Benchmarking Using MATLAB

1. Introduction

As computer architecture has been developed, it is still very difficult to compare the performance of different computer systems simply by comparing their specifications. For this, few tests were developed and this allows comparison between different computer architectures.

In computing, benchmark is used to assess the relative performance of an object by running few test programming cases. Benchmarking is usually associated with assessing performance characteristics of computer hardware. Since benchmarking contains performance sensitive aspects, we can use this clues to improve performance. Thus, benchmarking is necessary for evaluating and comparing difference features or configurations of computer architectures.

In this project, we have focused on performing detailed benchmarking and analysis of the performance bottlenecks for these programs on two machines. Specifically, we have observed by profiling performances with 4 test cases using MATLAB: single thread simple for-loop, single thread for-loop with pre-allocated memory, implicit multi-thread, and explicit multi-thread examples.

2. Background

1) Serial computing:

Serial programming means consecutive and ordered execution of processes. This also means that in serial programming, codes and processes are run in order. This can degrade the execution performance since the execution speed is dependent on transmission speed and it is expensive to make a single processor faster.

2) Parallel computing:

Parallel programming means concurrent computation or simultaneous execution of processes or threads at the same time. To support parallel programming, there need to be multiple processes to execute at the same time. This decompose an algorithm or data into parts and distribute tasks in multiple processors and execute simultaneously.

3) Preallocating:

Specifically in Matlab, the user doesn't need to declare the types and sizes of variables before using them. This is very convenient but this can slow down the program in certain cases: when user uses array which is not preallocated in a for-loop, each time the user uses it, MATLAB needs to allocate memory for a new larger array and then copy the existing data into it. This procedure will be repeated because of for-loop and this becomes very inefficient.

4) Profiling in MATLAB:

Profiling is a user interface and a method to measure where does a program spends time. By analyzing spending time of each functions, we can evaluate performance of program for possible performance improvements. This also helps debugging and finding error of the code.

3. Methodology

1) Machines Specification

We used 2 machines to analyze the performance of codes. Below is the specification of two machines:

Table 1: Specification of two machines

	Machine 1	Machine 2
CPU	2.9 GHz Intel Core i5	2.4 GHz Intel Core i7
Memory	8GB 1867 MHz DDR3	16GB DDR3L SDRAM(2 DIMM)
Disk	Macintosh HD	5400 RPM Hard drive
Operating System	Mac OS	Windows
Compiler	Matlab(R2015b) Compiler	Matlab(R2015b) Compiler

2) Approach

We have developed 4 test cases in MATLAB to analyze the performance: single thread simple for-loop, single thread for-loop with pre-allocated memory, implicit multi-thread, and explicit multi-thread program examples. This examples are attached below.

MATLAB automatically exploit the parallelism inherent to the computations and thus, no need to make any modifications. Therefore, we benchmarked two multi-threading code considering this implicit parallelism as well as dependencies along with it. In our test cases, the number of threads are increased in to see clear output. Lastly we benchmarked performance of multi-threading code comparing with simple for-loop code.

The following output is expected in our benchmark:

- Time consumption in each code
- Memory consumption for each code line
- Wait time for dependency
- Speed up by multi-thread
- Efficiency of each thread

3) Test Cases

- Forloop.m (Single thread, for-loop, not preallocated)

```
%%%%%Non-preallocated%%%%%
tic
%profile -memory on
n = 10e5;
x(1) = 0;
x(2) = 1;
for k=3:n
    x(k) = n*rand*x(k-2)+n*rand*x(k-1);
end
toc
%profreport
```

- Preallocate.m (Single thread, for-loop, preallocated)

```
%%%%%Preallocated%%%%%%
tic
%profile -memory on
n = 10e5;
x=zeros(n,1);
x(1) = 0;
x(2) = 1;
for k=3:n
    x(k) = n*rand*x(k-2)+n*rand*x(k-1);
end
toc
%profreport
```

- MultiThread_implicit.m (Implicit multi-thread automatically supported with dependency)

: maxNumCompThreads() controls the parallelism of the multithreading approach. This command is used to adjust number of threads.

```
%%%Multi-Thread(implitcit)%%%

n = 1000;
x = rand(n);
b = rand(n);

for k=3:n
    x(k) = n*rand*x(k-2)+n*rand*x(k-1);
end

for i=1:5
  m = 2^(i-1);
  maxNumCompThreads(m);
  tic
  y = x*b;
  wait_time(i) = toc;
  Spd_up(i) = wait_time(1)/wait_time(i);
  Efficiency(i) = 100*Spd_up(i)/m;
end
```

- MultiThread_explicit.m (Explicit multi-thread vs. simple for-loop)

```
%%%Multi-Thread(explitcit)%%%
myCluster = parcluster('local');
myCluster.NumWorkers = 4;
saveProfile(myCluster);
tic
y = zeros(1,10e6);
parfor i=1:10e6
%parfor i=1:10e4
%for i=1:10e6
%for i=1:10e4
    x = 0;
    for j=1:100
        x=i+1;
    end
    y(i) = x;
end
toc
```

4. Results (screenshots)

1) Forloop.m

- **Machine 1 (Single thread, for-loop, not preallocated)**

Preallocate (Calls: 1, Time: 0.490 sec, 8357.02 Kb, 22.61 Kb, 5764.00 Kb)

Generated 01-Dec-2016 02:28:33 using performance time.
script in file /Users/Grrrr/Documents/MATLAB/Preallocate.m
Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Sort busy lines and graph according to time

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
10	x(k) = n*rand*x(k-2)+n*rand*x(...	999998	0.309 s	8282.28 Kb	16.72 Kb	5764.00 Kb	63.0%	
11	end	999998	0.171 s	10.41 Kb	5.89 Kb	2.33 Kb	34.8%	
13	profreport	1	0.009 s	27.75 Kb	0.00 Kb	27.75 Kb	1.9%	
12	toc	1	0.000 s	6.67 Kb	0.00 Kb	6.67 Kb	0.1%	
7	x(1) = 0;	1	0.000 s	2.56 Kb	0.00 Kb	2.56 Kb	0.0%	
All other lines			0.001 s	27.34 Kb	0.00 Kb	5764.00 Kb	0.2%	
Totals			0.490 s	8357.02 Kb	22.61 Kb	5764.00 Kb	100%	

Children (called functions)
No children

Code Analyzer results
No Code Analyzer messages.

Coverage results
Show coverage for parent directory

Total lines in function	10
Non-code lines (comments, blank lines)	0
Code lines (lines that can run)	10
Code lines that did run	8
Code lines that did not run	2
Coverage (did run/can run)	80.00 %

Function listing

Color highlight code according to time

```
 time     Calls                     mem                                line
                (Calls: 1, Time: 0.490 sec, 8357.02 Kb, 22.61 Kb, 5764.00 Kb)   4 tic
                (Calls: 1, Time: 0.490 sec, 8357.02 Kb, 22.61 Kb, 5764.00 Kb)   5 profile -memory on
< 0.01       1  (Calls: 1, Time: 0.490 sec, 8357.02 Kb, 22.61 Kb, 5764.00 Kb)   6 n = 10e5;
< 0.01       1                  2.56k/0b/2.56k                                  7 x(1) = 0;
< 0.01       1                  2.94k/0b/2.94k                                  8 x(2) = 1;
< 0.01       1                        2k/0b/2k                                  9 for k=3:n
  0.31  999998                8.09m/16.7k/5.63m                                10     x(k) = n*rand*x(k-2)+n*rand*x(k-1);
  0.17  999998                10.4k/5.89k/2.33k                                11 end
< 0.01       1                  6.67k/0b/6.67k                                 12 toc
< 0.01       1                  27.8k/0b/27.8k                                 13 profreport
```

Other subfunctions in this file are not included in this listing.

Total execution time of single thread, for-loop, without preallocated memory in machine 1:

```
Elapsed time is 0.154138 seconds.
```

- **Machine 2 (Single thread, for-loop, not preallocated)**

Preallocate (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)

Generated 01-Dec-2016 19:11:29 using performance time.

script in file C:\Users\Grrrr's HP\Documents\MATLAB\Preallocate.m

Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions

☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Sort busy lines and graph according to time

Parents (calling functions)

No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
10	x(k) = n*rand*x(k-2)+n*rand*x(...	999998	0.382 s	9128.00 Kb	56.00 Kb	4104.00 Kb	60.4%	
11	end	999998	0.226 s	0.00 Kb	12.00 Kb	0.00 Kb	35.8%	
12	toc	1	0.018 s	0.00 Kb	0.00 Kb	0.00 Kb	2.8%	
13	profreport	1	0.005 s	0.00 Kb	0.00 Kb	0.00 Kb	0.8%	
7	x(1) = 0;	1	0.000 s	0.00 Kb	0.00 Kb	0.00 Kb	0.1%	
All other lines			0.001 s	0.00 Kb	0.00 Kb	4104.00 Kb	0.2%	
Totals			0.633 s	9128.00 Kb	68.00 Kb	4104.00 Kb	100%	

Children (called functions)

No children

Code Analyzer results

No Code Analyzer messages.

Coverage results

Show coverage for parent directory

Total lines in function	10
Non-code lines (comments, blank lines)	0
Code lines (lines that can run)	10
Code lines that did run	8
Code lines that did not run	2
Coverage (did run/can run)	80.00 %

Function listing

Color highlight code according to time

```
time     Calls                          mem                              line
                 (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   4  tic
                 (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   5  profile -memory on
< 0.01       1   (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   6  n = 10e5;
< 0.01       1   (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   7  x(1) = 0;
< 0.01       1   (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   8  x(2) = 1;
< 0.01       1   (Calls: 1, Time: 0.633 sec, 9128.00 Kb, 68.00 Kb, 4104.00 Kb)   9  for k=3:n
  0.38  999998                    8.91m/56k/4.01m                               10      x(k) = n*rand*x(k-2)+n*rand*x(k-1);
  0.23  999998                          0b/12k/0b                               11  end
  0.02       1                          0b/12k/0b                               12  toc
< 0.01       1                          0b/12k/0b                               13  profreport
```

Other subfunctions in this file are not included in this listing.

Total execution time of single thread, for-loop, without preallocated memory in machine 2:

```
Elapsed time is 0.162137 seconds.
```

2) Preallocate.m

- **Machine 1 (Single thread, for-loop, preallocated)**

Preallocate (Calls: 1, Time: 0.431 sec, 9078.36 Kb, 1117.83 Kb, 7823.53 Kb)

Generated 01-Dec-2016 02:31:39 using performance time.
script in file /Users/Grrrr/Documents/MATLAB/Preallocate.m
Copy to new window for comparing multiple runs

Refresh

- [x] Show parent functions
- [x] Show busy lines
- [x] Show child functions
- [x] Show Code Analyzer results
- [x] Show file coverage
- [x] Show function listing

Sort busy lines and graph according to time

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
24	x(k) = n*rand*x(k-2)+n*rand*x(...	999998	0.248 s	1003.83 Kb	546.00 Kb	444.00 Kb	57.6%	
25	end	999998	0.170 s	132.73 Kb	571.83 Kb	64.00 Kb	39.5%	
27	profreport	1	0.009 s	87.38 Kb	0.00 Kb	87.38 Kb	2.1%	
20	x=zeros(n,1);	1	0.002 s	7823.53 Kb	0.00 Kb	7823.53 Kb	0.5%	
26	toc	1	0.000 s	6.81 Kb	0.00 Kb	6.81 Kb	0.1%	
All other lines			0.001 s	24.08 Kb	0.00 Kb	7823.53 Kb	0.2%	
Totals			0.431 s	9078.36 Kb	1117.83 Kb	7823.53 Kb	100%	

Children (called functions)
No children

Code Analyzer results
No Code Analyzer messages.

Coverage results
Show coverage for parent directory

Total lines in function	11
Non-code lines (comments, blank lines)	0
Code lines (lines that can run)	11
Code lines that did run	9
Code lines that did not run	2
Coverage (did run/can run)	81.82 %

Function listing

Color highlight code according to time

time	Calls	mem	line	
		(Calls: 1, Time: 0.426 sec, 124.23 Kb, 22.23 Kb, 70.06 Kb)	17	tic
		(Calls: 1, Time: 0.426 sec, 124.23 Kb, 22.23 Kb, 70.06 Kb)	18	profile -memory on
< 0.01	1	(Calls: 1, Time: 0.426 sec, 124.23 Kb, 22.23 Kb, 70.06 Kb)	19	n = 10e5;
< 0.01	1	6.78k/0b/6.78k	20	x=zeros(n,1);
< 0.01	1	6.78k/0b/6.78k	21	x(1) = 0;
< 0.01	1	6.78k/0b/6.78k	22	x(2) = 1;
< 0.01	1	1.38k/0b/1.38k	23	for k=3:n
0.24	999998	19.3k/16.6k/8.38k	24	x(k) = n*rand*x(k-2)+n*rand*x(k-1);
0.17	999998	14k/5.61k/4k	25	end
< 0.01	1	5.88k/0b/5.88k	26	toc
< 0.01	1	70.1k/0b/70.1k	27	profreport

Other subfunctions in this file are not included in this listing.

Total execution time of single thread, for-loop, preallocated memory in machine 1:

```
Elapsed time is 0.095933 seconds.
```

- **Machine 2 (Single thread, for-loop, preallocated)**

Preallocate (Calls: 1, Time: 0.539 sec, 8072.00 Kb, 76.00 Kb, 7832.00 Kb)

Generated 01-Dec-2016 19:23:33 using performance time.
script in file C:\Users\Grrr's HP\Documents\MATLAB\Preallocate.m
Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing
Sort busy lines and graph according to time

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
25	x(k) = n*rand*x(k-2)+n*rand*x(...	999998	0.307 s	12.00 Kb	60.00 Kb	12.00 Kb	56.9%	
26	end	999998	0.225 s	0.00 Kb	16.00 Kb	0.00 Kb	41.9%	
28	profreport	1	0.005 s	228.00 Kb	0.00 Kb	228.00 Kb	0.8%	
21	x=zeros(n,1);	1	0.001 s	7832.00 Kb	0.00 Kb	7832.00 Kb	0.1%	
27	toc	1	0.000 s	0.00 Kb	0.00 Kb	0.00 Kb	0.1%	
All other lines			0.001 s	0.00 Kb	0.00 Kb	7832.00 Kb	0.2%	
Totals			0.539 s	8072.00 Kb	76.00 Kb	7832.00 Kb	100%	

Children (called functions)
No children

Code Analyzer results
No Code Analyzer messages.

Coverage results
Show coverage for parent directory

Total lines in function	11
Non-code lines (comments, blank lines)	0
Code lines (lines that can run)	11
Code lines that did run	9
Code lines that did not run	2
Coverage (did run/can run)	81.82 %

Function listing
Color highlight code according to time

```
time    Calls                    mem                                          line
                (Calls: 1, Time: 0.539 sec, 8072.00 Kb, 76.00 Kb, 7832.00 Kb)  18  tic
                (Calls: 1, Time: 0.539 sec, 8072.00 Kb, 76.00 Kb, 7832.00 Kb)  19  profile -memory on
< 0.01       1  (Calls: 1, Time: 0.539 sec, 8072.00 Kb, 76.00 Kb, 7832.00 Kb)  20  n = 10e5;
< 0.01       1                 7.65m/0b/7.65m                                  21  x=zeros(n,1);
< 0.01       1                 7.65m/0b/7.65m                                  22  x(1) = 0;
< 0.01       1                 7.65m/0b/7.65m                                  23  x(2) = 1;
< 0.01       1                 7.65m/0b/7.65m                                  24  for k=3:n
  0.31  999998                   12k/60k/12k                                   25      x(k) = n*rand*x(k-2)+n*rand*x(k-1);
  0.23  999998                      0b/16k/0b                                  26  end
< 0.01       1                      0b/16k/0b                                  27  toc
< 0.01       1                   228k/0b/228k                                  28  profreport
```

Total execution time of single thread, for-loop, preallocated memory in machine 2:

```
Elapsed time is 0.091066 seconds.
```

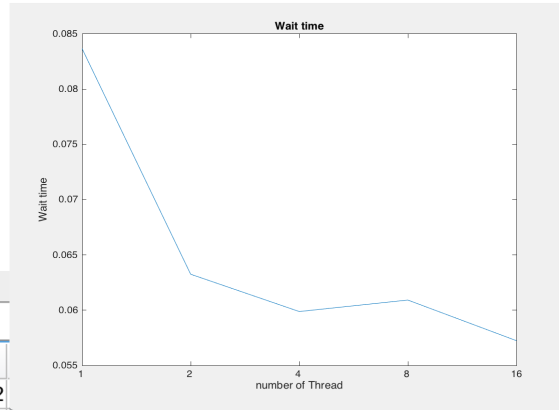
3) **MultiThread_implicit.m**

- **Machine 1 (Implicit multi-thread automatically supported with dependency)**

wait_time | Spd_up | Efficiency

1x5 double

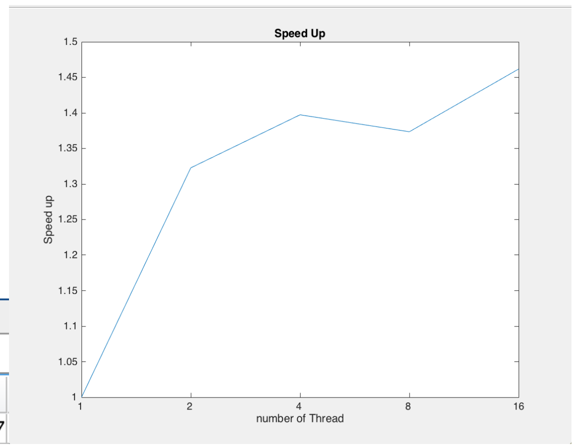
	1	2	3	4	5
1	0.0837	0.0633	0.0599	0.0609	0.0572



wait_time | Spd_up | Efficiency

1x5 double

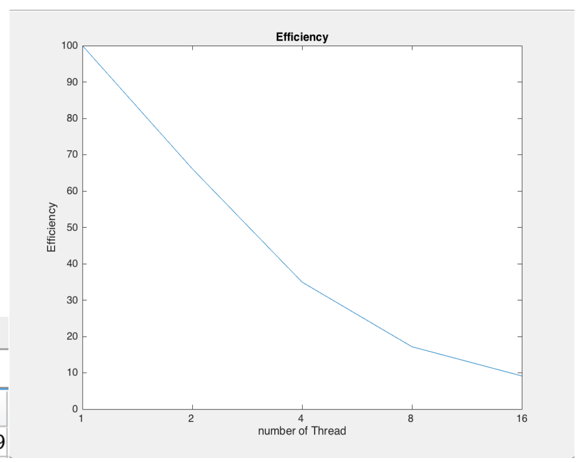
	1	2	3	4	5
1	1	1.3228	1.3974	1.3735	1.4617



wait_time | Spd_up | Efficiency

1x5 double

	1	2	3	4	5
1	100	66.1384	34.9340	17.1689	9.1359

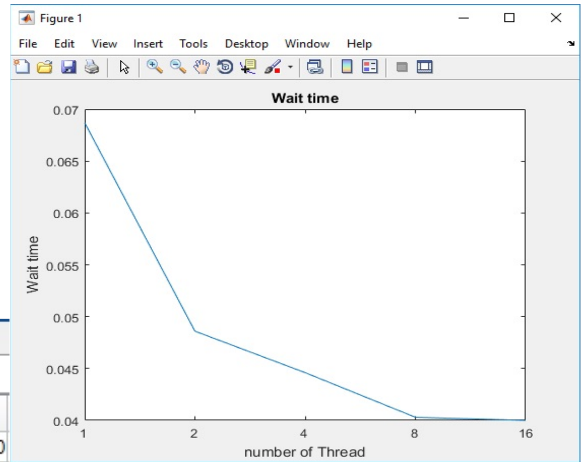


- **Machine 2 (Implicit multi-thread automatically supported with dependency)**

wait_time | Spd_up | Efficiency

1x5 double

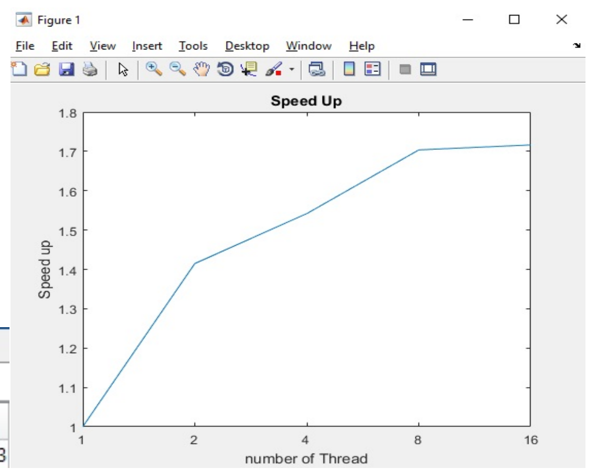
	1	2	3	4	5
1	0.0687	0.0486	0.0446	0.0403	0.0400



wait_time | Spd_up | Efficiency

1x5 double

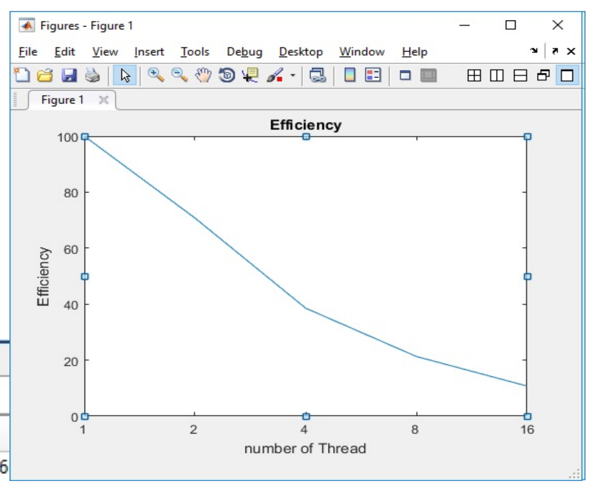
	1	2	3	4	5
1	1	1.4147	1.5417	1.7036	1.7163



wait_time | Spd_up | Efficiency

1x5 double

	1	2	3	4	5
1	100	70.7344	38.5427	21.2951	10.7266



4) MultiThread_explicit.m (Explicit multi-thread vs. simple for-loop)

Case 1: Large number of loop (10^7):

- **Machine 1**

  Explicit multi-thread: `Elapsed time is 2.365082 seconds.`

  Simple for-loop **:** `Elapsed time is 3.795548 seconds.`

- **Machine 2**

  Explicit multi-thread: `Elapsed time is 2.544409 seconds.`

  Simple for-loop **:** `Elapsed time is 4.058486 seconds.`

Case 2: Small number of loop (10^5):

- **Machine 1**

  Explicit multi-thread: `Elapsed time is 0.317262 seconds.`

  Simple for-loop: `Elapsed time is 0.042123 seconds.`

- **Machine 2**

  Explicit multi-thread: `Elapsed time is 0.152614 seconds.`

  Simple for-loop **:** `Elapsed time is 0.049406 seconds.`

5. Discussion

Benchmark result shows about memory consumption in every code line. In the profile result there are few specific memory types that are used during the program: allocated memory, freed memory, and peak memory. Allocated memory is the total amount of memory allocated within the function and any it calls. Freed memory is the total amount of memory released within the function and any it calls. Peak Memory is the maximum amount of memory in use at any one time during the execution of the function.

Figure 1: Memory usage of two machines with simple for-loop and preallocate

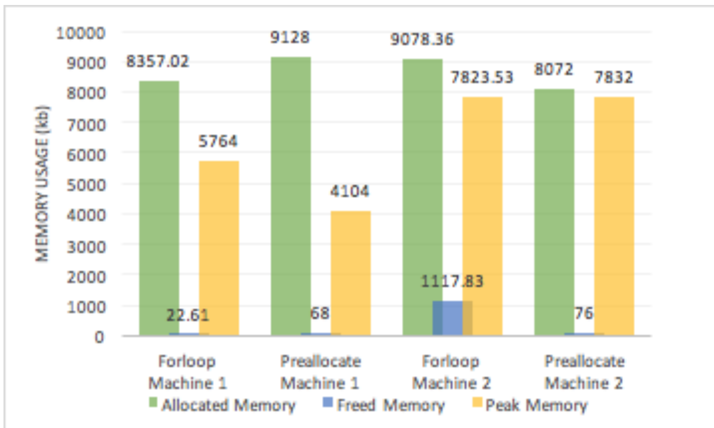
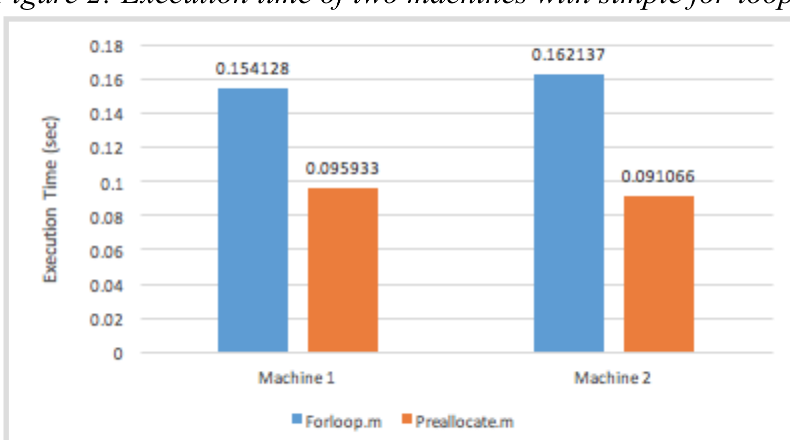


Figure 1 illustrates different memory usage of the two different machines with simple for-loop program and preallocate program. There is no explicit different between memory usage of the two programs. In machine 1, both for-loop code and preallocate code's usage trend are very similar and this trend also shown in machine 2. We will further look into execution time in addition to the memory usage.

Figure 2: Execution time of two machines with simple for-loop and preallocate



As illustrated in Figure 2, simple for-loop program took more time to execute than memory pre-allocated code. This is proved in both machine 1 and machine 2. The difference of execution time is very clear and this is because in MATLAB, it is necessary to pre-allocate memory space before use it. If the memory space isn't allocated but is newly made every time accessing at the for-loop, then MATLAB need to make new memory space to add at the previous array space at every time it runs it. On the other hand, if the memory space is preallocated, then there is no need to access the memory of array to add another space.
Moreover, in Figure 1, there was no difference in memory usage between simple for-loop code and memory pre-allocated code. Since the memory usage is not a big issue for the programs, the main concern for improving performance becomes the speed. Therefore memory preallocation, especially when using array in the program before accessing them, is needed to speed-up and to improve performance.

Using Multi-Threads:

Threads are a common software solution for parallel programming on multicore systems and the best performance is often obtained when the number of threads and the number of cores correspond, but there are circumstances when there should be fewer threads. We experimented implicit multi thread to see the optimal number of threads for our computation in each machine: machine 1 is using dual core processor and machine 2 is using quad core processor.

Figure 3: Implicit multi-thread execution times & speed-ups of two machines

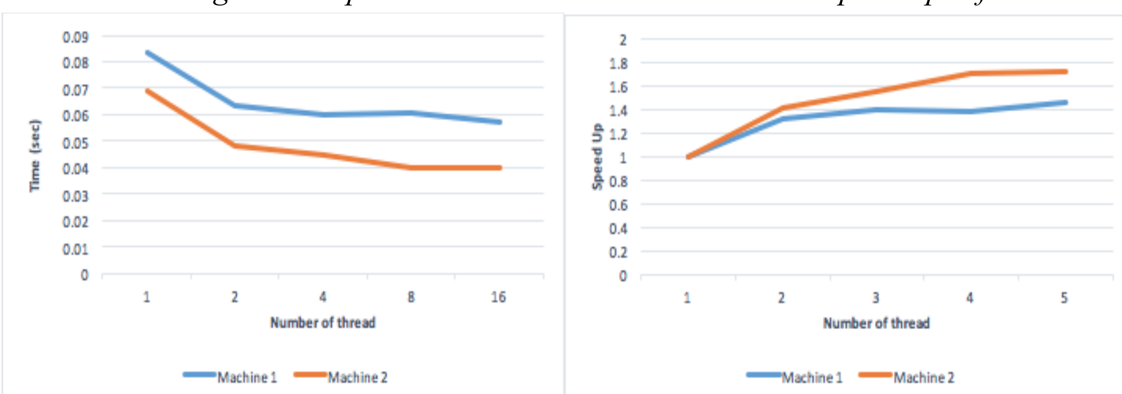
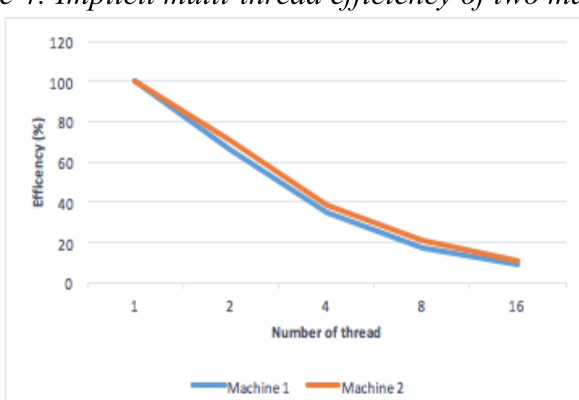


Figure 4: Implicit multi-thread efficiency of two machines



As shown in Figure 3, multi-threading has advantage of increasing speed of performance. As the number of thread increases, there has been speed-ups in the program and thus the execution time is reduced. However, the efficiency of each threads in parallel programming reduced significantly. Even though there was slight difference between two machines since their specification is different, overall trend of the result was the same. In addition, the result of speed was initially dramatically improved but eventually, the improvement become decreased. This follows the fact that parallel programming needs limit when producing threads in the system.

Limitation of parallel programming:

Programming explicit parallelism is the representation of concurrent computations by means of primitives in the form of special-purpose directives or function calls. Most of these parallel primitives are related to process synchronization, communication or task partitioning. However, when they carry out the intended computation of the program, they sometimes cause parallelization overhead. Sources of this parallelization overhead are communication and synchronization, contention, extra computation, and extra memory. Thus in some cases, sequential execution instead of parallel programming would be better solution to improve performance.

Figure 5: Execution time of Machine 1 with simple for-loop and multi-thread

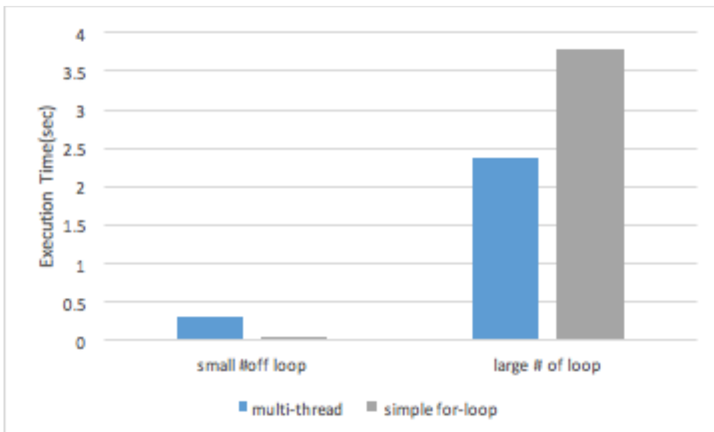
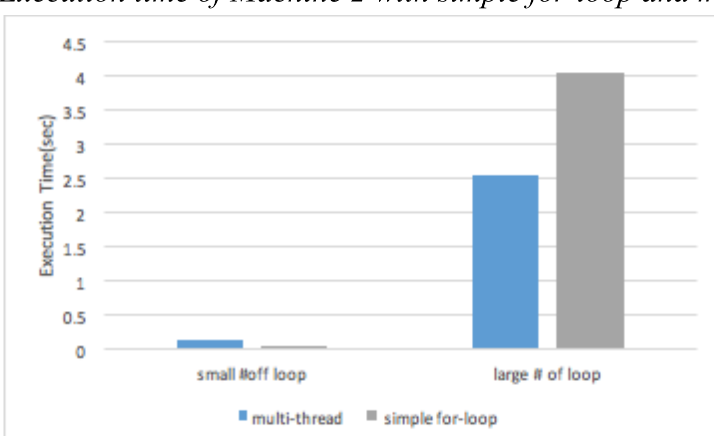


Figure 6: Execution time of Machine 2 with simple for-loop and multi-thread



As we predicted, there was a big difference in execution time when using multi-thread code and simple for-loop code. When there is small amount data to process, serial programming and parallel programming shows not much difference in speed. However this difference becomes larger when running with a lot of data to process. This result is well-depicted in Figure 5 and Figure 6. Machines' specification is different but they somehow showing the same result. Therefore, when considering performance it is clear that it's better to use serial programming when there is not much to process, and do parallel programming when there is much to process.

6. Conclusion

Through this project, we found out several ways that can improve overall performance. Memory usage, overall speed, total execution time, and parallel computing was all considered in this context. To improve the performance, not always the thread programming and less memory usage is the best choice but sometimes compromising and optimizing programming environment and code are necessary. In addition, the hardware specification is another aspect of improving performance. We tried to compare the result according to the difference of clock frequency but somehow MATLAB result was similar. We also thought that we can try with Raspberry PI 3 because it's CPU is 1.2 GHZ quad-core ARM Cortex A53. However since MATLAB requires a lot of memory to install and it's very heavy program, we are leaving this part as a future work.

MATLAB supports several parallelization methods already. We have tested the implicit multi-thread example and observed the thread performance and efficiency by increasing the number of thread in the program. In the future, we predict that we will see more cores inside the computers and there will be more advanced parallel programming to support this parallelization methods to improve performance.

7. References

[1] https://www.mathworks.com/help/matlab/matlab_prog/profiling-for-improving-performance.html

[2] *SERIAL COMPUTING vs. PARALLEL COMPUTING: A COMPARATIVE STUDY USING MATLAB* Abhay B. Rathod , Rajratna Khadse , M Faruk Bagwan

[3] *Programming Patterns: Maximizing Code Performance by Optimizing Memory Access*, Stuart McGarrity, Mathwork, 2007

[4] *Parallelism Provided by MATLAB: A Survey* Divya Kundra, International Journal of Advanced Research in Computer Science and Software Engineering, December, 2015

[5] *Parallel Programming in MATLAB*, Piotr Luszczek, July 20, 2009

[6] https://en.wikipedia.org/wiki/Explicit_parallelism

[7] *A Comparison of Implicit and Explicit Parallel Programming*, Vincent W. Freeh, Science Direct, April 10, 1996