Performance Evaluation of a Randomized Input Vector for Square Root Function between Single-threaded and Multi-threaded OpenMP Implementations

Michael Anthony Jay B. Regis
Visayas State University
Visca, Baybay City, Leyte, Philippines

Abstract
Parallel processors have been the recent trend in the mainstream computing market. But taking advantage of multiple processors is still a challenge since most algorithms are written in a sequential-based manner. An example of a sequential algorithm that is commonly used and implemented is the "square root extraction." Recently, the popularity of inverse square root in the image processing community poses even more computational challenge due to large amount of data inputs. In this paper, the problem of square root extraction on a large data input was addressed using a parallel-based algorithm in OpenMP. The algorithm used varying amounts of thread count and scheduling methods to handle the square root operations then it was executed in a parallel for loop statement. Results showed that the optimum parallel implementation was more than four times its serial counterpart given that the input vector of 20,000 elements is executed in a four physical core (8 logical core) shared memory platform processor. It is also important to note that using too few (i.e. threads lesser than the physical core count) or too many (i.e. threads equal to the input vector) would drastically reduce performance. In this study, a 1:16 logical core count to thread ratio is the optimum. The efficiency of multi-threaded operation using static and guided scheduling is 55.57% and 54.60%, respectively.

Keywords: Parallel processors, square root, OpenMP, shared memory platform

Introduction
The computation of square root has been considered essential in numerous scientific and engineering applications. Moreover, interest on inverse square root has gained substantial attention since it is widely used for multimedia and graphics applications (Ercegovac, Lang, Muller, & Tisserand, 2000). However, the Newton-Raphson Method — the most common multiplicative method for extracting high precision square root is relatively slow since the iteration needed to achieve quadratic convergence consist of multiplication and addition (Ito, Takagi, & Yajima, 1995).

Parallel processing has the sole purpose to enhance the performance of serial algorithms. It consists of multiple processors (or processing cores) linked through an interconnection network with the software required to enable the processing units to function together. Single-processor performance is overshadowed by the significant cost advantage (i.e. raw performance and price) of multiprocessors (El-Rewini & Abd-El-Barr, 2005). This is because increasing the clock frequency of a single core processor to increase performance would entail producing a lot of wasted heat. To settle this constraint, multiple-core processor with lower clock frequency than their single counterpart are designed to provide more throughput yet maintain the necessary thermal envelope (Pase & Eckl, 2006).

Recent trends in computing indicates that the industry is heading towards transition from serial to parallel computing. Almost all consumer computers shipped since the year
2010 was packaged with multi-core processors. Gone are the days when parallel computing is exclusive to supercomputers and mainframes from dual-core low end machines to high end 8- to 16-core workstations which are now widely available (Sanders & Kandrot, 2011). Consequently, software developers need to address the growing demand of computing performance in order to stay competitive with the ever increasing sophistication of the user base (Sanders & Kandrot, 2011).

In this study, the advantages of multi-threading to extract the square root of an input vector processed by a shared memory platform [i.e. multi-core, common memory processing unit (Grama, Gupta, Karypis, & Kumar, 2003)] using the approach of OpenMP was examined. An approach with OpenMP is more robust compared to manual threading because basic aspects of parallel programming such as defining parallel execution, thread communication, and thread synchronization are automatically addressed. Moreover, the directive based approach for supporting parallelism would allow same code base for both single and multi-processors in addition to incremental enhancements from serial oriented task into parallel oriented ones (Chandra et al., 2001). OpenMP is used to improve fundamental algorithms [e.g. Fast Fourier Transform (Weng, Huang, Perng, Hsu, & Li, 2009)] to the more sophisticated ones [e.g. Clustering Biological Graphs (Chapman & Kalyanaraman, 2011)].

The average running time was used as the primary metrics to evaluate performance. Significant differences among the averages was implemented using ANOVA (Analyses of Variance). If there is a significant difference, a pairwise comparison using Tukey’s HSD (Honest Significant Difference) would be used. Furthermore, the optimum thread count that would maximize performance was also determined. Finally, the speed-up and efficiency of single versus multi-threading was also computed.

Methodology

An Intel® Core I7 2670QM (code name “Sandy Bridge”) was used as the experimental platform. It is a quadcore hyperthreading (i.e. effectively creating twice the number of logical cores from the physical cores) processor, operating at 2.2Ghz. The platform also consisted of 8GB of shared DDR3 memory and 6MB of L3 Shared Cache memory. On the other hand, C/C++ version of OpenMP distributed in the Windows® GNU C package was used as compiler.

Initializing Randomized Values and Square Root Function

An array with a size of 20,000 elements containing randomized integer values ranging from 10,000 to 20,000 acted as the vector input. A square root function extracted all the possible square root values starting from one (1) up to the value of that element. (e.g. input_vec[0] = 14312 → √0, √1, √2…√14312).

Single-Threaded Operation

The single-threaded operation only used a thread (i.e. the main thread) to execute the square root function for each element in the input vector. The C++ code below shows the implementation.

```c
void serial()
{
    for (long int x=0;x<20000;x++)
    {
        squareRootFunction (input_vec[x]);
    }
}
```

N-Threads Operation

This multi-threaded approach created threads equal to the size of input vector. Static scheduling was then employed effectively providing a single thread for each square root operation (refer to the OpenMP C++ implementation below).
Another variation of multi-threading using static scheduling, the E-threads method, created threads that were equal to the number of logical processing cores. This means that the task per thread is the size of the vector input divided by the number of processor cores (i.e. 2,500 square root operations per thread, since there are 8 logical cores on the platform).

E-Threads Operation

The source code below summarizes the entire process.

```c
void pNThreads()
{
    int threads = 20000;
    #pragma omp parallel
    for (num_threads(threads)
        schedule (static, threads)
        for (int x=0; x<20000; x++)
        {
            squareRootFunction (input_vec[x]);
        }
}
```

F-Threads Operation

Same as E-threads, F-threads operation created threads equal the number of logical cores. But instead of using static scheduling, it used guided scheduling. In a way, the task is dynamically divided into chunk sizes that starts off large then decreases each time a portion of work is given to a thread. Initially the chunk size is:

\[
\frac{(number \_ of \_ iterations)}{(number \_ of \_ threads)}
\]

Subsequent chunk size was:

\[
\frac{(number \_ of \_ iterations)}{(number \_ of \_ threads)}
\]

Performance Evaluation

Each algorithm was executed 30 times with a time function to track the running time of each execution. Average and standard deviation was computed from the 30 repetitions. The results were further analyzed for significant difference using ANOVA followed by Tukey’s HSD if there is a significant difference.

Post-Processing Evaluation

Optimum thread count was tested to determine the maximize performance using the different scheduling techniques. The thread count to be tested was \(2^k\) threads where \(1 \leq k \leq 14\) (i.e. 2 - 16,384). After finding out the optimum thread count, speedup (Equation 1) and efficiency (Equation 2) were computed.

\[
Speedup = \frac{T_s}{T_p}
\]  \hspace{1cm} (1)

\[
Efficiency = \frac{T_s}{pT_p}
\]  \hspace{1cm} (2)

Where:

- \(p\) = number of processing cores
- \(T_s\) = average time for single threaded implementation
- \(T_p\) = average time for multi-threaded implementation
Results and Discussion

After compiling the source code, an initial report of the running time (in seconds) from 30 repetitions of each operation was generated. Table 1 summarizes the results (the average and standard deviation). Furthermore, a visual representation of the averages is shown in Figure 1. Given the results in Table 1, it shows that having a thread count equal to the number of processing cores with static and guided scheduling (i.e. E-Threads and F-Threads) are more than 4x faster than the single thread implementation. With a standard deviation less than 1, it strongly suggests of few outliers. But creating thread count equal to the size of the input vector (i.e. N-Threads) is 20.81% slower than single threaded worsened with many outliers since the standard deviation is more than 1.

Table 1: Average and Standard Deviation for the running time of each operation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average ($\mu$)</th>
<th>St Dev ($\sigma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Thread</td>
<td>2.5822</td>
<td>0.0029</td>
</tr>
<tr>
<td>N-Threads</td>
<td>3.1195</td>
<td>1.2565</td>
</tr>
<tr>
<td>E-Threads</td>
<td>0.6419</td>
<td>0.0335</td>
</tr>
<tr>
<td>F-Threads</td>
<td>0.5964</td>
<td>0.0393</td>
</tr>
</tbody>
</table>

Figure 1: Average running time graph of the different operation

Analysis for significant difference among the average reveals a $p$ value of less than 0.0001 (Table 2). This indicates that each average is significantly different at 99% confidence level. This would lead further to a pairwise comparison using Tukey’s HSD as shown in Table 3. In this comparison, E-Threads and F-Threads are significantly faster than single threaded, while N-Threads is significantly slower than single threaded at 99% confidence level. On the other hand, no significant difference existed when using different scheduling techniques, given that both have similar thread count.

Table 2: Tukey’s HSD table of the various operations

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>$p$-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treatments</td>
<td>3</td>
<td>153.778</td>
<td>51.2594</td>
<td>122.66</td>
<td>&lt;.0001</td>
</tr>
<tr>
<td>Error</td>
<td>16</td>
<td>45.8601</td>
<td>0.3953</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>119</td>
<td>119.638</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: ANOVA table of the various operations.

<table>
<thead>
<tr>
<th></th>
<th>N-Threads</th>
<th>E-Threads</th>
<th>F-Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Threaded</td>
<td>p&lt;0.01</td>
<td>p&lt;0.01</td>
<td>p&lt;0.01</td>
</tr>
<tr>
<td>N-Threads</td>
<td>p&lt;0.01</td>
<td>p&lt;0.01</td>
<td>*n/s</td>
</tr>
<tr>
<td>E-Threads</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n/s – not significant

After the first report, another report was generated to show the varying thread count with its corresponding performance (running time). Graphing the results of the different scheduling techniques (Figure 2) indicated that the optimum thread count is 128. Since the test machine has eight (8) logical cores, the optimum processor core to thread count ratio is 1:16. Furthermore, when too few (e.g. 2) or too many threads (e.g. 16384) were used, performance degraded.

Finally, performance of multi-threading over single threading is summed up on Table 4. Although multi-threading achieved significant speedup, its efficiency is still not as good as single threading.
Conclusion and Recommendation

This evaluation presents the significant speedup due to multi-threading in a multi-core processor. To optimize the execution time of a parallelizable function, it is recommended to create 1:16 processor core to thread count ratio. Threads that are thousands of times more than the processing cores would significantly degrade the performance and is even worse than traditional single-threaded implementation. Moreover, it doesn’t matter which scheduling technique is to be used when multi-threading for this particular study.

Multi-core processor can significantly increase the speedup of a parallelizable algorithm. Thus, consumer computing products are advertised as more processing cores equals more performance. While this statement is generally true, not all algorithms are parallelizable. Some are inherently serial. We suggest that further performance evaluations would be done when inherently serial and embarrassingly parallel tasks would be combined.

References


