpi.*

Figure 7 shows the effect of the virtual processor grid algorithm on 124 processors of a CRAY X1. Both the 4×31 and 31×4 grids show similar performance, while the 32×31 virtual processor grid is up to 10% faster. In the LINPACK benchmark, which is so dominated by matrix multiply that improvements to other parts of the code are usually in the 1% range, this degree of improvement is significant.

7. Summary

In optimizing the LINPACK benchmark for the CRAY X1, we have found new avenues for performance improvements in the code originally developed for a CRAY T3E. The storage order of the distribution blocks is an often overlooked parameter that affects how the data is mapped to the memory hierarchy in the main computational kernels. By parallelizing these kernels across the SSPs of an MSP, we achieved performance of 95% of peak for matrix multiplication and over 90% for the LINPACK benchmark overall. To further reduce inefficiencies in the row or column operations, which involve only a subset of the processors, we introduce the concept of a virtual processor grid. A side benefit of this technique is that one can model the solution of a problem on $k{\cdot}N_p$ processors using only N_p processors if it will fit in memory.

Acknowledgements

The authors wish to acknowledge Jeff Brooks for his insights in the development of the original Cray LINPACK Benchmark code, and Mike Aamodt of the Benchmarking group for his support of the CRAY X1 benchmarking efforts.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third Edition, SIAM, Philadelphia, 1999. (www.netlib.org/lapack)

[2] E. Anderson, J. Brooks, and T. Hewitt, "The Benchmarker's Guide to Single-processor Optimization for CRAY T3E Systems", Cray Research technical report, 1997. (available at ftp://ftp.arsc.edu/pub/mpp/docs/bmguide.ps.Z)

[3] L. S. Blackford et al., *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997. (www.netlib.org/scalapack)

[4] Cray Inc., *Man Page Collection: Shared Memory Access (SHMEM)*, S-2383-23, 2003. (available at www.cray.com/craydoc)

[5] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software", Technical Report CS-89-85, University of Tennessee. (updated version at www.netlib.org/benchmark/performance.ps)

[6] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers", Version 1.0a, Jan. 2004. (available at www.netlib.org/benchmark/hpl/)

About the Authors

Ed Anderson is a consultant to Cray Inc. working with the Benchmarking group. He worked for Cray Research in the Scientific Libraries and Benchmarking groups from 1991—1998. Ed can be reached at eanderson@na-net.ornl.gov. Maynard Brandt retired from Cray Inc. in 2004 after many years in the Benchmarking group. He still lives in the Minneapolis/St. Paul area and can be found at Gopher basketball games. Chao Yang is a Senior Applications Analyst with Cray Inc., specializing in mathematical software development. Chao can be reached at Cray Inc., 1340 Mendota Heights Road, Mendota Heights, MN, 55120, cwy@cray.com.

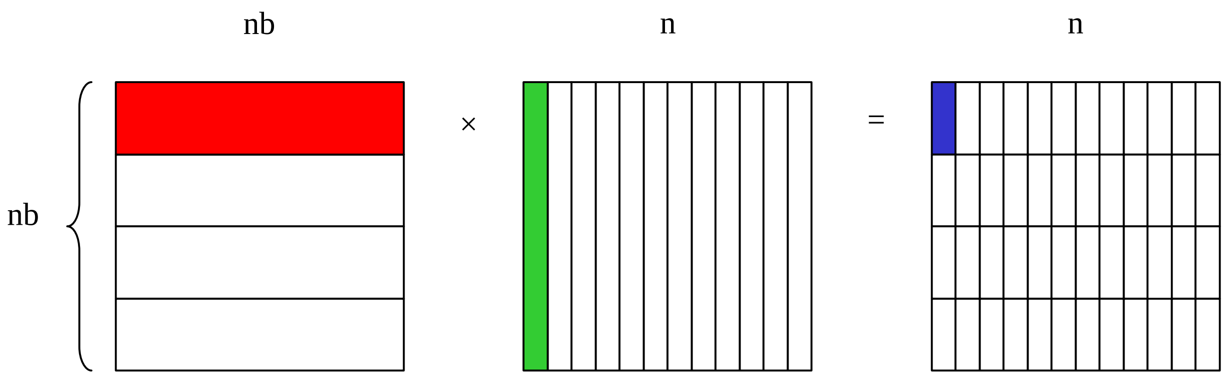

Figure 2: Matrix-multiply C = A*B on the CRAY T3E, highlighting the matrix-vector kernel

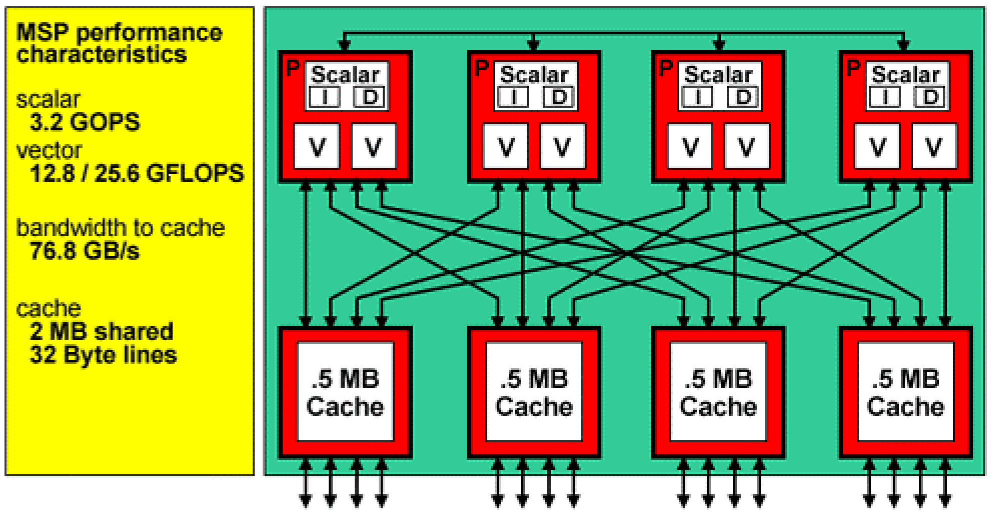

Figure 3: CRAY X1 multi-streaming processor design

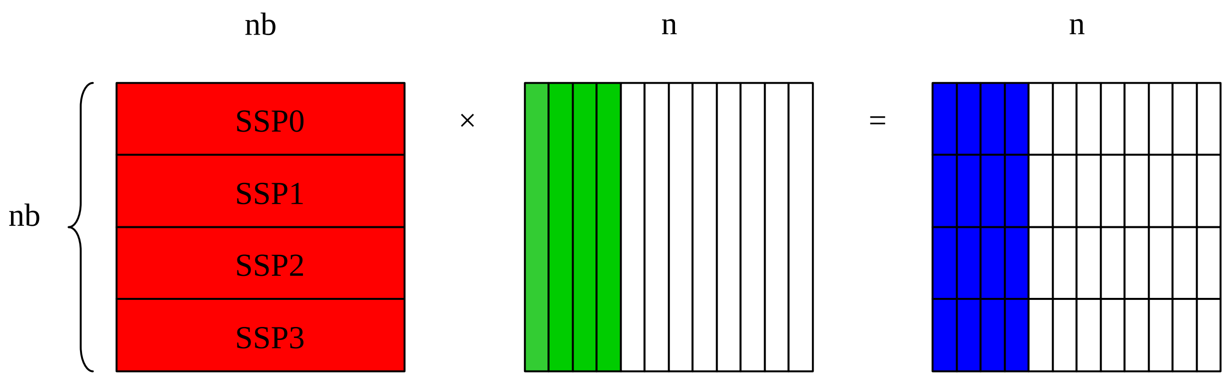

Figure 4: Matrix-multiply C = A*B on the CRAY X1, highlighting the SSP parallelism and computation of four vectors concurrently.

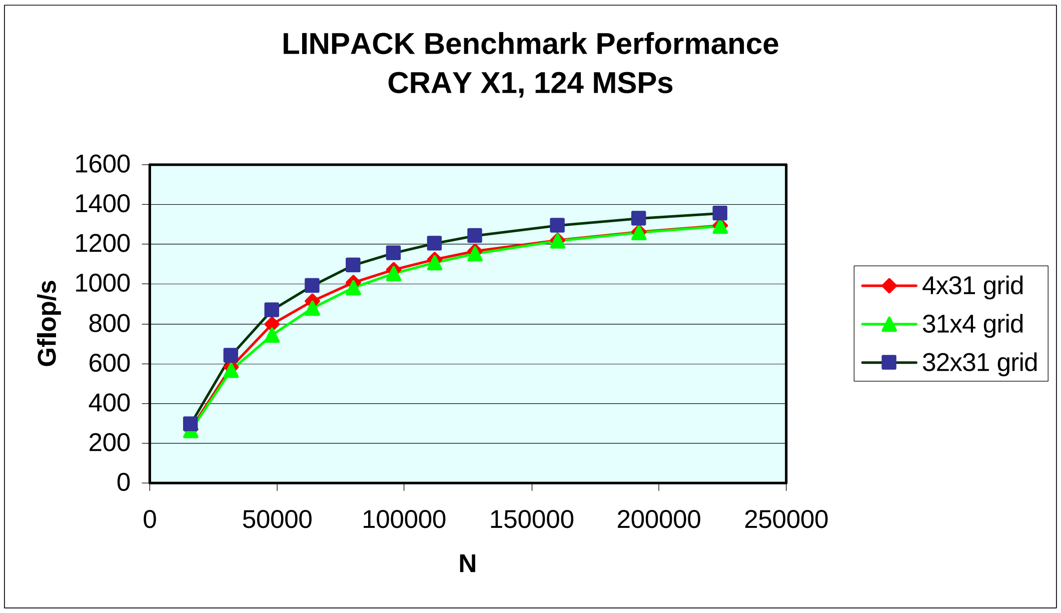

Figure 7: LINPACK Benchmark performance on a virtual processor grid, compared to a conventional processor grid