A Study on Parallel Computing and its Approaches

Prashant Chaudhary, Hari Mohan

Abstract— Engineering tools are used for a variety of analyses work ranging from bolted joints, air-flow analysis to finite element methods. Despite the computing power present today, engineering analyses often take long time to complete on a single machine. The Honeywell Engineering tools portfolio consists of applications both old and new but not all of them are adaptable to a parallelized or clustered environment. This paper tries to act as guide to engineers in using modern software libraries and clustered hardware to migrate from single core applications to multi-core applications.

Index Terms— parallel computing, engineering tools, multithreading, optimization, concurrency, analyses tools, parallelism.

1 Introduction

ENGINEERING tools division has a wide portfolio of projects which covers numerical analyses, estimations and interpolations. Often these applications are desktop based and are meant for a single computer. While most problems and requirements are satisfied by the current application setup; new updates and scalability to larger problem sets/requirements can be tedious for a single core system. Jobs that could significantly affect performance include numerical corrections and algorithms. On non-numerical side, there could be aspects such as user interface complexity, report generations and database retrieval.
Parallel computing is a broad field and could cover multiprocessor, multicore as well as multi-node setups. While parallel computing is proven to improve processing speed and efficiency, it is still limited by Amdahl's Law. Amdahl's Law can be stated per definition in Rodgers [1] as "The theoretical speedup of the entire task increases by increase in improvement of resources of the system regardless of the magnitude of its improvement"
This means that the performance of a parallelized system is limited by the amount of code parallelized.

$$S_{latency} = \frac{1}{(1-p)+\frac{p}{s}}$$

Where:
- is the theoretical speedup of the whole task.
- is the speedup in latency of parts that benefit from improvement in resources.
- is the percentage of execution time which benefits from improvement in resources before the improvement.

2 Parallel Computing Approaches

The usage of parallel computing could be governed by the following queries:

- *Compute resources* – How many and of what specifications?

Compute resources may mean multiple processor core on a single chip or multiple processors on a single die or even multiple systems on a network. The number gives us a scalability limit and the specification helps us to decide how much computationally intensive operations could be allocated along the compute units.

- *Memory resources* – How many and of what nature? Are they shared or independent?

Memory resources help us identify bottlenecks on access. If memory is shared, then we may need a mechanism to avoid race conditions from occurring. Independent memory resources allow compute units to work on their local memories hence, we achieve a higher degree of parallelism.

- *Communication/IO* – How do the computers communicate with each other?

Communication and IO could be via a shared memory (multicore), a common bus (multi-processor) and via networks (multinode). Due to this variation, different standards are used for communication such as: *OpenMP* for multi-threaded setup; *Message Passing Interface* and *Remote Procedure Calls* for multinode setup.

Apart from that, the type of networks also govern performance. A supercomputer may use *Infiniband* or torus-interconnect networks for a high throughput - low latency architecture. On the other hand, distributed systems may use local LANs or Internet for communication. The performance for the latter is governed by network speed, cables etc.

- *Control* – What strategies control use of resources?

Concurrency/Multithreading libraries found in many programming languages have the facility to handle race conditions or contentions. These facilities help avoid deadlocks in resource management and impose a certain order in access of resources. Based on the need, one may use critical sections or mutex locks to govern access of non-parallelizable sections of code or memory. In context of distributed computing, this corresponds to distributed lock managers.

The memory and communication elements could be subsumed as *Resources* because there is an overlap of these two categories in certain systems.

2.1 Multithreaded Parallelism

Parallelism at program level deals with usage of standardized libraries or language specific APIs which provide thread level control. The main standard adopted across C/C++ and FORTRAN is OpenMP, which is suitable for shared memory or distributed shared memory based architecture models. OpenMP in programming uses #pragma directives in C to specify sections of concurrent control.

Apart from standards, there are platform independent multiprocessing solutions available in various programming languages such as Thread class and Runnable interface in Java, TLP (Task Level Parallelism) in C#, thread classes in C++ etc. However, it must be noted that Ruby MRI and Python has threading implementations which cannot run in parallel due to GIL (Global Interpreter Lock). It could be used for concurrent programs though.
Multithreading is possible in processors with multiple GPU or CPU cores. It poses an advantage over simply running processes in parallel by reducing load on processors.

Advantages:

o Threads don't need any communication mechanism like message passing or mailboxes.

o They help keep various aspects of a process responsive to the user while performing some task in background.

o They run as single process and occupy the memory footprint for the same.

o Most threading libraries follow fork-join model [2] which means that all threads post-execution will join with the parent process which spawned them, hence ensuring no process zombies exist in memory.

Disadvantages:

o Multithreading is possible which only architecture that has multiple cores.

o Prone to deadlocks if resources are mismanaged.

o Can be applied only to tasks which can be decomposed into independent sub-tasks.

o If one of the tasks was finished earlier, that thread remains idle. This is counter-productive to what we would like to achieve in multithreading.

2.2 Multi-node Parallelism

This idea includes separating a gigantic undertaking among a wide range of hubs/frameworks. This is from multithreading as we have entry to whole arrangements of processors, memory and IO to play out a few assignments. At this scale, we need to manage disintegration of issues, association between the figure assets, performing calculations and combining the results. [3] Multi-hub parallelism is shown in supercomputers where there is tight coupling between numerous processors and in addition appropriated registering; prevalently in setting of Big Data. In Multi-hub, we can arrange parallel design into two classes: -

- *Distributed Memory*
- *Hybrid Distributed Shared Memory*

• *Prashant Chaudhary is currently an Engineer at Honeywell Technology Solutions Pvt. Ltd at Bangalore, India.*
• *Hari Mohan is an Engineer at Honeywell Technology Solutions Pvt. Ltd at Bangalore, India.*

Distributed Memory

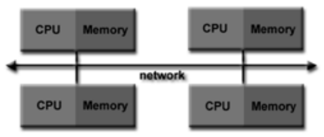

Figure 1: Distributed Memory Architecture [3]

General Characteristics:

•Like shared memory frameworks, distributed memory frameworks differ generally yet share a typical trademark. Distributed memory frameworks require a correspondence system to associate inter process or memory.

• Processors have their own nearby memory. Memory addresses in one processor don't guide to another processor, so there is no understanding of global address space over all processors.

•Because every processor has its own neighborhood memory, it works freely. Transforms it makes to its nearby memory have no impact on the memory of different processors. Subsequently, the idea of cache coherency does not have any significant bearing.

•When a processor needs access to information in another processor, it is normally the assignment of the software engineer to expressly characterize how and when information is conveyed.

• Synchronization between errands is in like manner the developer's duty.

Advantages:

• Memory is adaptable with the quantity of processors. Increment the quantity of processors and the span of memory increments proportionately.

• Each processor can quickly get to its own memory without impedance and without the overhead caused with attempting to keep up worldwide store coherency.

• Cost viability: can utilize item, off the rack processors and systems administration.

Disadvantages:

• The developer oversees a significant number of the points of interest related with information correspondence between processors.

• It might be hard to outline information structures, in view of worldwide memory, to this memory association.

• Non-uniform memory get to times information dwelling on a remote hub takes more time to access than hub nearby information.

Hybrid Distributed Shared Memory

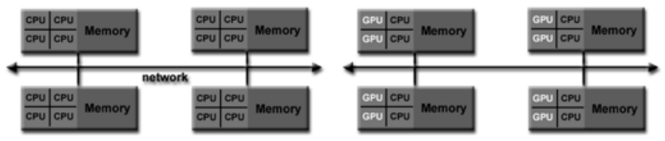

Figure 1: Distributed Shared Memory Architecture [3]

General Characteristics:

- The biggest and speediest PCs on the planet today utilize both shared and conveyed memory models.
- The shared memory part can be a mutual memory machine as well as design handling units (GPU).
- The circulated memory segment is the systems administration of numerous common memory/GPU machines, which know just about their own memory not the memory on another machine. In this manner, organize correspondences are required to move information starting with one machine then onto the next.
- Current patterns appear to demonstrate that this kind of memory design will proceed to win and increment at the high end of processing for a long time to come.

Advantages and Disadvantages:

- Whatever is regular to both shared and appropriated memory structures?
- Increased adaptability is an imperative preferred standpoint
- Increased developer multifaceted nature is an imperative drawback

3 PARRALLELIZATION PARADIGM

Most legacy toolkits are written in C/C++, FORTRAN and MATLAB. Many use iterative methods for approximation of numerical problems, which could make it tricky to implement parallel programming. Certain observations from past projects (HAM, CFCAD):

- Batch processing could be easily parallelized if there is no interdependence between data within iterative jobs.
- Notably, comparative statements such as equals (==), not equals (! =), greater than (>) etc., take greater resources than simple assignments. One can use atomic statements or critical sections to separate out assignments.
    - Atomic statements would work only if they are not special type assignments such as structs or classes.
    - Critical sections have an overload of their own, hence one must leverage the program size with critical section usage to attain optimal performance.
- Certain aspects are not parallelizable:
    - GUI tasks
    - I/O tasks

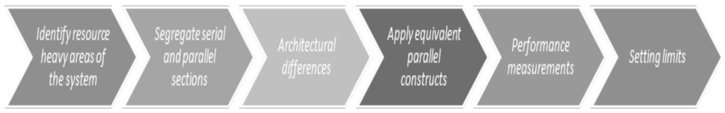
Identify resource heavy areas of the system
Segregate serial and parallel sections
Architectural differences
Apply equivalent parallel constructs
Performance measurements
Setting limits

Figure 3: A single threaded application can be transformed into a parallelized one, step-wise

1. Identify resource heavy areas of the system: One can approach various stakeholders of the project to discuss features which take long time to process in the application. Developers and Leads can guide us to areas of code which bring the system to a drag.

2. Segregate areas which are strictly serial or capable of concurrency: Note that GUI usually runs on a single thread which cannot be sub-divided any further. Similarly, I/O cannot be parallelized either. Certain computations are inherently serial in nature, for e.g.: Fibonacci series requires previous results for computing next. Newton-Raphson method for matrices is sequential as well.

3. Architectural differences: Performances in 64-bit architectures are higher than 32bit architectures especially if 64-bit primitives are being used in the application.

4. Apply equivalent parallel constructs: Once we've identified target areas which could be parallelized, we need to start enveloping the code areas around parallel constructs. Sometimes, this can be done readily using macros; as in case of OpenMP. But in others, like MKL or MPI, we may need to re-implement certain methods with appropriate data types and functions. In case OO languages like Java/C#, we may have to use threading frameworks.

5. Performance measurement: Unless there is a performance benefit, parallelization is of no use. We must use appropriate benchmarking facilities to check difference in running time. Note that most timer functions measure CPU cycle time, which may be misinterpreted as actual running time. To avoid this, use time elapsed from the computer's inner clock time or use external measuring device such as a stopwatch (however it may not be as precise as inner clock time differences).

6. Set limits: Per Amdahl's law, there is a limit to amount of parallelization possible. Adaptive code sections which control amount of multi-threading based on need will make sure the application runs at optimal performance.

4 PRELIMINARY RESULTS

In order to understand how parallelism can be beneficial to execution of tasks, we tested two different programs on a single machine with and without enabling multithreading. The first program to be used is matrix multiplication [4], which is an easily parallelizable program and is used in many benchmarks. Matrix multiplication offers the opportunity to compute each cell independently of the other values in matrix, which makes it easy to split into sub-tasks. The target machine runs on Intel i5-5300U processor, which has 2 cores and 4 threads available for multithreading. We have used OpenMP 2.0 for multithreading the execution. In these graphs, the lower the line, the better is the performance exhibited.

In the next test, we tested Floyd-Warshall algorithm for single threaded and multithreaded environments. Floyd-Warshall is also known as 'All Pairs Shortest Paths' algorithm. It is a network protocol used for finding shortest paths between any two given nodes, and this is done using transitive closure. This nature of transitivity somewhat restricts the amount of parallelization further, as there are more dependent variables which must be handled to avoid race conditions.

A naïve implementation of this algorithm involves three nested loops to get distances via transitive closures. We have used the same for judging performance between its single-threaded and multi-threaded programs.

TABLE 1
FLOYD-WARSHALL ALGORITHM RESULTS TABULATED FOR SINGLE-THREADED AND MULTI-THREADED VARIANTS WITH OBSERVED SPEED-UP

Matrix Orders	Multi-threaded program time (in seconds)	Single-threaded program time (in seconds)	Speed-up observed
100	0.014	0.024	1.714
200	0.066	0.118	1.204
300	0.169	0.251	1.485
400	0.341	0.509	1.492
500	0.565	1.192	2.109
1000	4.700	11.169	2.376
2500	127.947	277.306	2.167
5000	1190.56	1779.33	1.494

A few results covering higher matrix orders (2500 and 5000) have been pruned from the graph to fit its scale. In case Floyd-Warshall implementations, the single-threaded program was roughly 1.4 times slower than the multi-threaded program. The peak speed-up of 2.1x was achieved at matrix order of 500.

Its observed from both cases that:

- There is a speed-up factor associated with every increment or subsequent matrix orders
- Increasing size does not necessarily translate to better performance, as observed in second experiment, this could mean hardware limitations or the amount of parallelizability one could induce into the program/algorithm
- In case of matrix multiplication, we get slightly higher speedups with increasing matrix orders. But even this is expected to break down, as we reach hardware limits.

5 PERFORMANCE STUDY

As a part of our study, we applied parallel programming paradigm to two different internal projects used at Honeywell MCoE.

5.1 HAM

HAM (Honeywell Autocode Manager) is a productivity tool set and process for end-to-end controls analysis, design capture, automated requirements-based testing, and embedded code generation. The program's SCV test suite largely uses MATLAB code, and so the parallelization constructs within MATLAB were used for increasing performance.

We ran the SCV tests on a workstation to better scale our resources for parallel computing.

TABLE 2
MATLAB PROFILER RESULTS AFTER SCV RUN

Function name	Total time	% time
runSCVAllModels	4472 s	78.2
runSCVAllModels_parallel	1243 s	21.8

In this case:

Time taken without parallelism = 4472 seconds (approx. ~74 minutes)

Time taken with parallelism = 1243 seconds (approx. ~20 minutes)

Here *"runSCVAllModels"* is a serial implementation which conducts the tests in sequential order, whereas *"runSCVAllModels_parallel"* conducts them in parallel.

We see a performance improvement of about **54 minutes,** roughly three times the performance.

Conclusion

1. MATLAB Parallel computing framework scales better to larger problem sets.
2. Better hardware leads to better results.
3. There is a substantial increase in performance of parallelized version, almost three times.

5.2 CFCAD

Cooled Airfoil Design Tool is used for interpolation of HTC and temperature points using boundary conditions on blade surfaces. The program uses Intel MKL integrated with BLAS and LAPACK environments for parallelized matrix operations. We looked at certain points which could be used for parallelization and identified 3D space Octree generation and Solver stages as suitable for our purposes.

Octree Generation

The coordinate information from boundary conditions are used to generate Octrees for HTC and temperature respectively. We implemented parallelization by multi-threading the overall for-loop which called the code for generation of Octree at each node, and encapsulated certain sequential operations into critical sections. The end results of our comparison are tabulated below:

TABLE 3
CFCAD RESULTS ON GAS AND COOLANT MODELS

Code sections	**Time taken for gas model**	**Time taken for coolant model**
Temperature Octree	96.54 s	2674.49 s
HTC Octree	0.045 s	1172.84 s
Temp. Octree (parallel)	92.07 s	2321.96 s
HTC Octree (parallel)	0.041 s	1092.75

It is observed that:

- Due to movement of certain parts of code into critical sections, we have a certain loss of performance.
- However, we see roughly a 6% increase in performance with multithreaded Octree generation.
- The time taken for Coolant temperature points reduces by 300 seconds which is quite significant.
- A roughly 80 second reduction in case of Coolant temperature points is witnessed.
- It could be assumed that heavier processing when multi-threaded may see a consistent increase, nothing more or less than 6%.

Solver

In case of Solver code, much of the implementation uses routines specifically targeted towards Intel multi-core architectures. To measure performance benefits, we set certain Environment variables to a certain value to effectively turn off the parallelization present in these routines. The result for few segments of the code are tabulated as follows:

TABLE 4
SOLVER FUNCTIONS PERFORMANCE COMPARISONS

Code sections	**Time taken**	**Time taken in parallel**
Assemble()	35.28 s	29.74 s
IterateConduction()	156.50 s	18.89 s
SparseFunc()	18.361 s	15.254 s

- The most noticeable difference was in IterateConduction's first iteration. There is a performance increase of about 87.8%.
- There is a saving of about 15.6% in case of Assemble iterations with parallelization reducing roughly 5 seconds of the original duration.
- Given that the project ran on powerful Intel Xeon processors equipped with roughly 24GB RAM, the performance differences are not very much noticeable in many cases.

6 WHEN TO USE PARALLELISM

The fundamental questions a developer needs to consider before designing parallelized solutions to a problem are:

- Are there no ways to algorithmically increase performance of the software?
Many problems have advanced algorithms to significantly reduce time to solve a problem. If algorithms exist to solve the problem in nearly linear time (O (1) time complexity), there is no need to parallelize the solution.
- Could the problem size scale substantially over time?
If the software will deal with larger problem sets over time, it must scale accordingly to deal with them. One cannot guarantee up-to-date hardware all the time, so we might need to use concurrency, e.g. request traffic on an e-commerce website.
- Is it possible to parallelize the solution?
Parallelization is bound by the technologies worked upon, both in terms of software and hardware. Some systems may not have advanced multi-threading capabilities or the platform we work upon may not have support for concurrency. The problem itself may not be parallelizable if it cannot be resolved into independent sub-problems or if it is IO bound or strictly sequential, e.g. Factorials cannot be parallelized as we always need the next number's factorial to compute the first.
- Is it necessary to use parallel computing?
Sometimes the problem may not be suitable for concurrency. Such problems may be too trivial or may not require a speedup, if hardware advancements are sufficient.

7 CONCLUSION

We have tried to cover basic understanding and approaches one could utilize in parallel computing and tried to demonstrate the overall performance benefits and even a few possible pitfalls one may face. Developments in today's environment have led parallel computing to split into many diverse and sometimes overlapping fields of study, and is expected only to grow in importance in the coming years. Given our wide tools portfolio, we may benefit hugely from the resources and tools available today.

8 FUTURE SCOPE

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multiprocessor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing.
- There has been a greater than 500,000x increase in supercomputer performance, and it only seems to be growing.
- There have been ideas which intend to fuse Big Data with HPC to process extremely large data sets without compromising on fault tolerance nor performance.
- Big Data and Analytics sales are projected to reach $187 billion by 2019 from about $122 billion in 2015 [5]; this would require massive strides in the field of parallel computing.
- We could develop abstractions around IoT concepts

to handle parallel processing across diverse range of devices.

- Supercomputing performance has already reached Peta-Flops in speed, with the world's fastest being 93 Peta-Flops (Sunway TaihuLight) [6]; Exaflop computing is not far away.

Acknowledgment

We would like to thank Venkatarao Ganji for his encouragement to write this article. We would also like to extend our gratitude to Ravi Kiran from HAM team. It would be unfair to leave out the many comments and suggestions which came from reviewers of this article.

References

[1] Rodgers, David P. (June 1985). "Improvements in multiprocessor system design". ACM SIGARCH Computer Architecture News archive. New York, NY, USA: ACM. 13 (3): 225–231. Doi: 10.1145/327070.327215. ISBN 0-8186-0634-7. ISSN 0163-5964

[2] Michael McCool; James Reinders; Arch Robison (2013). Structured Parallel Programming: Patterns for Efficient Computation. Elsevier.

[3] Blaise Barney. Introduction to Parallel Computing. Livermore Computing.

[4] Chowdhury, Rifat. Parallel Computing with OpenMP to solve matrix Multiplication. UCONN BIOGRID REU Summer 2010. University of Connecticut.

[5] Davis, Jessica (2016-05-24). "Big Data, Analytics Sales Will Reach $187 Billion By 2019". Information Week.

[6] Clark, Jack; King, Ian (2016-06-20). "World's Fastest Supercomputer Now Has Chinese Chip Technology". Bloomberg.com.

[7] Java Documentation. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

[8] Parallel Processing in Java, https://blog.pavelsklenar.com/parallel-processing-java/

[9] Supercomputer performances over the years, Wikipedia.org.