Process Scheduling on Multi-Core Computers

D.C.K.Uduwarage
University of Colombo School of Computing

June 18, 2010

Author : D.C.K.Uduwarage
07001231
2007CS123
chethakau@gmail.com
+94716406688

Supervisor’s name : Mr. Malik Silva
Declaration

I hereby declare that this literature survey report has been prepared by Don Chethaka Kasunpriya Uduwarage based on mainly the reference material listed under the bibliography of this report. No major components (sentences/paragraphs etc.) of other publications are directly inserted into this report without being duly cited.

Name of Candidate: D.C.K.Uduwarage

Signature of Candidate: .................................................. Date: 18 June, 2010
Abstract

Multi-Core technology is used everywhere and process scheduling on multi core computers is important topic to be discussed. Traditional parallel programming techniques like message passing and shared memory threads are too difficult and inefficiency for most developers. In those methods programmers should manually manage each process and it is very difficult to write correct and scalable parallel code for important algorithms. Therefore parallel programs that can exploit more than one processor are needed for working with multi-core computers.

Current on going researches are encouraging architectural changes in CPU design such as multi-core processors with asymmetric core performance. It also called many-core architectures that integrate up to 100 cores in one package. Such architectures will exhibit a fundamentally different behavior with regard to shared resource utilization and performance of non parallelizable code compared to current CPUs. In process scheduling on multi-core computers, the responsibility of the operating system is to spare the programmer as much platform-specific knowledge as possible and optimize overall performance by making use of intelligent and configurable scheduling mechanisms.
Acknowledgements

I would like to thankful to my supervisor, Mr. Malik Silva, whose encouragement, supervision and support from the preliminary to the concluding level enabled me to develop an understanding of the subject by giving web sites links for search research papers and guiding me on writing the Literature survey.

Also I would like to thankful to Coordinator of the Literature Survey, Dr. H.L. Premarathne for providing a great support during the survey and would like to thankful to UCSC library for giving me chance to refer past surveys which are done by previous year students.

Lastly, I offer my regards and blessings to my parents, colleagues and all of those who supported me in any respect during the completion of the project.
Contents

1 Introduction .......................................................... 6
  1.1 Need of using multi-core computers .......................... 6
  1.2 Multi-core scheduling and its architecture .................. 7
    1.2.1 Hardware failures ........................................... 7
    1.2.2 Design consideration of different cores .................. 7
    1.2.3 Power-saving policies for energy efficient ................. 8

2 Technologies for process scheduling on Multi-Core Computers ...... 9
  2.1 Phoenix ............................................................ 9
  2.2 Hyper-Threading ................................................. 9
  2.3 Design Tradeoffs for Process Scheduling ....................... 10
  2.4 Performance asymmetry ......................................... 10

3 Operating systems process scheduling ............................. 11
  3.1 Linux scheduler for multi-core Processing .................... 11
    3.1.1 Hyperthreaded processor level .............................. 12
    3.1.2 physical processor level .................................. 12
    3.1.3 NUMA node level ............................................ 12
  3.2 Microsoft Windows scheduler for multi-core processing ....... 13

4 Conclusion and Future Directions .................................. 14

5 Bibliography .......................................................... 15

List of Figures

1 Multi-core Architecture ............................................. 7
2 The basic data flow for the Phoenix runtime ...................... 9
3 Hierarchy in the Linux scheduling domains ........................ 12
1 Introduction

1.1 Need of using multi-core computers

Today multi core computers are using everywhere and in the last few years, multi-core CPUs have become a standard component in nearly all sorts of computers. In multi core technology processors that have two or more working processor chips which also called as cores are working simultaneously as one system. Also the idea of multi core computers can be used to make parallel computing possible. This technology is used even in desktop and laptop PCs for consumers as well as servers and high-end workstations. Also game consoles nowadays usually use multi core technology with CPUs.

When considering the development history of this technology in 1965 Gordon Moore predicted that the numbers of transistors that can be cost-effectively built onto integrated circuits were going to double every year. In 1975, Moore corrected that assumption to a period of two years. Nowadays this period is frequently assumed to be 18 months. Increase in transistor density does not always lead to an equal in increase in computing speed but Moore’s projection has more or less been accurate up to today and users have gotten used to the constant speedup of computer hardware. In order to implement the exponential increase of integrated circuits, the transistor structures have to become gradually smaller. The extra transistors were used for the integration of more and more specialized instruction sets on CISC chips and smaller transistor sizes lead to higher clock rates of the CPUs, because due to physical factors, the gates in the transistors could perform faster state switches.

Multiple processing cores could be placed in a single processor die, since the CMOS technology met its limits for the additional increase of CPU clock frequency and the number of transistors that could be integrated on a single die that allowed for it. Significant increases in the clock speed of processors will not be seen longer. The power consumed by the fastest possible processors generates too much heat to dissipate effectively in known technologies. Instead processor manufacturers are adding multiple processors cores to each chip [1]. In 2006, Intel released the Core microprocessor, a die package with two processor cores with their own level 1 caches and shared level 2 caches. In 2006, AMD released the Athlon X2, a processor with similar architecture to the Core platform. The concept of also sharing a CPU-integrated memory-controller among the cores is additional feature of this Athlon X2 technology [2].

At the operating system level, the design of efficient multi-core scheduling mechanisms or scheduling frameworks is to be done, because various CPU components can be shared among cores. The performance of different cores in a multi-core system may be asymmetric regarding the performance of the different cores. Therefore current operating systems have to be evolved in support of the new core-architectures.
1.2 Multi-core scheduling and its architecture

Current multi-core architectures have some properties which are not support to the basic approach of the conventional scheduling. In conventional scheduling the run-queue would just have to be replaced by \( n \) run-queues, where \( n \) is the number of cores and processes would simply be scheduled to the currently shortest run-queue with some additional process-priority. In much multi-core architecture, each core manages its own level 1 cache (figure 1). Parts of the processes cache working set may become unnecessarily lost and the overall performance may slow down when task migration, which is rescheduling interrupted processes to a shorter queue which belongs to another core.

![Multi-core Architecture](image)

Figure 1: Multi-core Architecture

Also in a multi-core system, performance of each different core can be asymmetric because of the performance difference of each other cores [3]. Here are the reasons for this effect,

1.2.1 Hardware failures

Some hardware parts of the CPU may be automatically disabled because of gets damaged over time. Performance asymmetry may take place in symmetric cores over time when such components may disable and fail to work properly in certain cores independently from the other cores.

1.2.2 Design consideration of different cores

Many slow cores can be used for increasing the throughput in parallel computation while a few faster cores contribute to the efficient processing of costly tasks which can not be parallelized. Even algorithms that are parallelizable contain parts that have to be executed consecutively, which will benefit from the higher speed of the fastest core.
Hence performance-asymmetry has been shown to be a very efficient approach in multi-core architectures.

1.2.3 Power-saving policies for energy efficient

In order to save power, different cores may switch to different P- or C-power-states at different times. At different P-states, equal cores show a different clock-frequency. If an OS scheduler manages to take this into account for processes not in need of all system resources, the system can remain more energy efficient.

At the operating system level, the design of efficient multi-core scheduling mechanisms or scheduling frameworks is to be done, because various CPU components can be shared among cores. The performance of different cores in a multi-core system may be asymmetric regarding the performance of the different cores. Therefore current operating systems have to be evolved in support of the new core-architectures.
2 Technologies for process scheduling on Multi-Core Computers

2.1 Phoenix

Phoenix is an implementation of MapReduce model which allows programmers to write automatically parallelized and scheduled functional style code in a distributed system for shared memory systems. Also it is an efficient runtime system. Thread creation, dynamic task scheduling, data partitioning and fault tolerance across processor nodes are managed by the phoenix runtime automatically. It supports to programmers by efficient execution on multiple cores without concerning them with concurrency management. The efficient runtime which consists with phoenix that handles parallelization, resource management, and fault recovery. Also there is a simple API that is visible to application programmers [4].

![Phoenix runtime diagram](image)

Figure 2: The basic data flow for the Phoenix runtime

The Map Reduce programming model is encouraged by functional languages and objectives are data-intensive computations. The input data format is specific by the application and the user. The output is a set of \( <key,value> \) pairs. The user states an algorithm using two functions, Map and Reduce. The Map function is applied on the input data and produces a list of intermediate \( <key,value> \) pairs. The Reduce function is applied to all intermediate pairs with the same key. It typically performs some kind of merging operation and produces zero or more output pairs (Figure 2). Then their key value sorts the output pairs. The programmer provides the Map function in the simplest form of Map Reduce programs. Including The grouping of the intermediate pairs and all other functionality which have the same key and the final sorting, is provided by the runtime.

2.2 Hyper-Threading

Intel Corporation developed Hyper-Threading Technology to bring the simultaneous multi-threading approach to the Intel architecture. Processors resources can be used
efficiently when two threads are scheduled on the same physical core rather than context switching between the threads. Two threads can execute on the same single processor core simultaneously in parallel when using Hyper-Threading technology. The shared resources include items such as cache, registers, and execution units to execute two separate programs or two threads simultaneously. HT Technology is available on Intel Xeon processors and some Intel Pentium 4 processors [5].

2.3 Design Tradeoffs for Process Scheduling

A potential system software bottleneck is used as Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems. It is an efficient process scheduling method for multiprocessor systems with shared-memory communication mechanism. Finding the design tradeoffs between monitor bottleneck due to scheduling overhead and low process utilization due to load unbalancing is the main task of this method. Here the process scheduling overhead is considered and choosing an optimum number of scheduling monitors is the key to resolve the bottlenecks. To find the optimum number of monitors, it is using a method called Hill-climbing simulation [6].

2.4 Performance asymmetry

Building multi-core chips with asymmetric performance of the different cores can have advantages for the overall processing speed of a CPU. As an example, it can prove useful if one fast core can be used to speed up parts that can hardly be parallelized while multiple slower cores come to play when parallel code parts are executed. By keeping the cores for parallel execution slower than the cores for serial execution, die area and cost can be saved while power consumption may be reduced. Closely analyzes the impact of performance asymmetry on the average speedup of an application with increase of cores. Asymmetric multi core chips can offer maximum speedups that are much greater than symmetric multi core chips. Hence, performance asymmetry at the processor core level seems to be a promising approach for future multi-core architectures [7].
3 Operating systems process scheduling

3.1 Linux scheduler for multi-core Processing

Current kernel version for Linux operating system is version 2.6.34. The Linux scheduler for multi-core processing is based on the kernel version 2.6.23 and it has complexity $O(1)$ by basically using per-CPU run-queues and priority arrays. New kernel versions are released because failures in old version and they should support well for applications are interactive and CPU-intensive. Kernel version 2.6.23 was introduced as a completely fair scheduler, which was released on October 9, 2007 [8].

The complete fair scheduler treats all processes in a multi-core system, equally and has completely neglected the concept of different kinds of processes [9]. The data-structure of a red-black tree is used to align the tasks according to their priority to use the processor resources for a predefined interval until context-switch. The process positioned at the leftmost node in the data structure is allowed most to use the processor resources at the time it occupies that position in the tree. The position of a process in the tree is only dependent on the wait-time which is including the time the process is actually waiting for events of the process in the running queue and the process priority [10]. This is a simple concept which works with all kinds of interactive processes which process at once and normal processes, since they get a enhance by getting account for their I/O-waiting time. The total scheduling complexity is increased to $O(\log n)$ where $n$ is the number of processes in the run-queue and the process has to be inserted again into the red-black tree at every context-switch. The red-black trees of complete fair scheduler are managed per run queue which support with the Linux load-balancer [11].

The scheduling domain architecture in Linux scheduler is built based on the actual hardware resources of a computing element and scheduling domains are hierarchical sets of computation units on which scheduling is possible. Scheduling domains contain lists with scheduling groups that share common properties for each process. Computing architectures and different cache models are concerned by Linux load balancing. But performance asymmetry is not necessary here. The underlying model of the Linux load balancer is the concept of scheduling domains, which was introduced in Kernel version 2.6.7 due to the unsatisfying performance of Linux scheduling on SMP and NUMA systems in pervious kernel versions. In multi-core systems or Symmetric multiprocessing (SMP) system the cache or parts of the cache is controlled by each core separately and transferring tasks with a large working set may become difficult. Also in NUMA machines different CPU may be closer or more remote to the memory which is using by the process. All architectures have to be treated differently because of these reasons [12].

Scheduling domains in Linux scheduler are hierarchically nested and there is a top-level domain. This top-level domain containing all other domains of the physical system the Linux is running on (Figure 3). In a respective hierarchical nesting the sub-domains represent physical CPU groups, NUMA node groups, multi-core groups and SMT groups.
depending on the actual architecture. This structure is built automatically based on the actual topology of the system. Each CPU keeps a copy of every domain for reasons of efficiency which belongs to its topology.

![Hierarchy in the Linux scheduling domains](image)

**Figure 3: Hierarchy in the Linux scheduling domains**

Policy information for control, how decisions are made at that level of the hierarchy are contained in each scheduling domain. How far the loads on the component processors are allowed to get out of sync before a balancing attempt is made, how often attempts should be made to balance loads across the domain and how long a process can sit idle before it is considered to no longer have any significant cache similarity are the policy parameters which contains in scheduling domains [12]. Policies are set, based on followings,

3.1.1 **Hyperthreaded processor level**

Even when the imbalance between processors is small balancing attempts happen in every 1-2ms. Process does not want to move from one to another because of hyperthreaded processors share cache and there is no cache similarity. Domains at this level are also marked as sharing CPU power.

3.1.2 **physical processor level**

Balancing efforts for scheduler do not want to happen and processor loads must be somewhat farther out of balance before processes will be moved within the domain. Here processes lose their cache similarity after a few milliseconds.

3.1.3 **NUMA node level**

Cache similarity is not end and balancing attempts are made relatively on the odd occasion. Between NUMA nodes, the cost of moving a process is relatively high.
3.2 Microsoft Windows scheduler for multi-core processing

Multi-core process scheduling on Microsoft Windows is done by threads. These threads are scheduled to run based on the scheduling priority for each job and each thread is allocated a scheduling priority. The scheduler is priority-based with priorities ranging from 0 (lowest priority) to 31 (highest priority). When there are no other threads that need to run, Zero page thread is zeroing any free pages and only the zero-page thread can have a priority of zero.

All threads with same priority are treated as equal by the system. The system assigns time slices in a round-robin fashion to all threads with the highest priority. Lower priority threads may receive the time slice only if know thread of a given priority is ready to run at a certain time. If a higher-priority thread becomes available to run, the system stops execution of the lower-priority thread without allowing it to finish using its time slices, and assigns a full time slice to the higher-priority thread [13].

Scheduling on Symmetric multi processing systems (SMP) is also following the above steps. But windows keep the notion of a threads processor affinity and an ideal processor for a thread is not done in SMP systems. The ideal processor is the processor with for example the highest cache-locality for a certain thread. The thread may run on another processor if the ideal processor is not idle at the time of lookup.

The progress of job scheduling on windows scheduler is done by three generations. Batch processing, workload management and workload automation. Different vendors use different terms to describe their products and features. Batch processing is the scheduling of non-interactive jobs to optimize use of the resources of a single computer. Later workload management improvements distributed the job scheduling across clusters of servers, and offered calendar scheduling features. Recently, the integration of Web-based applications and the scheduling of jobs based on real-time events have defined workload automation [14].
4 Conclusion and Future Directions

The beginning of multi-core architectures has encouraged a lot of excitement in recent years. It is widely considered as the most capable direction towards increasing computer performance in the current developments of power-consumption-limited processor design. This development of computer architectures and operating system to increase performance of multi-core scheduling was discussed in this survey.

As introduction section, in the future, the processor architectures will experience wide-ranging changes in order to keep up with Moores law. Many-core CPUs and AMPs are just two proposals for innovative new architectures that may help in extending the time possibility within which Moores law can stay valid. Operating system schedulers are going to have to adapt to the changing underlying architectures for process scheduling on multi-core computers.

The scheduler domains of Linux scheduler use some essential needed flexibility for different operating systems. Computing architectures that have different behaviors with considering memory access or single-threaded performance can be quite easily integrated into the load-balancing hierarchy is the reason for that. There are arguments that in the future probably more will have to be done in the scheduling algorithms themselves. But scheduler domains at least provide the required flexibility at the level of threads. But when considering the windows schedulers, they are not scale with the number of cores or performance asymmetry at any rate. The Windows scheduler treats multi-core architecture like SMP systems and hence can not give proper care to the peculiarities of such CPUs, like the shared L2 cache. However, the Linux scheduler has the best performance when comparing with windows scheduler for multi-core scheduling.

Technologies and current researches for process scheduling on multi-core computers like Phoenix (map reduce model), Hyper-Threading Design, and tradeoffs for Process Scheduling and Performance asymmetry are discussed in this paper and many other technologies are there for developing multi-core scheduling. As described in Phoenix, the programmer provides a simple, functional expression of the algorithm and leaves parallelization and scheduling to the runtime system in the technology. Phoenix with map reduces model leads to scalable performance for both multi-core chips and conventional symmetric multiprocessors. The approach described in Performance asymmetry performs well for multi-core processors with asymmetric performance. Once asymmetric architectures expand to widespread changes the load balancing algorithm can be implemented as a modification to current operating system kernels, can be made available quickly is an advantage. Experimental evaluation of the scheduling algorithm describes good results, also with regard to equality for multi-core scheduling.
5 Bibliography

References


