Application of Fortran Pthreads to Linear Algebra and Scientific Computing

Clay P. Breshears (clay@turing.wes.hpc.mil)

Mark R. Fahey (mfahey@nrcmail.wes.hpc.mil)

Henry A. Gabb (gabb@ibm.wes.hpc.mil)

Abstract

Pthreads is a POSIX standard library for expressing concurrency on uniprocessor and symmetric multiprocessor computers. Typical multithreaded applications include database manipulation, operating systems, or any algorithm displaying task-level concurrency. These types of programs are generally coded in C. Hence, the POSIX standard only defines a C interface to Pthreads. The lack of a standard Fortran interface has limited the use of Pthreads for scientific and numerically intensive applications. However, many of these applications would benefit from multithreading. For example, the manipulation and solution of dense linear systems with 10,000 to 40,000 simultaneous equations are made up of many independent tasks that may be executed concurrently. The effectiveness of Fortran and Pthreads for scientific programming will be demonstrated using two linear algebra routines (matrix multiplication and Gaussian elimination) and two algorithms from the Command, Control, Communication, and Intelligence Benchmark. This work builds on the Fortran interface to Pthreads presented at the 40th Cray Users Group Conference.

Introduction

Threads is a programming model for expressing concurrency on uniprocessor or symmetric multiprocessor (SMP) computers (Ben-Ari, 1982). Each thread is a separate execution stream with its own stack and instruction counter. Typically, a thread represents an independent task or function. Though these tasks execute concurrently, they do not necessarily execute in parallel, even on a SMP. It is up to the operating system to perform thread scheduling. Threads exist within a single process, sharing the resources (memory, files, I/O channels) of the spawning process. This relationship confers better resource utilization when running threaded applications.

Several thread standards are available: Microsoft Win32, DCE, Sun Solaris, OpenMP, and POSIX. Pthreads is the most common and most portable standard (Nichols *et al*., 1996). It is available for nearly all UNIX dialects. DCE threads is based on an earlier draft of the Pthreads standard (Lewis and Berg, 1998). Solaris threads are very similar to Pthreads in terms of functionality. However, the two standards differ significantly in the way thread properties are set (Robbins and Robbins, 1996). OpenMP (OpenMP ARB, 1997), the newest standard for multithreading, though capable of expressing task-level concurrency, is designed for loop-level parallelism.

The purpose of this work is not to develop a Fortran and Pthreads combination superior to other high performance computing methods or algorithms. Rather, this work illustrates that scientific applications display functional concurrency and may benefit from multithreading. Fortran is the traditional language of scientific programming; Pthreads is designed to express task-level, or functional, concurrency. The POSIX standard, however, does not define a Fortran interface to Pthreads.

A Fortran 90 application programming interface (API) to the Pthreads library was proposed last year (Gabb *et al*., 1998; Breshears *et al*., 1998). The present work demonstrates the efficacy of this API for scientific computing and numerical analysis. Multithreaded matrix multiplication and direct solution of linear systems of equations demonstrate the latter. Two algorithms from the Command, Control, Communication, and Intelligence (C3I) benchmark (Metzger *et al*., 1996; Brunett *et al*., 1998; Pires and Elder, 1998) demonstrate the former.

Fortran Pthreads Applications

Linear Algebra

Matrix Multiplication

Matrix-matrix multiplication is one of the most studied algorithms in numerical computation. The simplest algorithm is a triply nested loop structure:

```
do i = 1, N
   do j = 1, N
     do k = 1, N
       C(i,j) = C(i,j) + A(i,k) * B(k,j)
     end do
   end do
end do
```

The ordering of the loops is arbitrary and leads to different memory access patterns.

Our first attempt at threaded matrix multiplication used sparse matrices. To conserve memory by not storing zero entries, a linked-list scheme (Duff *et al*., 1989) was used in place of the standard 2-D array layout. This method employs four single-dimensioned arrays: 1) VALUE holds the nonzero data values, 2) JCN holds the column index of the corresponding element in VALUE, 3) LINK holds the index of the next element within the row, and 4) IROWSTART holds the index (in VALUE) of the first nonzero element within each row. Elements from a given row can be found by starting at the element in VALUE indexed by the appropriate entry in the IROWSTART array and following the LINKs. A special end-of-list value (say, -1) is stored in LINK of the last nonzero element within the row.

While Fortran uses column-major ordering of data, the linked-list scheme lends itself to row-major computation. Thus, the IKJ loop ordering was considered first. Rewriting the innermost statement of the loops in Fortran 90 array syntax gives the following:

```
C(I, :) = C(I, :) + A(I, K) * B(K, :)
```

Under this execution, the Ith row of C is generated by the summation of scalar multiples of rows from B. These scalars are taken from the Ith row of A; i.e., the Kth element in the Ith row of A is multiplied by the Kth row of B. All data is accessed in row-major order. It should be noted that the sparse matrix data structure could have stored values in a column-major order with the second array holding row numbers of elements. However, because the actual implementation uses only one-dimensional arrays, unless there was some matrix structural reason there would have been no practical difference.

The threaded algorithm dynamically assigns rows of matrix C to individual threads using a shared counter variable to indicate the next available row for processing. After a row is assigned, threads repeatedly copy relevant rows of B into a local vector (filling in the zeros) and multiplying this local vector by the appropriate scalar from A. Of course, only those rows from B that correspond to a nonzero value in the row of A under consideration need to be copied to the local vector. The results of the vector scaling are added to a local summation vector, which, after all scalars from the relevant row of A have been processed, is added to the row of matrix C.

Access to the data structure holding matrix C must be mutually exclusive. For those nonzero elements in the summation vector that correspond to nonzero elements in the original row of C there is no possibility of two threads attempting to store a value to those locations because each row is handled by a single thread. However, new nonzero elements from the summation vector must be stored within the data structure arrays and linked correctly. Because two or more threads may be simultaneously updating previous zero positions within the C matrix, only a single thread must be allowed to add new nonzero elements to the C matrix at any time.

Timing results (in seconds) from test cases run using 1000 x 1000 matrices each holding just under 10,000 nonzero elements on a SGI/Cray Origin2000 system (IRIX 6.4) are shown in Table 1.

# of Threads	Time (seconds)
1	0.259
2	0.261
4	0.261
8	0.264
16	0.270
32	0.301
64	0.324
128	0.364

Table 1. Sparse matrix multiplication (1000 x 1000 with almost 10,000 nonzero elements) on a 112-CPU SGI/Cray Origin2000 (IRIX 6.4)

# of Threads	IKJ Time (seconds)	JKI Time (seconds)
1	75.5	29.8
2	42.4	20.2
4	22.4	11.2
8	11.9	6.1
16	6.3	3.4
32	3.6	2.0
64	3.2	1.9
128	3.0	2.3

Table 2. Dense matrix multiplication (1000 x 1000) on a 112-CPU SGI/Cray Origin2000 (IRIX 6.4)

The results for threaded, sparse matrix multiplication are somewhat disappointing. There is simply not enough work to keep all threads busy. Also, the overhead of creating threads and memory contention increases as the number of threads increases for a fixed-size problem.

The threaded sparse matrix multiply algorithm was adapted for use on dense matrices. The algorithm remained the same for the dense version as was used for the sparse version with one alteration: the update of each row of the C matrix by the local summation vector need not be made exclusive. Because each row of C is being computed by a unique thread and all the memory space needed to hold the matrix has been allocated in a two-dimensioned array, it can be guaranteed that no two threads will attempt to write into the same memory locations.

Because the dense matrices are now represented as 2-D arrays, memory access patterns must be taken into account. From the triply-nested loop algorithm, the JKI loop ordering is well-suited. Writing the innermost statement of this matrix multiplication algorithm with the inner loop collapsed with Fortran 90 array syntax yields:

```
C(:, J) = C(:, J) + A(:, K) * B(K, J)
```

This loop ordering updates the Jth column of C by the summation of scalar multiples of columns from A. These scalars are taken from the Jth column of B; i.e., the Kth element in the Jth column of B is multiplied by the Kth column of A. Thus, the second threaded, dense matrix multiplication algorithm dynamically assigns columns of C to unique threads which are updated by the local summation of scalar multiples of columns from A. Timing results for both threaded, dense matrix codes are shown in Table 2.

Two things are apparent from this data. First, using the correct data access pattern (in this case, column-major order of multi-dimensioned arrays) is essential for achieving the best performance. Second, the efficacy of using Pthreads is demonstrated by the almost linear speed-up shown as the number of threads are increased. Here again, though, thread overhead and memory contention begin to overshadow execution time when too many threads are used on a relatively small, fixed problem size.

Though not covered here, other matrix-matrix multiplication algorithms worth considering for threading are the parallel Cannon's algorithm (Kumar *et al.*, 1994) and Strassen's algorithm (Strassen, 1969).

Gaussian Elimination and Back Substitution

Solving systems of linear equations is an important numerical computation that is an integral part of the solution to a variety of problems. One simple, direct method is Gaussian elimination with back substitution. During the elimination phase of the algorithm, the update of each row below the current one can be done concurrently. Thus, as was done in matrix multiplication, the row update is mapped to threads.

As the algorithm progresses through the elimination steps, rows above the current row are never again accessed. In order to ensure that the execution is properly load-balanced, roughly equal numbers of rows should be updated by each thread. This load balancing was accomplished through dynamic scheduling of rows to threads in the matrix multiplication codes. This type of scheme would work for Gaussian elimination. However, because each row update has the same amount of work involved within each elimination step, it was thought that access to a shared counter might cause an undesirable amount of overhead. A much simpler division of work was needed.

For Gaussian elimination, the rows of the matrix are statically divided in a cyclic manner. For N threads, the first thread is assigned rows 1, N+1, 2N+1, …; the second thread is assigned rows 2, N+2, 2N+2, …; and so on. As the I^{th} row becomes the current row, the thread assigned to that row moves the diagonal element to the I^{th} element of a global pivot vector while all other threads wait for this pivot element to be stored. After the pivot value is found each thread copies the current row into a local vector in order to apply a scaled multiple of this row to all other rows assigned to the thread. Using this local vector to compute the scaling vector cuts down on memory contention to the global matrix because this scaling vector will be computed many times by each thread. After all the assigned rows have been updated, threads determine if the $(I+1)^{th}$ row has been assigned to it and, consequently, pull out the next pivot value. Otherwise, threads wait for the pivot to be assigned into the proper pivot vector position.

For the back substitution phase, each thread is again assigned the same rows as in the elimination phase. Working backwards, threads wait for all preceding elements of the solution vector to be computed. After computing the assigned solution vector element, a thread waits for the time when it can calculate the next element of the solution. The coordination of this phase is done slightly differently than was done when determining the current row in the elimination phase. The pivot vector was initialized with zero. Because no pivot value can be zero (for a nonsingular matrix), threads need only test the current pivot value for a nonzero value to know when the new pivot value had been stored by a thread. Because any element of the solution vector could be zero, this method cannot be used for the coordination of threads in the back substitution. In this case, a global index is used. The index is decremented by the thread that computes the corresponding element of the solution vector. Other threads simply wait until this index contains the value corresponding to one of the assigned rows. The solution vector element corresponding to the index is then computed. Once the global index reaches zero, the algorithm is complete and all threads exit.

Timing results for multithreaded Gaussian elimination of a linear system of 2000 equations is shown in Table 3. As expected, a row-major algorithm does poorly when coded in Fortran. The algorithm was converted to column-major by transposing the matrix (using the Fortran 90 TRANSPOSE intrinsic) and interchanging the indices [i.e., A(I,J) to A(J,I)]. The number of threads does not necessarily correspond to the number of processors used by the program. The operating system decides where to execute threads. The Phreads standard does not provide the programmer with a means to map threads to processors. Two non-standard functions, `pthread_setconcurrency` and `pthread_getconcurrency`, are included in the IRIX Pthreads implementation. On SGI systems, these functions are often necessary to achieve better parallelism for Pthreads programs. Notice the performance degradation for two threads when `pthread_setconcurrency` is not used (Table 3). Both threads may be competing for the same processor or the threads may be competing for memory bandwidth on the same SGI Origin2000 node.

A larger system of 10000 equations was also solved. The best performance was achieved for the transpose column-major algorithm using 32 threads: 2.8 hours without pthread_setconcurrency and 2.4 hours with `pthread_setconcurrency`.

# of Threads	Row-Major Time (seconds)	Transpose Column-Major Time (seconds)	Transpose Column-Major with pthread_setconcurrency Time (seconds)
1	672	162	162
2	735	232	125
4	556	115	68
8	1030	100	26
16	1642	147	53
32	1667	131	64

Table 3. Gaussian elimination with back substitution for a double precision linear system of 2000 equations on a 112-CPU SGI/Cray Origin2000 (IRIX 6.5).

Command, Control, Communication, and Intelligence (C3I) Benchmark

Map-Image Correlation

Suppose, for example, that a harbor is under surveillance. A satellite photo reveals three ships docked in the harbor. A few hours later, a spy plane flying over the area from a different angle takes a new picture. In order to extract useful information from these surveillance images, it is necessary to align image features with a map of the area of interest. Accurate alignment depends upon finding the correct translation and rotation of the image with respect to the map. In the C3I benchmark, this is known as map-image correlation (Metzger *et al*., 1996).

In this simplified example, the map (f_M) and image (f_I) are digitized in 2-D grids with every node (l,m={1…N}) assigned a value:

$$f_{M_{l,m}} = \begin{cases} 1 & on \\ -1 & off \end{cases}$$

and

$$f_{I_{l,m}} = \begin{cases} 1 & on \\ -1 & off \end{cases}$$

depending on whether or not the pixel associated with that node is illuminated. More detailed analyses might digitize based on color, temperature, or elevation.

The correlation function of map f_M and image f_I is:

$$f_{C_{i,j}} = \sum_{l=1}^{N_1} \sum_{m=1}^{N_2} f_{M_{l,m}} \times f_{I_{l+i,m+j}}$$

where N_1 x N_2 is the number of grid points and i,j is the translation vector of the image with respect to the map. High correlation denotes similarity between the overlapping portions of the map and image.

Calculating the correlation function by direct summation is very inefficient, requiring N_1 x N_2 multiplications and additions for every N_1 x N_2 shifts i,j. Because the discretized map and image are discrete functions, f_M and f_I, one can calculate f_C more rapidly using the fast Fourier transform (FFT). The FFT requires on the order of (N_1 x N_2) ln (N_1 x N_2) calculations, which is fewer than $(N_1 \times N_2)^2$.

The Fourier correlation algorithm is shown schematically in Figure 1 (adapted from Gabb *et al*., 1997). Briefly, the discrete functions f_M and f_I are transformed (denoted DFT for discrete Fourier transform) and the complex conjugate F_M^* and F_I are multiplied:

$$F_M = DFT(f_M)$$
$$F_I = DFT(f_I)$$
$$F_C = (F_M^*)(F_I)$$

The correlation function describing the optimum overlap for given orientations of f_M and f_I is the inverse Fourier transform (IFT) of the transform product:

$$f_C = IFT(F_C)$$

(See Press et al., 1996 and Bracewell, 1990 for thorough reviews of fast transforms and Fourier correlation.)

The Fourier correlation algorithm exhibits two levels of concurrency (Figure 1). First, a 2-D FFT is commonly implemented as a collection of independent 1-D FFTs, which can be mapped to threads. Second, the rotational scan of image orientations with respect to the map consists of independent, and therefore concurrent, iterations. The ability to easily express nested parallelism gives Pthreads an algorithmic advantage over higher-level thread methods (e.g., OpenMP).

A threaded 2-D FFT was developed to test the efficacy of Fortran and Pthreads to the map-image correlation benchmark. The 2-D FFT was coded as a collection of 1-D FFTs. The nonthreaded 1-D FFT routines (scfft and csfft) from the Cray Scientific Library were used. The Cray Scientific Library contains a threaded 2-D FFT. However, the purpose of this exercise is to create a 2-D FFT in which the smallest functional unit (i.e., 1-D FFT) is mapped to threads. Because Fortran is column-major, the individual columns of the first dimension are transformed, the matrix transposed, and the columns are transformed again. The columnar transforms are mapped to threads. The order of the column transforms is irrelevant and threads never compete for the same memory locations. Therefore, synchronization is only required after each array dimension is transformed. The timing results for the Fourier correlation of two 1024 x 1024 grids shown in Table 4 demonstrate good parallel speed-up. However, the non-threaded 2-D FFT routines in the Cray Scientific Library can correlate two 1024 x 1024 matrices in less than two seconds. So, there is an obvious advantage to using an optimized FFT library. This also indicates that coding a 2-D FFT as a collection of 1-D FFTs is not an efficient algorithm.

Terrain Masking

Terrain masking is the process of determining whether a location or altitude is visible from any of a set of vantage points, called threats, or whether it is obscured, or “masked,” by intervening terrain (Schultz *et al*., 1995; Pires and Elder, 1998) (Figure 2). The threats for this type of problem could be light sources, radio towers, antennae, radar receivers, etc. The threats are effective only in their sensor range. Terrain masking computes the safe altitude, called the masking altitude, at each grid point. Aircraft flying above this altitude can be observed by at least one enemy sensor.

Terrain masking is an important application in ground-based C3I systems and flight planning computers. To support real-time mission planning, terrain masks must be computed as quickly as possible. Terrain masking takes as input a relief map of a region (i.e., 2-D grid of surface elevations), and the position and range of a set of threats located within the region. Output consists of the original map plus masking altitudes. (Terrain masking output is used as input to another method in the C3I benchmark called route optimization, which also contains opportunities for concurrency.)

At each grid point within range of a threat, the masking height associated with that threat is computed. The final masking height at each grid point is the minimum of the computed masking altitudes associated with each threat covering the given grid point. For a given threat, the masking altitude at each grid point is computed by a line-of-sight calculation.

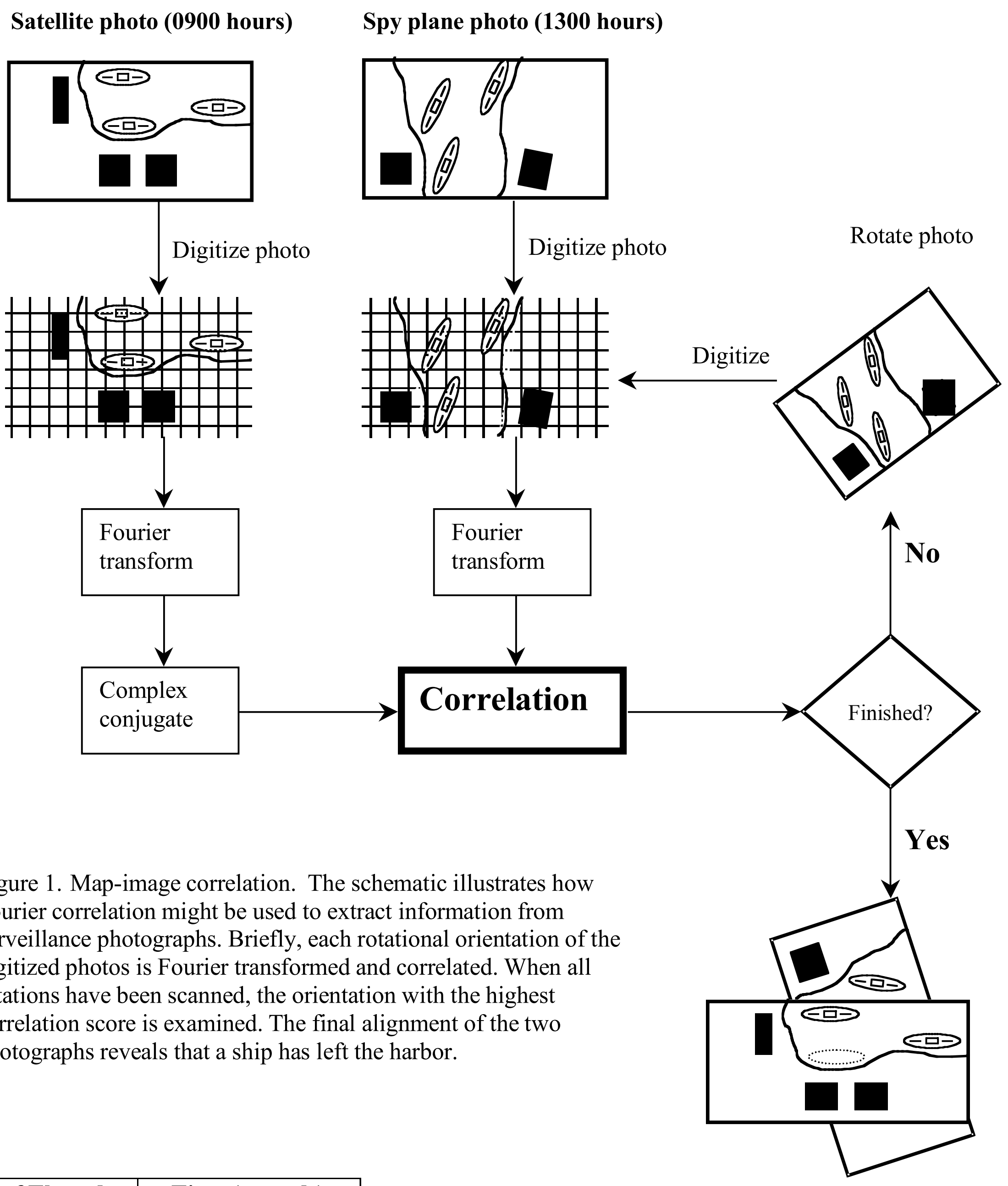

Figure 1. Map-image correlation. The schematic illustrates how Fourier correlation might be used to extract information from surveillance photographs. Briefly, each rotational orientation of the digitized photos is Fourier transformed and correlated. When all rotations have been scanned, the orientation with the highest correlation score is examined. The final alignment of the two photographs reveals that a ship has left the harbor.

# of Threads	Time (seconds)
1	155
2	85
4	47
8	28
16	19
32	14
64	16

Table 4. Fourier correlation of two 1024 x 1024 grids on a 112-CPU SGI/Cray Origin2000 (IRIX 6.5) using pthread_setconcurrency.

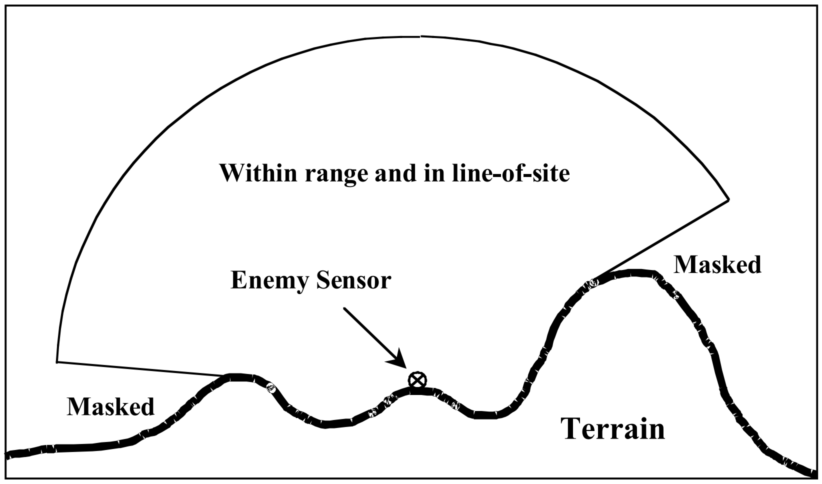

Figure 2. Terrain masking.

Terrain masking contains task concurrency, it is also computationally intensive. Therefore, a Fortran and Pthreads combination is well-suited to this algorithm. There are two opportunities for concurrency. First, each threat's area of coverage can be divided into N equal-sized sectors with a thread mapped to each. Load balancing is near ideal, because each thread will received an equal part of the threat's computation. This approach is simple to code because there is no memory contention between threads. However, it is not scalable. Large values of N can be used, but as the number of threads increases the area (and the amount of work) in each sector decreases. For a 6000 x 6000 terrain grid and 90 threats where at most only three threats are nonoverlapping, serial execution gives a baseline time of 27 minutes. Using four threads, one for each quadrant, the terrain mask takes 11 minutes to calculate (a 2.5-fold speed-up).

A second approach to concurrent terrain masking maps threads to threats rather than sectors. This introduces additional complexity because the areas of influence for threats almost always overlap. Race conditions, in this case write-write conflicts, are possible if multiple threads simultaneously access the same memory location (i.e., grid nodes in overlapping regions). Masking heights in a column are computed via a line-of-sight calculation using the masking heights from the previous column. Again, race conditions, in this case read-write conflicts, are possible if one thread is reading a column while another thread is writing to it. Memory contention is avoided using mutual exclusion (mutex) variables. A mutex variable is associated with every column of the grid. Whenever a thread processes a column of the grid, it first locks the two associated mutex variables before proceeding.

This algorithm scales with the number of threats rather than sectors. However, the overhead associated with mutual exclusion limits parallel efficiency as the number of threads increases. Timing results for this threaded algorithm are shown in Table 5.

# of Threads	**Time (minutes)**
1	25
2	14
4	8
8	5.5
16	3.5

Table 5. Terrain masking for a 6000 x 6000 grid containing 90 threads.

Conclusions

The results of this study show the applicability and effectiveness of Fortran and Pthreads to a range of applications. Each of the test problems is easy to thread once the functional unit of concurrent computing is determined. Parallel speed-up is demonstrated for each of the test problems. Excessive synchronization does degrade the parallel performance of a Pthreads program. However, this is not unique to Pthreads.

Pthreads expresses concurrency by mapping functions to threads. The programmer must therefore understand the algorithm being coded in order to identify functional concurrency and to compartmentalize work into concurrent functions. High-level thread methods like OpenMP concentrate on loop-level parallelism. The programmer need only find the work-intensive loops and add compiler directives. In this sense, OpenMP is superior to Pthreads. However, none of the currently available OpenMP implementations currently supports nested parallelism. Expressing multiple levels of concurrency is relatively simple using Pthreads.

Acknowledgements

This work was funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation.

References

Ben-Ari M., Principles of Concurrent Programming, Prentice-Hall International, 1982.

Bracewell R.N., "Numerical Transforms," *Science*, 248, 697-704, 1990.

Breshears C.P., Gabb H.A., Bova S.W., "Towards a Fortran 90 Interface to Pthreads," DoD User Group Proceedings, 1998.

Brunett S., Thornley J., and Ellenbecker M., "An Initial Evaluation of the Tera Multithreaded Architecture and Programming System Using the C3I Parallel Benchmark Suite," SC98 Technical Papers, 1998.

Duff I.S., Erisman A.M., and Reid J.K., Direct Methods for Sparse Matrices, Oxford University Press, 1989.

Gabb H.A., Bording R.P., Bova S.W., Breshears C.P., "A Fortran 90 Application Programming Interface to the POSIX Threads Library," 40th Cray User Group Conference Proceedings, 1998.

Gabb H.A., Jackson R.M., and Sternberg M.J.E., “Modelling Protein Docking Using Shape Complementarity, Electrostatics and Biochemical Information,” *Journal of Molecular Biology*, **272**, 106-120, 1997.

Kumar V., Grama A., Gupta A., and Karypis G., Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin/Cummings Publishing Company, Inc., 1994.

Lewis B. and Berg D.J., Multithreaded Programming with Pthreads, Sun Microsystems Press, 1998.

Metzger R.C. *et al*., "The C3I Parallel Benchmark Suite - Introduction and Preliminary Results," SC96 Technical Papers, 1996.

Nichols B., Buttlar D., and Farrell J.P., Pthreads Programming, O'Reilly and Associates, 1996.

OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," Version 1.0, October 1997.

Pires L. and Elder A., "Effective Parallelization of Terrain Masking Applications," Proceedings of the Second IASTED International Conference on Parallel and Distributed Computing and Networks, 1998.

Press W.H., Teukolsky S.A., Vetterling W.T., and Flannery B.P., Numerical Recipes in Fortran, Cambridge University Press, 1986.

Robbins K.A. and Robbins S., Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading, Prentice Hall PTR, 1996.

Schultz R., Jha R., Pires L., "Terrain Masking, Technical Information Report (A006)," Honeywell Technology Center, Minneapolis, MN, 1995.

Strassen V., "Gaussian Elimination is not Optimal," *Numerical Mathematics*, **13**, 354-356, 1969.