EXPERIMENTS WITH MULTITHREADING IN PARALLEL COMPUTING

Leonardo Jelenković, Goran Omrčen-Čeko
Faculty of Electrical Engineering and Computing, University of Zagreb
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Unska 3, 10000 Zagreb, Croatia
leonj@zemris.fer.hr, goc@zemris.fer.hr

Abstract: *This paper presents experiments with multithreading in several computationally intensive examples in which the time of parallel, multithreaded execution is significantly shorter than sequential, single-threaded. Algorithms used in these examples are not fully optimized for that wasn't the goal. The intention was to find out multithreading mechanism and structure and how to improve the performance on a multitasking, multi-user and multiprocessor operating system. The target operating system was SunOS™ 5.5.1 on two-processor workstation Ultra sparc 2.*

Keywords: multithreading, LU factorization, simulated annealing, sorting, task scheduling

1. Introduction

The word multithreading can be translated as many threads of control. While a traditional process always contains a single thread of control, multithreading separates a process into many execution threads, each of which runs independently.

The main benefits that arise from multithreading are:

- improved application responsiveness and better program structure - any program in which many activities do not depend upon each other can be redesigned so that each activity is executed as a thread,
- efficient use of multiple processors - numerical algorithms and applications with a high degree of parallelism, such as matrix multiplication, can run much faster when implemented with threads on a multiprocessor,
- use fewer system resources - the cost of creating and maintaining threads is much smaller than the cost for processes, both in system resources and time.

All threads created from the same initial thread (standard process) exist as a part of the same process, sharing its resources (address space, operating system state ...). Beside that the multithreaded applications use fewer system resources than multiprocess applications, communication between threads can be made without involving the operating system, thus improving performance over standard inter-process communication. From these reasons multithreading is so populat today, and modern operating systems support it.

Multithreading also brings some problems, like signal handling, function safety under possible parallel threads execution (parallel use and change of global variables), alarms, interval timers and profiling. The problem is how to change this process oriented and defined elements to support threads and to be defined on a thread level. One of the main problems in this work was a time measurment (real, user and system time) for a single thread in a multithreaded application. This was resolved using specific system dependent features (interval timers provided for each lightweight process [1]).

This work presents several examples of algorithms that can be implemented using threads with low dependencies between them. Experimental results are shown for a various number of threads and compared with single-threaded results (speedup). All programs were made using C programming language linked with thread library. Measurements were made on a low loaded system with no other users logged on. The first example is matrix multiplication presented in section 2. LU factorization is presented in section 3, Q-sort in section 4, TSP in Section 5 and simulator for a task system in Section 6.

2. Matrix Multiplication

Matrix multiplication is a good example of computationally intensive operation that requires $M \times N \times P$ operations of multiplication, where $M \times N$ is the size of the first matrix $\underline{A}$ and $N \times P$ of the second matrix $\underline{B}$. The number of multiplications grows with the third power of the problem size (assuming $M \cong N \cong P$). This operation demands long computation time for slightly greater problems and that is why there are plenty of methods for parallelizing this operation. However, the method presented here is very simple and, what is very important, the communication between threads is minimal. Basic idea is that each thread computes the entire row of the product matrix $\underline{C}$ and then checks for the next non-computed row and computes it, if such row exists. The synchronization is obtained using mutually exclusive locks provided for multithreading.

Each thread executes the same code which is outlined below:

```
procedure thread;
  lock;
  while lastrow < M do
    my_row := lastrow;
    lastrow++;
    unlock;
    for i:=1 to P do
      C(my_row,i) := 0;
      for j:=1 to N do
        C(my_row,i) := C(my_row,i)+A(my_row,j)×B(j,i);
    lock;
  unlock;
end.
```

Variable `lastrow` is a global variable, initially set to `1`. It points to the next non-computed row, and it is protected from simultaneous use from different threads. This is done with exclusive locks which in that way determine thread dependencies. Penalties for exclusive locking and thread creation are significant and comparable to the computation time when the dimensions of matrices are relatively small. As the matrix size increases the resulting overhead gets smaller.

Experimental results for $M=N=P=100$ are given in Fig. 1. As results show, speedup (execution time of one-threaded sequentional algorithm compared with parallel multithreaded algorithm) is almost independent of the number of threads when that number is equal or greater than the number of system processors, which are actually responsible for speedup. In the case of two processors the computation time is almost two time shorter and shows a very slow decrease when the number of threads increases. Reasons for this decrease are heavier thread communication and context switching between multiple threads.

#threads	t_{100}(s)	t_{200}(s)	t_{500}(s)
1	0.12	0.995	40.14
2	0.062	0.500	20.27
4	0.062	0.501	20.34
8	0.064	0.504	20.47

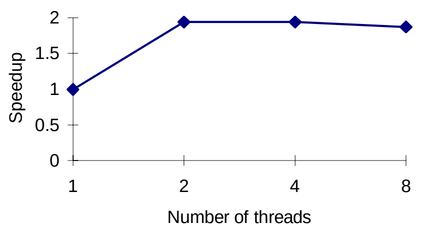

Figure 1: Matrix multiplication results

3. LU Factorization

LU factorization is a common algorithm used for solving systems of linear equations. This system can be presented in a matrix form:

$$\underline{\mathbf{A}}\,\underline{\mathbf{x}} = \underline{\mathbf{b}}\,,$$

where $\underline{\mathbf{A}}$ is the coefficient matrix, $\underline{\mathbf{x}}$ is the unknown vector, and $\underline{\mathbf{b}}$ is the excitation vector. With LU factorization, the original coefficient matrix is factored, or decomposed, into the product of a lower-triangular matrix $\underline{\mathbf{L}}$ and upper triangular matrix $\underline{\mathbf{U}}$. To attain a unique decomposition, the dialog terms of $\underline{\mathbf{L}}$ or $\underline{\mathbf{U}}$ are set to unity. After the decomposition is performed, the solution is determined by a forward substitution step and a backward substitution.

The proposed parallel algorithm is based on the independence of inner loops of a sequential algorithm which can run in parallel. In this algorithm, one thread (main) must take care of synchronization after a sub-matrix is computed, waiting for all threads to terminate their executions in a current sub-matrix. Then the same thread gives a signal to proceed with a new sub-matrix. All other threads execute the same code.

The algorithm is listed below:

```
procedure thread_slave;
  lock;
  while i < N do
    wait_for_start;
    while j ≤ N do
      my_j := j;
      j++;
      unlock;
      A(my_j,i) := A(my_j,i)/A(i,i);
      for k:=i+1 to N do
        A(my_j,k) := A(my_j,k)-A(my_j,i)×A(i,k);
      lock;
    signal_end;
  unlock;
end.
```

Variable `i` is a global variable that shows the actual step of the algorithm and actual sub-matrix. `i` is changed only by the main thread. Variable `j` is also a global variable that shows the next row of a current sub-matrix which is not computed yet. In each step the main thread initializes this variable. Results are presented in Fig. 2.

#threads	t_{100}(s)	t_{200}(s)	t_{500}(s)
1	0.053	0.387	6.504
2	0.058	0.333	4.181
4	0.082	0.354	4.321
8	0.162	0.402	4.642

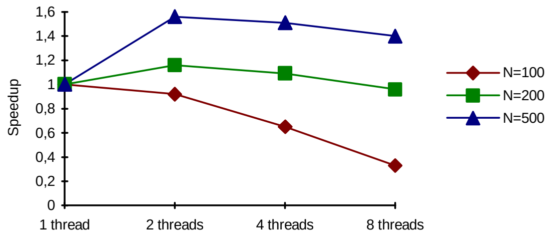

Figure 2: LU Factorization results

Speedup is reached only for matrices with dimension greater than 200×200. The reason for this is the synchronization time between threads, especially at the end of each step, when all threads must be waited, and then started again. If matrices are smaller than 200×200 this synchronization time is comparable with computatio time for one sub-matrix. Results also show that speedup decreases as the number of threads increases over the number of available processors. Optimum is reached when the number of threads is equal to the number of processors.

4. Q-sort

Sorting a large array of data is a computationally demanding operation which can be efficiently parallelized, resulting in shorter sorting time. Proposed Q-sort (quick sort) algorithm is just an experiment of parallel sorting. Original Q-sort is modified in a way such that the sorting is performed by more than one thread. Sorting begins with one initial thread, like in a sequential algorithm, dividing the array into two parts which elements are smaller (one part) and greater (other part) than one middle element. Sequential algorithm continues recursive divisions while the proposed algorithm creates a new thread after every division, only if the parts are large enough so that the thread creation time is not greater than sorting time.

Arrays sorted in this example were arrays of strings with a constant length. First array contains 20000 elements and each element is 70 characters long, second 50000 elements with 50-characters strings, and third 100000 elements with 40-character strings. Results are presented in Fig. 3.

#threads	t_1(s)	t_2(s)	t_3(s)
1	0.546	1.28	2.5
2	0.352	0.808	1.52
4	0.313	0.725	1.34
8	0.313	0.733	1.37

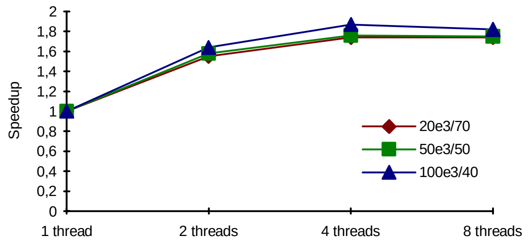

Figure 3: Q-sort results

Speedup achieves maximum with four threads. This is due to the nature of the algorithm because it limits the number of simultaneously running threads. Speedup increases as the problem size increases both in the number and the size of elements.

5. TSP

Traveling salesman problem (TSP) is a well known problem of combinatorial optimization. TSP is a problem where the salesman has to find the shortest path while visiting the *n* cities once each. A highly efficient method for this type of problems is *simulated annealing* [3]. A major hurdle in simulated annealing, however, is the long computation time due to its sequential nature of the slow annealing process. Parallel simulated annealing method, that follows the same decision sequence as the sequential method, computes several speculative iterations in parallel. Iteration with accept status and with the lowest iteration number is accepted. Results of all computations with higher iteration number are not valid and must be recomputed in each new configuration at the next computation level.

Used algorithm was originally presented by Sohn [2] for a large scale massively parallel distributed-memory multiprocessor, where a 20-fold speedup on 100 processors was obtained. We implemented this algorithm using threads. The execution time and the number of computed levels for different number of threads for 100 cities are shown in Fig. 4.

#threads	#levels	t_{REAL} (s)
1	233310	1.59
2	135462	15.17
4	88403	36.21
8	66337	108

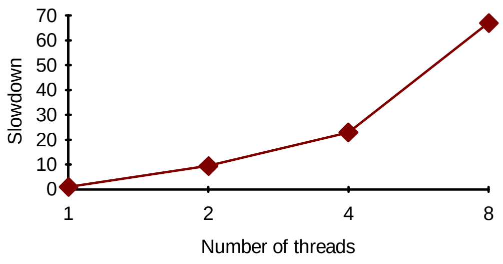

Figure 4: TSP results

Although the number of computed levels notable decrease there is no speedup. In fact Fig. 4 shows that multithreaded version is many times slower than the sequential algorithm. What is the reason for that? If we take a closer look at the table in Fig. 4 we can see that the serial algorithm (1 thread) computes 233310 iterations in 1.59 seconds, including changes. One iteration is computed in less than 7 µsec. This computation time must be the same in the multithreaded version (for each thread). After computing one level of iterations, in each thread, one result is chosen and a change is made. The next level then starts with a new configuration. Communication and synchronization time is obviously much greater than the computation time of one iteration, so no speedup can be reached. Also when there are more threads than processors, context switching time cannot be ignored if the computation time is so small.

This example indicates that the use of multithreading in algorithms with short computation time and heavy communication between threads brings no speedup but only slowdown.

6. Simulating a Task System

With the task system simulator all mentioned characteristic of multithreaded applications in previous sections show up. The used task system, shown in Fig.5, is constructed from ten tasks, with dependencies that permit some parallelism.

Simulations were performed in three steps. The first step is a sequential execution of tasks, like in the uniprocessor system. The second step is a parallel multithreaded simulation with one thread for each task. Threads are then scheduled by operating system and synchronized by locks and the conditions variables. Third step is also a multithreaded parallel simulation but with the same number of threads and processors. These threads execute tasks according to their structure with a simple scheduling. The simulator can also execute the task system continuously if the dependencies

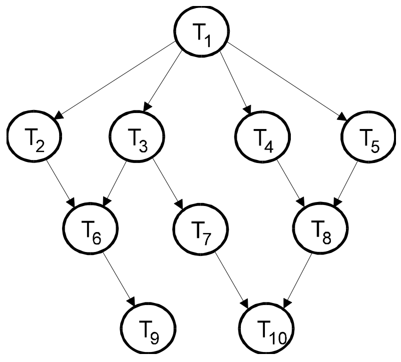

Figure 5: Task system

between the last and the first task are defined.

Simulation is done for four different numbers of iterations: 1, 10^2, 10^3 and 10^4, but the sum of operations per task is kept constant (10^7). Execution time and speedup are shown in Fig.6.

#iterations	t_{seq}(s)	$t_{task\text{-}thread}$(s)	$t_{thread\text{-}proc}$(s)
1	12.53	7.01	7.52
100	12.61	8.23	8.09
1000	12.80	9.53	8.33
10000	13.63	18.22	9.85

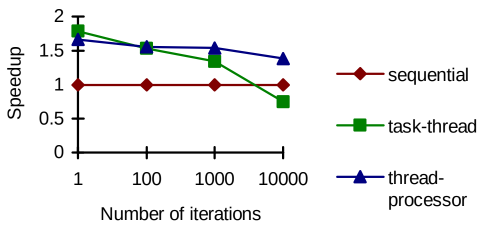

Figure 6: Task system simulation results

Speedup decreases as the number of iterations increases which is the result of less computation and more synchronization. Synchronization between threads is so slow that at 10000 iterations method task-thread is even slower than the sequential method. Method thread-processor is faster than sequential even at 10000 iterations, because threads made internal task scheduling without involving the operating system. Speedups for both parallel methods for a small number of iterations are almost the same.

7. Conclusion

Based on the results presented in this paper, we conclude that under certain conditions the multithreading can improve the performance of given algorithms which are running on multiprocessor system. If threads run independently or with very low communication, speedup is only limited by the number of processors. If the communication between threads is heavier, speedup can be reached only if the time of computation between synchronization is at least several times greater than the synchronization time.

Although all results presents execution times on a lightly loaded system, speedup is also reached at higher loads when all times increase but with almost the same ratio. Speedup slowly decreases as load increases, which is the result of a slower communication through the operating system.

8. Acknowledgments

This work was done within the research project "*Problem-Solving Environments in Engineering*", supported by the Ministry of Science and Technology of the Republic of Croatia.

9. References

[1] SunSoft, (1994), *Solaris 2.4: Multithreaded Programming Guide*, Sun Microsystems, Mountain View, California.

[2] Sohn, A. (1995), “Parallel N-ary Speculative Computation of Simulated Annealing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 997-1005.

[3] van Laarhoven, P.J.M, Aarts, E.H.L. (1987), *Simulated annealing: Theory and Applications*, D. Reidel Publishing Company, Dordrecht.

[4] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D., “Solving problems on concurrent procesors: Volume I, General Techniques and Regular Problems”, California Institute of Technology, Prentice-Hall International, Englewood Clifts, N. J., 1988.

[5] Jelenković, L. (1996), *Colection of Functions for Multithreaded Programs* (in Croatian), Undergraduate dissertation, Faculty of El.Engineering and Computing, University of Zagreb