IJCSC Volume 4 • Number 2 September 2013 pp.161-166 ISSN-0973-7391

Determining Effectiveness of Multithreading for Solving Problems with Low Computational Complexity

[1] Ms.Rupsa Saha, [2] Ms. Shwetha Rai, [3]Dr.Srikanth Prabhu, [4]Dr. Geetha M.
Department of Computer Science, MIT, Manipal University Manipal, Karnataka, India.
[1]rupsa.saha@gmail.com,[2]shwetha.rai@manipal.edu,[3]srikanth.prabhu@manipal.edu,[4]geetha.maiya@manipal.edu

Abstract - Multithreading is an important aspect of modern computing and also an established model for carrying out complex computational problems for achieving better performance. For simple computations, however, the effectiveness of multithreading over sequential programming is not apparent. This paper attempts to determine the effectiveness of multithreading by performing simple computation through single and multiple threads for a range of very small number of computations to a very large number of computations.

Keywords– multithreading, parallel computing, performance, simple computation, POSIX.

I.INTRODUCTION

In multiprocessor architectures, threads are used to implement parallelism. Achieving parallelism is important for getting computational gains [1]. A number of procedures are scheduled to run simultaneously by the operating system to make better use of resources and CPU cycles. Threads created by a single process exist within the process resources. However, they can run as independent entities because they duplicate only the resources necessary for them to be executable code. Multithreaded models are an improvement on multiprocessor models. This is because threads allow for cheaper context switch as well as memory considerations as compared to processes.

Threads change the timing of operations, hence are used for performance related problems. For complex computational problems, threads allow much better performance from a computer than sequential programming models. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events [3].

In distributed computing, threads are often not a practical abstraction because creating the illusion of shared memory is often too costly. In cases where simple computations are only required the overhead of creating, despatching and exiting threads can nullify the computational gains that threads attempt to provide in the first place. Even though threads provide very cheap context switching, still in such cases the context switch is overweight with respect to the actual computation done by the machine, leading to degraded performance.

II.RELATED WORK

High performance applications on shared memory machines have typically been written in a coarse grained style, with one heavyweight thread per processor. The programmer can use multiple lightweight threads and express a new thread to execute each individual parallel task; the implementation dynamically creates and schedules these threads onto the processors, and effectively balances the load. However, unless the threads scheduler is designed carefully, the parallel program may suffer poor space and time performance. G.J. Narlikar [2] studies the performance of a native, lightweight POSIX threads (Pthreads) library on a shared memory machine running Solaris using a set of parallel programs that dynamically create a large number of threads. The programs include dense and sparse matrix multiplies two N-body codes, a data classifier, a volume rendering benchmark, and a high performance FFT package. The results indicate that, provided we use a good scheduler, the rich functionality and standard API of Pthreads can be combined with the advantages of dynamic, lightweight threads to result in high performance.

Linux itself has "tasks", and originally tasks mapped one-to-one to processes. There was no multithreading. Then it was decided to add threads to Linux. This was done by dissociating tasks from processes. The idea was to provide a general-purpose kernel mechanism, whereby the kernel knew only of "tasks", which could optionally share various common resources, such as address spaces, file descriptor tables, and so forth, with one another. This was intended to provide a flexible mechanism on top of which various process/thread models, presented by the application-mode system library, could potentially be built, including the POSIX threading model [4].

There are several reasons why the parallelization of sequential programs is important. The most frequently mentioned reason is that there are many sequential programs that would be convenient to execute on parallel computers [5].

III.OBJECTIVE

The objective of this work is to find relevancy on multithreading in extremely simple computational jobs. By varying the number of computational jobs performed and varying the number of threads used, an attempt is made to show the usefulness of multithreading in such cases.

Performing an extremely large number of the same basic operations can be considered a problem for the algorithms having medium level of complexity, whereas small number of simple operations is a problem for algorithms having very low complexity. In particular, this attempt is to see how the performances of threads vary according to the various levels of complexities while the task itself remains simple and complexity arises only from the number of computations to be performed.

IV. METHODOLOGY USED

The threading model used here is the POSIX threading interface, often called Pthreads.

The program *threading.c* (P1) adds a certain number or pseudo-random integers. The program uses multithreading and takes two arguments. The first argument signifies the number of integers to be added. The second argument specifies the number of threads to be created in order to process the job.

Let *num* be the number of integers. Let *thr* be the number of threads.

Each thread handles the summation of n/k numbers. The final sum is obtained by the sum of the results produced by the threads. Pseudo-random integers less than 100 are generated by the in-built function rand(), seeded using the current system time.

Pseudocode of Program P1:

```
//Program to calculate sum of n random integers parallel
//Input: Number of integers: num, Number of threads: thr
//Output: Final sum: sum
main()
{
        //calculate the number of integers to be
        handled //by each thread
        max=num/thr;

        seed the pseudo-random generator with
        current system time
        srand(time(0));

        create the threads
        for i: 0 to thr
            pthread_create(...thread[i]…runner, max);
        done

        wait for the thread to exit
        for i: 0 to thr
                pthread_join(thread[i], NULL);
        done

        print final sum;
}

runner()
{
        for i: 0 to max
                sum+=rand()%100;
        done
        exit the thread
}
```

The program *sequential.c* (P0) does the same work as P1, i.e. add a certain number or pseudo-random integers less than 100. This program takes one argument and uses sequential programing. The number of integers to be added is passed on to the program as the argument.

Pseudocode of Program P0:

```
//Program to calculate sum of n random integers sequentially
//Input: Number of integers: num
//Output: Final sum: sum
main()
{
        seed the pseudo-random generator with
        current system time
        srand(time(0));
        for i: 0 to num
                sum+=rand()%100;
        done

        print final sum;
}
```

V. EXPERIMENTAL ANALYSIS

In order to find how threading affects performance in case of computations with low level of complexity, P1and P0 are run on a machine with 2 GB RAM, Intel i3 core processor, using Linux Ubuntu distribution, with varying number of threads in P1 for a given number of integers.

The time required to execute the program is obtained by the *time* system call on the terminal. The utility 'time' takes a program name as an input and displays information about the resources used by the program. This information displayed consists of (i) the elapsed real time between invocation and termination, (ii) the user CPU time, and (iii) the system CPU time.

The program P0 and P1 are executed for different datasets by varying the number of integers.

Case 1:

For P0, num=5, thr=0, the execution time is shown in the Table 1.

Table 1: Execution time of P0

Dataset	Real time in sec	User time in sec	System time in sec
1	0.002	0	0
2	0.002	0.004	0
3	0.002	0	0
4	0.001	0	0
5	0.002	0	0
6	0.002	0	0
7	0.001	0	0
8	0.002	0	0
9	0.002	0	0
10	0.002	0	0
Average	**0.0018**	**0.0004**	**0**

For P1, num=5, thr=1 and thr=5, the execution time is shown in the Table 2 and Table 3 respectively.

Table 2: Execution time of P1 using one thread

Dataset	Real time in sec	User time in sec	System time in sec
1	0.002	0	0
2	0.002	0	0
3	0.002	0	0
4	0.003	0	0
5	0.002	0.004	0
6	0.002	0	0
7	0.003	0	0
8	0.003	0.004	0
9	0.001	0.004	0
10	0.002	0	0
Average	**0.0022**	**0.0012**	**0**

Table 3: Execution time of P1 using five thread

Dataset	Real time in sec	User time in sec	System time in sec
1	0.003	0	0
2	0.004	0	0
3	0.003	0	0
4	0.002	0.004	0
5	0.003	0	0
6	0.003	0	0
7	0.004	0	0
8	0.002	0	0
9	0.007	0.004	0
10	0.003	0	0
Average	**0.0034**	**0.0008**	**0**

The average execution time for the execution of P0 and P1 is given in the Table 4 and Figure1 respectively.

Table 4: Average results for 5 integers

No. of Threads	Real time in sec	User time in sec	System time in sec
0	0.0018	0.0004	0
1	0.0022	0.0012	0
5	0.0034	0.0008	0

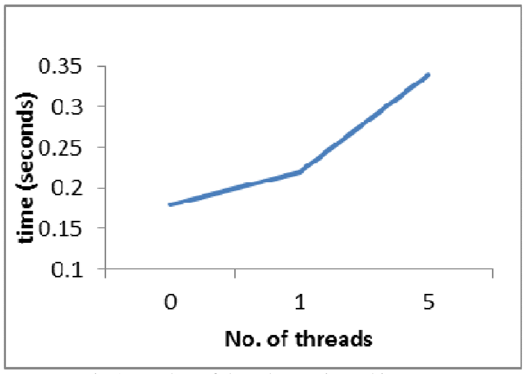

Fig 1.Number of threads vs. Time: 5 integers Real time of execution

It is observed that when *num* is small, sequential execution is better than using multithreading. In this case P0, which does not use threads, takes less time than P1. Even if a single thread is despatched separately through calling of P1, it leads to marginally higher times. Increasing the number of threads increases the time taken to complete the task.

Case 2:

For P0, num=1000000, thr=0, the execution time is shown in the Table 5.

Table 5: Execution time of P0

Dataset	Real time in sec	User time in sec	System time in sec
1	0.032	0.028	0
2	0.034	0.032	0
3	0.029	0.028	0
4	0.04	0.036	0
5	0.041	0.032	0
6	0.035	0.027	0.004
7	0.04	0.032	0
8	0.045	0.036	0
9	0.04	0.036	0
10	0.038	0.036	0
Average	**0.0374**	**0.0323**	**0.0004**

For P1, num=1000000, thr=1, thr=10 and thr=100, the execution time is shown in the Table 6, Table 7 and Table 8 respectively.

Table 6: Execution time of P1 using one thread

Dataset	Real time in sec	User time in sec	System time in sec
1	0.043	0.04	0
2	0.046	0.044	0
3	0.047	0.048	0
4	0.037	0.036	0
5	0.046	0.036	0.008
6	0.038	0.04	0
7	0.044	0.036	0.004
8	0.041	0.04	0

9	0.046	0.044	0
10	0.044	0.048	0
Average	**0.0432**	**0.0412**	**0.0012**

Table 7: Execution time of P1 using 10 threads

Dataset	Real time in sec	User time in sec	System time in sec
1	0.14	0.14	0.304
2	0.162	0.172	0.38
3	0.168	0.184	0.376
4	0.16	0.192	0.324
5	0.178	0.184	0.384
6	0.172	0.18	0.424
7	0.17	0.188	0.368
8	0.16	0.204	0.332
9	0.147	172	0.392
10	0.127	0.152	0.236
Average	**0.1584**	**17.3596**	**0.352**

Table 8: Execution time of P1 using hundred threads

Dataset	Real time in sec	User time in sec	System time in sec
1	0.162	0.148	0.228
2	0.186	0.16	0.236
3	0.148	0.156	0.252
4	0.16	0.128	0.208
5	0.174	0.144	0.188
6	0.157	0.164	0.26
7	0.178	0.176	0.264
8	0.189	0.188	0.204
9	0.174	0.204	0.46
10	0.184	0.12	0.548
Average	**0.1712**	**0.1588**	**0.2848**

The average execution time for the execution of P0 and P1 is given in the Table 9 and Figure 2 respectively.

Table 9: Average results for 1000000 integers

No. of Threads	Real time in sec	User time in sec	System time in sec
0	0.0374	0.0323	0.0004
1	0.0432	0.0412	0.0012
10	0.1584	17.3596	0.352
100	0.1712	0.1588	0.2848

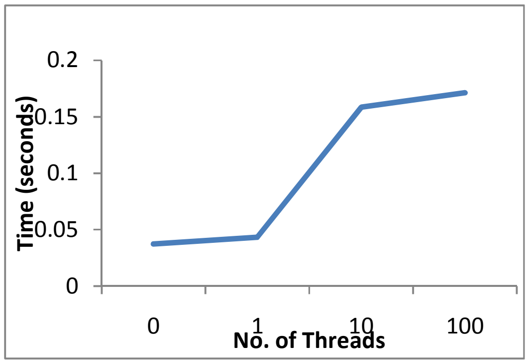

Fig 2.Number of threads vs. Time: 1000000 integers
Real time of execution

The second sample has a fairly large number (1000000) of integers to be added. It is observed from the experimental analysis that P1 with 1 thread gives slightly worse performance than P0 with 0 threads. However, performance falls drastically as the number of threads in P1 is increased first to 10 and then to 100.

Case 3:

P0 does not successfully execute for 10000000000 integers.

For P1, num=10000000000, thr=10, thr=100 and thr=379, the execution time is shown in the Table 10, Table 11 and Table 12 respectively.

Table 10: Execution time of P1 using ten threads

Dataset	Real time in sec	User time in sec	System time in sec
1	221.374	214.997	239.383
2	204.497	108.815	103.3
3	160.814	153.106	61.804
4	185.387	173.415	66.126
5	163.938	156.682	72.021
6	172.028	164.19	82.873
7	348.44	380.08	345.687
8	185.609	175.347	104.191
9	193.841	211.069	163.682
10	263.485	255.264	306.011
Average	**209.9413**	**199.2965**	**154.5078**

Table 11: Execution time of P1 using hundred threads

Dataset	Real time in sec	User time in sec	System time in sec
1	180.028	159.973	69.7
2	180.028	218.698	69.7
3	146.073	118.403	10.513
4	126.139	116.975	11.745
5	182.459	171.459	94.786
6	208.1	114.795	61.324

7	172.073	120.072	20.785
8	180.909	168.306	104.767
9	183.568	184.665	154.834
10	136.158	103.963	31.726
Average	**169.5535**	**147.7309**	**62.988**

Table 12: Execution time of P1 using three hundred and seventy nine threads

Dataset	Real time in sec	User time in sec	System time in sec
1	250.018	236.111	196.024
2	181.728	113.191	12.173
3	146.236	117.003	11.833
4	200.222	182.283	104.415
5	173.385	137.909	19.377
6	170.598	117.039	16.417
7	190.094	131.944	39.15
8	172.567	155.118	69.728
9	158.545	114.795	14.516
10	240.751	191.608	164.098
Average	**188.4144**	**149.7001**	**64.7731**

The average execution time for the execution of P1 is given in the Table 13 and Figure 3 respectively.

TABLE 13: Average results for 10000000000 integers

No. of Threads	Real time in sec	User time in sec	System time in sec
10	209.9413	199.2965	154.5078
100	169.5535	147.7309	62.988
379	188.4144	149.7001	64.7731

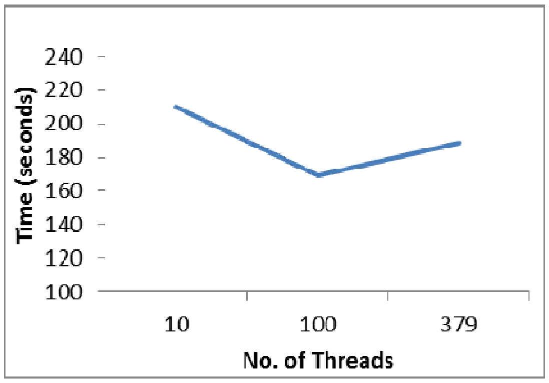

Fig 3.Number of threads vs. Time: 10000000000 integers Real time of execution

The third sample adds a large number of (10000000000) integer. It is observed from the experimental analysis that the amount of computation being large increases the complexity of the program in this case. Here, according to the collected timings, running P1 with 100 threads gives a better performance than running P1 with just 10 threads. On increasing the number of threads to 379 however, the performance of P1 declines.

In the first case, adding 5 integers, the amount of computation required is extremely low. In such a situation, the overhead involved in creating, dispatching and terminating threads decreases the performance and hence is not justified.

The second case, involving the addition of 1000000 integers is of greater complexity than the first. Results obtained however show a performance pattern similar to that of the previous case. Increasing number of threads contribute to decreasing the performance. Similar to the first sample, the amount of computation is still too low to justify the usage of threads.

In the third case, there are 10000000000 integers to be added, leading to increased complexity of the problem. The performance pattern observed here is different from that in the previous two cases. Since using 100 threads gives a better performance than using just 10 threads, here the usage of threads is justified. Even considering the overhead required for handling 100 threads, performance is still better than results obtained for lesser number of threads. However, as the number of threads is further increased to 379, performance declines, signifying that for larger number of threads the overhead again fails to justify the amount of computation required.

VI. CONCLUSION AND FUTURE SCOPE

This paper attempts to determine the effectiveness of multithreading by performing simple computation through single and multiple threads for a range of very small number of computations to a very large number of computations.

The research and analysis of this project has some limitations as follows. Firstly, the data represented here is a snapshot of the total data collected. Secondly, the maximum number of threads that the test machine supported was 379. Larger number of threads led to segmentation faults. The total data collected can be more extensive, from various machine configurations and under different constraints in order to have a more accurate diagnosis of the behaviour of threads.

Developing analyses for multithreaded programs can be a challenging activity. The primary complication is characterizing the effect of the interactions between threads. The obvious approach of analysing all interleaving of statements from parallel threads fails because of the resulting exponential analysis times [3].

This project can be further extended to include a wider range of variations of the parameters (i.e. number of threads, and number of integers to be added) in order to come to more accurate conclusions. This can help to set a standard on the minimum complexity of a problem that calls for usage of the multiple threads. In turn it can also prevent misuse of multithreading, in situations where threading actually decreases performance and go on to clearly define at least the lower bound of the capabilities of the multithreading model. Further research and analysis in this field will contribute to multithreading being used appropriately and to its full functional capability.

REFERENCES

[1] Edward A. Lee, "The Problem with Threads", University of California, Berkeley, Tech. Rep UCB/EECS-2006-1, 2006.

[2] G.J. Narlikar, G.E. Blelloch, "Pthreads for Dynamic and Irregular Parallelism", IEEE/ACM Conference on Supercomputing, SC98, 1998.

[3] Martin Rinard, "Analysis of Multithreaded Programs", Laboratory for Computer Science, Massachusetts Institute of Technology, Proceedings of the 8th Static Analysis Symposium, July 2001.

[4] http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/linux-thread-problems.html

[5] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. Proceedings of the IEEE, 81(2):211–243, 1993.

[6] Robert D. Blumofe, Charles E. Lieserson, "Space-Efficient Scheduling of Multithreaded Computations", MIT Laboratory for Computer Science. In the Proceedings of ACM symposium on Theory of computing, 1993.

[7] Kai Hwang, Faye A. Briggs, "Computer Architecture and Parallel Processing", 1st ed., 2012.

[8] Kenneth C. Louden, "Design and Implementation of Programming Languages", 3rd ed., 2012.