Nita, I.; Rapan, A.; Lazarescu, V.; Seceleanu, T.;
8th International Conference on Communications (COMM),
Digital Object Identifier: 10.1109/ICCOMM.2010.5508993, Publication Year: 2010 , Page(s): 53 - 56

Efficient threads mapping on multicore architecture

Iulian Nita, Adrian Rapan, Vasile Lazarescu
Faculty of Electronics, Telecomm. and IT
"Polytechnics" University of Bucharest
Bucharest, Romania
Iulian.nita@upb.ro

Tiberiu Seceleanu
ABB Corporate Research and
Mälardalen University
Västerås, Sweden

***Abstract* - Considering today's hardware performance, in order to obtain best results, a proper programming strategy for optimum mapping of all processes to existing resources is necessary. The presence of multiple cores in a single chip requires applications with a higher level of parallelism. The use of suited mapping algorithms can lead to a great performance improvement considering computing time at a smaller energy consumption. We've realized a comparison between the parallel computing with an efficient mapping algorithm of threads to specific cores and parallel computing with threads mapping maintained by Linux kernel process scheduler. We observed that a good strategy regarding thread mapping on different processing units, balancing all available cores and allocating a specific amount of work, can lead to improved computational time. In our simulation we used Kubuntu Linux operating system, a system with Intel Core 2 Duo processor and another system with an Intel Quad Core.**

Keywords: multicore, affinity, mapping, threads, mpsoc, multiprocessor, linux

I. INTRODUCTION

Multicore processors were recently available on home and server markets, but they have a rich history considering the embedded computing and strict requirements considering real time processing. MPSoC (Multiprocessor SoC) have been developed as an answer to a quicker processing need in multimedia applications which are more complex every day. General use embedded systems with a single processor are not capable anymore to satisfy the increasing requirement of the complex software applications. Even if the multiprocessor architecture was developed long time ago, the approach was focused especially on multi-chip accomplishments. Considering the architecture, MPSoC combines the features of the distributed systems and also of the systems on chip (SoC). Motivated by the technology admitted by billions of transistors, MPSOC becomes more and more the preferred target of the embedded systems implementations. Despite their significant potentials, a main impediment in their design remains the development of programming paradigms and designed tools in order to reduce their complexity. Some progress was made in the architecture area with customizable multiprocessors and also in design methodology in order to reduce the designing time. The designers are still forced to manually map the application tasks on different processors and, in the same time, to customize each processor, so that the performance requirements are generally accomplished.

We observed tha,t when we developed a parallel algorithm on a multicore platform, mapping threads on available cores implied directly by the programmer is much more efficient than kernel scheduler.

The results obtained by such a method are significantly better than those obtained by classical method when the kernel governs tasks allocation. Although the kernel has mechanisms and advanced techniques for control and management of threads, in some cases, such as problems with a fine granularity in terms of parallelism, the method we suggest is more efficient because it uses a balanced load of tasks on each processor in hand. For better performance in parallel computing, we should maximize granularity and minimize synchronization and communication. [1]

One of the main advantages of the proposed algorithm is that it is independent of the hardware platform which runs: automatically identify every available core and split the application in a number of tasks proportional to the core's number, in order to have a balanced load of the operations on each core.

To support this third-party theory we chose as a starting point for simulation a simple operation of multiplying two large matrices with real number elements. There are three obvious reasons of why we chose this third-party approach:

1.Matrix multiplication is a fundamental parallel algorithm with a high degree of parallelization (calculating each element is an independent task)

2. Practical applicability is obvious, since multiplication of matrices is an operation widely used in image processing applications, in digital signal processing applications or multimedia applications.

3. Multiplying matrices is a good example of an algorithm that seems simple on the surface, but that can go quite wrong if you're not careful.

II. CHALLENGES

The election of the processing algorithm must be made considering the application and must be platform specific, knowing all the advantages and limitations in order to obtain best results. The explosive growth of digital content it may be seen in all devices used daily and also on the Internet. Due to strong computational of media algorithms, the processing is suitable to parallel processing. New challenges appeared considering compression, analysis and synthesis of media content in real time. The semiconductor industry has shifted from increasing clock speed to increasing core counts.

Multicore (dual, quad) processors are now present in many home based systems. This chance presents a massive challenge to application developers who must design a sufficient and suited parallelism onto each parallel algorithm. Mapping a set of algorithms onto a multi core platform requires using a parallel programming model, which describes and controls the communication, concurrences, and synchronization of all components involved. The correct use synchronization and locking mechanisms is complex and it has proven to be a challenging implementation.

III. PARALLEL PROGRAMMING

Like any time-sharing system, Linux achieves the effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time-frame. The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on. The set of rules used to determine when and how selecting a new process to run is called *scheduling policy*.[2]

Linux scheduling is based on the time-sharing technique: several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or quantum expires, a process switch may take place. Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that denotes how appropriate it is to be assigned to the CPU. In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of the CPU for a long time interval are boosted by dynamically increasing their priority. Correspondingly, processes running for a long time are penalized by decreasing their priority.[2]

The Linux scheduler must be slightly modified in order to support the symmetric multiprocessor (SMP) architecture. Actually, each processor runs the schedule function on its own, but processors must exchange information in order to boost system performance. When the scheduler computes the goodness of a runnable process, it should consider whether that process was previously running on the same CPU or on another one. A process that was running on the same CPU is always preferred, since the hardware cache of the CPU could still include useful data. This rule helps in reducing the number of cache misses. In order to achieve good system performance, Linux/SMP adopts an empirical rule to solve the dilemma. The adopted choice is always a compromise, and the trade-off mainly depends on the size of the hardware caches integrated into each CPU: the larger the CPU cache is, the more convenient it is to keep a process bound on that CPU.

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and its arguments which will be processed in the thread. Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction. [3]

Processor affinity is a modification of the native central queue scheduling algorithm. Each task (be it process or thread) in the queue has a tag indicating its preferred / kin processor. At allocation time, each task is allocated to its kin processor in preference to others. Processor affinity takes advantage of the fact that some remnants of a process may remain in one processor's state (in particular, in its cache) from the last time the process ran, and so scheduling it to run on the same processor the next time could result in the process running more efficiently than if it were to run on another processor. Actual scheduling algorithm implementations vary in how strongly they will adhere to processor affinity. Processor affinity can effectively reduce cache problems but it does not curb the persistent load-balancing problem. [4]

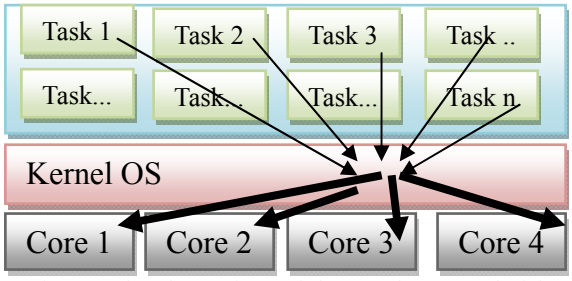

Figure 1. Threads mapping made by Kernel process scheduler

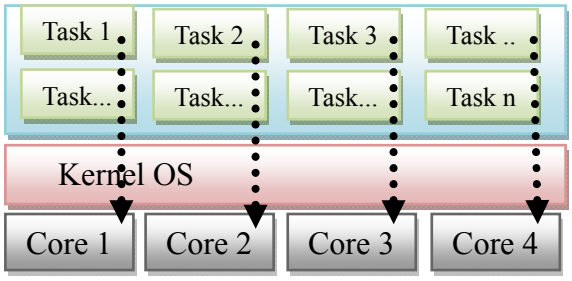

Figure 2. Manual thread mapping using CPU affinity

Processor affinity becomes more complicated in systems with non-uniform architectures. As an example, a system with two dual-core hyper-threaded CPUs presents a challenge to a scheduling algorithm. There is complete affinity between two virtual CPUs implemented on the same core via hyper-threading; partial affinity between two cores on the same physical chip (as the cores share some, but not all, cache), and no affinity between separate physical chips. Processor affinity alone cannot be used as the basis for dispatching processes to specific CPUs, however, as other resources are also shared. For instance, if a process has recently run on one virtual hyper-threaded CPU in a given core, and that virtual CPU is currently busy but its partner is not, cache affinity would suggest that the process should be dispatched to the idle partner. However, since the two virtual CPUs compete for essentially all computing, cache, and memory resources, it would typically be more efficient to dispatch the process to a different core or CPU if one is available; while this would likely incur a penalty in that the process would have to repopulate the cache, overall performance would likely be higher as the process would not have to compete for resources such as functional units within the CPU. [4]

Improving memory effectiveness is an important technique to achieve high program performance. While there exist tools and runtime systems to schedule threads efficiently, little is known about what would be an optimal affinity thread schedule to maximize the memory effectiveness and why it is optimal.[5]

Recent studies advocate explicit thread affinity management using the sched setaffinity system call and better system load balancing mechanisms.[6][7][8][9]

An affinity mask is a bit mask indicating what processor(s) a thread or process should be run on by the scheduler of an operating system. Setting the affinity mask for certain processes running under Linux kernel can be useful. This might lead to better application performance.

The Linux kernel contains a mechanism that allows developers to programmatically enforce hard CPU affinity. This means your applications can explicitly specify which processor (or set of processors) a given process may run on. In the Linux kernel, all processes have a data structure associated with them called the task_struct. This structure is important for a number of reasons, most pertinent being the cpus_allowed bitmask. This bitmask consists of a series of n bits, one for each of n logical processors in the system. A system with four physical CPUs would have four bits. If those CPUs were hyperthread-enabled, they would have an eight-bit bitmask. If a given bit is set for a given process, that process may run on the associated CPU. [10]

The Linux kernel API includes some methods to allow users to alter the bitmask or view the current bitmask:

- sched_set_affinity() (for altering the bitmask),
- sched_get_affinity() (for viewing the current bitmask).[10]

IV. SIMULATION

Parallel programming models are provided as extensions of existing languages, added to C/C++, rather than an entirely new parallel programming language. Some models have the property that concurrent entities are separated by different memory address spaces. Such models as MPI (Message Passing Interface) and UPC (Unified Parallel C) are useful for large scale distributed systems with separate address spaces. On general interest hardware these are usually not found. On smaller systems where are the cores that have access to a common address space, programmers develop applications using the threading support of the underlying operating systems. These include POSIX threads and the windows thread interface. All threads can access each other's data, an for that threading libraries provide complex synchronization and

locking mechanisms.

In order to develop applications on multiprocessor systems on chip, three factors must be taken into consideration: application parallelism degree , available hardware resources, and real time constraints. For a program to run on a parallel processors system it must be split in a series of tasks which can depend one to another or can be interdependent. The split implies partitioning and mapping. Mapping is the task distribution to each processor and can be static or dynamic.

We have used C language and POSIX threads for paralleling of multiplication of 2 big size matrix with random generated real elements. The program was developed to automatically detect the number of cores and to compare 2 parallel computation times : the time with manual dynamic thread mapping on each core by setting the affinity versus the time when Linux scheduler maps all started threads.

The number of threads for the multiplication is variable to determine in tests the good balancing between computation and started tasks. Each thread deals with the same computational charge. Processors used in simulations were Intel Core 2 Duo T5879 2 GHz and Intel Quad Core Q6600 2.4GHz.

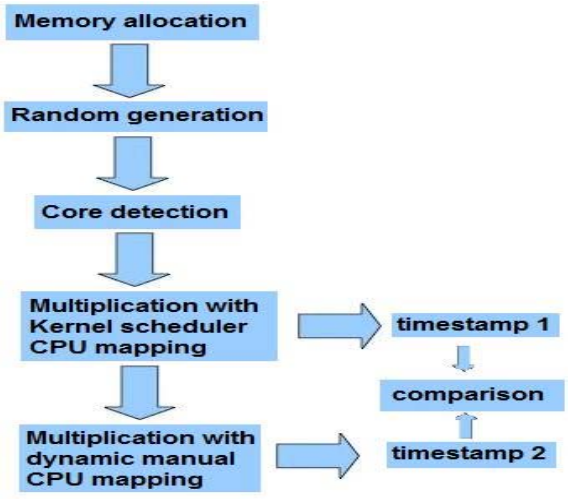

Figure 3. Program block scheme

TABLE I. SIMULATION RESULTS

Cores	Mat. Dim	No. threads	Manual mapping time (s)	Kernel mapping time (s)	Manual mapping gain
2	500	100	0.92	1.08	17%
2	500	250	0.91	1.11	22%
2	500	500	0.87	1.12	29%
2	1000	500	9.88	10.63	8%
2	1000	1000	9.68	11.9	23%
4	500	100	0.51	0.75	47%
4	500	250	0.6	1.79	198%
4	500	500	0.75	1.85	147%
4	1000	500	3.86	5.21	35%
4	1000	1000	4.45	8.92	100%

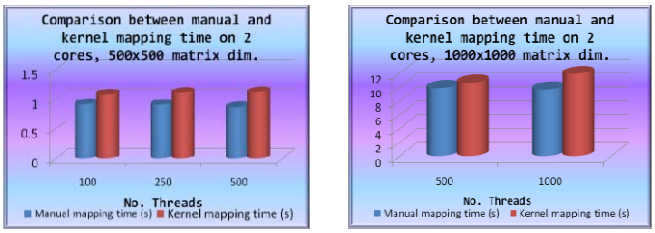

Figure 4. Time comparison of 2 cores matrix multiplication

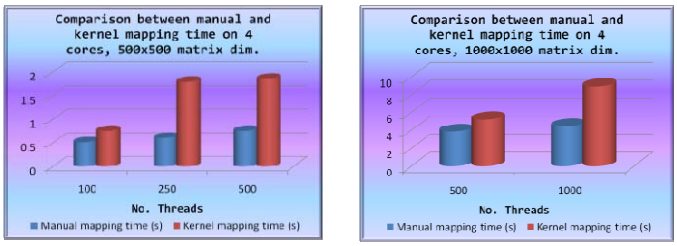

Figure 5. Time comparison of 4 cores matrix multiplication

V. CONCLUSIONS

Our simulations reported a gain up to 200% of parallel computing time with dynamic allocation of processing units over computing time with kernel scheduler mapping. Also it is highly necessary to balance the work load on each core and not have a massive queue of pending threads. For some bigger payload (matrix dimensions extremely large, ex. 8000 x 8000) the cache size (quad core: 4 MB/Core, dual core: 2 MB/Core) is exceeded and both algorithms have the same performance.

Besides that, in simulations, even if it is not the most efficient manual mapping, it is superior to kernel scheduling. Even if the hardware offers great solutions for parallelism, as the presence of a large number of different processing cores on a single chip, this fact must be handled properly: partitioning, communication, synchronization between threads are key features for an efficient algorithm with a higher degree of parallelism. Using some intelligent algorithms for task mapping may determine, in some situations, improved performances and in the same time a lower energy consumption.

REFERENCES

[1] F. Song, S. Moore, and J. Dongarra, “Analytical modeling for affinitybased thread scheduling on multicore platforms,” University of Tennessee, Computer Science Tech. Rep. UT-CS-08-626, 2008.

[2] Daniel P. Bovet, Marco Cesati. ” Understanding the Linux Kernel”, 2nd Edition, O'Reilly, 2002, pp 378-379.

[3] Greg Ippolito.”YoLinux Tutorial: POSIX thread (pthread) libraries”. Internet:www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html , Jan 2007 [Jan. 10, 2010].

[4] TMurgent Technologies, White Paper: ”Processor Affinity”. Internet: www.tmurgent.com/WhitePapers\ProcessorAffinity.pdf Nov. 3, 2003 [Jan. 5 2010].

[5] Fengguang Song, Shirley Moore, Jack Dongarra – ”Analytical Modeling and Optimization for Affinity Based Thread Scheduling on Multicore Systems”, EECS Department, University of Tennessee – USA, IEEE CLUSTER, New Orleans, Louisiana, Sep. 3, 2009.

[6] Costin Iancu, Steven Hofmeyr, Filip Blagojevi´c, Yili Zheng - Oversubscription on Multicore Processors, Lawrence Berkeley National Laboratory.

[7] S. Hofmeyr, C. Iancu, and F. Blagojevic. Load Balancing on Speed. To appear in Proceedings of Principles and Practice of Parallel Programming (PPoPP’10), 2010.

[8] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In SC ’07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007.

[9] A. Mandal, A. Porterfield, R. J. Fowler, and M. Y. Lim. Performance Consistency on Multi-Socket AMD Opteron Systems. Technical Report TR-08-07, RENCI, 2008.

[10] Eli Dow. “Take charge of processor affinity”. Internet http://www.ibm.com/developerworks/linux/library/l-affinity.html, Sep. 29, 2005 [Jan. 7, 2010].

[11] Felicia Ionescu, "Principiile Calculului Paralel", Editura Tehnica, Bucuresti, 1999, ISBN: 973-31-1331-X

[12] Michela Becchi, Patrick Crowley, “Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures”, *Conference Journal of Instruction-Level Parallelism 10* (2008) 1-26, Submitted 10/06; published 6/08, 2008.

[13] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, “A Survey of Multicore Processors”, *IEEE Signal Processing Magazine* (Volume 26 Number 6 November 2009 : Multicore Platforms in the signal processing world, Part 1)

[14] Wayne Wolf, “Multiprocessor System-On-Chip Technology”, *IEEE Signal Processing Magazine* (Volume 26 Number 6 November 2009 : Multicore Platforms in the signal processing world, Part 1)

[15] Philip Mucci – ”Linux Multicore Perforrmance Analysis and Optimization in a Nutshell”, *NOTUR 2009*,Trondheim, Norway