The LINPACK Benchmark on the AP 1000: Preliminary Report*

Richard P. Brent†
Computer Sciences Laboratory
Australian National University

Abstract

We describe a preliminary implementation of the *LINPACK Benchmark* on the Fujitsu AP 1000. Design considerations include communication primitives, data distribution, use of blocking to reduce memory references, effective use of the cache, and use of virtual cells on non-square AP 1000 configurations. Preliminary results show that the LINPACK Benchmark can be implemented efficiently on the AP 1000, and it is possible to attain about 80% of the theoretical peak performance for large problems. Planned improvements and extensions are mentioned.

1 Introduction

This paper describes an implementation of the LINPACK Benchmark [7] on the Fujitsu AP 1000. The report given here is preliminary in the sense that the implementation is currently being refined and generalised.

The LINPACK Benchmark involves the solution of a nonsingular system of n linear equations in n unknown, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. Three cases are considered:

1. $n = 100$ – the original benchmark.
2. $n = 1000$ – gives more opportunity for vector pipeline machines (and to some extent parallel machines) to exhibit high performance.
3. n as large as desired – gives maximum opportunity for vector pipeline and parallel machines to exhibit high performance.

We are only concerned with the cases 2 and 3 here, since case 1 is trivial to solve on a machine as powerful as the AP 1000.

For historical reasons and brevity we shall refer to the Fujitsu AP 1000 computer as the CAP. The CAP is a MIMD machine with up to 1024 independent processors which are called *cells*. For details of the CAP architecture and software environment, see [4].

The manner in which matrices are distributed across the cells is very important on a distributed-memory message-passing machine such as the CAP. In Section 2 we discuss several matrix storage conventions including the one adopted for the benchmark program.

Section 3 is concerned with the algorithms used for parallel triangular factorisation and solution of the resulting triangular systems, Various other aspects of the design of the benchmark program are mentioned in Section 4. For example, we discuss the use of blocking to improve efficiency, effective use of the cache, and communication primitives.

*Appeared in *Proceedings of the Second Fujitsu-ANU CAP Workshop* (edited by R. P. Brent), Department of Computer Science, ANU, November 1991, G1–G13. Copyright © 1991, ANU and Fujitsu Laboratories Ltd.

†E-mail address: rpb@cslab.anu.edu.au rpb130pr typeset using LaTeX

In Section 5 we present preliminary results obtained on the 64-cell CAP at ANU and a 512-cell CAP at Fujitsu Research Laboratories, Kawasaki. Some conclusions and plans for future software development on the CAP are given in Section 6.

1.1 Conventions

Apart from the optional use of some assembler for linear algebra kernels, our programs are written in the C language. For this reason, we shall 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 rows and n columns is declared as $a[m][n]$ and has elements $a[i][j], 0 \le i < m$, $0 \le j < n$. 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.)

When referring to the CAP configuration as $ncelx$ by $ncely$, we mean that the CAP 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 $0 \le x < ncelx$, $0 <= y < ncely$) is connected to cell $(x \pm 1 \bmod ncelx, y \pm 1 \bmod ncely)$. Wrap-around is not important for the LINPACK benchmark – our algorithms would work almost as well on a machine with a rectangular grid without wrap-around.

The total number of cells is $P = ncelx \cdot ncely$. Occasionally P denotes a permutation matrix instead of the number of cells, but no confusion should arise. 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 $2s^2$.

1.2 Restrictions

Our first implementation was restricted to a square CAP array, i.e. one with aspect ratio 1. Later this was extended in an *ad hoc* manner to 2:1 aspect ratio. This was sufficient for benchmark purposes since all CAP 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). Conversion of the benchmark program to general aspect ratios is in progress, but incomplete at the time of writing this report.

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 CAP 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. (The exception is for very small problems which may as well be solved in a single cell.) Since vectors are a special case of matrices, we consider the partitioning of an m by n matrix A.

Before considering how to map data onto the CAP, 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 have to be interchanged. (This could be done by broadcasting both rows, but it might be more efficient to explicitly send a row to the appropriate destination if it does not need to be broadcast.)

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 via an associative operator θ defined on pairs:

$$(a,i)\ \theta\ (b,j) = \begin{cases} (a,i), & \text{if } |a| \geq |b|; \\ (b,j), & \text{otherwise.} \end{cases}$$

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 described above can be implemented efficiently. This is possible if a rectangular grid with aspect ratio close to 1 is a subgraph of the connection graph of the parallel machine.

A simple mapping of data to cells is the *column-wrapped* (or *row-wrapped*) representation. Here column (or row) 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 [12, 13, 19, 20], the column-wrapped (or row-wrapped) representation has some disadvantages –

Lack of symmetry – rows are treated differently from columns. It is instructive to consider the data communication involved in transposing a matrix.

Poor load-balancing for moderate-sized problems – if $n < P$ some cells store no columns, so presumably perform no useful work. On the other hand, if n increases from P to $P+1$ then the load on cell 0 doubles. Thus, the performance curve as a function of n (the number of columns in the matrix) will be "jagged" – there will be jumps at each multiple of P.

Poor communication bandwidth for column broadcast – since each column is stored in the memory associated with only one cell, the speed of column broadcast is constrained by the communication bandwidth of a single cell (compare the blocked/scattered storage representations below).

Another conceptually simple mapping is the *blocked* representation. Assume for the time being that the aspect ratio is 1, so the cells form an s by s grid ($P = s^2$). The matrix A is padded with zero rows and columns if necessary, so that m and n can be assumed to be multiples of s. A is partitioned into an s by s matrix of blocks. Each block, of dimension m/s by n/s, is assigned to one cell in the natural way. This avoids the lack of symmetry inherent in the row/column-wrapped representation. It also improves the communication bandwidth for column broadcast, because each column is shared by $s = P^{1/2}$ cells. However, it is inconvenient for matrix operations if several matrices are stored with different block dimensions. The blocked representation also suffers from a load-balancing problem – if A is upper triangular then about

half the cells (those storing the strict lower triangle of blocks) are only storing zeros and can probably not do any useful computation.

Harder to visualize, but often better than the row/column-wrapped or blocked representations, is the *scattered* representation [10] (also called *dot mode* [18] 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 $(j \bmod s, i \bmod s)$. Now the matrices stored locally on each cell have the same shape (e.g. triangular, band, ...) as the global matrix A, so the computational load on each cell is approximately equal.

It is sometimes useful to regard a blocked representation of a matrix as a scattered representation of the same matrix with its rows and columns permuted [3].

The blocked and scattered representations do not require a configuration with aspect ratio 1. In the general case, the scattered representation of A on an $ncelx$ by $ncely$ configuration stores matrix element $a_{i,j}$ on cell $(j \bmod ncelx, i \bmod ncely)$. Matrix multiplication and transposition are easier to implement if the aspect ratio is 1 than in the general case [23].

Some recent implementations of the LINPACK benchmark use a *blocked panel-wrapped* representation [11] which may be regarded as a generalisation 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 in 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 our blocked representation.

Because of its generality, the blocked panel-wrapped representation makes programming complicated. Its use may be desirable on a machine with low communication speed (relative to computation speed) or with high communication startup times. The CAP has a higher ratio of communication speed to computation speed than the Intel iPSC/860 or Intel DELTA [11], and we have found that it is possible to obtain good performance on the CAP with $b = 1$. Thus, for the benchmark programs we used the scattered representation as described above, with $b = 1$. (Our programs do use a different form of blocking, which is described in Section 4.1.)

3 The Parallel Algorithm

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

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

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

It is known [16, 22, 25] 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

$$PA = LU \tag{3.2}$$

where P is a permutation matrix (not the number of cells here). 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

$$P\bar{A} = L\bar{U} \tag{3.3}$$

where $\bar{U} = [U|\bar{b}]$ and (3.1) has been transformed into the upper triangular system

$$Ux = \bar{b} \tag{3.4}$$

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

2. Broadcast the pivot row vertically.

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. Alternatively, the exchanges could be kept implicit, but this would lead 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 operations (a *saxpy* is the addition of a scalar multiple of one vector to another vector), but the computation can also be formulated as a rank-1 update by combining several saxpys.

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 –

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. (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 CAP.) With our assumption, the overall communication time is $O(n^2/s) + O(n)$, where the $O(n)$ term is due to startup costs.

On the CAP 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 is simply the sum of A and B above, i.e.

$$T_P \simeq \alpha n^3/s^2 + \beta n^2/s + \gamma n, \tag{3.5}$$

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

$$T_1 \simeq \alpha n^3, \tag{3.6}$$

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.

From (3.5) and (3.6), the efficiency E_P is

$$E_P \simeq \frac{1}{1 + (1 + \bar{\gamma}/\bar{n})\bar{\beta}/\bar{n}}, \tag{3.7}$$

where $\bar{\beta} = \beta/\alpha$ is proportional to the ratio of communication to computation speed, $\bar{\gamma} = \gamma/\beta$ measures the importance of the communication startup time, and $\bar{n} = n/s$ is the number of rows or columns of A stored in a single cell. From (3.7), the efficiency is close to 1 only if $\bar{n} \gg \bar{\beta}$.

If $\bar{\gamma}$ is negligible, we can write (3.7) in the form

$$E_P \simeq \frac{1}{1 + n_{half}/n}, \tag{3.8}$$

where $n_{half} = \bar{\beta}s$ is the problem size giving efficiency 0.5.

3.1 Solution of triangular systems

In this preliminary report we omit details of the "back-substitution" phase, i.e. the solution of the upper triangular system (3.4), because this can be performed in time much less than (3.5) (see [10, 19]). For example, with $n = 1000$ on a 64-cell CAP, back-substitution takes less than 0.1 sec but the LU factorisation takes about 3.5 sec. Implementation of back-substitution is relatively straightforward and does not involve the solution of any new problems. The main differences between the back-substitution phase and the LU factorisation phase are that

1. No pivoting is required.
2. The high-order term in (3.5) is reduced to $O(n^2/s^2)$, so the low-order terms become relatively more important.

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

On many machines, including the CAP 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.

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. 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 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 is best determined by experiment, but

$$\omega \simeq n^{1/2}$$

is a reasonable choice. For simplicity we use a fixed ω throughout the LU factorisation (except at the last step, if n is not a multiple of ω). It is also convenient (though not essential) to choose ω to be a multiple of s. Slightly improved performance might be obtained by varying ω as the LU factorisation proceeds.

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 BLAS project [23]. However, this approach is not the best one to use on the CAP. Instead, we take advantage of each CAP 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. 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 factorisation is independent of blocking with parameter b in the storage representation. Van de Geijn [11] appears to use $\omega = b > 1$, but we use $\omega > b = 1$.

The effect of blocking is to reduce the constant α in (3.5) 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 the companion paper [23], it is not possible to obtain peak performance when writing in C, because of deficiencies in the C compiler which is currently used. A simple-minded C compiler, which took notice of *register* declarations and did not attempt to reorder loads, stores, and floating-point operations, would actually perform better than our current "optimising" compiler.

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 [24].

2. Keep all scalar variables in registers. (If this is not possible, simplify the specification of the routine.)

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 ratio R of loads or stores from/to memory to floating-point operations. 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 [23], matrix-matrix multiplication can be implemented with $R << 1$. Our current implementation has $R = 0.25$ (single-precision) and $R = 0.375$ (double-precision). The difference between single and double-precision is because we have 32 single-precision floating-point registers, but only 16 double-precision registers.

4.3 Effective use of the cache

Each CAP cell has a 128 KByte direct-mapped, copy-back cache [4], and performance can be degraded for large problems unless care is taken to keep the cache-miss ratio low. In the LU factorisation 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 factorisation routine.

4.4 Communication primitives

The CAP 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. Thus, we always use synchronous routines for small messages. Use of the synchronous routines for large messages may cause a problem with buffer overflow, as all such messages are received in a ring buffer of size 64-512 Kbyte. The problem can be avoided by replacing xy_send with r_rsend for large messages and splitting up the row and column broadcasts via x_brd and y_brd into several smaller broadcasts which wait for an acknowledgement before proceeding. The acknowledgement can conveniently be implemented using a function such as x_amax or y_amax. Currently we use r_rsend for messages larger than 1 KByte and limit x/y_brd to 16 Kbyte. This ensures that a ring buffer of 64 Kbyte is adequate, but does not significantly reduce performance.

4.5 Context switching and virtual cells

Our description above has focussed on the case of aspect ratio 1, because this is the simplest to understand and is optimal (or close to optimal) because of the balance between communication bandwidth in the horizontal and vertical directions. On a CAP with $P = 2s^2$ cells, e.g. $P = 512$, we are faced with a dilemma. We can –

1. Choose $\sigma = \lfloor P^{1/2} \rfloor$, e.g. $\sigma = 22$ if $P = 512$, and configure the CAP as a σ by σ array.
2. Choose $s = (P/2)^{1/2}$ and configure the CAP as an s by $2s$ array.
3. As for case 2, but configure the CAP as a $2s$ by $2s$ array with each real cell simulating two virtual cells.

Method 1 is possible, but is inefficient because some cells are not used (e.g. only 484 of 512). Even worse, it turns out that the communication routines x_brd and y_brd are much slower in this case than in case 2, because the hardware broadcast capability of the T-net can not be used.

Simulating x_brd/y_brd using xy_send/xy_recvs and a binary tree is possible, but several times slower than when we can use the broadcast capability of the T-net.

Method 2 is probably the fastest, but complicates the programming task.

Method 3 is easy to program, but turns out to be slow on the CAP if each virtual cell is simulated by a separate task. The reason is that context-switching is slow (taking considerably longer than the time to send a short message from one cell to another using xy_send/xy_recvs). Also, the synchronous communication routines (xy_send, x_brd etc) can not specify a task ID, so can not easily be used if more than one user task is sending/receiving messages.

Our *ad hoc* solution was to use method 3 conceptually, but to simulate the two virtual cells explicitly in the program for each cell. Thus each cell has only one user task and the synchronous communication routines can be used. The difficulty with this solution is that a separate program has to be written and maintained for the two cases $P = s^2$ and $P = 2s^2$.

In future we plan to write LU factorization and back-substitution routines which work for all aspect ratios (not just 1 or 2), but are not significantly slower than the routines specialised for aspect ratios of 1 or 2. This work is not yet complete, but we have already implemented matrix transposition and multiplication for general aspect ratios [23].

5 Preliminary results on the AP 1000

In this Section we present preliminary 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. Single-precision is about 50 percent faster.

The CAP cells run at 25Mhz and the time for a double-precision floating-point multiply and add is $6 + 3 = 9$ cycles, giving a theoretical peak speed of 5.56 Mflop per cell when performing computations with an equal number of multiplies and adds. (The peak speed is 6.25 Mflop per cell if two adds are performed for every multiply.)

The results in Table 1 are for $n = 1000$ and should be compared with those in Table 2 of [7]. 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 [7]. 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).

The results for the CAP are good when compared with reported results for other distributed memory MIMD machines such as the nCUBE, 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 $n = 21376$ [7] with $r_{peak} = 2.4$ Gflop. Our results indicate that a P-cell CAP is consistently faster than a $2P$-cell nCUBE 2. The 512-cell Intel Delta achieves 11.9 Gflop but this is less than 60 percent of its theoretical peak of 20 Gflop [11]. The 128-cell Intel iPSC/860 achieves 2.4 Gflop, slightly more than the 512-cell CAP, but this is only 48 percent of its theoretical peak of 5 Gflop. For large n the CAP consistently achieves in the range 79 to 82 percent of its theoretical peak (with the ratio slightly better when the number of cells is a perfect square, e.g. 64 or 256, than when it is not).

An encouraging aspect of the results is that the CAP has relatively low n_{half}. For example, on the 64-cell CAP at ANU we obtain at least half the maximum performance (i.e. at least

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

Table 2: LINPACK Benchmark results for large n

145 Mflop) for problem sizes in the wide range $648 \leq n \leq 10000$. (On the 64-cell Intel Delta, the corresponding range is $3000 \leq n \leq 8000$ [11].) As expected from (3.8), 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 CAP, 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,

Problem size n	With blocking				Without blocking			
	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: Linpack speeds (%) on 64-cell CAP (see text)

these routines were simulated using binary tree algorithms and the basic communication routines xy_send, xy_recvs (or, for messages of more than 1024 bytes, r_rsend, r_rrecv).

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 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 CAP 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 (3.5) gives a good fit to the benchmark results only if n is sufficiently small and ω is fixed (or blocking is not used).

6 Conclusion and Future Plans

The LINPACK Benchmark results show that the CAP 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. The high-bandwidth hardware broadcast capability of the T-net (x_brd, y_brd) and the low latency of the synchronous communication routines (xy_send etc) are significant.

Conversely, one might argue that the floating-point units on the CAP cells are too slow to take full advantage of the high communication speed of the T-net. The performance of the CAP on the LINPACK Benchmark could be increased significantly if each cell had a faster, pipelined floating-point unit. However, it must be remembered that three cycles are required for each 8-byte load from the cache, so it would be difficult to keep the floating-point unit busy except for computations with $R << 1$ (see Section 4.2 and Section 5.3 of [23]).

In the future we intend to implement the LINPACK Benchmark on CAP configurations with general aspect ratio (not just 1 and 2). We also intend to implement other common linear algebra routines, e.g. –

Orthogonal (QR) factorisation using blocked Householder transformations [1, 21].

Solution of the symmetric eigenvalue problem by the recursive splitting method of Dongarra and Sorensen [9] and/or a Jacobi method [5].

The singular value decomposition (SVD) by various methods [2, 5, 14]. For some preliminary results, see [6].

7 Acknowledgements

Thanks to Takeshi Horie and his colleagues at Fujitsu Laboratories for assistance in running the Benchmark programs on a 512-cell CAP, to Peter Strazdins for many suggestions related to

the use of the BLAS, and to Peter Price for his assistance in optimising some of the assembler routines.

References

[1] C. Bischof and C. Van Loan, "The WY representation for products of Householder matrices", *SIAM J. Sci. and Statist. Computing* 8 (1987), s2-s13.

[2] R. P. Brent, "Parallel algorithms for digital signal processing", in [15], 93-110.

[3] 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.

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

[5] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays", *SIAM J. Sci. and Statist. Computing* 6 (1985), 69-84.

[6] A. Czezowski and P. Strazdins, "SVD computation on the AP 1000", *Proceedings CAP Workshop 91*, Canberra, November 1991.

[7] J. J. Dongarra, "Performance of various computers using standard linear equations software", Report CS-89-05, Computer Science Department, University of Tennessee, version of November 12, 1991 (available from `netlib@ornl.gov`).

[8] J. J. Dongarra, F. G. Gustavson and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine", *SIAM Review* 26 (1984), 91-112.

[9] J. J. Dongarra and D. C. Sorensen, "A fully parallel algorithm for the symmetric eigenproblem", *SIAM J. Sci. and Statist. Computing* 8 (1987), 139-154.

[10] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors, Vol. I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[11] 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.

[12] G. A. Geist and M. Heath, "Matrix factorization on a hypercube", in [17], 161-180.

[13] 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.

[14] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions", *Numer. Math.* 14 (1970), 403-420.

[15] G. H. Golub and P. Van Dooren, *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, Springer-Verlag, 1990, 93-110.

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

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

[18] 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.

[19] 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.

[20] C. Moler, "Matrix computations on distributed memory multiprocessors", in [17], 181-195.

[21] R. Schreiber and C. Van Loan, "A storage-efficient WY representation for products of Householder transformations", *SIAM J. Sci. and Statist. Computing* 10 (1989), 53-57.

[22] G. W. Stewart, *Introduction to Matrix Computations*, Academic Press, New York, 1973.

[23] P. E. Strazdins and R. P. Brent, "Implementing BLAS level 3 on the AP 1000", *Proceedings CAP Workshop 91*, Canberra, November 1991.

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

[25] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon, Oxford, 1965.