Software Overhead and Blocking Issues in Parallel BLAS

Peter Strazdins,
Australian National University, Acton, ACT 0200. AUSTRALIA
ph +61 6 249 5140. fax +61 6 249 0010. e-mail Peter.Strazdins@cs.anu.edu.au

February 17, 1998

Abstract

This paper compares the performance of three parallel BLAS implementations on the Fujitsu AP1000 and AP+ parallel computers. The comparison is based on LU and LLT decomposition benchmarks, with a secondary parallel rank-1 update benchmark to illustrate the extent of software overheads. These show that algorithmic blocking and reduced software overheads can enhance the performance of LU decomposition on such an architecture by a factor of 1.70 (LU) and 1.25 (LLT) for small matrices, and by a factor of 1.27 (LU) and 1.23 (LLT) for moderate sized matrices, on the AP1000. On the AP+, the enhancement was a factor of 1.52 (LU) and 1.13 (LLT) for small matrices, and by a factor of 1.27 (LU) and 1.31 (LLT) for moderate sized matrices. For large matrices, the performance differences persists for the case of LU, and reduces by half for the case of LLT. For these reasons, it is important for parallel BLAS implementations to have low software overheads and support algorithmic blocking effectively.

1 Introduction

ScaLAPACK [3, 5, 1] is a parallel dense linear algebra library based on *block-partitioned* algorithms, with essentially all computations (and most communication) being expressed the Level 1, 2 and 3 Parallel Basic Linear Algebra Subroutines (PBLAS).

This report compares the following three implementation of the PBLAS:

1. UTK PBLAS 1.2 (released on NetLib, May 1996, [4]).
2. ANU PBLAS (3rd Edition, released to Fujitsu Parallel Research Facilities, May 1997; a slightly enhanced version produced in June 1997 was used in this report).

   The ANU PBLAS is essentially a front end to another version of parallel BLAS, called the DBLAS. This was primarily designed for the Fujitsu AP series of multicomputers, but can be compiled to have portable (BLAS/BLACS) interfaces [11].

   The DBLAS Home Page can be found at:
   `http://cs.anu.edu.au/people/Peter.Strazdins/projects/DBLAS/index.html`.
3. UTK PBLAS 2.0 Alpha (a pre-release version of that to be released on NetLib, November 1997).

   A description of some of the methods used in this release can be found in [8].

LU decomposition with partial pivoting will be used to illustrate the issues of algorithmic blocking and software overheads. In this context, *algorithmic blocking* simply means choosing the block-partitioned panel width ω independently to the storage block size r, where an $r \times r$ block-cyclic matrix distribution is used over a $P \times P$ logical grid of processors. For the AP1000, it has been found that $r \approx 1$ and $\omega \approx 64$ is optimal for LU and LLT on an 8×8 Fujitsu AP1000 [10]. For the AP+, it has been found that $r \approx 1$ and $\omega \approx 32$ is optimal.

Algorithmic blocking has two potential advantages over storage blocking. It can achieve better load balance on the panel formation stages of a block-partitioned algorithm. This can not only yield a performance gain in its own right, but can allow a wider blocking factor, which in turn can enable a somewhat higher `dgemm()` performance on memory-hierarchy processors. This advantage increases linearly with ω and P (up to at least moderate-sized processor configurations). Its potential disadvantages include increased communication startup and volume overheads, usually by a constant factor. More details on this can be found in [10].

Software overheads, if high enough, can significantly degrade performance for small to moderate sized block-partitioned computations [6, 12, 9]. A comparison of the software architectures of the UTK PBLAS 1.2 and of the ANU PBLAS, relevant to the potential for software overheads, can be found in Section 3 of [12].

This paper is organized as follows. Section 2 describes features of the Fujitsu AP1000 and AP+, including its BLAS and BLACS implementations, that are relevant for this study. A secondary benchmark of the parallel rank-1 update (`PDGER()`) is given in Section 3 which isolates some software overhead issues. Sections 4 and 5 describe the LU and LLT decomposition benchmarks, respectively, which are then discussed in Section 6.

2 The Fujitsu AP Machines

The Fujitsu AP1000 [7] is a SPARC 1-based multiprocessor; it's cells have a theoretical peak speed of 5.5 MFLOPS (double precision). The AP1000 uses a physical torus communication network using wormhole routing. A useful feature of this is the ability to perform a row or column broadcast for the cost of a normal point-to-point message (ie. $\kappa_{bc} = 1$ in the notation of [3]).

The ANU configuration has 128 cells, arranged in a 16×8 torus (see `http://cap.anu.edu.au/` for more details).

The AP1000 cells have a 128KB direct-mapped copy-back cache (shared by instructions and data). This has two consequences of relevance to this report:

- a logical blocking factor $\omega \approx 64$ is optimal for this architecture.
- for small computations, placement of code and data in memory can have a greater impact due to the shared and direct-mapped properties of the cache.

  This is further explained in the Section 2.3.

On the AP1000, benchmark programs have yielded values of $\alpha = 25\mu s, \beta = 1.2\mu s, \gamma_3(16) = .25\mu s$ and $\gamma_3(64) = .20\mu s$, where $\gamma_3(\omega)$ is the cost per floating point operation in a local matrix multiply having input operands of width ω.

While it is not a fast machine by today's standards, it still can be used to illustrate the issue of algorithmic blocking vs storage blocking for machines with large cache size (this is more significant than the relatively low α/γ_3 ratio of the AP1000 [10]) and the issue of software overheads.

2.1 The Fujitsu AP+

The AP+ is similar to the AP1000, except that its cells are based on a 50 MHz Viking SuperSparc chip, having a 4-way set-associative write-through 16 KB data cache, and a 5-way set-associative 20 KB instruction cache. There is no second-level cache; this reduces level-3 and especially level-2 computation performance to a large extent; also it increases the cost of an instruction or data cache miss. The optimal logical blocking factor is $\omega \approx 32$, and the cache associativity means that placement of code and data in memory generally has little effect on program performance.

The AP+ has source code compatability with the AP1000, and it uses an identical communication network[1]. On the AP+, benchmark programs have yielded values of $\alpha = 12\mu s, \beta = 0.64\mu s, \gamma_3(16) = .035\mu s$ and $\gamma_3(32) = .030\mu s$.

The AP+ cells have many of the features modern RISC processors, eg.

multiple instruction issue, integer multiply and divide instructions, virtual memory.

[1]Because a memory copy is part of a communication on the AP1000/AP+, the AP+ has a lower value of β due to a faster memory bus.

Hence a smaller portion of the software overhead due to indexing arithmetic is expected on the AP+[2]; on the other hand, the effects of instruction cache misses are potentially larger. With is smaller data cache size, it would appear to favor algorithmic blocking less than does the AP1000.

The Fujitsu Parallel Computing Facilities has a 64 node AP+ arranged in a 8×8 torus, which was mainly used for the results in this paper[3].

2.2 AP BLAS and BLACS Implementations

A low-overhead implementation of the BLAS and BLACS (C and Fortran) has been made for the AP1000 and AP+. It is not complete, but includes all routines used in the benchmarks for this paper.

These implementations have minimal software overheads (and share some software overhead reducing techniques, and even some codes, eg. for fast buffer allocation, as used for the DBLAS [12]). No error checking is performed, and in the cases of the BLACS, all BLACS contexts must be the same as the 'natural' cell layout

Note that broadcast topologies are ignored in the AP BLACS implementation. This is because the hardware broadcast is superior in any situation to tree or ring topologies simulated by point-to-point messages.

2.3 Reducing the Effects of Direct-Mapped Cache Conflicts on the AP1000

As previously mentioned, the placement of code and data in memory can have a greater than usual impact due to the shared and direct-mapped properties of the 128 KB cache on AP cells for small computations. To minimize this impact, the following two techniques have been used for the benchmarks in this paper:

1. to reduce instruction-instruction cache conflicts,
   all *active code* (or, in the case of LU, that used by level 2 part of the decomposition, which is the most critical with this respect) used in the timing was packed into a single object file before being linked into the executable program[4].

   The *active code* also included all low-level AP system communication routines.

   This technique is only useful if the size of the active code does not exceed the cache size significantly. In the case of the LU benchmark, it improved performance for the ANU PBLAS by up to 7% for the smallest matrix size.

   The tests were run with this technique and without (signified by the the active code size marked with a '*'), with the best result being reported.

2. to reduce instruction-data cache conflicts,
   the relative offset of the heap to the code in memory was changed in 32KB increments; the best offset for the smallest data size ($N = 512$ in the case of LU) was used for the timings.

   For the smallest matrix size of the LU benchmark, there was at most a 6% difference between the best and worst cases.

On the AP+, these techniques did not make a significant impact on program performance.

3 The `PDGER()` Benchmark

`PDGER()` was used to perform $C = C - A * B$ on a 4×4 processor configuration. Here A, B, C are 4×4 matrices, with storage block size $r = 1$ (ie. 1 element per cell), and $C = \mathtt{M(5,5)}, A = \mathtt{M(5,1)}, B = \mathtt{M(1,5)}$ and the super-matrix `M` has column-major storage.

[2] Indeed, relative to floating point speed, the integer multiply and divide instructions are faster than on the UltraSparc [12].

[3] Due to difficulties porting the ScaLAPACK test programs (cells cannot read files or standard input), equivalent C test programs, interfacing to ScaLAPACK or PBLAS, were used.

[4] Ideally, in the case of LU, all code should fit within a 128 KB segment (this is not possible with the current parallel BLAS implementations, and indeed for the ANU PBLAS, the active (level 2) codes are uncomfortably close to or exceeding 128 KB. Note that within the ANU PBLAS's *active code*, the 'important' routines will be mainly at either end of the segment, with large sections of unused code in the middle.)

routine used	uses BLAS & BLACS	s/w overhead ($\times\alpha = 25\mu s$)		*active code* size
ANU PDGER()	no	3.2	(1.2)	90 KB
ANU PDGER()	yes	4.1	(1.2)	112 KB
UTK 1.2 PDGER()	yes	12.9	(6.3)	76 KB
UTK 2.0 PDGER()	yes	11.0	(4.6)	176 KB

Table 1: Software overheads in the PDGER() benchmark (in parentheses, the component overhead due to PBLAS descriptor checking & indexing) on the AP1000

routine used	uses BLAS & BLACS	s/w overhead ($\times\alpha = 12\mu s$)	
ANU PDGER()	no	3.9	(1.1)
ANU PDGER()	yes	5.5	(1.1)
UTK 1.2 PDGER()	yes	12.1	(3.5)
UTK 2.0 PDGER()	yes	12.0	(2.3)

Table 2: Software overheads in the PDGER() benchmark (in parentheses, the component overhead due to PBLAS descriptor checking & indexing) on the AP+

This benchmark tests the overhead of a fully parallel PDGER() operation, with, at each step, a different cell column/row broadcasts a column/row of A/B, ie.

```
DO I = 1, 4
  C = C - A(1:4,i) * B(i,1:4)
END DO
```

While this does not exactly simulate the sequence of PDGER() calls in LU decomposition, it is near enough from the point of view of software overheads, and PDGER() is probably the most critical single PBLAS routine with respect to software overhead in LU decomposition.

Table 1 gives the AP1000 software overheads per call (total execution time minus 2.0α for communication startup overhead minus 0.8α for the software overhead of the matrix multiply routine). Note that for the UTK 1.2 PDGER(), the *matrix descriptor checking & indexing* component is essentially 2 calls to pbchkvect() and one to pbchkmat(); for the UTK 2.0 PDGER(), the equivalent code was isolated for this test. For the ANU PDGER(), inter-call *descriptor caching* was used to accelerate the equivalent of this component [12].

The reduction in software overheads of the UTK 2.0 PBLAS over the UTK 1.2 PBLAS is largely due to more efficient indexing arithmetic. In particular, almost all integer modulus operations have either been avoided or replaced by (generally) less expensive integer multiplications or conditional additions/subtractions. Approximately half of the the *matrix descriptor checking & indexing* component of the overheads here were due to descriptor checking, and about half is due to indexing in the Minfog2l() macro.

From the first two rows of this table, it can be deduced that the use of the (portable) BLAS / BLACS interfaces adds another 0.9α overhead; this is mainly from the BLAS.

The above table indicates that software overheads are smaller by a factor of 3.4 in the ANU PBLAS, relative to the UTK PBLAS.

Table 2 gives the AP+ software overheads per call (total execution time minus 2.0α for communication startup overhead minus 0.5α for the software overhead of the matrix multiply routine); however a 2×2 configuration was used instead of a 4×4. It indicates that the role of indexing arithmetic for software overhead is less on the AP+, as expected. This table indicates that software overheads are smaller by a correspondingly smaller factor of 2.1 in the ANU PBLAS, relative to the UTK PBLAS.

N	512	1024	1536	2048	2560	(active code size)
`ANU-Al-N`	1.66	2.87	3.53	3.87	4.09	(128 KB)
`ANU-St-N`	1.22	2.20	2.75	3.08	3.31	
`ANU-Al-P`	1.51	2.74	3.43	3.50	4.05	(144 KB)
`ANU-St-P`	1.14	2.09	2.68	3.03	3.29	
`UTK1.2-St-P`	0.83	1.80	2.43	2.84	3.11	(128 KB*)
`UTK2.0-Al-P`	0.66	1.33	1.84	2.23	2.55	(250 KB*)
`UTK2.0-St-P`	0.89	1.87	2.51	2.89	3.17	

Table 3: Performance of the LU benchmark on an 8×8 AP1000, with speeds given in MFLOPs/cell

N	512	1024	1536	2048	2560	3072	3584
`ANU-Al-P`	3.5	8.6	12.8	15.8	18.1	19.9	21.3
`ANU-St-P`	2.9	6.8	10.0	12.1	14.1	15.6	16.8
`UTK1.2-St-P`	2.3	6.0	9.2	11.5	13.6	15.2	16.4
`UTK2.0-Al-P`	2.3	6.0	9.2	11.6	13.8	15.4	16.7

Table 4: Performance of the LU benchmark on an 8×8 AP+, with speeds given in MFLOPs/cell

4 The LU Benchmark

This section describes the benchmark of a parallel column-oriented LU decomposition of an $N \times N$ matrix. The codes used were the ScaLAPACK 1.2 `PDGETRF()` and `PDGETF2()` routines extended to support algorithmic blocking[5]. These can be found at:

http://cs.anu.edu.au/people/Peter.Strazdins/projects/ExtendScaLAPACK/index.html

Note that for algorithmic blocking, extra communication is required when column-replicated pivot vectors are used; also some compression of the pivot vector is required, adding a small amount of software overhead.

Table 3 gives the performance of various versions of the LU benchmark on the AP1000, with the tests being named as follows. `Al` signifies algorithmic blocking was used, with $\omega = 64$ and $r = 1$. `St` signifies storage blocking, with $\omega = r = 16$ [6]. `P` signifies a portable BLAS/BLACS interface was used; `N` signifies a non-portable (APLib) interface.

Table 4 gives the benchmark for the AP+. The naming of the tests is as for the AP1000, except for the `Al` tests, $\omega = 32$ was used[7]. The `ANU-*-N` tests gave a consistent 0.2 MFLOP increase over the corresponding `ANU-*-P` tests, so they were omitted from this table. The `UTK2.0-Al-P` tests yielded no improvement over `UTK2.0-St-P`, and so they too were omitted.

5 The LLT Benchmark

Similarly to the LU benchmark, one for the LLT decomposition of a lower triangular $N \times N$ matrix was performed. With respect to algorithmic blocking and software overheads, LLT has the following

[5] The test program `pdludriver.f` had also be extended to allocate more space for the pivot vector.

[6] For LU and LLT, this is the optimal storage blocking factor on an 8×8 AP1000, for $2048 \leq N \leq 3072$. For $512 \leq N \leq 1536$, $r = 12$ is optimal, being at most 4% faster than for $r = 16$.

[7] For storage blocking an 8×8 AP+, $r = 16$ was again found to be optimal for $1024 \leq N \leq 8192$, except for a few points in the middle of this range where $r = 24$ was marginally faster.

N	512	1024	1536	2048	2560	3072	(active code size)
ANU-Al-P-pdsyr	1.23	2.39	3.05	3.48	3.79	3.96	(168 KB)
ANU-St-P-pdsyr	1.00	1.93	2.49	2.83	3.06	3.23	
ANU-St-P	1.05	2.00	2.55	2.87	3.10	3.26	
UTK1.2-St-P-pdsyr	0.57	1.35	1.96	2.38	2.68	2.92	(120 KB)
UTK1.2-St-P	0.83	1.66	2.22	2.57	2.83	3.03	
UTK2.0-Al-P-pdsyr	0.51	1.19	1.71	2.12	2.42	2.63	(240 KB)
UTK2.0-St-P-pdsyr	0.69	1.55	2.18	2.58	2.87	3.09	
UTK2.0-St-P	0.97	1.90	2.44	2.79	3.03	3.22	

Table 5: Performance of the LLT benchmarks on an 8×8 AP1000, with speeds given in MFLOPs/cell

differences from LU:

- with algorithmic blocking, LLT increases the number of communication startups relatively more, from $O(N/r)$ to $O(N)$.

  Also, the level-2 factorization must be distributed, requiring in our implementation N calls to each of `PDSYR()`, `PDSCAL()`, `PDELGET()` and `PDELSET()`. This leads to a potential increase in software overheads.

  From both of these factors, performance increases for small N are expected to be less than in LU.

- an $O(\frac{N^2 r}{P})$ load imbalance with cells on the right-upper half of the grid having less work to do favors a low value of r, ie. algorithmic blocking.

- the issue of software overheads is greater in `PDSYRK()` than in `PDGEMM()`, since the former (in all 3 implementations) uses a much larger number of calls to the cell BLAS routines [8].

For this benchmark, `netlib` (vanilla) versions of `dgemv()`, `ddot()` and `dsyr()` were used; this is because they were faster[9] over the range of sizes in which they would be called ($n \leq r = 16$).

Table 5 gives the performance of various versions of the LLT benchmark, with the naming scheme as for to Table 3. Here, `pdsyr` indicates a `PDSYR()`- based level 2 factorization routines was used; note that `Al` (algorithmic blocking) requires this. Thus, `UTK1.2-St-P` represents the original ScaLAPACK 1.2 performance. Note that for `UTK2.0-Al-P-pdsyr`, unlike in LU, a block size of $r = 8$ was found to be optimal and was used in these results; $r = 1$ was optimal for `ANU-Al-P-pdsyr`.

Table 6 gives the performance of various versions of the LLT benchmark on the AP+, a similar naming scheme as for Table 4.

6 Discussion

Several points can be raised by the results of Sections 4 and 5 (together with those of Section 3).

1. algorithmic blocking had the most impact on performance, even though it doubled the communication volume of the L and U panels in LU, and of the L panel in LLT[10]. This improvement is sustained for larger matrix sizes.

[8] Note that the ANU PBLAS implementation is `dgemm()`-based, ie. does not use any other BLAS for matrix multiplication.

[9] This is due to software overheads. For example, the `netlib dsyr()` has a latency of 1.0α less than the 'optimized' general triangular multiply routine ultimately called by the ANU PBLAS.

[10] The second communication of the U panel is redundant. An equivalent DBLAS-based LU algorithm has been modified to eliminate this, resulting in a 7% performance increase for $N = 512$ on the AP1000. Similarly for the second communication of the L panel in LLT.

N	512	1024	1536	2048	2560	3072	3584
ANU-Al-P-pdsyr	2.7	7.1	11.1	14.4	16.9	18.9	20.5
ANU-St-P-pdsyr	2.8	7.0	10.5	13.2	15.4	17.1	18.4
ANU-St-P	4.2	8.7	12.2	14.6	16.6	18.1	19.2
UTK1.2-St-P-pdsyr	1.5	4.3	7.1	9.3	11.2	12.8	14.0
UTK1.2-St-P	3.1	6.7	9.6	11.5	13.1	14.4	15.4
UTK2.0-St-P-pdsyr	1.9	5.3	8.5	10.5	12.3	13.7	14.8
UTK2.0-St-P	3.5	7.3	10.0	11.9	13.4	14.6	15.6

Table 6: Performance of the LLT benchmarks on an 8×8 AP+, with speeds given in MFLOPs/cell

For the UTK PBLAS 2.0, algorithmic blocking did not improve performance. The reasons for this is not yet understood – it is possible that it was not installed optimally on the AP1000[11]. A trace of BLACS calls for `UTK2.0-Al-P` indicates however that there is a large volume of communication occurring within the 'aggregate' stages of the hybrid LCM blocking method [8] used in `PDGEMM()`, which could account for some of the performance loss.

2. the (software) overheads introduced by using portable interfaces, ie. the BLAS/BLACS, are not large but still discernible, being about 5% for $N = 512$ (cf. `ANU-*-P` and `ANU-*-N`). Most of the performance loss is due to[12]:

   (a) the BLAS must perform a scaled copy via a copy followed by a scale. This increases overhead and decreases computation speed.

   (b) there is a low-latency version of the 'native' AP matrix multiply routine that can be used when packing and scaling of the input operands is not required. The BLAS interface can only provide a single (higher latency) matrix multiply routine.

   (c) an extra level of procedure call is introduced by the AP BLAS, in which the DBLAS local matrix descriptor structures [11] get converted to and from the equivalent Fortran representation.

   (d) the BLACS requires the index to be broadcast separately in `PDAMAX()`.

3. the impact of software overheads can be seen by comparing `*-St-P`, since all these tests should be exactly the same with respect to communication and cell computation.

   As is also indicated by the results of Section 3, the UTK PBLAS 2.0 shows an improvement over UTK PBLAS 1.2, with the ANU PBLAS having the least overhead, yielding a 25% improvement for LU at $N = 512$ on the AP1000 and the AP+.

4. the overall difference between the portable LU tests `ANU-Al-P` & `UTK*-St-P` is a factor of 1.70 (1.52) for $N = 512$ and a factor of 1.27 (1.31) for $N = 2560$ on the AP1000 (AP+), This combines the effect of both algorithmic blocking (the most significant) and reduced software overheads.

5. using a DBLAS-based version of the same LU decomposition code[13], the performance can be further enhanced slightly from `ANU-Al-*`, both from having less software overhead than in the corresponding ScaLAPACK routines (`PDGETRF()`, `PDGETF2()` and `PDLASWP()`) and from being able to use row-major storage.

[11] `PILAENV()` was modified to return a logical blocking factor of 64. At the time of writing, the author had no installation guide document specific for this release.

[12] A potential secondary effect on performance is that portability increased the code size (and hence worsened the effects of instruction-instruction cache clashes).

[13] See Section 2.1 of [12]; the codes here have been modified use a column-replicated instead of a fully replicated pivot vector, to be more comparable to the ScaLAPACK codes.

On the AP1000, the former can yields a 6% improvement at $N = 512$. The latter is generally faster ($\approx$ 5% faster at $N = 512$) since with partial pivoting there will be slightly more accesses across rows than columns.

On the AP+, column-major storage is always optimal. The DBLAS-based LU and LLT algorithms, using non-portable interfaces, are faster than the modified ScaLAPACK routines calling the ANU PBLAS, by $\approx$ 25% at $N = 512$ and $\approx$ 6% for $N \geq 2560$. That this difference is much greater on the AP+ is thought to partially due to reduced instruction cache misses, since the DBLAS-based codes are considerably more compact.

6. in Table 5, the effect of the overheads of using the PBLAS (mainly `PDSYR()`) for the level-2 factorization can be seen, by comparing the corresponding pairs of `*-St-P` with `*-St-P-pdsyr`. This indicates a relative degree of software overheads applies for `PDSYR()` as for `PDGER()`, as shown in Tables 1 and 2.

   The difference is far greater in Table 6; it is conjectured that the extra difference is due to instruction cache misses being avoided in the BLAS-based level-2 factorization on the AP+.

   Comparing the various `*-St-P` gives an indication of the effect of software overheads (and possibly different transposition algorithms) in `PDTRSM()` and `PDSYRK()`.

7. due to factors described in Section 5, the relative improvement from algorithmic blocking and software overheads are less for LLT than for LU for small N.

   However, for moderate N, the results are similar, with other trends as for the LU benchmark.

8. the overall difference between the LLT tests `ANU-Al-P-pdsyr` & `UTK2.0-St-P` is a factor of ≈ 1.25 for $512 \leq N \leq 3072$ on the AP1000.

   For the AP+, algorithmic blocking is not always faster for small N, especially when a BLAS-based `pdpotf2()` is used for storage blocking. The overall difference between the best of `ANU` tests and `UTK2.0-St-P` is 13% at $N = 512$, increasing to 30% at $N = 3584$.

9. large matrix tests ($4096 \leq N \leq 8192$) were carried out on the 64-node AP+, using the ANU-based codes. The performance advantage of algorithmic blocking over storage blocking was still significant, being 20–25% for LU and 12–15% for LLT. For LU, this result was rather surprising: a performance model [12] calibrated for the AP+ predicted only half this difference. A high degree of cache and TLB misses in the formation of the U panel, occurring for large N, could account for this discrepancy: these would affect algorithmically blocked code less, since the local width of the panel is reduced by a factor of $1/P$ (where $P = 8$).

7 Future Work

Once the performance problems for algorithmic blocking on the PBLAS 2.0 are understood and rectified, the corresponding tests should be repeated.

It would be interesting to extend this comparison to other distributed memory machines, particularly those with large (second-level) caches, and lower α/γ_3 ratios (which is further heightened by having $\kappa_{bc} > 1$). The lower β/γ_3 ratios of these machines will also require the redundant communications of algorithmic blocking to be eliminated.

The 170 MHz UltraSparc-based Fujitsu AP3000 is such a machine; a performance model for LU decomposition calibrated for a 64-node AP3000 predicts a 15–20% performance gain for algorithmically blocked LU, over moderate–large matrix sizes[14]. This is assuming parameters of $\alpha = 18\mu s$, $\beta = 0.22\mu s$ (36 MB/s) and $\gamma_3(88) = 4ns$ (250 MFLOPs). The ANU codes will require some performance enhancements for such a machine.

It is felt that algorithmic blocking could improve the performance of the 64-node machine configurations in `PDGESV()` in Table 5.9 of [2] by at least half this amount, excepting the Cluster of Sun Ultra

[14] The author expects to gain access to a small configuration AP3000 in the next few months.

2s. For most of these machines, the LU performance model, calibrated by Table 5.5 of [2][15], predicts $4 \le r \le 8$ and $P = 4Q$ to be optimal.

It would also be interesting to extend this comparison for other computations (eg. QR, symmetric tri-diagonal reduction).

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, J. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Linear Algebra Library for Message Passing Computers. In *SIAM Conference on Parallel Processing*, March 1997.

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, J. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK User's Guide*. SIAM Press, Philadelphia, 1997.

[3] J. Choi, J. Demmel, J. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Applied Parallel Computing*, pages 95–106, Berlin, 1995. Springer-Verlag.

[4] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, and D. Walker amd R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra SubPrograms. In *Applied Parallel Computing*, pages 107–114, Berlin, 1995. Springer-Verlag.

[5] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D.W. Walker, and R.C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

[6] B. Hendrickson, E. Jessup, and C. Smith. A Parallel Eigensolver for Dense Symmetric Matrices. *submitted to SIAM J. Sci. Comput.*, 1996.

[7] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata. Improving AP1000 Parallel Computer Performance with Message Communication. In *International Symposium of Computer Architecture (ISCA '93)*, 1993.

[8] A. Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions.* PhD thesis, University of Tennessee, Knoxville, December 1996. xv+193p.

[9] K. Stanley. *Execution Time of Symmetric Eigensolvers.* PhD thesis, University of California, Berkelely, 1997. viii+184p.

[10] P.E. Strazdins. Matrix Factorization using Distributed Panels on the Fujitsu AP1000. In *IEEE First International Conference on Algorithms And Architectures for Parallel Processing (ICA3PP-95)*, pages 263–73, Brisbane, April 1995.

[11] P.E. Strazdins. A High Performance, Portable Distributed BLAS Implementation. In *Sixth Parallel Computing Workshop*, pages P2–K–1 – P2–K–10, Kawasaki, November 1996. Fujitsu Parallel Computing Research Center.

[12] P.E. Strazdins. Reducing Software Overheads in Parallel Linear Algebra Libraries. In *The 4th Annual Australasian Conference on Parallel And Real-Time Systems*, pages 73–84, Newcastle Australia, September 1997. Springer.

[15] This model requires level-2 and level-1 computations speed, among various other parameters, which have had to be estimated at this point. Also, whether γ_3 improves with larger blocking factors than given in Table 5.5 is yet unknown, except for the UltraSparc-based machines.