Notes on LINPACK NxN Benchmark on Hewlett-Packard Systems

Piotr Luszczek*

August 3, 2001

Benchmark name	Matrix dimension	Optimizations allowed	Parallel Processing
100	100	complier	No
1000*	1000	manual	No
Parallel	1000	manual	Yes
NxN**	arbitrary	manual	Yes

* also known as TPP (Towards Peak Performance) or Best Effort

** also known as Highly-Parallel LINPACK Benchmark

Table 1: The LINPACK Benchmark details.

1 LINPACK Benchmark Overview

As a yardstick of performance the "best" performance is used as measured by the LINPACK Benchmark [1, 2]. It is set of benchmarks that indicate speed at which a given computer solves a system of linear equations. Details of the types of the LINPACK benchmark suite are given in table 1.

The TOP500 list contains computers ranked by their performance on the LINPACK NxN Benchmark. The list has been compiled twice a year since June 1993 and is published in June at the International Supercomputer Conference in Europe and in November at the SC conference in the U.S.A. Systems having the same performance rate are sorted according to their manufacturer's name.

Here is the original description of the NxN benchmark [2]:
"[...] The ground rules are as follows: Solve systems of linear equations by some method, allow the size of the problem to vary, and measure the execution time for each size problem. In computing the floating-point execution rate, use $2n^3/3 + 2n^2$ operations independent of the actual method used. (If you choose to do Gaussian Elimination, partial pivoting must be used.) Compute and report a residual for the accuracy of solution as $\|Ax - b\|/(\|A\|\|x\|)$.

The columns in Table 3 are defined as follows:
R_{max} the performance in Gflop/s for the largest problem run on a machine.
N_{max} the size of the largest problem run on a machine.
$N_{1/2}$ the size where half the R_{max} execution rate is achieved.
R_{peak} the theoretical peak performance Gflop/s for the machine. [...]"

*Mathematical Software Group (`http://texas.rsn.hp.com/`), Richardson, TX 75080, E-mail: `luszczek@rsn.hp.com`

2 The HPL Code

HPL [9] is a software package that generates and solves a random dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It is perfectly suitable as a code to run the Highly-Parallel LINPACK Benchmark. It is portable through the use of Basic Linear Algebra Subprograms (BLAS) [3, 4] and the Message Passing Interface (MPI) [5, 6, 7] and its source code is available free of charge. The only assumption made in the code is that all CPUs in the system have the same processing power and the implementation of MPI's grid topology routines efficiently balances the communication load over the interconnect.

Here are some general remarks of the authors about HPL [9]:
"[...] HPL solves a linear system of order n of the form:

$$Ax = b$$

by first computing the LU factorization with row partial pivoting of the n-by-$(n + 1)$ coefficient matrix:

$$[A, b] = [[L, U], y].$$

Since the lower triangular factor L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system:

$$Ux = y.$$

The lower triangular matrix L is left unpivoted and the array of pivots is not returned. [...]"

This immediately implies that this is not a general LU factorization software and it cannot be used to solve for multiple right hand sides simultaneously.

HPL performs two-dimensional blocking [9] illustrated in the Fig. 1:
"[...] The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The n-by-$(n+1)$ coefficient matrix is first logically partitioned into nb-by-nb blocks, that are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix. [...]"

The type of panel broadcast algorithm can be chosen in HPL out of the following:

- Increasing ring
- Modified increasing ring
- Increasing 2-ring

Global view:

A_{11}	B_{12}	C_{13}	A_{14}	B_{15}	C_{16}
D_{21}	E_{22}	F_{23}	D_{24}	E_{25}	F_{26}
A_{31}	B_{32}	C_{33}	A_{34}	B_{35}	C_{36}
D_{41}	E_{42}	F_{43}	D_{44}	E_{45}	F_{46}

Distributed view:

A_{11}	A_{14}	B_{12}	B_{15}	C_{13}	C_{16}
A_{31}	A_{34}	B_{32}	B_{35}	C_{33}	C_{36}
D_{21}	D_{24}	E_{22}	E_{25}	F_{23}	F_{26}
D_{41}	D_{44}	E_{42}	E_{45}	F_{43}	F_{46}

Figure 1: Block-cyclic distribution scheme for 6 CPUs named A, ..., F of a matrix consisting of 4 by 6 blocks.

- Modified increasing 2-ring
- Long (bandwidth reducing)

It also allows to vary the look-ahead depth, communication pattern for updates U, and recursive algorithm for panel factorization. The backward substitution is done with look-ahead of depth 1.

In order to run, HPL requires implementation of the BLAS and Message Passing Interface standards.

The timing procedure of the solve phase is illustrated by the following pseudo C code from Fig. 2

```
/* Generate and partition matrix data among MPI
   computing nodes. */
/* ... */

/* All the nodes start at the same time. */
MPI_Barrier(...);

/* Start wall-clock timer. */

dgesv(...); /* Solve system of equations. */

/* Stop  wall-clock timer. */

/* Obtain the maximum wall-clock time. */
MPI_Reduce(...);

/* Gather statistics about performance rate
   (base on the maximum wall-clock time) and
   accuracy of the solution. */
/* ... */
```

Figure 2: Timing of the LINPACK Benchmark distributed memory code in HPL.

Tuning of HPL is thoroughly described by the authors. The most influential parameters are: the blocking factor NB, matrix dimension N, process grid shape and communication algorithms.

HPL is most suitable for cluster systems, i.e. relatively many low-performance CPUs connected with a relatively low-speed network. It is not suitable for SMPs as MPI incurs overhead which might be acceptable for a regular application but causes substantial deterioration of results for a benchmark code. Another performance consideration is the use of look-ahead technique. In an MPI setting it requires additional memory to be allocated on each CPU for communication buffer of the look-ahead pipe. In an SMP environment, existence of such buffers is rendered unnecessary by the shared memory mechanism. The buffer memory can grow extensively for large number of CPUs and in SMP system it should rather be used to store matrix data as the performance of the benchmark is closely related to the size of the matrix.

3 Former Benchmark Code for SMP Systems

The old SMP code for LINPACK NxN Benchmark on Hewlett-Packard systems was derived from the freely available LAPACK's implementation of LU factorization [10]. It is a panel-based factorization code with look-ahead of depth 1. It uses advanced optimization techniques [12] and with its blocking factors fine-tuned for the architectures available at the time when the code was created. The code is written in Fortran 90 and the most computationally intensive parts (such as Schur's complement operation) are written in assembly. Threading is performed with the CPS system which performs thread-caching for faster spawning of threads. The code is able to achieve very good benchmark results but unfortunately it is limited as to the dimension of matrices it can factor. The limitation comes from the fact that Fortran 90 runtime system allows the size of arrays that can be dynamically allocated to be no larger than 16 GBytes when 4-byte integers are used. Thus, the biggest double-precision matrix that can be allocated has dimension 46340, even in a 64-bit addressing mode. For bigger matrices a 4-byte integer overflows. This problem may be easily circumvented with the use of 8-byte integers and a change to 64-bit integer arithmetic may be achieved with a single compilation flag. However, such integers would require use of 64-bit version of MLIB [13] and might not work with calls to other libraries. Still, if conformance with all external routines is achieved, the code would use twice as much memory bandwidth to operate on 8-byte integers which is unacceptable for a benchmark code.

4 Current SMP Implementation

The new SMP code for the LINPACK NxN Benchmark implements the same functionality as HPL, i.e. it is not a general LU factorization code, however, it uses one-dimensional data distribution between CPUs. The matrix is partitioned into equal size vertical panels, each of which is owned by a certain CPU. The ownership is determined in a cyclic fashion as illustrated in Fig. 3. In general, panel i is owned by CPU k where: $k = i \pmod{N_{CPUs}}$.

Ownership of a panel implies that a given CPU is responsible for updating and factorizing the panel. This scheme prevents multiple CPUs to update or factor the same panel. Once the data is logically assigned to the CPUs, factorization may proceed. A sample computational sequence for LU factorization

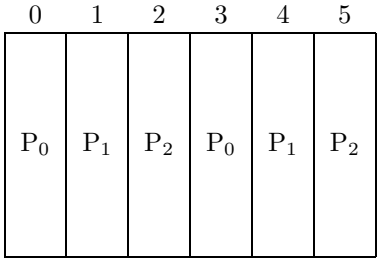
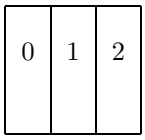


Figure 3: Cyclic 1D division of a matrix into 6 panels. Each panel is owned by one of 3 processors: P_0, P_1, P_2.

is shown in Fig. 4. The important observation is that updates can be performed independently, whereas factorization steps have to be scheduled in a strictly defined order. It is said that factorization steps lie on a *critical path* of the algorithm. The consequence of the existence of a critical path is the need to perform factorization steps ahead of time so that CPUs that do not own a given panel do not have to wait for this panel to be factored. This is usually done with a *look-ahead technique*: during an update stage of computation each CPU checks if it is possible to factor one of its panels ahead of time. Fig. 5 shows a general algorithm that performs a look-ahead during factorization.



Factor panel 0
Update panels 1, 2 using panel 0
Factor panel 1
Update panel 2 using panel 1
Factor panel 2

Figure 4: Computational sequence for factorization of a matrix divided into 3 panels.

The new SMP code for LINPACK benchmark implements three variants of the algorithm from Fig. 5:

- look-ahead of depth 2,
- critical-path, and
- dynamically bounded critical-path.

The variant with look-ahead of depth 2 allows each thread to factor only up to two panels ahead of the current panel. The advantage is that only two threads are accessing thread-shared data. These are the two threads that own panels to be factored through the in look-ahead pipe. The remaining threads perform updates on their own panels. This minimizes contention for shared data. On the other hand, some panels might not be factored soon enough. This, in turn, would cause some CPUs to wait. Details of this variant are shown in Fig. 6.

In the critical path variant, every thread constantly checks for possibility to factor yet another panel. These constant checks may cause excessive contention for shared data but any panel (except for the first one) is always ready (i.e. it is factored) when it's needed. Details of this variant are shown in Fig. 7.

To aliavate the contention of the critical path variant, the dynamically bounded critical path variant has been introduced.

```
for each panel p
begin
    Panel factoriazation phase
    if panel p belongs to this thread
    begin
        Factor panel p.
        Mark panel p as factored.
        Notify threads waiting for panel p.
    end
    else
        if panel p hasn't been factored yet
            Wait for the owner of panel p to factor it.

    Update phase
    for each panel u such that:
        u > p and panel u belongs to this thread
    begin
        Update panel u using panel p.

        Look-ahead technique
        if panel u − 1 has been factored
        begin
            Perform necessary updates to panel u.
            Factor panel u.
            Mark panel u as factored.
            Notify threads waiting for panel u.
        end
    end
end
```

Figure 5: General thread-based panel factorization algorithm with a look-ahead technique. This code is executed by each thread (according to SIMD paradigm).

Still, every thread performs checks whether it's possible to factor yet another panel. However, the checks stop as soon as at least one panel gets factored during a single update phase. The checks resume at the next update phase (i.e. for the updates from the next global panel). To some extend then, this variant tries to minimizes contention for shared data and, at the same time, allows for look-ahead panels to be factored promptly. The algorithm can easily be derived from what is shown in Fig. 7. The change that needs to be made is adding a check whether at least one panel has been factored during a factorization phase. If so, only update operations are performed.

The new code is written in C and uses fine-tuned BLAS computational kernels of the old code. They are necessary ingredients of the code since most of the factorization time is spent in them. The panel factorization uses the recursive right-looking LU decomposition algorithm with partial pivoting [11] presented in Fig. 8. Threading is performed with POSIX thread API (`pthread`) [14] with simple thread-caching mechanism added to shorten thread's start-up time. As synchronization mechanism, `pthread`'s mutexes and condition variables are used. The important remark to make at this point is that threaded applications that use MLIB a single thread stack size should be at least 4 MBytes. Explicit shared data is minized

and is limited to a global integer array (referred in the code as `panel_ready`) of size equal to the number of panels. Initially set to 0, its entries are set to 1 if the corresponding panel has been factored. Accesses to this array are minimized as it might cause excessive contention among threads as well as cache-thrashing when neighboring entries of the array are accessed.

To deal with the aforementioned problem with allocating and indexing of large arrays the code uses 64-bit addressing model. It allows the Standard C Library's `malloc()` function to use memory storage on the heap which is as big as the physical memory constraints permit. However, for such large matrices 64-bit indexing has to be used. Thus, whenever an integer arithmetic operation might overflow a 32-bit integer, variables of type `size_t` (from the standard header file `stddef.h`) are used or appropriate casts are performed.

5 The Code for Constellation Systems

The new MPI LINPACK NxN Benchmark code performs the same operation as its SMP counterpart but is meant for Hewlett-Packard's constellation systems. A constellation system consists of a small number of high performance SMP nodes. It's been designed to take full advantage of the computational power of a single SMP node (as exhibited by the aforementioned SMP code) and to minimize performance loss incurred by the interconnect between the nodes.

Experimental data proved the look-ahead mechanism very successful in achieving high performance in constellation setting. Two variants of the algorithm are implemented: with look-ahead depth of 1 and 2; the latter achieving somewhat better performance. The lack of load balancing solution was determined by the types of systems that would be eligible for LINPACK Benchmark. These systems are mostly SMP nodes with differing number of CPUs. The CPU types, however, are the same across multiple nodes (this is also an assumption made in HPL). This substantially simplified the design of the algorithm. Each computational node runs the same number of threads. However, by assigning multiple computational nodes to a single SMP node, it is possible to balance the load in the system. For example, in a setting with two SMP nodes one with 64 CPUs and the other with 32 CPUs, the preferred configuration is to have two computational nodes on the 64-CPU machine and one node on the 32-CPU machine. Each node should run 32 threads so there is one-to-one mapping between threads and CPUs in the system.

Communication between nodes is performed by a separate thread using MPI. Standard MPI collective communication routine, namely the `MPI_Bcast()` routine, is used to communicate matrix data between nodes.

6 Tuning the Benchmark Codes

The performance obtained on a given system for the LINPACK NxN Benchmark heavily depends on tuning the code and system parameters. The code tries to choose appropriate values of blocking parameters which initially were obtained experimentally and now are used as default values. In real applications, the user is not able to choose them, though. However, they are chosen for the user by system libraries such as MLIB to obtain close to optimal performance. In the new benchmark code, it is still possible to change blocking parameters to (possibly) obtain a better performance a new system.

For the benchmark run it is beneficial for the matrix dimension to be a multiple of the panel width. This prevents running *clean-up code* which has lower performance than the code for matrix with appropriate dimension. Also, matrix dimension has to be large enough for the code to fully take advantage of techniques to hide memory and network latencies. These techniques are look-ahead and communication/computation overlapping. The effectiveness of these techniques deteriorates at the end of factorization as the parallelism inherent in the process decreases and there is less possibility to take advantage of them. Large matrices, however, require for the system to have enough main memory not to trigger the paging mechanism to slow the factorization.

For SMP codes (including the benchmark code) it is important to pay close attention to data shared by multiple threads. Correctness requires synchronization protocols to be employed to avoid multiple or incomplete accesses to these data. Efficiency, on the other hand, requires the accesses to be minimized. This can be done with algorithmic changes in the code as well as proper data layout which avoids cache-thrashing for logically independent data elements which happen to be physically neighboring each other in main memory.

Important consideration for memory-intensive codes is the proper handling of memory by CPU. This may be chosen at compilation or linking time. At compilation time, it has to be decided whether 32 or 64-bit addressing model should be chosen. While it seems obvious to always choose 64-bit mode as the 64-bit integer arithmetic is as fast as its 32-bit counterpart and it allows larger memory for a program, it might have negative influence on the performance for applications which store many pointers in memory (rather than address registers). A 64-bit pointer requires twice as much bandwidth to be fetched. In the benchmark code we have chosen the 64-bit model because it gives access to memory areas bigger than 4 GBytes. Another issue worth considering is the data page size. `chatr(1)` utility allows setting an appropriate page size for an existing executables. It is desirable for a portion of a matrix owned by a single CPU to be retained at all times in this CPU's Translation Look-aside Buffer (TLB) which on Hewlett-Packard's PA-RISC processors has usually 128 entries. Thus, for a matrix of dimension 108000, there is a total of some 89 GBytes of memory required to store it which translates to about 1.4 GBytes per each CPU. If one wants each CPU to retain such an amount of data in its TLB the data page size has to be set with the `chatr(1)` command to 16 MBytes.

In addition, there are also *kernel tunables* which can affect the performance, especially for memory-intensive applications. Two important kernel parameters: `data_size` and `stack_size` may limit the amount of memory that process has if they are not set properly. Also, the amount of swap space has to be at least as big as the available physical memory since otherwise the memory will be scaled down to the size of the swap space. Kernel's buffer cache can sometimes occupy excessive

amount of the main memory. For the benchmark code it was sufficient to have less than 3% of main memory to be assigned to buffer cache. This of course does not hold for applications with intensive file I/O use.

The important feature of a LINPACK benchmark code is a short completion time for a single run. This allows more tests to be performed in order to find optimal execution parameters. This is extremely important since usually machines, on which benchmarking is performed, allow only a short period of time of a dedicated access due to their heavy use. To facilitate it, the parts of the codes that are not timed (matrix generation, computation of norms and residual) are also optimized and are run in parallel where possible. The matrix generation part is particularly sensitive to optimizations in a multi-threaded MPI environment. It has to deliver large non-singular matrices. To meet all these goals, the original random number generator from the LINPACK benchmark code was modified so that each thread generates a globally unique random sequence of period 2^{45}. Computing the $\|Ax-b\|$ residual has to be done in a fashion which does not cause a *catastrophic cancellation* effect. This is especially viable in an MPI environment where the partial sums might be truncated while being transferred between higher precision floating point registers and memory when they get sent across the network.

7 Performance Results

In the tables throughout this section the standard LINPACK benchmark parameters are used as described in section 1. In addition, symbol F denotes the fraction of the theoretical peak of a machine that has been obtained by the benchmark code. Tests that used message passing were using HP MPI verion 1.7.

Tables 2 and 3 compare performance of two benchmark codes on Hewlett-Packard's SMP systems with two different processor architectures. Quite substantial improvement of performance for the new code can be attributed to ability to run much bigger matrices. This shows how important is large memory available for the benchmark run. However, the new code is still able to outperform the old one for the same matrix sizes: for the matrix of order 41000 it is able to deliver 121 Gflop/s as compared with 117 Gflop/s for the old code. This 4 Gflop/s difference may (to some extent) be atributed to the following:

- the old code does only look-ahead of depth 1 which might occasionally make some CPUs wait for a panel to get factored; the new code has multiple look-ahead strategies,
- the old code performs backward pivoting which is not necessary for a single right hand side,
- the old code performs two calls to `DTRSM()` function during solve phase as opposed to just one call in the new code,
- the old code's panel factorization is a blocked `DGEMM()`-based variant versus recursive one in the new code.

The other interesting aspect is the fact that the performance of the LINPACK 1000 Benchmark for a single CPU of SuperDome is 1.583 Gflop/s whereas for a 64-CPU LINPACK NxN benchmark it is 1.562 Gflop/s per CPU which translates into over 98% parallel efficiency.

No. of proc.	Old algorithm N_{max} [10^3]	Old algorithm R_{max} [Gflop]	Old algorithm F [%]	New algorithm N_{max} [10^3]	New algorithm R_{max} [Gflop]	New algorithm F [%]	R_{peak} [Gflop]
32	41	50.3	71	40	50.6	72	70.7
64	41	90.9	64	108	100.0	71	141.3

Table 2: Performance of SuperDome PA-8600 552MHz SMP system for LINPACK NxN Benchmark.

No. of proc.	Old algorithm N_{max} [10^3]	Old algorithm R_{max} [Gflop]	Old algorithm F [%]	New algorithm N_{max} [10^3]	New algorithm R_{max} [Gflop]	New algorithm F [%]	R_{peak} [Gflop]
32	41	65.4	68	75	67.3	70	96
64	41	117.0	61	120	133.2	69	192

Table 3: Performance of Caribe PA-8700 750MHz SMP system for LINPACK NxN Benchmark.

Tables 4 and 5 show performance results for the MPI code that runs the LINPACK NxN Benchmark for two kinds of interconnect. Table 4 shows the performance for the unsymmetric Caribe system where one of SMP nodes has 32 CPUs and the other has only 16. In such a setting, the code was run with two MPI nodes on the 32-CPU system and one MPI node on the 16-CPU machine. Table 5 compares the performance of two symmetric Caribe systems with two different MPI node assignments.

Table 6 presents comparisons of comunication parameters of the interconnects and protocols used in tests. HyperFabric is a high speed cluster interconnect product that enables hosts to communicate between each other at link speeds of 160MB/s in each direction. With longer cable lengths (greater than 35ft), the speed is limited to 80MB/s. HyperFabric technology consists of hi-speed interface cards and fast switches which use worm-hole routing technology for hi-speed switching. The latency of the core switching chip is 100ns. Lowfat is a specialized low-latency/high bandwidth optimized protocol implemented on top of the HyperFabric interconnect. Table 7 shows performance of an N-class system with different interconnects and protocols. A rather poor scalability of Lowfat protocol for four nodes should be attributed to much smaller matrices that has been factored on this system. The size could not be increased due to memory being occupied by the Lowfat protocol buffers.

Interconnect	N_{max} [10^3]	R_{max} [Gflop]	F [%]
Ethernet 100	90	90.9	63
Gigabit	92	97.9	68
System characteristics: 32-CPU Caribe + 16-CPU Caribe $R_{peak} = 144$ Gflop 3 MPI nodes × 16 CPUs = 48 CPUs			

Table 4: Performance of two unsymmetric Caribe PA-8700 750MHz SMP systems for LINPACK NxN Benchmark.

		2×32 CPUs		4×16 CPUs	
Interconnect	N_{max} [10^3]	R_{max} [Gflop]	F [%]	R_{max} [Gflop]	F [%]
Ethernet 100	90	108.8	57	106.4	55
Gigabit	92	126.4	66	126.8	66
System characteristics: 2×32-CPU Caribe $R_{peak} = 192$ Gflop					

Table 5: Performance of two symmetric Caribe PA-8700 750MHz SMP systems for LINPACK NxN Benchmark.

Interconnect (protocol)	Bandwidth [Mb/s]	Latency [μs]
Gigabit Ethernet	35-50	100-200
HyperFabric	79	57
Lowfat	100	33

Table 6: MPI performance over HyperFabric and Gigabit Ethernet interconnects and Lowfat protocol.

	2×8 CPUs		3×8 CPUs		4×8 CPUs	
Interconnect (protocol)	R_{max} [Gflop]	F [%]	R_{max} [Gflop]	F [%]	R_{max} [Gflop]	F [%]
Ethernet 100	15.0	53	19.3	46	21.6	38
HyperFabric	15.7	56	23.2	55	29.7	53
Lowfat	16.5	58	23.5	56	28.4	50
R_{peak}[Gflop]	28.2		42.2		56.3	
System characteristics: $\{2, 3, 4\}$×8-CPU N-class						

Table 7: Performance of up to four N-class PA-8500 440MHz SMP systems for LINPACK NxN Benchmark.

References

[1] J. Dongarra, J. Bunch, C. Moler and G. W. Stewart, LINPACK Users Guide, SIAM, Philadelphia, PA, 1979.

[2] J. Dongarra, Performance of Various Computers Using Standard Linear Equations Software, Technical Report CS-89-85, University of Tennessee, 1989. (An updated version of this report can be found at `http://www.netlib.org/benchmark/performance.ps`)

[3] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software **14**, pp. 1-17, March 1988.

[4] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, A set of Level 3 FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software **16**, pp. 1-17, March 1990.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. `http://www.mpi-forum.org/`.

[7] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/`.

[8] LINPACK Benchmark on-line FAQ: `http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html`.

[9] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, `http://www.netlib.org/benchmark/hpl/` (or `http://icl.cs.utk.edu/hpl/`).

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.

[11] F. Gustavson, Recursion leads to automatic variable blocking for dense linear-algebra algorithms, IBM Journal of Research and Development **41**(6), pp. 737-755, November 1997.

[12] K. R. Wadleigh and I. L. Crawford, Software Optimization for High-Performance Computing, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2000.

[13] HP MLIB User's Guide, VECLIB and LAPACK, First Edition, Hewlett-Packard Company, December, 1999.

[14] International Organization for Standardization. Information technology – Portable operating system interface

(POSIX) – Part 1: System Application Programming Interface (API) [C language]. ISO/IEC 9945-1:1996, Geneva, Switzerland, 1996.

```
for each panel p
begin
    Panel factoriazation phase
    if panel p belongs to this thread
    begin
        Factor panel p.
        Mark panel p as factored.
        Notify threads waiting for panel p.
    end
    else
        if panel p hasn't been factored yet
            Wait for the owner of panel p to factor it.

    Look-ahead of depth 1
    if panel p + 1 belongs to this thread
        begin
            Update panel p + 1 from panel p.
            Factor panel p + 1.
            Mark panel p + 1 as factored.
            Notify threads waiting for panel p + 1.
        end

    Update phase
    for each panel u such that:
        u > p and panel u belongs to this thread
    begin
        Update panel u using panel p.

    Look-ahead of depth 2
        if panel p + 2 belongs to this thread and
            it hasn't been factored and
            panel p + 1 has been factored
        begin
            Perform necessary updates to panel p + 2.
            Factor panel p + 2.
            Mark panel p + 2 as factored.
            Notify threads waiting for panel p + 2.
        end
    end
end
```

Figure 6: Thread-based panel factorization algorithm with a look-ahead of depth 2.

```
Let f be the first panel that belongs to this thread
    (not counting the first panel)
for each panel p
begin
    Panel factoriazation phase
    if panel p belongs to this thread
    begin
        Factor panel p.
        Mark panel p as factored.
        Notify threads waiting for panel p.
    end
    else
        if panel p hasn't been factored yet
            Wait for the owner of panel p to factor it.

    Update phase
    for each panel u such that:
        u > p and panel u belongs to this thread
    begin
        Update panel u using panel p.

        Look-ahead technique
        if panel f − 1 has been factored
        begin
            Perform necessary updates to panel f.
            Factor panel f.
            Mark panel f as factored.
            Notify threads waiting for panel f.
            f := f+number_of_threads
        end
    end
end
```

Figure 7: Critical path variant of thread-based panel factorization algorithm.

function dgetrf($A \in \mathbf{R}^{m \times n}; m \geq n$)
returns: $(\mathbf{R}^{m \times n} \ni) A' \equiv (L - I) + U$; where:
L - lower triangular (trapezoidal) matrix with unitary diagonal
U - upper triangular matrix
$P \cdot A = L \cdot U$
P - (row) permutation matrix
begin
    if ($A \in \mathbf{R}^{m \times 1}$) begin
        {*find largest element w.r.t absolute value*}
        $k = \texttt{idamax}(a_k := max(A_{:,1}))$;
        $a_0 :=: a_k$; {*swap elements*}
        {*scale column by pivot's reciprocal*}
        $\texttt{dscal}(A_{2:m,1} := \frac{1}{a_k} \cdot A_{2:m,1})$;
    else begin
        $\texttt{dgetrf}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}\right)$;
        {*apply pivoting sequence to trailing matrix*}
        $\texttt{dlaswp}\left(\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}\right)$;
        $\texttt{dtrsm}(A_{12} := L_{11}^{-1} \cdot A_{12})$;
        $\texttt{dgemm}(A_{22} := A_{22} - A_{21} \cdot A_{12})$; {*Schur's complement*}
        $\texttt{dgetrf}(A_{22})$;
        {*apply pivoting sequence*}
        $\texttt{dlaswp}(A_{21})$;
    end
end function dgetrf

Figure 8: Recursive LU factorization routine for double precision matrices.