The LINPACK Benchmark on the Fujitsu AP 1000*

Richard P. Brent†
Computer Sciences Laboratory
Australian National University
Canberra, Australia

Abstract

We describe an implementation of the LINPACK *Benchmark on the Fujitsu AP 1000. Design considerations include communication primitives, data distribution, use of blocking to reduce memory references, and effective use of the cache. Some details of our blocking strategy appear to be new. The results show that the LINPACK Benchmark can be implemented efficiently on the AP 1000, and it is possible to attain about 80 percent of the theoretical peak performance for large problems.*

1 Introduction

This paper describes an implementation of the LINPACK Benchmark [6] on the Fujitsu AP 1000. Preliminary results were reported in [2].

The LINPACK Benchmark involves the solution of a nonsingular system of n linear equations in n unknowns, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. We are interested in large systems since there is no point in using a machine such at the AP 1000 to solve single small systems.

The manner in which matrices are stored is very important on a distributed-memory machine such as the AP 1000. In Section 2 we discuss several matrix storage conventions, including the one adopted for the benchmark programs.

Section 3 is concerned with the algorithm used for parallel LU factorization. Various other aspects of the design of the benchmark programs are mentioned in Section 4. For example, we discuss the use of blocking to improve efficiency, effective use of the cache, and communication primitives.

In Section 5 we present results obtained on AP 1000 machines with up to 512 cells. Some conclusions are given in Section 6.

*Appeared in *Proc. Frontiers '92* (McLean, VA, October 1992), IEEE Press, 1992, 128–135.

†E-mail address: `rpb@cslab.anu.edu.au`

1.1 The Fujitsu AP 1000

The Fujitsu AP 1000 (also known as the "CAP 2") is a MIMD machine with up to 1024 independent 25 Mhz SPARC[1] processors which are called *cells*. Each cell has 16 MByte of dynamic RAM and 128 KByte cache. In some respects the AP 1000 is similar to the CM 5, but the AP 1000 does not have vector units, so the floating-point speed of a cell is constrained by the performance of its Weitek floating-point unit (5.56 Mflop/cell for overlapped multiply and add in 64-bit arithmetic). The topology of the AP 1000 is a torus, with hardware support for wormhole routing [5]. There are three communication networks – the B-net, for communication with the host; the S-net, for synchronization; and the T-net, for communication between cells. The T-net is the most significant for us. It provides a theoretical bandwidth of 25 MByte/sec between cells. In practice, due to overheads such as copying buffers, about 6 MByte/sec is achievable by user programs. For details of the AP 1000 architecture and software environment, see [3, 4].

1.2 Notation

Apart from the optional use of assembler for some linear algebra kernels, our programs are written in the C language. For this reason, we use C indexing conventions, i.e. rows and columns are indexed from 0. Because we often need to take row/column indices modulo some number related to the machine configuration, indexing from 0 is actually more convenient than indexing from 1. In C a matrix with $m \le$ `MMAX` rows and $n \le$ `NMAX` columns may be declared as a[`MMAX`][`NMAX`], where `MMAX` and `NMAX` are constants known at compile-time. The matrix elements are $a[i][j], 0 \le i < m,\ 0 \le j < n$. Rows are stored in contiguous locations, i.e. $a[i][j]$ is adjacent to $a[i][j+1]$. (Fortran conventions are to store columns in contiguous locations, and to start indexing from 1 rather than from 0.)

[1]SPARC^{TM} is a trademark of Sun Microsystems, Inc.

rpb130 typeset using LaTeX

When referring to the AP 1000 configuration as *ncelx* by *ncely*, we mean that the AP 1000 is configured as a torus with *ncelx* cells in the horizontal (X) direction (corresponding to index j above) and *ncely* cells in the vertical (Y) direction (corresponding to index i). Here "torus" implies that the cell connections wrap around, i.e. cell (x, y) (for $0 \leq x < ncelx$, $0 <= y < ncely$) is connected to cell $(x \pm 1 \bmod ncelx, y \pm 1 \bmod ncely)$. Wrap-around is not important for the LINPACK Benchmark – our algorithm would work almost as well on a machine with a rectangular grid without wrap-around.

The total number of cells is $P = ncelx \cdot ncely$. The *aspect ratio* is the ratio $ncely : ncelx$. If the aspect ratio is 1 or 2 we sometimes use $s = ncelx$. In this case $ncely = s$ or $2s$ and $P = s^2$ or $2s^2$.

1.3 Restrictions

Our first implementation was restricted to a square AP 1000 array, i.e. one with aspect ratio 1. Later this was extended in an *ad hoc* manner to aspect ratio 2. This was sufficient for benchmark purposes since all AP 1000 machines have 2^k cells for some integer k, and can be configured with aspect ratio 1 (if k is even) or 2 (if k is odd). For simplicity, the description in Sections 3 and 4 assumes aspect ratio 1.

A minor restriction is that n is a multiple of *ncely*. This is not a serious constraint, since it is easy to add up to $ncely-1$ rows and columns (zero except for unit diagonal entries) to get a slightly larger system with essentially the same solution.

2 Matrix storage conventions

On the AP 1000 each cell has a local memory which is accessible to other cells only via explicit message passing. It is customary to partition data such as matrices and vectors across the local memories of several cells. This is essential for problems which are too large to fit in the memory of one cell, and in any case it is usually desirable for load-balancing reasons. Since vectors are a special case of matrices, we consider the partitioning of a matrix A.

Before considering how to map data onto the AP 1000, it is worth considering what forms of data movement are common in Gaussian elimination with partial pivoting. Most other linear algebra algorithms have similar data movement requirements. To perform Gaussian elimination we need –

> Row/column broadcast. For example, the pivot row has to be sent to processors responsible for other rows, so that they can be modified by the addition of a multiple of the pivot row. The column which defines the multipliers also has to be broadcast.
>
> Row/column send/receive. For example, if pivoting is implemented by explicitly interchanging rows, then at each pivoting step two rows may have to be interchanged.
>
> Row/column scan. Here we want to apply an associative operator θ to data in one (or more) rows or columns. For example, when selecting the pivot row it is necessary to find the index of the element of maximum absolute value in (part of) a column. This may be computed with a scan if we define an associative operator θ as follows:
>
> $$(a, i)\ \theta\ (b, j) = \begin{cases} (a, i), & \text{if } |a| \geq |b|; \\ (b, j), & \text{otherwise.} \end{cases}$$
>
> Other useful associative operators are addition (of scalars or vectors) and concatenation (of vectors).

It is desirable for data to be distributed in a "natural" manner, so that the operations of row/column broadcast/send/receive/scan can be implemented efficiently [1].

A simple mapping of data to cells is the *column-wrapped* representation. Here column i of a matrix is stored in the memory associated with cell $i \bmod P$, assuming that the P cells are numbered $0, 1, \ldots, P-1$.

Although simple, and widely used in parallel implementations of Gaussian elimination [13, 14, 18, 19], the column-wrapped representation has some disadvantages – lack of symmetry, poor load balancing, and poor communication bandwidth for column broadcast – see [1]. Similar comments apply to the analogous *row-wrapped* representation.

Another conceptually simple mapping is the *blocked* representation. Here A is partioned into at most P blocks of contiguous elements, and at most one block is stored in each cell. However, as described in [1], the blocked representation is inconvenient for matrix operations if several matrices are stored with different block dimensions, and it suffers from a load-balancing problem.

Harder to visualize, but often better than the row/column-wrapped or blocked representations, is the *scattered* representation [8], sometimes called *dot mode* [17] or *cut-and-pile*. Assume as above that the cells form an s by s grid. Then the matrix element $a_{i,j}$ is stored in cell $(j \bmod s, i \bmod s)$ with local indices $(\lfloor i/s \rfloor, \lfloor j/s \rfloor)$. The matrices stored locally on each cell have the same shape (e.g. triangular, ...) as the global matrix A, so the computational load on each cell is approximately the same.

The blocked and scattered representations do not require a configuration with aspect ratio 1, although matrix multiplication and transposition are easier to implement if the aspect ratio is 1 than in the general case [20]. In the general case, the scattered representation of A on an *ncelx* by *ncely* configuration stores matrix element $a_{i,j}$ on cell ($j \bmod ncelx, i \bmod ncely$).

Some recent implementations of the LINPACK Benchmark use a *blocked panel-wrapped* representation [12] which may be regarded as a generalization of the blocked and scattered representations described above. An integer blocking factor b is chosen, the matrix A is partitioned into b by b blocks $A_{i,j}$ (with padding if necessary), and then the blocks $A_{i,j}$ are distributed as for our scattered representation, i.e. the block $A_{i,j}$ is stored on cell ($j \bmod ncelx, i \bmod ncely$). If $b = 1$ this is just our scattered representation, but if b is large enough that the "mod" operation has no effect then it reduces to a straightforward blocked representation.

The blocked panel-wrapped representation is difficult to program, but its use may be desirable on a machine with high communication startup times. The AP 1000 has a lower ratio of communication startup time to floating-point multiply/add time than the Intel iPSC/860 or Intel Delta [12], and we have found that it is possible to obtain good performance on the AP 1000 with $b = 1$. Thus, for the benchmark programs we used the scattered representation as described above, with $b = 1$. Our programs use a different form of blocking, described in Section 4.1, in order to obtain good floating-point performance within each cell.

3 The parallel algorithm

Suppose we want to solve a nonsingular n by n linear system

$$Ax = b \tag{3.1}$$

on an AP 1000 with $ncelx = ncely = s$. Assume that the augmented matrix $[A|b]$ is stored using the scattered representation.

It is known [15] that Gaussian elimination is equivalent to triangular factorization. More precisely, Gaussian elimination with partial pivoting produces an upper triangular matrix U and a lower triangular matrix L (with unit diagonal) such that $\mathcal{P}A = LU$, where $\mathcal{P}$ is a permutation matrix. In the usual implementation A is overwritten by L and U (the diagonal of L need not be stored). If the same procedure is applied to the augmented matrix $\bar{A} = [A|b]$, we obtain $\mathcal{P}\bar{A} = L\bar{U}$ where $\bar{U} = [U|\bar{b}]$ and (3.1) has been transformed into the upper triangular system $Ux = \bar{b}$.

In the following we shall only consider the transformation of A to U, as the transformation of $\bar{A}$ to $\bar{U}$ is similar.

If A has n rows, the following steps have to be repeated $n-1$ times, where the k-th iteration completes computation of the k-th row of U –

1. Find the index of the next pivot row by finding an element of maximal absolute value in the current (k-th) column, considering only elements on and below the diagonal. With the scattered representation this involves s cells, each of which have to find a local maximum and then apply an associative operator. The AP 1000 provides hardware and system support for such operations (`y_damax`).

2. Broadcast the pivot row vertically. The AP 1000 provides hardware and system support for such row (or column) broadcasts (`y_brd`, `x_brd`) so they only involve one communication overhead. On machines without such support, row and column broadcasts are usually implemented via message passing from the root to the leaves of s binary trees. This requires $O(\log s)$ communication steps.

3. Exchange the pivot row with the current k-th row, and keep a record of the row permutation. Generally the exchange requires communication between two rows of s cells. Since the pivot row has been broadcast at step 2, only the current k-th row has to be sent at this step. The exchanges can be kept implicit, but this leads to load-balancing problems and difficulties in implementing block updates, so explicit exchanges are usually preferable.

4. Compute the "multipliers" (elements of L) from the k-th column and broadcast them horizontally.

5. Perform Gaussian elimination (using the portion of the pivot row and the other rows held in each cell). If done in the obvious way, this involves saxpy[2] operations, but the computation can also be formulated as a rank-1 update by combining several saxpys. In Section 4.1 we describe how most of the operations may be performed using matrix-matrix multiplication, which is faster.

We can make an estimate of the parallel time T_P required to perform the transformation of A to upper triangular form. There are two main contributions –

[2] A *saxpy* is the addition of a scalar multiple of one vector to another vector.

A. *Floating-point arithmetic.* The overall computation involves $2n^3/3 + O(n^2)$ floating-point operations (counting additions and multiplications separately). Because of the scattered representation each of the $P = s^2$ cells performs approximately the same amount of arithmetic. Thus floating-point arithmetic contributes a term $O(n^3/s^2)$ to the computation time.

B. *Communication.* At each iteration of steps 1-5 above, a given cell sends or receives $O(n/s)$ words. We shall assume that the time required to send or receive a message of w words is $c_0 + c_1 w$, where c_0 is a "startup" time and $1/c_1$ is the transfer rate.[3] With this assumption, the overall communication time is $O(n^2/s) + O(n)$, where the $O(n)$ term is due to startup costs.

On the AP 1000 it is difficult to effectively overlap arithmetic and communication when the "synchronous" message-passing routines are used for communication (see Section 4.4). Ignoring any such overlap, the overall time T_P may be approximated by

$$T_P \simeq \alpha n^3/s^2 + \beta n^2/s + \gamma n, \qquad (3.2)$$

where α depends on the floating-point and memory speed of each cell, β depends mainly on the communication transfer rate between cells, and γ depends mainly on the communication startup time. We would expect the time on a single cell to be $T_1 \simeq \alpha n^3$, although this may be inaccurate for various reasons – e.g. the problem may fit in memory caches on a parallel machine, but not on a single cell.

3.1 Solution of triangular systems

Due to lack of space we omit details of the "back-substitution" phase, i.e. the solution of the upper triangular system $Ux = \bar{b}$. This can be performed in time much less than (3.2): see [8, 18, 20]. For example, with $n = 1000$ on a 64-cell AP 1000, back-substitution takes less than 0.1 sec but the LU factorization takes about 3.4 sec. Implementation of back-substitution is relatively straightforward. The main differences between the back-substitution phase and the LU factorization phase are –

1. No pivoting is required.

2. The high-order term $\alpha n^3/s^2$ in (3.2) is reduced to $O(n^2/s^2)$, so the low-order terms become relatively more important.

[3] The time may depend on other factors, such as the distance between the sender and the receiver and the overall load on the communication network, but our approximation is reasonable on the AP 1000.

4 Design considerations

In this section we consider various factors which influenced the design of the LINPACK Benchmark programs.

4.1 The need for blocking within cells

On many serial machines, including the AP 1000 cells, it is impossible to achieve peak performance if the Gaussian elimination is performed via saxpys or rank-1 updates. This is because performance is limited by memory accesses rather than by floating-point arithmetic, and saxpys or rank-1 updates have a high ratio of memory references to floating-point operations. Closer to peak performance can be obtained for matrix-vector or (better) matrix-matrix multiplication [7, 10, 11, 20].

It is possible to reformulate Gaussian elimination so that most of the floating-point arithmetic is performed in matrix-matrix multiplications, without compromising the error analysis. Partial pivoting introduces some difficulties, but they are surmountable. There are several possibilities – the one described here is a variant of the "right-looking" scheme of [7]. The idea is to introduce a "blocksize" or "bandwidth" parameter ω. Gaussian elimination is performed via saxpys or rank-1 updates in vertical strips of width ω. Once ω pivots have been chosen, a horizontal strip of height ω containing these pivots can be updated. At this point, a matrix-matrix multiplication can be used to update the lower right corner of A.

The optimal choice of ω may be determined by experiment, but $\omega \simeq n^{1/2}$ is a reasonable choice. For simplicity we use a fixed ω throughout the LU factorization (except at the last step, if n is not a multiple of ω). It is convenient to choose ω to be a multiple of s where possible.

We originally planned to use a distributed matrix-matrix multiply-add routine for the block update step. Such routines have been written as part of the CAP BLAS project [20]. However, this approach involves unnecessary communication costs. Instead, we take advantage of each AP 1000 cell's relatively large memory (16 MByte) and save the relevant part of each pivot row and multiplier column as it is broadcast during the horizontal and vertical strip updates. The block update step can then be performed independently in each cell, *without any further communication.* This idea may be new – we have not seen it described in the literature. On the AP 1000 it improves performance significantly by reducing communication costs. Each cell requires working storage of about $2\omega n/s$ floating-point words, in addition to the

$(n^2 + O(n))/s^2$ words required for the cell's share of the augmented matrix and the triangular factors.

To avoid any possible confusion, we emphasise two points –

1. Blocking as described above does not change the error bound for Gaussian elimination. Arithmetically, the only change is that certain inner products may be summed in a different order. Thus, blocking does not violate an important rule of the LINPACK Benchmark: that the algorithm used must be no less stable numerically than Gaussian elimination with partial pivoting. (This rule precludes the use of Gaussian elimination without pivoting.)

2. Blocking with parameter ω in the LU factorization is independent of blocking with parameter b in the storage representation. Van de Geijn [12] appears to use $\omega = b > 1$, but we use $\omega > b = 1$. There is a tradeoff here: with $\omega = b > 1$ the number of communication steps is less but the load balance is worse than with our choice of $\omega > b = 1$.

The effect of blocking with $\omega \simeq n^{1/2}$ and $b = 1$ is to reduce the constant α in (3.2) at the expense of introducing another term $O(n^{5/2}/s^2)$ and changing the lower-order terms. Thus, a blocked implementation should be faster for sufficiently large n, but may be slower than an unblocked implementation for small n. This is what we observed – with our implementation the crossover occurs at $n \simeq 40s$.

4.2 Kernels for single cells

As described in [20], it is not possible to obtain peak performance when writing in C, because of deficiencies in the C compiler which is currently used (the standard Sun release 4.1 cc compiler). A simple-minded C compiler, which took notice of *register* declarations and did not attempt to reorder loads, stores, and floating-point operations, would perform better than our current optimizing compiler on carefully written C code.

We wrote some simple loops (essentially a subset of the level 1 and level 2 BLAS) in assembly language in order to obtain better performance. In all cases an equivalent C routine exists and can be used by changing a single constant definition.

The key to obtaining high performance in simple floating-point computations such as inner and outer products is –

1. Write "leaf" routines to minimise procedure call overheads [21].

2. Keep all scalar variables in registers. The routines are sufficiently simple that this is possible.

3. Separate loads, floating multiplies, floating adds, and stores which depend on each other by a sufficiently large number of instructions that operands are always available when needed. This process can be combined with loop unrolling.

Even when written in assembler, the kernels for saxpys and rank-1 updates are significantly slower than the kernels for matrix-vector products, and these are slightly slower than the kernels for matrix-matrix multiplication. The important parameter here is the number R of loads or stores from/to memory per floating-point operation [7, 10]. A saxpy has $R \geq 1.5$, since there are two loads and one store for each floating-point multiply and add; inner products and rank-1 updates have $R > 1$; matrix-vector multiplication has $R > 0.5$. As described in [20], matrix-matrix multiplication can be implemented with $R << 1$. A lower bound on R is imposed by the number of floating-point registers (32 in single-precision, 16 in double-precision). Our current implementation has $R = 0.25$ (single-precision) and $R = 0.375$ (double-precision).

4.3 Effective use of the cache

Each AP 1000 cell has a 128 KByte direct-mapped, copy-back cache, which is used for both instructions and data [4]. Performance can be severely degraded unless care is taken to keep the cache-miss ratio low. In the LU factorization it is important that each cell's share of the vertical and horizontal strips of width ω should fit in the cell's cache. This implies that ω should be chosen so that $\omega n/s^2$ is no greater than the cache size. If $\omega n/s$ exceeds the cache-size then care has to be taken when the block update is performed in each cell, but this is handled by the single-cell BLAS routines, so no special care has to be taken in the LU factorization routine.

4.4 Communication primitives

The AP 1000 operating system CellOS provides both "synchronous" (e.g. `xy_send`, `x_brd`) and "asynchronous" (e.g. `r_rsend`) communication routines. In both cases the send is non-blocking, but for the synchronous routines the receive is blocking. Because of the manner in which the communication routines are currently implemented [4], both synchronous and asynchronous routines give about the same transfer rate (about 6 MByte/sec for user programs), but the synchronous routines have a much lower startup overhead (about 15 μsec) than the asynchronous routines.

4.5 Context switching and virtual cells

Our description has focussed on the case of aspect ratio 1. This is the simplest case to understand and is close to optimal because of the balance between communication bandwidth in the horizontal and vertical directions. Due to space limitations, we refer to [2] for a discussion of design considerations for aspect ratio greater than 1, including the use of "virtual cells" to simulate a virtual machine with aspect ratio 1.

5 Results on the AP 1000

In this section we present the LINPACK Benchmark results. As described above, the benchmark programs are written in C, with some assembler for the single-cell linear algebra kernels. The programs implement Gaussian elimination with partial pivoting and check the size of the residual on the cells, so the amount of data to be communicated between the host and the cells is very small. All results given here are for double-precision.[4]

The AP 1000 cells run at 25Mhz and the time for overlapped independent floating-point multiply and add operations is 9 cycles, giving a theoretical peak speed of 5.56 Mflop per cell when performing computations with an equal number of multiplies and adds.[5]

The results in Table 1 are for $n = 1000$ and should be compared with those in Table 2 of [6]. The results in Table 2 are for n almost as large as possible (constrained by the storage of 16 MByte/cell), and should be compared with those in Table 3 of [6]. In Table 2 –

> n_{max} is the problem size giving the best performance r_{max},
>
> n_{half} is the problem size giving performance $r_{max}/2$, and
>
> r_{peak} is the theoretical peak performance, ignoring everything but the speed of the floating-point units. r_{peak} is sometimes called the "guaranteed not to exceed" speed.

The results for the AP 1000 are good when compared with reported results [6, 12] for other distributed memory MIMD machines such as the nCUBE 2, Intel iPSC/860, and Intel Delta, if allowance is made for the different theoretical peak speeds. For example –

> The 1024-cell nCUBE 2 achieves 2.59 sec for $n = 1000$ and 1.91 Gflop for $n = 21376$ with $r_{peak} = 2.4$ Gflop. Our results indicate that a P-cell AP 1000 is consistently faster than a $2P$-cell nCUBE 2.
>
> The 512-cell Intel Delta achieves 1.5 sec for $n = 1000$ (an efficiency of 3 percent). It achieves 13.9 Gflop for $n = 25000$ but this is less than 70 percent of its theoretical peak of 20 Gflop.[6]
>
> The 128-cell Intel iPSC/860 achieves 2.6 Gflop, slightly more than the 512-cell AP 1000, but this is only 52 percent of its theoretical peak of 5 Gflop.[7]

For large n the AP 1000 consistently achieves in the range 79 to 82 percent of its theoretical peak, with the figure slightly better when the aspect ratio is 1 than when it is not.

An encouraging aspect of the results is that the AP 1000 has relatively low n_{half}. For example, on a 64-cell AP 1000 at ANU we obtained at least half the best performance for problem sizes in the wide range $648 \le n \le 10000$.[7] As expected from (3.2), n_{half} is roughly proportional to $P^{1/2}$.

Table 3 gives the speed (as a percentage of the theoretical peak speed of 356 Mflop) of the LINPACK Benchmark program on a 64-cell AP 1000, for various problem sizes n, with and without blocking, and with four different compilation options –

1. The fastest version, with assembler kernels for some single-cell BLAS.
2. As for 1, but with carefully written C code replacing assembler kernels.
3. As for 2, but using "simple" C code without any attempt to unroll loops or remove dependencies between adjacent multiplies and adds.
4. As for 1, but not using the system routines `x_brd`, `y_brd`, `y_damax`, `x_dsum` etc. Instead, these routines were simulated using binary tree algorithms and the basic communication routines `xy_send`, `xy_recvs`.

Table 3 shows that coding the single-cell BLAS in assembler is worthwhile, at least if the alternative is use of the current C compiler. The "simple" C code (option 3) is significantly slower than the more sophisticated code (option 2) for saxpys and rank-1 updates, but not for matrix multiplication. This is because the

[4] Single-precision is about 50 percent faster.

[5] The peak double-precision speed is 6.25 Mflop per cell if two adds are performed for every multiply.

[6] Assuming 40 Gflop per cell, although 60 Gflop per cell is sometimes quoted as the peak.

[7] On the 64-cell Intel Delta, the corresponding range is $2500 \le n \le 8000$.

Time for one cell	cells	time (sec)	speedup	efficiency
160	512	1.10	147	0.29
160	256	1.50	108	0.42
160	128	2.42	66.5	0.52
160	64	3.51	46.0	0.72
160	32	6.71	24.0	0.75
160	16	11.5	13.9	0.87
160	8	22.6	7.12	0.89
160	4	41.3	3.90	0.97
160	2	81.4	1.98	0.99

Table 1: LINPACK Benchmark results for $n = 1000$

cells	r_{max} Gflop	n_{max} order	n_{half} order	r_{peak} Gflop	r_{max}/r_{peak}
512	2.251	25600	2500	2.844	0.79
256	1.162	18000	1600	1.422	0.82
128	0.566	12800	1100	0.711	0.80
64	0.291	10000	648	0.356	0.82
32	0.143	7000	520	0.178	0.80
16	0.073	5000	320	0.089	0.82

Table 2: LINPACK Benchmark results for large n

Problem	With blocking				Without blocking			
size n	1	2	3	4	1	2	3	4
200	15	13	12	5	14	13	11	5
400	28	24	24	13	27	24	19	14
1000	53	40	39	35	44	37	26	32
2000	64	47	46	46	35	30	24	28
4000	74	52	51	61	37	31	25	33
8000	80	55	55	73	38	32	25	36
10000	82	56	55	75	37	32	25	35

Table 3: Speeds (% of peak) on 64-cell AP 1000

C compiler fails to use floating-point registers effectively in the matrix multiplication routine, resulting in unnecessary loads and stores in the inner loop.

Option 4 (avoiding `x_brd` etc) shows how the AP 1000 would perform without hardware/CellOS support for `x_brd`, `y_brd`, `y_damax` etc. Performance would be significantly lower, especially for small n.

As n increases the performance without blocking first increases (as expected) but then decreases or fluctuates. This is because of the effect of finite cache size. The rank-1 updates run slowly if the result does not fit in the cache. This effect is not noticeable if blocking is used, because our choice of ω and our implementation of single-cell matrix multiplication take the cache size into account (see Section 4.3).

Because of the influence of the cache and the effect of blocking, the formula (3.2) gives a good fit to the benchmark results only if n is sufficiently small and either ω is fixed or blocking is not used. Nevertheless, it is interesting to estimate the constant γ due to communication startup costs (for both LU factorization and backsubstitution). For aspect ratio 1 there are approximately $8n$ communication startups, each taking about 15 μsec, so $\gamma \simeq 120$ μsec.[8] Thus, about 0.12 sec of the time in the third column of Table 1 is due to startup costs. Use of a blocked panel-wrapped storage representation would at best decrease these startup costs, so it could not give much of an improvement over our results.

6 Conclusion

The LINPACK Benchmark results show that the AP 1000 is a good machine for numerical linear algebra, and that we can consistently achieve close to 80 percent of its theoretical peak performance on moderate to large problems. The main reason for this is the high ratio of communication speed to floating-point speed compared to machines such as the Intel Delta and nCUBE 2. The high-bandwidth hardware row/column broadcast capability of the T-net (`x_brd`, `y_brd`) and the low latency of the synchronous communication routines are significant.

Acknowledgements

Support by Fujitsu Laboratories, Fujitsu Limited, and Fujitsu Australia Limited via the Fujitsu-ANU CAP Project is gratefully acknowledged. Thanks are due to Takeshi Horie and his colleagues at Fujitsu Laboratories for assistance in running the benchmark programs on a 512-cell AP 1000, to Peter Strazdins for

[8]For aspect ratio 2, γ is about 50 percent larger.

many helpful discussions, to Peter Price for his assistance in optimizing the assembler routines, and to the referees for their assistance in improving the exposition.

References

[1] R. P. Brent, "Parallel algorithms in linear algebra", *Proceedings Second NEC Research Symposium* (held at Tsukuba, Japan, August 1991), to appear. Available as Report TR-CS-91-06, Computer Sciences Laboratory, ANU, August 1991.

[2] R. P. Brent, "The LINPACK Benchmark on the AP 1000: Preliminary Report", in [3], G1-G13.

[3] R. P. Brent (editor), *Proceedings of the CAP Workshop '91*, Australian National University, Canberra, Australia, November 1991.

[4] R. P. Brent and M. Ishii (editors), *Proceedings of the First CAP Workshop*, Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

[5] W. J. Dally and C. L. Seitz, "Deadlock free message routing in multiprocessor interconnection networks", *IEEE Trans. on Computers* C-36 (1987), 547-553.

[6] J. J. Dongarra, "Performance of various computers using standard linear equations software", Report CS-89-05, Computer Science Department, University of Tennessee, version of June 2, 1992 (available from `netlib@ornl.gov`).

[7] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1990.

[8] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 1*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[9] K. Gallivan, W. Jalby, A. Malony and H. Wijshoff, "Performance prediction for parallel numerical algorithms", *Int. J. High Speed Computing* 3 (1991), 31-62.

[10] K. Gallivan, W. Jalby and U. Meier, "The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory", *SIAM J. Sci. and Statist. Computing* 8 (1987), 1079-1084.

[11] K. A. Gallivan, R. J. Plemmons and A. H. Sameh, "Parallel algorithms for dense linear algebra computations", *SIAM Review* 32 (1990), 54-135.

[12] R. A. van de Geijn, "Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems", Report TR-91-92, Department of Computer Sciences, University of Texas, August 1991.

[13] G. A. Geist and M. Heath, "Matrix factorization on a hypercube", in [16], 161-180.

[14] G. A. Geist and C. H. Romine, "LU factorization algorithms on distributed-memory multiprocessor architectures", *SIAM J. Sci. and Statist. Computing* 9 (1988), 639-649.

[15] G. H. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, Maryland, 1983.

[16] M. T. Heath (editor), *Hypercube Multiprocessors 1986*, SIAM, Philadelphia, 1986.

[17] M. Ishii, G. Goto and Y. Hatano, "Cellular array processor CAP and its application to computer graphics", *Fujitsu Sci. Tech. J.* 23 (1987), 379-390.

[18] G. Li and T. F. Coleman, "A new method for solving triangular systems on distributed-memory message-passing multiprocessors", *SIAM J. Sci. and Statist. Computing* 10 (1989), 382-396.

[19] C. Moler, "Matrix computations on distributed memory multiprocessors", in [16], 181-195.

[20] P. E. Strazdins and R. P. Brent, "The Implementation BLAS Level 3 on the AP 1000: Preliminary Report", in [3], H1-H17.

[21] Sun Microsystems, *Sun-4 Assembly Language Reference Manual*, May 1988.