Volume 1, No. 4, Nov-Dec 2010
ISSN No. 0976-5697
International Journal of Advanced Research in Computer Science
RESEARCH PAPER
Available Online at www.ijarcs.info

Overview of Application Performance on Multicore Environment

M.Narayana Moorthi.[*] , P.Mohankumar and Dr.J.Vaideeswaran,

School of Computing Science and Engineering,
VIT University, Vellore-14., TN ,India,
Mnarayanamoorthy@Vit.Ac.In
Pmohankumar@Vit.Ac.In
j_vaideeswaran@vit.ac.in

Abstract: When Programming Multicore Applications we need to consider the method of making use of the existing powerful multicore processors. The multicore processor provides a new challenge or issues to be taken into account for software developers to achieve higher performance in computing applications. The applications requires high speed process need to migrate the software from single core to multicore processor is the real challenge in front of us. Several factors determine whether the performance of an application improves on a multicore system. Bottlenecks in parallelism may arise at several levels of software application stack, and avoiding this problem is a challenging task. This proposed paper discusses the benchmarking metrics for applications on Multicore systems and to tune the performance of threaded applications with balanced load for each core and a basic understanding of how to gather meaningful benchmark data and tune applications on multicore systems. Before benchmarking our application on a multicore system, we need to understand the characteristics of the application being measured. If the application is multithreaded then in theory the performance of the application will increase as the number of cores increases in a system. How much the performance of the application actually increases will depend on what else is running on the system, how effectively the application is threaded, and the nature of the hardware platform. Here are few issues we need to analyze. They are 1.How many threads are used by our application. If our application is multithreaded, whether the application is designed to make use of all cores in a multicore system .2. Knowing performance on a single core platform, what is the expected performance on a system with N-Cores? The Amdahl's Law basically predicts how the performance of a parallel application changes as the number of cores increases based on the amount of serial and parallel work in that application. The mismatch between measured parallel application performance and predicted performance are to be investigated.

Keywords: Multicore, Parallel Program, Interprocessor Communication

I. INTRODUCTION

Building parallel versions of software can enable applications to run a given data set in less time, run multiple data sets in a fixed amount of time, or run large-scale data sets that are prohibitive with unthreaded software [4]. The success of parallelization is typically quantified by measuring the speedup of the parallel version relative to the serial version. In addition to that comparison, however, it is also useful to compare that speedup relative to the upper limit of the potential speedup. That issue can be addressed using Amdahl's Law and Gustafson's Law. The faster an application runs, the less time a user will need to wait for results. Shorter execution time also enables users to run larger data sets in an acceptable amount of time. One computed number that offers a tangible comparison of serial and parallel execution time is speedup. Speedup is the ratio of serial execution time to parallel execution time. For example, if the serial application executes in 6720 seconds and a corresponding parallel application runs in 126.7 seconds (using 64 threads and cores), the speedup of the parallel application is 53X (6720/126.7 = 53.038).

Related to speedup is the metric of efficiency. While speedup is a metric to determine how much faster parallel execution is versus serial execution, efficiency indicates how well software utilizes the computational resources of the system. To calculate the efficiency of parallel execution, take the observed speedup and divide by the number of cores used. This number is then expressed as a percentage. For example, a 53X speedup on 64 cores equates to an efficiency of 82.8% (53/64 = 0.828). This means that, on average, over the course of the execution, each of the cores is idle about 17% of the time.

II. NEED FOR PARALLELISM

Recent change in computing and communication with Multicore processors [7] [8] [9] is developing an application that use all of the cores to their full support is a challenge for all. Writing concurrent programs for multicore is difficult and is of increasing practical importance. If we want to run our program faster, we need to learn parallel programs. The Amdahl's Law is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. In case of parallelization , Amdahl's Law [1][2][3][5] states that if P is the proportion of a program that can be made parallel and (1-P) is the proportion that can not be parallelized, then the maximum speedup that can be achieved by using N –Processors is 1/(1-P)+P/N. This analysis is required to know the cause of poor application scaling with increasing number of cores. We should always benchmark our multithreaded application on systems with 2, 4, ---N cores in order to get an accurate picture of how our application scales. The processor Affinity can also be tested to ensure that data critical to the thread stays in a given cores cache which can improve performance. When benchmarking applications on multicore systems we need to pay attention to not only the overall performance of the application, but also to how the performance of the application changes as the number of cores increases. We need to decide benchmark metrics that are important to us, and then test our application using benchmark workloads that provide meaningful benchmark [6] results to tune our applications. The performance of our application changes as the number of cores increases. One of the performance inhibiting factors in threaded applications is load imbalance. Balancing the workload among threads is critical to application performance. [6] The key objective for load balancing is to minimize idle time on threads and share the workload equally across all threads with minimal work sharing overheads. Generally mapping or scheduling of independent tasks to threads can happen in two ways: Static and Dynamic. When all tasks are the same length, a simple static division of tasks among available threads dividing the total number of tasks into equal sized groups assigned to each thread is the best solution. Alternatively when the lengths of individual tasks differ, dynamic assignment of tasks to threads yields better solution.

III. LITERATURE SURVEY

For the past 30 years CPU designers have achieved performance gains in three main areas like clock speed , execution optimization and cache design [13] [14]. Increasing clock speed is about getting more cycles, running the CPU faster more or less directly means doing the same work faster. Optimizing execution flow is about doing more work per cycle. Increasing the size of On-Chip Cache is about staying away for RAM. A fundamentally important thing to recognize about this list is that all of these are concurrency agnostic. Speedups in any of these areas will directly lead to speedups in sequential (Non-parallel, single threaded) applications. For the near term future the performance gains in new chips will be fueled by three main approaches. They are Hyper Threading, Multicore and Cache. Hyper threading is about running two or more threads in parallel inside a single CPU.A limiting factor in hyper threaded CPU is it has one cache, one integer math unit and one Floating point unit. Hyper threading is sometimes cited as offering a 5% to 15% performance boost for reasonably well written multithreaded applications. Multicore is about running two or more actual CPUs on one chip. And the die cache sizes can be expected to continue to grow. The following table 1, and 2 describes the existing multicore processors and the different applications [12] and their performance improvements.

Table I: Specifications Data for Top Performing Multicore Processors

Capability	Intel	AMD
Processor Nomenclature	Core 2 Duo E6850	64 X 2 6000t
Processor Speed	3GHZ	3GHZ
Instruction Set	SSE,SSE2,SSE3	SSE,SSE2,SSE3
Power	65W	125W
Transistors	291 Million	227 Million
Cache	L1-32KB / L2 - 4MB	L1-64KB / L2 – 1MB

Table II: Specifications of different applications

Application Environment	Dual Core Intel	Dual Core AMD
3D Gaming	80 % increase over top Intel single core	80 % Increase over top single core AMD
Anti Virus Scans	100%gains	30%gains
File Archiving(Backups)	30% gains	30%gains
Floating Point Calculations	20 % gains	50% gains

The following are the issues or challenges [9] the application developers need to focus on.

1. Software Decomposition into instructions (or) sets of tasks that need to execute simultaneously.
2. Communication between two or more tasks that are executing in parallel
3. Concurrency accessing or updating data by two or more instructions or tasks
4. Identifying the relationships between concurrently executing pieces of tasks
5. Determining the optimum or acceptable number of units that need to execute in parallel
6. Creating a test environment that simulates the parallel processing requirements and conditions

IV. LOAD BALANCING TECHNIQUES

Load balancing an application workload among threads is critical to performance. The key objective for load balancing is to minimize idle time on threads. Sharing the workload equally across all threads with minimal work sharing overheads results in fewer cycles wasted with idle threads not advancing the computation, and thereby leads to improved performance. However, achieving perfect load balance is non-trivial, and it depends on the parallelism within the application, workload, the number of threads, load balancing policy, and the threading implementation. An idle core during computation is a wasted resource, and when effective parallel execution could be running on that core, it increases the overall execution time of a threaded application. This idleness can result from many different causes, such as fetching from memory or I/O. While it may not be possible to completely avoid cores being idle at times, there are measures that programmers can apply to reduce idle time, such as overlapped I/O, memory prefetching, and reordering data access patterns for better cache utilization.

Similarly, idle threads are wasted resources in multithreaded executions. An unequal amount of work being assigned to threads results in a condition known as a "load imbalance." The greater the imbalance, the more threads will remain idle and the greater the time needed to complete the computation. The more equitable the distribution of computational tasks to available threads, the lower the overall execution time will be.
As an example [4], consider a set of twelve independent tasks with the following set of execution times: {10, 6, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1}. Assuming that four threads are available for computing this set of tasks, a simple method of task assignment would be to schedule each thread with three total tasks distributed in order. Thus, Thread 0 would be

assigned work totaling 20 time units (10+6+4), Thread 1 would require 8 time units (4+2+2), Thread 2 would require 5 time units (2+2+1), and Thread 3 would be able to execute the three tasks assigned in only 3 time units (1+1+1). Figure 1(a) illustrates this distribution of work and shows that the overall execution time for these twelve tasks would be 20 time units (time runs from top to bottom).

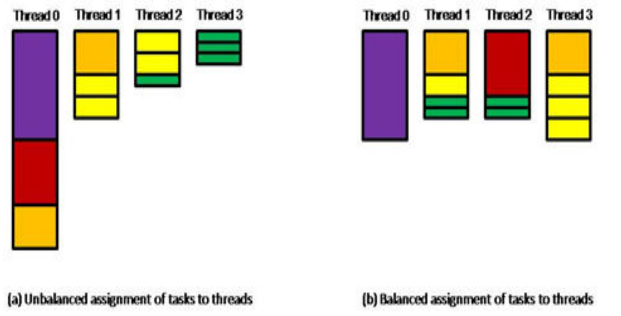

Thread 0
Thread 1
Thread 2
Thread 3
(a) Unbalanced assignment of tasks to threads
(b) Balanced assignment of tasks to threads

Figure 1. Examples of task distribution among four threads.

A better distribution of work would have been Thread 0: {10}, Thread 1: {4, 2, 1, 1}, Thread 2: {6, 1, 1}, and Thread 3: {4, 2, 2, 2}, as shown in Figure 1(b). This schedule would take only 10 time units to complete and with only have two of the four threads idle for 2 time units each.

The load balancing is the method or technique by which the overall task is getting distributed or assigned to all the cores evenly. The consideration here is that when the work load is large we should ensure that all the cores are busy and when the work load is small we should ensure that the energy is saved by not utilizing many cores. So the application developers for parallel programming should have necessary tools and techniques for load balancing. We should design the model for parallel programming which is flexible enough for tomorrow's processor evolution. The issues or areas we need to focus includes the potential parallel programming model, IO and File systems , Operating system issues and memory utilization, and the compilers. We need to analyze the scalability of a number of load balancing algorithms which can be applied to problems that have the following characteristics: The processor can be assigned a work which is the part of the overall work load., and the size of the work assigned is not constant ., ie it may vary. These kind of problems can be assigned to the multiple cores by proper load balancing techniques. The load balancing assures good performance in the multicore systems. The simple concept of dynamic load balancing is as follows. The task or work load is given to sub task generator which generates the mutually independent subtasks and puts them in to the allocation unit which distributes the sub tasks to the processing elements so as to balance the work load. Here the processing elements can be busy or idle. After finding the idle processing elements the allocation unit assign the work to them. Here the consideration or issue rises as follows. That is if the size of the sub task is less more no of subtasks are generated or if the size of the sub tasks are more the less no of sub tasks are generated. So we need to look for the optimal distribution policy which can balance the size of the task and the number of sun b tasks. All the processing elements are grouped together initially and one of the processor acts like master which takes care of allocation of sub tasks to other elements. If any processing element is busy it can not be assigned new work. But after the processor completes the work it can demand to the master processor for new work and it can be assigned. But here the delay in the request and allotment needs to be considered and it can be minimized by the buffering scheme. Keeping all the above in mind we need to find the dynamic scalable load balancing algorithm which can improve the performance of our system.

The difference between the end time and start time of the work load is known as the response time of a processor. The mechanism for achieving the optimal response time depends on how we distribute the work load equally among the multiple cores. There is no way to measure the processing time of a work load prior to actually executing it. So, approximate estimations are to be made for workload by the cores utilization or the cores wait queue length. The following are to be considered to get the optimal response time and good load balancing methods .They are size of the overall work load, the time needed to process the subtasks , the size of sub tasks , number of subtasks, number of processing elements or cores busy and idle, sequential execution or parallel execution.

V. PERFORMANCE METRICS

When evaluating application performance memory usage and execution time [1] [2] [6] are the main metrics to consider. Execution time is the amount of time required to process a group of instructions, usually measured in seconds. Memory usage is the amount of memory space required to process a group of instructions, usually measured in bytes. Another performance improvement seen on multicore systems is responsiveness. Multicore systems exhibit improved responsiveness due to multitasking with several cores available. **Communication Overhead**: - Memory organization in multicore computing systems affects communication overhead and program execution speed, common memory architectures are shared memory, distributed memory and hybrid shared distributed memory. Shared memory systems use one large global memory space, accessible by all processors, to provide fast communication. However, as more processors are connected to the same memory, a communication bottleneck between processors and memory occurs. Distributed memory systems use local memory space for each processor and communicate between processors via communication Network. **Interprocessor Communication**:-Physical distances between processors and the quality of interprocessor connections affect the program execution speed through communication overheads. **Code Organization**: - The degree of program parallelism has a large effect on program execution time, as does granularity, the ratio of computation to communication and load balancing.

VI. TOOLS FOR PERFORMANCE MEASUREMENTS

The VTune analyzer provides an integrated performance analysis and tuning environment that helps us to analyze our code's performance on systems with IA-32, Intel(R) 64, and IA-64 architecture [4]. VTune analyzer can plug in into Microsoft Visual Studio and Eclipse Integrated Development Environments. We can work with the VTune analyzer using the graphical interface and command line interface. All commands to create and run Activities must be preceded by vtl at the command line. The general tuning methodology begins at the system level and goes down to the micro architecture level. Regardless of your specific tuning goals, you should conduct the analysis level by level, in the following order, that is, from a high to a low level: System-level, Application-level and Micro architecture-level. There are three main strategies for improving application performance. Each strategy has an effect on processor utilization. **Balancing I/O and computation.** When processor utilization is low because processors are waiting for I/O to complete, balancing I/O and computation can speed up an application (since when balanced, I/O and computation can be performed simultaneously - the I/O time is masked by the computation time). Balancing I/O and computation is usually done during system-level and application-level tuning [4]. **Improving the threading model:** Adding multithreading to a single-threaded application, or improving the threading model of a multithreaded application, is an application-level tuning technique that can speed up your application by making more effective use of all available processor resources - this usually raises processor utilization [4]. **Improving the efficiency of computation:** Using less or more efficient computation can also speed your application. If the amount of I/O remains the same and the I/O time is not masked by computation time, then processor utilization will decrease (since a higher fraction of the total workload run time will be spent waiting for I/O) [4]. These types of changes are made during application-level and micro architecture-level tuning.

The Intel Thread Profiler collector of the Intel VTune Performance Analyzer [4][1][2] helps us to identify and locate bottlenecks that are limiting the parallel performance of our multithreaded application. Thread Profiler graphically displays the performance results of a parallel application that has been instrumented with calls to the OpenMp statistics gathering runtime engine. With the thread profiler we can, 1.compare the performance impact of using different configuration options when our program is run, such as thread scheduling methods or the number of threads used to run our application. 2. Locate areas that show large amounts of parallelization overhead, indicating inefficient parallelization.3.Compare alternative ways of parallelizing our program to see which set of directives or what locations for those directives work best. 4. Estimate the total run time that we would get if more processors were available. The time that our program spends executing is categorized by the Intel(R) Thread Profiler according to several different categories. The below table III showing the categories and their descriptions:

Table III

1) **Name**	*2)* **Description**
Sequential	The total amount of wall-clock time spent outside of parallel regions.
Seq. Overhead	Sequential Overhead is an estimate of the time the application spent in OpenMp* regions that were not executed within an OpenMp* parallel region.
Synchronized	Time spent inside critical sections and holding locks.
Lock	Time spent waiting to enter critical sections and to acquire locks.
Barrier	Time spent waiting for other threads to arrive at a barrier.
Imbalance	Time spent waiting for other threads to reach the end of a parallel region.
Par. Overhead	Parallel overhead is the estimated time spent inside of parallel regions in the OpenMp* Runtime Engine, which implements OpenMp.
Parallel	Total wall-clock time spent running code inside parallel regions. Well-tuned code spends the majority of its time in this category.
Total	Total time, equivalent to the summation from sequential time through parallel time columns.

Sequential Time is the amount of wall-clock time spent outside OpenMp* parallel regions. The goal of parallel programming is to minimize the amount of serial time spent in the execution of an application by parallelizing portions of the source code.

Use a profiler to identify parts of the code that spend the most serial execution time as the primary candidates for additional sections of parallel code. Sequential Overhead is an estimate of the time the application spent in OpenMp* regions that were not executed within an OpenMp* parallel region. These include serialized critical sections and orphaned OpenMp* constructs. If a critical section, used to protect code segments from being executed concurrently by multiple threads, will never be executed within an OpenMp* parallel region, that critical section will be adding unnecessary overhead and could be eliminated. Reducing Synchronization Time: Synchronization Time is the amount of time threads spent inside critical sections and holding locks. Large amounts of synchronization time indicate large critical sections or locked code segments. The longer one thread spends in a critical section or holding a lock, the longer other threads will be forced to wait for the release of these resources.

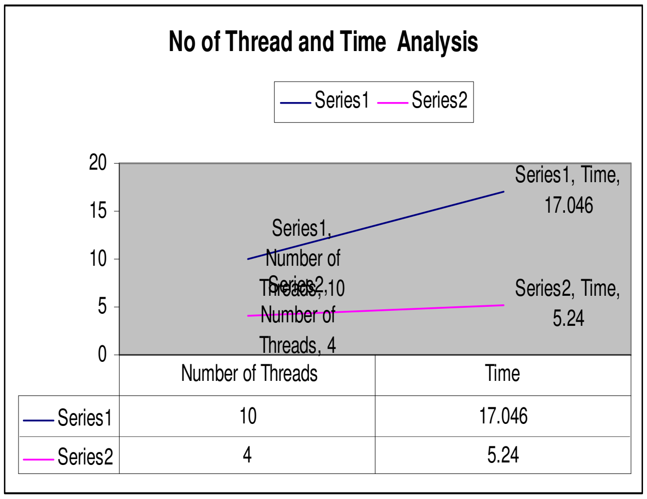
VII. EXPERIMENTAL ANALYSIS AND CONCLUSION

The parallel computing with Multicore Environment uses as many cores as possible. But here the speedup of an application is limited by the number of cores and problem size., Which is stated by Amdahl's law if p is denoted as parallel execution and (1-p) is denoted as serial execution and a system consist of "N" Cores then speed up is calculated by 1/(1-p)+p/N. Here if the value of N is increased we have limitations. The following experiment is made in the Multicore Machine with different thread numbers to perform the same task. Like to print the numbers from 1 to 50,000 different threads were created and the time is observed. The following table and the chart describe the total number of threads and the time taken to complete the task. It is observed that when the number of threads is more the time is also increasing for small task, but if the task to be performed is lengthy the larger number of threads is better.

The sample code using C with OpenMp is as follows,

```
#include<stdio.h>
#include<omp.h>
#include<conio.h>
Void main()
{ #pragma omp parallel {Printf ("hallo well done"); }}
```

The above code prints hallo well done which is equal to the number of cores in the system. The above code is modified to create as many threads as possible and print the values from 1 to 50,000 with different threads. The following is the result of it.



	Number of Threads	Time
Series1	10	17.046
Series2	4	5.24

Figure 2: Analysis of Thread Time

So, this is the time to look at the design of our applications and determine what operations are sensitive to processor now or future and identify how these places could benefit from concurrency[13] [14]. Also this is the time to learn the importance of concurrency programming and techniques and their requirements.

VIII. ACKNOLEDGMENT

We like to thank our Research advisor Dr.N.Ch.Sn.Iyengar., Senior Professor, School of Computing Science and Engineering , VIT University , Vellore-14 for his continuous Motivation towards our research. We would also like to thank Dr.J.Vaideeswaran, our Research Guide for his continuous support to conduct our research. We also extend our Gratefulness to the higher officials of VIT University for giving us the Opportunity to conduct this research.

IX. REFERENCES

[1] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishing Co. 1996.

[2] Frank Schirrmeister, "Multi-core Processors: Fundamentals, Trends, and Challenges', *Embedded Systems Conference* 2007 ESC351, Imp eras, Inc.

[3] Muti-Core Processors—The Next Evolution in Computing http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf

[4] Intel@ Multi-core technology, http://www.intel.com/multicore/

[5]http://multicore.amd.com/GLOBAL/WhitePapers/Multi-Core_Processors_WhitePaper.pdf

[6] http://www.cs.virginia.edu/stream/

[7] http://www.sun.com/processors/UltraSPARC-T1

[8]http://www.amd.com/usen/Processors/ProductInformation/0,,30_118_8825,00.html

[9] "Professional Multicore Programming: Design and Implementation for C++ Developers", chapter 3, page 36

[10]http://multicore.amd.com/en/[11] http://www.embedded.com/showArticle.jhtml?articleID=183702075

[12] Blair Guy, "An Analysis of Multicore Microprocessor capabilities and their suitability for current day application Requirements" , Bowie University , Maryland in Europe, Nov 2007.

[13] Herb Sutter, "The free lunch is over: A fundamental turn toward concurrency in software", Dr.Dobb's Journal, 30(3), March 2005

[14] Herb Sutter,"The concurrency revolution" in C /C++ users Journal, 23(2), February 2005.

AUTHORS

M.Narayana Moorthi is working as Assistant Professor., (senior) in School of Computing Sciences and Engineering VIT University Vellore. His area of Research Interest includes Parallel Computing, High Performance Computing and Advanced Computer Architecture.

P.Mohankumar is working as Assistant Professor., (senior) in School of Information Technology and Engineering VIT University, Vellore. His area of Research includes Advanced Database Management Systems and Data Mining Techniques.

Dr.J.Vaideeswaran. is working as Senior Professor in Architecture and Embedded systems division, in School of Computing Sciences and Engineering, VIT University, Vellore. His research includes High Performance Computing, and Computer Architecture.