Practical multicore programming using Java Executer Frame work and OpenMp – a comparison

By

Lasith Eranda Haputhanthiri

(lasithh@gmail.com)

Introduction

In the last 4-5 decades, the speed of the microprocessors was increased according to the Moore's law. Micro processor vendors tend to increase the number of transistors packed in a micro processor chip to increase the performance. But few years back (around 2003) they were forced to stop this approach. The reason was as the speed of a single processor was increased more than 5GHz, the chip was melt due to the heat produced by the transistors. So this fact defeated the Moore's law and also the single core micro processor architecture. Due to this upper bound of speed, micro processor vendors tend to increase the performance of the chips by increasing the number of cores of in a single micro processor. This scenario gives the birth to the Multicore Microprocessor architecture and Multicore programming.

So according to Mr. Herb Sutture for software developer's, the free lunch of performance is over by ever faster clock speeds [1]. So now they have to develop programs which are capable of working with multiple processing cores. But the picture is not attractive as it seems. Introduction of parallel computing introduce a set of side effects and challenges with it. So the task of software engineers is to make sure that the advantages of parallel computing over weight the disadvantages.

Parallel Computing – The Challenges

So to the point, how you make a program parallel. The answer is simple, using a multithreaded execution. In normal multithreaded program what you do is, you create a set of threads and let the processor to switch among these threads as the execution continues. As the switcing speed of the micro processor is very high, the end user will feels as all the threads are executed in parallel. This is called the psudo parallel execution. But in the jargon of multicore programming, you can execute a set of threads in parallel by assigning a thread per each core. Here you should notice that the fundamental never changes. At any given time a single core can perform only a single task. So the maximum number of thread which can be executed in parallel is bounded by the number of execution cores in the processor. If the number of active threads out numbered the number of execution cores, the threads will have switch among each other. This each thread switch comes with an overhead. So the programmer has to decide the optimal number of active threads in a given time.

And by nature the multi threaded programs are more complex and buggy than sequential programs due to the issues like race condition. So the programmer will have to take an extra effort in the maintenance and the development.

For the explanation purposes, I'll take an example program. The target of the program is to find the result of R = A*B + C * D^{-1} * E + F * G formula. Here each item is an n X n matrix. The execution of the program can be conducted as follow.

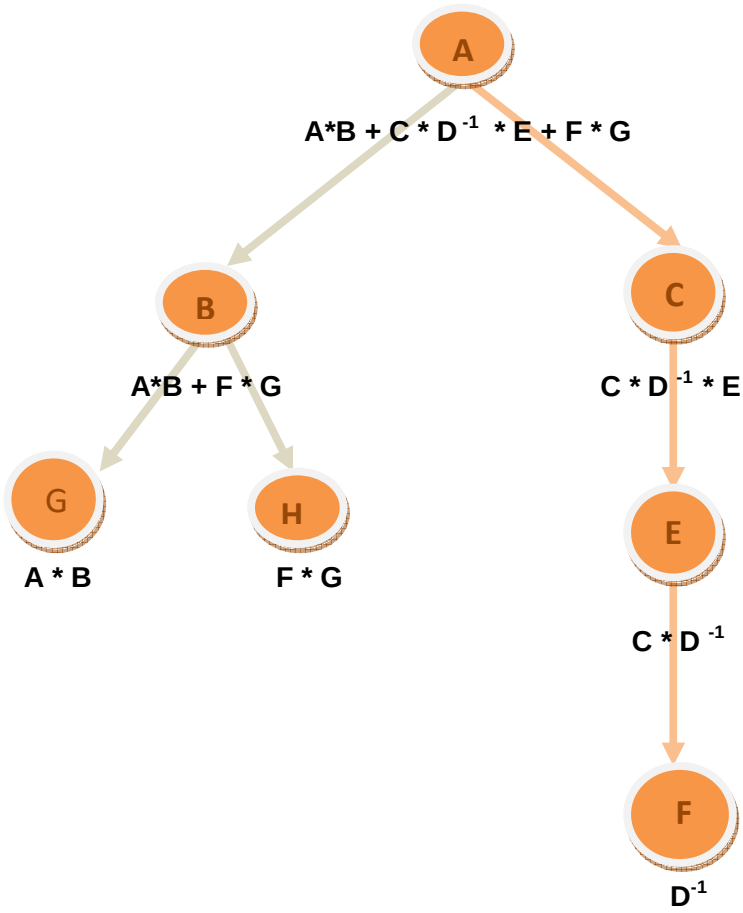


Figure 1:set of steps for the entire calculation

You should not take the wrong idea that you can get the speedup up to the number of available processing cores. As an example if you have a processor with 2 processing cores, normally you cannot get a speed up of 2. The reason is you cannot make a program 100% parallel. Many programs contain sections which requires serial execution. In the above example the F→E→C→A should execute serially. The reason is each node of above tree depends on the

results of its child nodes. More importantly, for any given program there exists this kind of dependencies for the top to bottom. And it also contains a critical path. This is the longest dependency chain of a given program[1]. So you cannot finish the program before this critical path finishes it's all the serial executions. In above example the critical path is A→C→E→F. If you delay the execution of a any step in this path in time t, the possible execution completion time also delayed by t. This is a very interesting property.

So while writing a multithreaded application you may give the priority to the tasks of the critical path. A good scenario may be, you should dedicate a single core for the execution of the steps of the critical path all the time and use other cores for the execution of steps of non-critical paths. This seems to be a good strategy, but not necessarily be the best.

Practical Parallel Computing

Major problem up to this point is how we can assign a thread to a particular core. Or how do we tell the computer that we want to execute two or more threads parallel? The good news is that you don't want to do any of these tasks by your self. There are a lot of frameworks available which helps you to develop multicore programs without a much effort. Intel's thread building block, OpenMp, Click++, Java Executor Framework are some examples. Here I consider OpenMP and JEF to evaluate efficiency of the multicore programs over traditional sequential programs.

Java Executor Framework (JEF)

As name implies JEF is for java. Being a part of java, this framework is fully object oriented. It has an interface called Executor which handles all the thread management tasks.

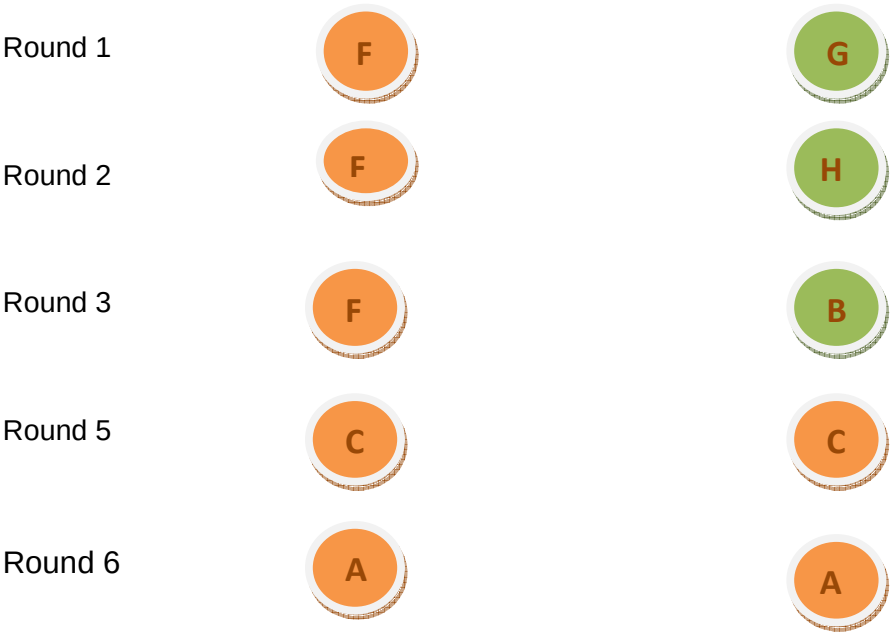
Programmer has to create the set of tasks and submit them to the executor framework appropriately. Each submitted task is assigned to a worker thread and these worker threads are assigned to execution cores. And if there are a large number of active threads, framework also handles the thread switching tasks. The entire above task are performed by the framework. This framework is well organized and easy to use.

Open Multi-Processing (OpenMp)

OpenMp is for c++. This framework provides very large set functionalities for parallel computing. Now OpenMp compatible compilers are provided by a set of well known organizations. This framework take the parallel computing in to a very deeper level such that a programmer can executes the iteration of a loop in parallel.

Experiment

I have implemented a two parallel algorithms to solve above mathematical problem using both JEF and OpenMp. I have divided the entire calculation in to a set of steps as shown in the above *figure1.*Here the step F takes a much longer time compared with other steps. I have executed these steps as follows.



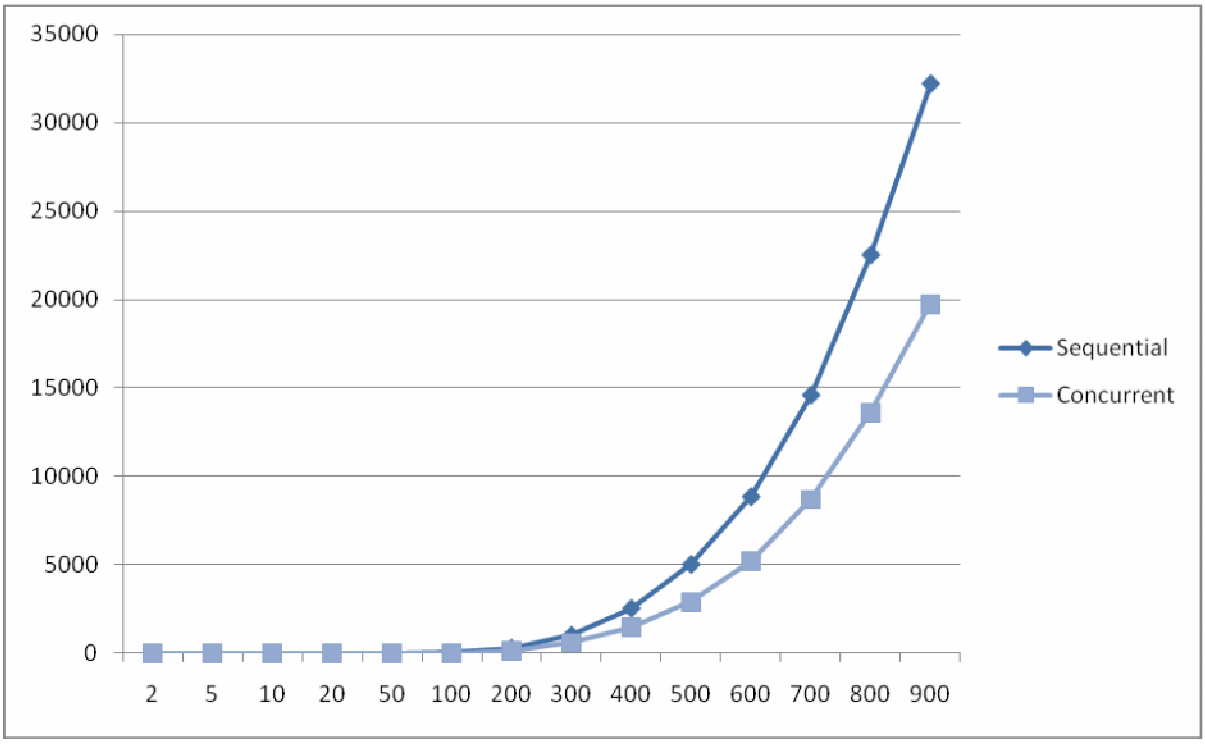
You may notice that I have dedicated at least one core for the executions of the steps of the critical path. As my test machine has two processing core I made sure that there is only two active threads available at any given moment. In round 5 and 6 I have decomposed a single steps in to two parallel steps so they can be executed in two parallel threads. You can notice that the execution cannot be continued further from round 3 until the result of step F is available. This is a dependency that the system have. If step F took a lot of time then we have to wait in round 3 until it finishes the execution.

Results

Here n is the degree of the matrixes. And sequential and Concurrent columns indicates the execution time in each approach.

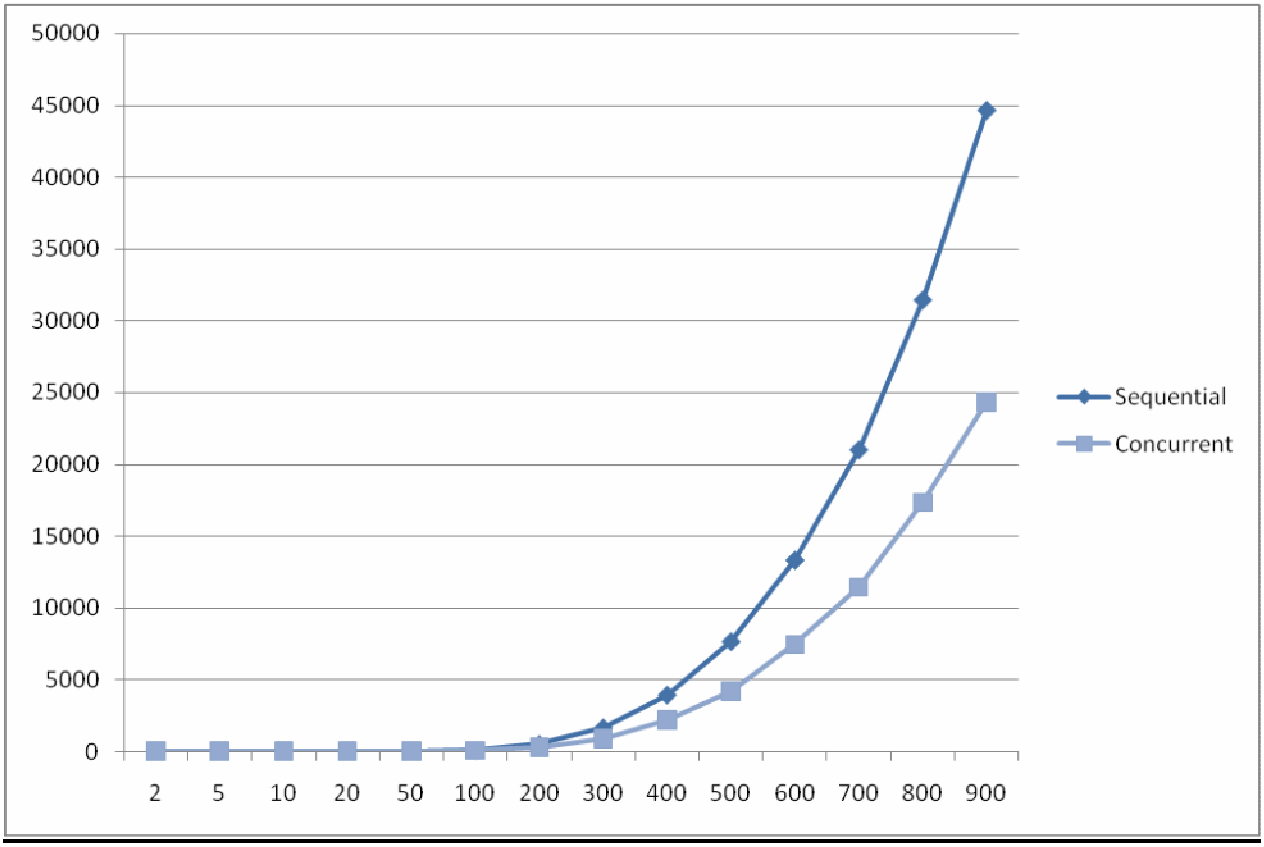
Using Java Executor Framework

n	Time in Sequential Approach(ms)	Time in Concurrent Approach(ms)
2	0	16
5	0	16
10	0	16
20	15	16
50	31	31
100	62	47
200	312	187
300	1047	625
400	2563	1500
500	5047	2937
600	8869	5234
700	14610	8719
800	22531	13625
900	32203	19766

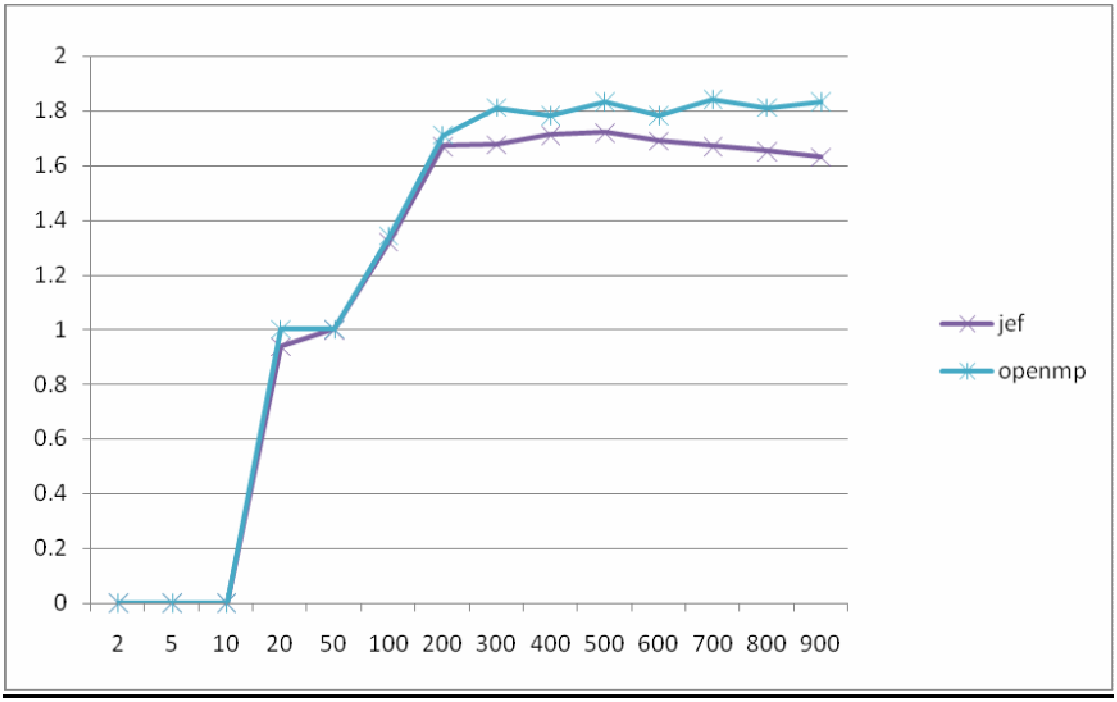


Using OpenMP

n	Sequential	Concurrent
2	0	0
5	0	0
10	0	0
20	10	10
50	16	16
100	63	47
200	484	282
300	1640	907
400	3922	2204
500	7641	4172
600	13313	7438
700	21000	11438
800	31438	17344
900	44640	24328



Performance Gain



Conclusion

In the sequential approach all the calculations were carried out in a sequential approach. In concurrent approach calculations were carried out parallel. The experiment was carried out in a AMD Athlon 64 X2 Dual Core 2.61processor. As processor has only two cores, only two threads can be executed parallel. So in the program there are only two threads at a given time. This reduces the unnecessary thread creation and switching overhead.

In both JEF and OpenMp approaches efficiency of the sequential approach is equivalent or better than the concurrent approach for small values of n. This is due to the thread creation and switching overhead. So for n < 50 values, thread switching and creation overhead is larger than the advantage gained by the parallel approach. But when the value of n grows larger, the concurrent approach has a significant performance gain.

Here you can notice that even though I have two processing cores I cannot have a performance gain of 2. For OpenMp maximum performance it lies around 1.81 and for JEF it's around 1.71. So you can notice that even though both frame works gives almost same results OpenMp is slightly better than JEF. The main reason for this situation is that OpenMP provides concurrency in a very lower level. Such that you can even parallel the iterations of a for loop. But while choosing an Multi-processing framework, there may be some other facts that you may want to consider. As an example if you have a java program and you want to make it multi-

core compatible, your best choice may be JEF. So it is up to the programmer to choose the best framework according to a particular situation.

References

How to survive the multicore software revolution by Charles E. Leiserson And Ilya B. Mirman. 1 - 2