Revista de Sistemas de Informação da FSMA
n. 9 (2012) pp. 45-53

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

A Comparative Study of Multithreaded Applications Performance in Different Scenarios

Alex G. C. de Sá, *MSc student in Computer Science, UFMG*,
Marluce R. Pereira, *Professor at the Computer Science Department, UFLA*,
Pedro M. Moura, *MSc student in Computer Science, UNICAMP* and
Luiz Henrique R. Peixoto, *MSc student in Computer Science and Technology, UNIFEI*

***Abstract*—This paper presents a comparative study of the performance of multithreaded applications in different scenarios. These applications use the concept of threads to solve tasks. To make each scenario being evaluated were considered four factors: threads programming platform (or library), operating system, operating system architecture and the priority of running processes. We analyzed the performance obtained in mathematical applications in a multicore processor. The runtime operations and the number of parameters were used in comparative performance. Results showed that the factors referred to are actually relevant to the parallelism inherent to multicore architectures.**

***Index Terms*—Performance, Parallelism, Comparison, Multicore, Multithreaded Applications.**

I. Introdução

PARALLEL computing arose with the main goal of diminishing the time to solve high computational cost problems whose resolution was not feasible time-wise, either because of its time or space complexity. Meteorology, Mathematics, Bioinformatics, Chemistry and Physic are some of the examples of the current challenges that demand a great effort in computational processing. These areas use algorithmic models to treat complex problems that need a huge amount of calculations. In this case, the time to obtain a solution for the problem at hand may be one of the the key issues in research.

Even though scientific problems require more processing power, there is a growing difficulty in keeping up with Moore's Law [9], that forecasted that processors would have their capacity doubled every 18 months. The reason for this difficulty is the inherent physical limitations in the reduction of transistors size inside processors. These limitations cause the impossibility for the industry to increase the processor's potential, resulting in a stabilization to processors speed.

In order to avoid the limitation in chips fabrication, the alternative was simply to multiply the number of processing cores in the processor. Nevertheless, with those new parallel architectures, there is the growing need to use parallel programming concepts in order to use the most of these platforms processing capacity.

Corresponding Author: Alex G. C. de Sá alexgcsa@dcc.ufmg.br

There are basically two different ways to get better performance implementing concurrent applications. The first approach is through the creation of processes, with all the communication made through messages, which are responsible for keeping all the necessary information for the programs, including register content and memory space [4]. The second approach uses thread, which are also known as light processes [12]. These threads exchange information only through shared memory and can be up to 20 times faster in their creation time when compared to processes [1].

This paper presents a study and an analysis of how to get better performance in parallel applications based on the second approach, that is, using threads. Our main motivation was to know which are the factors influencing the gain or loss of performance in applications with multiple threads (multithreading). We studied four different factors: operating systems (Windows *versus* Linux), differences between operating system architectures (32 bits *versus* 64 bits), usage of distinct thread manipulation libraries (PThreads *versus* WinAPI) and change on the process scheduling priority (common priority *versus* maximum priority). This way, we can understand which is the best way to model high computational cost problems and which of the applicable internal and external factors influences the most the performance.

These factors may improve or impoverish a concurrent application performance, but this depends on the context of the application. Hence, this work is clearly justified by the analysis we perform on different application types under this perspective.

One point that must be stressed is that the WinAPI, in its thread manipulation functionalities, offers similar operations to those of POSIX-Threads. Nevertheless, the implementations cannot be compared, since WinAPI source code in proprietary. The lack of ways to compare the source code justifies (and motivated) an experimental analysis of the applications that used the concurrent programming paradigm using for programmatic application interfaces.

This article is structured as follows. Section I introduced this paper. In order to understand how the concurrent applications were modelled, Section II explores the relevant characteristics of both PThreads and WinAPI platforms. Section III reports on works related to this paper and Section IV presents the methodology we used, that is, the problems used for the performance analysis of both libraries and the main project decisions. Section V presents the results with a brief discussion for every situation and, finally, Section VI closes this article with the conclusions we came to and the possible challenges for a future work.

II. Thread based programs

THIS section introduces the library PThreads and the WinAPI programming platform, so that the reader can understand how we modelled and developed the applications used in this paper.

A. PThreads

The main idea behind using threads consists in dividing a single program into several different parallel tasks, so that the heavy work gets done quicker [8]. Nowadays, the POSIX standard for threads in Linux is commonly used based on the standard library PThreads (POSIX-Threads). This library offers an interface for the creation and manipulation of threads, that execute on a program [1]. PThreads is standardized for the C programming language.

When the PThreads library was created, its developers sought to keep the principles outlined by the UNIX system kernel creators, imbibing them into the library. One of those principles, for instance, states that the context switch between related processes must be fast enough to deal with each segment at the user level in a kernel thread. Kernel processes may have different relationship levels, but the PThreads specification demands sharing of almost all resources [5].

Besides supplying an interface, the PThread standard specifies several services to support multithreaded applications, such as support to specific functions, error detection and managing functionalities [10]. Nevertheless, there are many services that are optional in the library implementation, which can make on implication quite different from the other. In Figure 1, we present the software layers for a PThread using application, based on a diagram in which the communication process between application, PThreads library and operating system are defined.

According to Figure 1, only C based programs can use directly the PThreads library. Any other programming language requires an interface for PThread in order to pass the parameters correctly, execute type conversions and other adjustments that depend on the language or the compiler. PThreads contains a set of functions whose interface and functionality are defined in the POSIX-Threads standard. The PThreads code executes partially in user mode and inside the critical sections, operate in kernel mode. This is a guarantee of mutual exclusion between threads. The implementation uses several UNIX standard libraries and also some kernel calls.

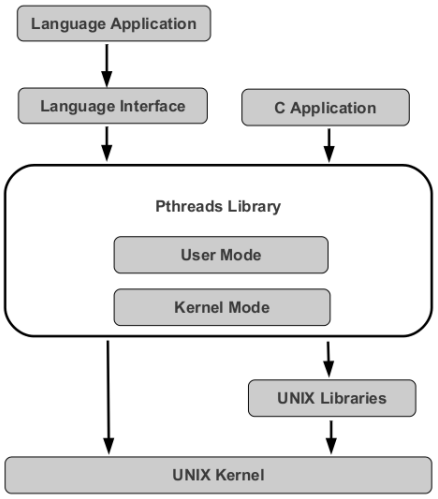

Fig. 1. PThread software layers for an application [10].

It is important to point out that the PThreads library defines functions, types and constants in the C programming language. In this paper, we used the header file *pthread.h*.

B. WinAPI

The Windows Application Programming Interface (WinAPI) is a programming platform that is supplied with all versions of the Windows operating systems. Its implementation is proprietary, with all rights reserved to Microsoft.

According with *Bodnar* [2], the WinAPI allows through its source code interface to create several kinds of applications. This platform was created specially for the C and C++ programming languages. Besides, it is the most direct way to create applications in Windows.

Figure 2 presents the four basic components of the WinAPI.

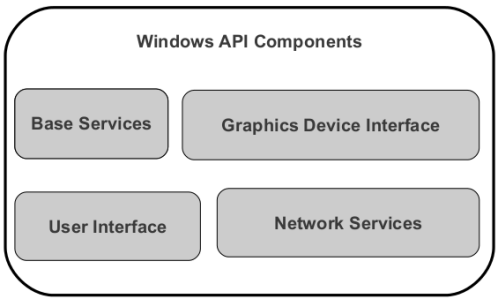

Fig. 2. The four basic components of the WinAPI [2].

In Figure 2, the Base Services supply access to the Windows basic resources, which include file system manipulators, devices, processes, threads, records and errors.The Graphics Device Interface (GDI) is an interface to work with graphics. It is used to interact with graphic devices

such as printer, monitor or a file. The User Interface offers functionalities to create windows and controls requested by an application. The Network Services offer access to network resources of the Windows operating system.

Complementing the last reference, *Spinellis* [14] reports that the WinAPI functionalities can be divided into eight categories:

- Managing and administration
- Diagnostics
- Graphics and multimedia
- Networks
- Security
- System services
- User interface

The thread and processes functionality used in this paper fit into the category System services.

III. Related Work

THE work of *Silva and Yokoyama* [13] has the goal of comparing the performance of threads libraries. The libraries compared were the GNU Pth [7], the Protothreads [6] and the PM2. Marcel [16]. During the comparison performed in this work, basic management, synchronization, initialization and union functions for thread functioning were evaluated. Operations that demand a lot or a little CPU and a lot of a little input and output (CPU Bound/IO Bound) also were evaluated. Nevertheless, the performance of the manipulation libraries was only measured based on run time averages, varying the number of operations and the amount of threads in the problem. This way, it is not analyzed in this reference the real parallelism achieved by multicore architectures. This could be done based on the usage of program performance metrics such as *speedup*. This is the main difference from the work by *Silva and Yokoyama* and the work described in this paper.

Torelli and Bruno [17] propose an approach similar to this work. It described a comparison between two different parallel programming tools, OpenMP and PThreads. The tests were performed in the symmetric multiprocessor (SMP) that has four processors and shared memory. An Euclidian distance transformation algorithm, that is used in image processing, was used to compare the results. The main idea in this related work was to evaluate and compare the type of explicit type of thread based programming (PThreads) with the directive based programming model (OpenMP). The results indicated that the PThreads tool has a smaller execution time. Besides, it was seen that the larger the image, he better the performance achieved by the tool.

The difference from this work to the one from *Torelli and Bruno* is the fact that in the latter they did not analyze and compare different applications or scenarios (operating systems and architectures). They only varies the size of the image to which the Euclidian distance transformation algorithm was applied. There are several types of applications and other scenarios in which different results could be found. In the present work, these points are widely discussed in the Section V.

Penha et al. [11] present a performance evaluation of different paradigms and programming languages using multithreading. They also performed their study comparing digital image processing algorithms. In this reference they compared both procedural and object oriented paradigms, as well as the C++ and Java programming languages. In this paper they observed that the procedural paradigm achieved better results than the object oriented one, considering only the C++ programming language. Another important point to stress was that the Java language outperformed the object oriented C++.

The difference from our work to this reference is that we did not compare different paradigms and programming languages. We only used the C programming language, in the procedural paradigm. In spite of its many interesting notes, *Penha et al.* did not fully explore the concepts on multicore architecture and did not contemplate many relevant factors, such as the usage and the study of different multithreading programming interfaces for the C language (or any other chosen language); different performance of parallel application in different operating systems; the study of the impact of a change in the operating system architecture and the difference in execution time and amount of performed operations when we change the process scheduling priority, which is orchestrated by a multithreading application.

Therefore, this work is relevant because it observes in minutiae the inherent real parallelism applications that execute on multicore processor and also because we analyze several different factors that have an influence on the performance of those applications.

IV. Methodology

IN order to analyze the performance of a set of thread based concurrent applications, four factors were evaluated. There factors determine scenarios that influence the performance of an application.

The first factor is the supporting operating system, considering the applications implemented both in the Linux and in the Windows environments.

For both operating system we used two distinct architectures, which consisted in the second factor. Tests on the applications were performed both in the 32 bits and in the 64 bits architectures.

Considering that the applications were compiled and executed in two different operating systems, a third factor became relevant. Two differnt platforms from concurrent applications programming were used - the WinAPI for the Windows environment and the PThreads for Unix-like systems.

Finally, the fourth factor is different from the others because it uses different priorities in the processes execution. A thread based process will be scheduled with common priority (configure by the operating system) and maximum priority (configured before execution).

Table I presents the configurations described, specifying what will be each scenario, that is, what factors are evaluated in each of the scenarios.

TABELA I
SCENARIOS EVALUATED AND THEIR FACTORS

Scenario	O.S.	Architecture	Platform	Priority
1	Ubuntu 11.04	32 bits	PThreads	Common
2	Ubuntu 11.04	32 bits	PThreads	Maximum
3	Ubuntu 11.04	64 bits	PThreads	Common
4	Windows 7	32 bits	WinAPI	Common
5	Windows 7	64 bits	WinAPI	Common
6	Windows XP	32 bits	WinAPI	Common

Due to the incompatibility of development standards between the Gnome C Compiler and the Windows 64 bits architecture, we performed tests only in its 32 bits version, in spite of the operating system 64 bits architecture

Another issue that should be pointed out is that in order to evaluate the applications both in the Windows and Linux operating systems we needed to adapt the implementations, specially in the choice of thread manipulation platform and in the change of operating system architecture.

Next we describe the set of applications we used. Four different problems were used to estimate the performance in the presented scenarios: calculation of millions of integer operations per second (MIPS), calculation of millions of floating point operations per second (MFLOPS), calculation of π using Leibniz's method and the calculation of a defined integral using the trapezes rule. In order to even the situations in the tests performed, all quantitative variables for calculations and iterations were the same from one scenario to another. The four problems used to develop the performance comparison are described in more detail in the next subsections.

A. MIPS

The Millions of Integer Operations Per Second (MIPS) algorithm seeks to estimate the number of integer operations a machine can calculate in a certain period of time. The problem uses all types of operations (sum, subtraction, division and multiplication) performing the same load on all operation types. In the simply multithreaded method, each created thread performs a calculation in parallel and in the end the result is integrated. The number of operations in our tests amounted to ten billion (1E+10).

B. MFLOPS

The Millions of Floating Point operations Per Second (MFLOPS) is similar to MIPS, with the difference that the operations are performed on fractions stored in floating point format, that have larger computational cost. The number of operations in our tests amounted to one hundred million (1E+8).

C. Leibniz Method

Leibniz method uses a Taylor expansion, whose sum tends to the value of $\frac{\pi}{4}$. This series is shown in Equation 1. The Leibniz method was parallelized dividing ranges of terms between the threads, and having each thread calculate the value of its range and then adding up the final results when all threads are synchronized. In the end, the final result is multiplied by four. The calculations amount to a billion (1E+9) series terms.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4} \qquad (1)$$

D. Integral using the trapeze rule

The trapeze rule is a method used to find an approximation of the result of a defined integral. The integration area is the region under the curve defined by a curve $f(x)$ and bounded by points such as a and b. The method consist in dividing the integration are into N trapezes with height h, dividing the whole area and the final value of the integral is the sum of the area of all trapezes under the curve. (Equation 2).

$$\int_b^a f(x)dx = h * \sum_{i=0}^{N-1} \left(\frac{f(x_i) + f(x_{i+1})}{2} \right) \qquad (2)$$

For the tests, we used one hundred million (1E+8) trapezes in each execution. The function used to calculate the integral was $f(x) = x^3 + x^2 + 2x + 1$ and the limiting points where $a = 1$ and $b = 2$.

E. Other project decisions

The source code for the described problems were implemented using the C programming language and compiled with GCC 4.5 in both operating systems. The operating systems used were Ubuntu 11.04, Windows XP Service Pack 3 and Windows 7 Service Pack 1. At Linux we used the header file *pthread.h* for the PThreads library and at Windows, the header file *windows.h* was used to invoke the threads with WinAPI platform. The computer used was an AMD Phenom™ II X4 B95 with 2,99 GHz processor clock and 3,00 GB RAM.

For the developed code to control correctly the number of operations/interactions specified in the previous subsections, the amount of threads and the process priority (as maximum or normal), we created scripts that manipulated the executable file created by the compiler, adjusting each of their parameters. Repetition loops also were coded in order to execute the tests several times.

In order to schedule a process with maximum priority in Linux, we used the following command:

nice -n -20 EXECUTABLE <PARAMETERS>.

The command *nice* sets the priority, determining that the executable creates a process with the highest or lowest priority from the beginning of the process. In this case, we used a priority equals to -20, the highest one available in a UNIX-Like system, where the priority remains inside the interval -20 to 19.

In the Windows environment, we used the following command to determine the maximum priority to the process:

start /high EXECUTABLE <PARAMETERS>.

The command *start* can initiate a process with a specific priority, which will remain constant until the end of the process. In the case of the applications we developed, the parameter *high* was used to determine that the process priority in the highest. Windows is limited in its priority classes, allowing only the following types: *low*, *bellownormal* , *normal* (normal), *abovenormal*, *high* and *realtime*.

V. Results and Discussions

EACH problem was executed 33 times with one, two, three and four simultaneous threads and the average execution time was calculated, as shown in figures 3 to 6. The number of executions was chosen in order to find statistically significative results. The highest number of threads was chosen because it was equal to the number of cores in the processor used in the tests. This real, the real parallelism inherent to this architeture can achieve the best results.

For the graphics in Figures 3, 4, 5 and 6, the legends have the definitions from Table I. Each scenario in that table has a different factor that is under evaluation in this work.

Tables II, III, IV and V also use the same legends it their lines. Their columns expose the speedup for each number of threads, that is, the performance gain using two, three and four threads when compared to the sequential execution with a single thread.

For all the graphics the results are based on averages. For this reason, we consider in the graphics the 95% confidence interval. This was done in order to determine the precision of the performed measurements, that is 95% of the times those programs were executed, the results were the same, considering the margins in the confidence intervals. This can be illustrated in each bar of the graphics by a solid black bar that delimits the maximum and minimum in that interval.

The results showed that the executions had a small standard deviation, given the small margins in the confidence intervals. There was a high standard deviation and in the confidence intervals in the execution of MFLOPS in all execution scenarios, with the highest values in the three and four threads scenario, which indicates a problem with thread scheduling in the process (user level threads) as the number of executable lines grows. This can be explained by the amount and type of operations (high precision) in this problem. In the graphic in Figure 4 this increase can be seen in details.

In Figure 3, in all execution scenarios the results showed that increasing the number of threads, so would the number MIPS. The **Scenario 6** showed close to 5,32% more MIPS than the other scenarios.

In terms of speedup, when increasing the number of threads, there was also an increase in performance in the same proportion in all evaluated scenarios, which can be seen in Table II. In this case, no specific scenario performs best.

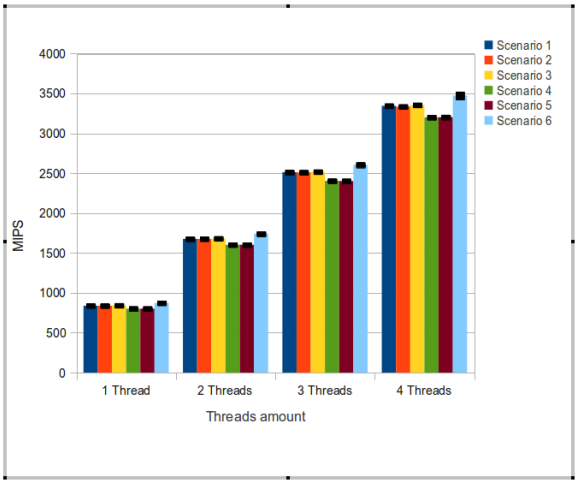

Fig. 3. Results for MIPS.

TABELA II
Speedup for MIPS

MIPS	2 threads	3 threads	4 threads
Scenario 1	1,99836	2,99522	3,98812
Scenario 2	1,99999	2,99439	3,97763
Scenario 3	1,99400	2,98604	3,97917
Scenario 4	1,99767	2,99662	3,98881
Scenario 5	1,99826	2,99513	3,99230
Scenario 6	1,99440	2,98423	3,98073

For the results in Figure 4, we can see that increasing the number of threads, the number of MFLOPS executed also increased in all contexts. Nevertheless, the **Scenario 3** increased this number 7,25% more than the other scenarios.

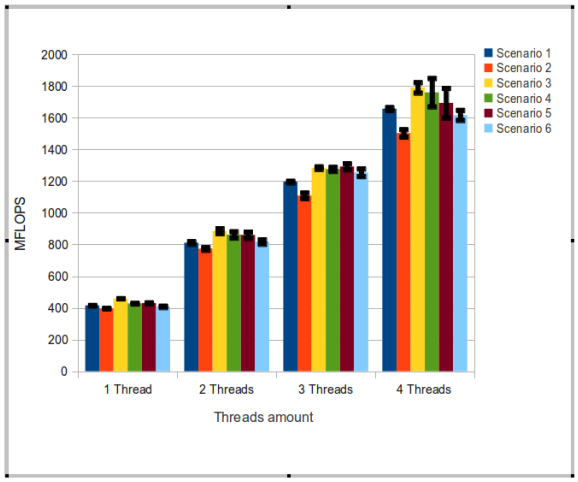

Fig. 4. Results for MFLOPS.

Another point seen in Figure 4 is that in spite of the fact that **Scenario 5** has the second largest number of MFLOPS executed, the speedup (Table III) that it achieved was highest than the one achieved by **Scenario 3**. Since both systems use 64 bits architecture, both benefit from the execution of this application, for several variables in this problem perform operations with floating points, that perform better in this architecture.

In the MFLOPS application we can also see that exceptions occurred that influenced the performance, given that the speedup achieved was not stable. The more stable speedup was found in **Scenario 6**. In **Scenario 4** for four threads and in **Scenario 6** for three threads we could see superlinear speedups. The Table III presents the speedup for the MFLOPS application.

TABELA III
SPEEDUP FOR MFLOPS

MFLOPS	2 threads	3 threads	4 threads
Scenario 1	1,95273	2,87901	3,98785
Scenario 2	1,94719	2,79152	3,78374
Scenario 3	1,92784	2,79359	3,89541
Scenario 4	2,01294	2,97793	4,10829
Scenario 5	2,00000	3,00323	3,93656
Scenario 6	2,00283	3,07327	3,96040

In the π calculation problem using Leibniz method, there was a performance gain for two, three and four threads. The confidence interval was also small, showing a certain stability in the executions. This can be seen in Table IV, where the speedup found was always close to the capacity limit of the four core processor.

TABELA IV
SPEEDUP FOR LEIBNIZ

Leibniz	2 threads	3 threads	4 threads
Scenario 1	1,98469	2,97873	3,95099
Scenario 2	1,98974	2,98022	3,97332
Scenario 3	1,98717	2,97128	3,95378
Scenario 4	1,99813	2,99383	3,98998
Scenario 5	1,99771	2,99461	3,98846
Scenario 6	2,00055	3,00082	3,99713

In Figure 5 we can understand that given that the operations in the Leibniz method are similar to the ones in MFLOPS (floating point operations), we can justify the smaller execution time for **Scenario 3**.

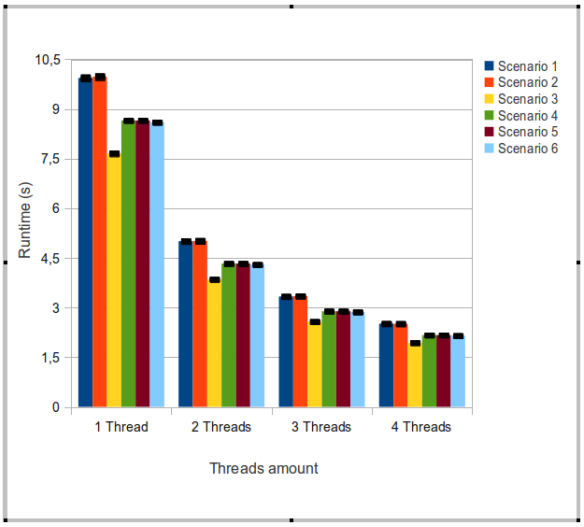

Fig. 5. Results for Leibniz.

In the problem of finding the integral solution using the trapeze method, the behavior of all solutions was almost the same in all the evaluated scenarios. As can be seen in the graphic in Figure 6, few behavior changes occurred when the executions were performed.

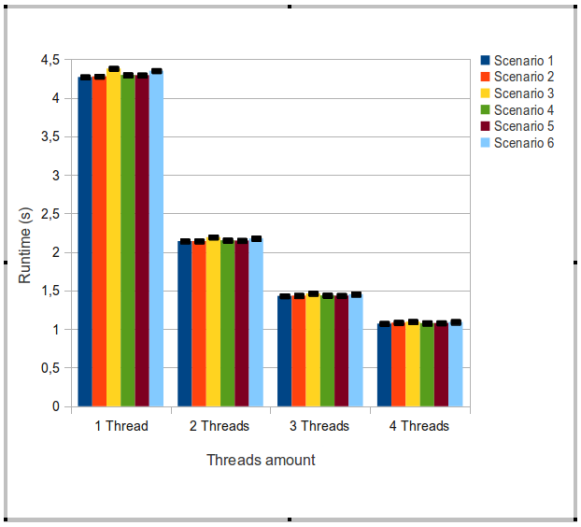

Fig. 6. Results for Integral.

Considering the speedup results shown in Table V, there were performance gain with two, three and four threads. We can see from that table that the results were also close to the four core processor capacity. The exceptions in performance achieve can be related to the operating system scheduling method.

TABELA V
SPEEDUP FOR INTEGRAL

Integral	2 threads	3 threads	4 threads
Scenario 1	1,99378	2,98423	3,98439
Scenario 2	1,99809	2,98152	3,94500
Scenario 3	1,99736	2,99384	3,99304
Scenario 4	1,99674	2,98969	3,98805
Scenario 5	1,99824	2,99309	3,97724
Scenario 6	1,99913	2,99543	3,98352

Analyzing the results for **Scenario 1** and for **Scenario 2** we can see that in almost all executions of the four problems, the performance from **Scenario 2** was inferior. The difference between both scenarios was that the second uses a priority coded by the operating system. We can find a possible explanation in the work of *Carissimi et al.* [3], where it is reported that in several case forcing a maximum priority for the application is not ideal. Since this priority is configured by the user, the same can compromise the performance, given that it does not consider the current scheduling queue and other important factors in the operating system that exploit the characteristics of the hardware architecture.

We can also consider that the type of thread used in this work was the user thread type that is controlled by the process [15], what reinforces this explanation.

Therefore, if an application creates four threads (as we did in this work) in a process scope and the user configures

it to have maximum priority throughout its execution, it is possible that when one of the processor cores free, an idle thread is immediately scheduled to execute its tasks in that core, without any respect for the system process scheduling needs and also for the thread scheduling queue controlled by the process. This way, if the aforementioned thread was performing in another processor core or if it was necessary for another process to execute automatically in the current core, the context switch for core exchange can increase the latency in the execution of the process as a whole, because this new condition can affect the standard configuration of the operating system (virtual memory descriptors, scheduler queues and other data structures), compromising its performance.

If the user leaves the scheduling to the operating system, it can use its supervision and event forecasting capabilities to expect the end of a task in there processor where the waiting process or thread wants to execute and this wait may be smaller than the context switch time for the processing core. Hence the global execution time may decrease, increasing the performance.

To validate this result and its explanation, new tests were performed in all scenarios, with the exception of **Scenario 2**, which already presents an excellent result with maximum execution priority. Table VI presents how the results changed for each scenario, considering their execution with maximum and common priorities.

TABELA VI
Results with maximum priority compared to those achieve with common priority

Scenario	MIPS	MFLOPS	Leibniz	Integral
1	<0,08971%	<6,87317%	<0,15905%	<0,42007%
3	<0,27061%	<3,87829%	<0,16772%	<0,81466%
4	<0,19856%	<2,73659%	>2,25321%	>0,10840%
5	<0,23265%	<2,22516%	>2,22593%	>0,04917%
6	<0,37546%	<0,09574%	>0,18584%	>0,06284%

The results presented in the Table VI show that the use of priority can increase the performance of parallel applications, but this depends on the operating system used and on the application itself. Only the Leibniz method and the Integral application were able to achieve a better performance with the setting of maximum priority than allowing the operating system to regulate this factor. For those two applications, the superior results were found in Windows. In all other cases, the setting of maximum priority did not help the performance of the application, as explained in the previous paragraphs.

VI. Conclusions

In this work we perform a study of the performance of parallel applications in different scenarios. Each scenario that was evaluate considered a set of factor that can influence the performance of those applications. These factors are related to the operating system, the different operating system architecture, the distinction between platforms of parallel programming libraries and the configuration of process execution priorities. These mathematical applications use intensely threads to help perform their calculations.

The results showed that the factors studies are relevant and help the applications achieve an adequate performance in a multicore architecture. This can be seen in the graphics of execution times and number of operations and also in the speedup tables, that show results adequate to parallel applications, which showed the expected results in most cases.

Besides, we saw that the application type also favours some scenarios, that is, the data structures used by some applications may favour or hinder its performance. For instance, applications that manipulate huge amount of floating point data with double precision are favoured by 64 bits architectures.

As to the process execution priority, we can state that in most cases it hindered the performance application, with some exceptions that depended solely on the type of application and of the characteristics of the operating system.

As future work, we intend to study other factors that may influence the performance of multithreaded applications. As seen in related works, the amount of iterations or operations may be considered a factor, for it affects the speedup achieved. Another issue that may help clear some question is the different processor architectures. Therefore, the use of a certain type of operating system architecture may be beneficial for the performance of applications treating different multicore processor architectures.

Besides, we also intend to study this issue using known benchmark applications in order to determine more precisely the exact influence of these factors in multithreaded applications.

Acknowledgmentss

The authors would like to thank The Computer Science Department at the Federal University of Lavras (UFLA) for the technical support to this work.

Referências

[1] BARNEY, B. “POSIX Threads Programing”. Lawrence Livermore National Laboratory. 2011. Disponível em: <https://computing.llnl.gov/tutorials/pthreads>. Acesso em: 08/01/2012.

[2] BODNAR, J. “The WinAPI (C Win32 API, No MFC) tutorial”. 2007. Disponível em <http://zetcode.com/tutorials/winapi/>. Acesso em: 07/02/2012.

[3] CARISSIMI, A., DUPROS, F., MÉHAUT, J. and POLANCZYK, R. V. “Aspectos de Programação Paralela em Arquiteturas NUMA”. Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD). 2007.

[4] CORSINI, J. F., FREITAS, L. G. and ROSSI, F. D. “Comparando Processos entre Unix e Windows”. Revista INFOCAMP. Março de 2006.

[5] DREPPER, U. and MOLNAR, I. “The Native POSIX Thread Library for Linux”. Technical Report. Red Hat, Inc. 2005.

[6] DUNKELS, A. “Protothreads - Lightweight, Stackless Threads in C”. 2005. Disponível em:<http://www.sics.se/adam/pt/>. Acesso em janeiro/2012.

[7] ENGELSCHALL, R. S. “The GNU Portable Threads”. 2006. Disponível em: <http://www.gnu.org/software/pth/>. Acesso em janeiro/2012.

[8] JOHNSON, P. "Pthread Performance in an MPI Model for Prime Number Generation". Technical report, University of Colorado. 2006.
[9] MOORE, G. E. "Cramming more Components onto Integrated Circuits". Readings in Computing Architecture. Editors: HILL, M. D., NORMAN, P. J. and GURINDAR S. S. . Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. Pages: 56-59. 2000.
[10] MUELLER, F. "A Library Implementation of POSIX Threads under UNIX". Proceedings of the USENIX Conference. Pages: 29-41. 1993.
[11] PENHA, D. O., CORRÊA, J. B. T., POUSA, C. R., RAMOS, L. E. S. and MARTINS, C. A. P. S. "Performance Evaluation of Programming Paradigms and Languages Using Multithreading on Digital Image Processing". Proceedings of 4th WSEAS International Conference on Applied Mathematics and Computer Science. 2005.
[12] SCHEFFER, R. "Uma visão Geral sobre Threads". Revista Campo Digit@l. Volume 2. Número 1. Páginas: 7-12. 2007.
[13] SILVA, R. R. and YOKOYAMA, R. S. "Avaliação do Desempenho da Utilização de Threads em User Level em Linux". Revista de Informática Teórica e Aplicada - RITA. Volume 18. Número 1. 2011.
[14] SPINELLIS, D. "A Critique of the Windows Application Programming Interface". Computer Standards and Interfaces. Volume 20, Issue 1, Pages: 1-8. 1998.
[15] TANENBAUM, A. S. "Modern Operating Systems (3rd edition)". Prentice Hall Press, Upper Saddle River, NJ, USA. 2008.
[16] THIBAULT, S. "PM2. Marcel: A Posix-Compliant Thread Library for Hierarchical Multiprocessor Machines". 2011. Disponível em: <http://runtime.bordeaux.inria.fr/marcel/>. Acesso em janeiro/2012.
[17] TORELLI, J. C. and BRUNO, O. M. "Programação Paralela em SMPS com OPENMP E POSIX Threads: Um estudo comparativo". Anais do IV Congresso Brasileiro de Computação (CBComp). Volume 1. Páginas: 486-491. 2004.

Alex G. C. de Sá graduated in 2011 in Computer Science by the Federal University of Lavras (UFLA) and is currently pursuing his studies at the Federal University of Minas Gerais (UFMG). He has worked with Artificial Neural Networks applied to biology (Bioinformatics) and currently has worked in several research areas, such as Heterogeneous Systems Modelling and Simulations (emphasis on communication), Computational Intelligence applied to Wireless Sensor Networks and Parallel and Distributes programming.

Marluce R. Pereira is a Professor at the Computer Science Department of the Federal University of Lavras (DCC–UFLA). She has an undergraduate degree given by Federal University of Juiz de Fora (UFJF - 1999), masters in Computer Science and Systems Engineering given by the Federal University of Rio de Janeiro (UFRJ - 2001) and a PhD from the same program (UFRJ - 2006). She has experience in Science, working mainly with Parallel and Distributed Programming, Parallel Processing, Intelligent Systems, Constraint Logic Programming and Software Development process.

Pedro M. Moura undergraduate degree in Computer Science by the Federal University of Lavras (UFLA - 2011) Works mainly with Wireless Networks, Next Generation Networks and Parallel and Distributed Programming. He is currently pursuing his graduate studies at the State University of Campinas (UNICAMP).

Luiz Henrique R. Peixoto undergraduate degree in Computer Science by the Federal University of Lavras (2011), where he researched on Parallel and Distributed Programming, Evolutionary Computation and GPU Programming with CUDA. He is currently pursuing his graduate studies on Computer Science at the Federal University of Itajubá (UNIFEI).