Performance Evaluation of Programming Paradigms and Languages Using Multithreading on Digital Image Processing

DULCINÉIA O. DA PENHA[1], JOÃO B. T. CORRÊA[2], LUIZ E. S. RAMOS[3], CHRISTIANE V. POUSA[4], CARLOS A. P. S. MARTINS[5]
[1, 2, 3, 4, 5] Computational and Digital Systems Laboratory / [5] Informatics Institute
[1, 3, 4, 5] Graduate Program in Electrical Engineering
Pontifical Catholic University of Minas Gerais
Av. Dom José Gaspar 500, 30535-610, Belo Horizonte, Minas Gerais
BRAZIL

http://www.ppgee.pucminas.br/lsdc/

Abstract: - We present a comparative performance evaluation of different programming paradigms and languages using multithreaded programming. We compare the procedural and object-oriented (OO) paradigms, as well as the C++ and Java languages, regarding both performance and programmability. The comparison is made upon sequential and parallel image convolution implementations based on those paradigms and languages. The parallel implementations used the shared-variable programming model and multithreading. They exploited not only pure parallelism, but also parallelism combined with concurrency. The performance evaluation was based on the response time of those implementations. The evaluation of system performance showed that pure parallelism yielded better performance results than parallelism combined with concurrency. Regarding the C++ implementations, the procedural paradigm led to better results than the OO paradigm. One of the most significant results in our work is the fact that Java yielded shorter response times than OO C++ for most of the multithreaded implementations.

Key-Words: - Programming, Paradigms, Languages, Parallelism, Concurrency, Image, Convolution, Performance

1 Introduction

Nowadays, a number of applications in many areas of knowledge (scientific, commercial, industrial, etc) demand very short response times. Due this fact, those applications require a great amount of computational resources for storage, transmission and information processing. A possible solution for this problem is the use of high performance computing (HPC). An HPC system may perform sequential or parallel processing [1] [2] [3].

The effective use of parallel systems is a very difficult task because it involves the design of correct and efficient parallel applications. Thus, in parallel architectures, the programming transparency is an important issue for the developers of parallel applications. Some techniques can be used to provide the desired transparency. Compiler directives, functions and classes from multithread support libraries are commonly used in operating systems that support multiprocessor computational systems.

The **problem** on which we focus is the correlation among architectures, paradigms, languages and performance. Thus, our **main goal** is to compare and evaluate the system's performance, whereas using different programming paradigms and languages with multithread programming. We compare the procedural and object-oriented (OO) paradigms, using the C++ and the Java languages.

Our evaluation was based on the overall system's performance, not concerning about specific features of other architectural blocks, such as: processor, memory and communications. The standard support libraries used to implement the multithreaded code were: WinThread for the Windows operating system and the Threads class for Java. Based on the evaluation of programmers and developers, in our evaluation, we also considered the: programmability, programming methods, portability, simplicity and transparency of those libraries.

We decided to vary some parameters while the others remained unchanged. The variable parameters were: the paradigms and languages and the non-variable parameters were: the execution system and the features of the implementations. Moreover, no compiler or language optimizations were used.

The workload for our experiments and comparative analysis is an image convolution, which is a digital image processing (DIP) operation. It is one of the most important DIP operations and demands a considerable amount of computational resources to be executed.

Digital images are composed of a great amount of data and are often stored in matrixes. Their manipulation usually has a high cost and consumes large computational resources. DIP operations have a parallel nature [1] [11] because they perform independent actions over independent data (pixels, consisting of the elements that compose the image representation matrix) [5]. Thus, in many situations the use of general-purpose parallel architectures using shared memory (shared-variables programming model) yields performance gains [4] [11].

So, we developed sequential and parallel image convolution implementations and varied the paradigm and the language, in a total of six combinations. There was a sequential and parallel version for each of the following implementations: procedural C++, OO C++ and Java (OO paradigm). The parallel versions are based on the shared-variable programming model, using multithread programming. They used: explicit compiler directives, classes and functions from the used multithread support libraries (WinThread and class Thread from Java).

In the procedural programming paradigm, the problem is broken into smaller pieces that can be solved algorithmically, within a specific number of steps. As soon as the variables are declared, the specified sequence of actions is followed. The program and the data are viewed as separate entities.

In the OO paradigm, the application to be implemented is a set of interacting objects. The programmer defines objects and their associated properties. The sequence of actions that occur in the running system depends on how the user interacts with the objects.

The use of threads is a possible way to obtain parallelism support in a program, usually by means of compilers and system libraries. In this work we use the WinThreads library from the C++ Builder Compiler and the Thread class from J2SDK.

The results in this paper are part of a larger research [6] [7] related with performance, architecture and programming on: different programming paradigms and languages, different parallel algorithm models, algorithm optimization, and compiler optimization using the shared-variable and the message-passing parallel programming models. Our larger research is a comparative evaluation of the correlation among the objects' combinations and their influence on the system's performance. We combined objects that belong to different computational abstraction levels and compared and evaluated them based on performance and programmability.

Our main contribution is the comparative performance evaluation of different programming paradigms and languages using multithread mechanisms applied on a digital image convolution operation.

2 The Convolution Operation

A filtering operation in space domain is called convolution. The term space domain refers to the aggregation of pixels that compose an image. Operations in space domain are the procedures applied directly on those pixels [5]. The equation (1) describes the convolution operation.

$$P[x][y] = \sum_{u=u-\frac{k}{2}}^{\frac{k}{2}} \sum_{v=v-\frac{k}{2}}^{\frac{k}{2}} I[x+u][y+v] \times M[u][v] \qquad (1)$$

The convolution is carried out for each pixel (P[row][column]) of an NxN image I, with a KxK mask. A convolution mask is applied on each pixel of the input image, resulting in a convolved (filtered) output image [5]. In this work, we use high-pass and low-pass spatial filters to carry out the tests and comparisons.

The convolution mask that characterizes a high-pass filter is composed of positive coefficients in its center (or next to it), and negative coefficients in the surroundings [5]. The high-pass filtering operation produces a highlight effect on the edges of the original image. It happens because the appliance of a high-pass mask on a constant area (or with a small gray level variation) generates the output with a zero value or near zero [5].

We applied the convolution operation with a high-pass filter on the original image showed in Fig.1 (a). We show this convolution result in Fig. 1 (b). Fig. 1 (c) presents the negative image to validate the convolution operation. We introduced a border on the negative image to show the actual image dimensions. So, the border is not a part of the image.

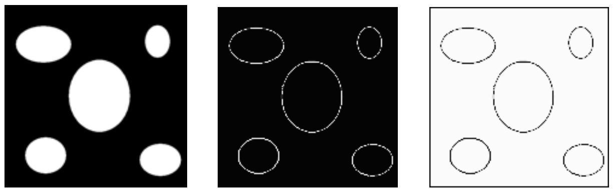

(a) (b) (c)

Fig. 1. Original image for high-pass convolution (a) Image of the convolution with a high-pass filter (b), and its negative image (with additional border) (c)

3 The Paradigms and Languages

The procedural paradigm represents the traditional approach for programming (for example, it is the basis for the CPU's fetch-decode-execute cycle). This paradigm defines a program as a sequence of instructions that manipulates data in order to produce the desired results [8]. The entire logic of the program is a series of instructions, in which the problem is divided into smaller pieces. A program based on the procedural paradigm executes efficiently, because the software matches the hardware. Nevertheless, the procedural paradigm is not enough for handling today's problems, because these may be too complex and/or too large to be implemented as functions. Moreover, another problem of this paradigm is that it does not facilitate code reusability.

Object-oriented (OO) programming has been presented as a technology that can fundamentally aid software engineering, because the underlying object model fits the real domain problems better [13]. The OO paradigm is focused on the behavior and the structural characteristics of entities as complete units. The main advantage of OO is the easiness of reutilization. When it is necessary to change the program's code, the programmer modifies specific classes and makes only the required adjustments. This eliminates excessive code browsing and dependency checking in order to make the changes [12].

C++ is an object-oriented language based on C. It can be viewed as a superset of C because almost all of the features and constructs available in that language are also available in C++. Its additional features support the OO programming paradigm [4] [9].

Java is a portable object-oriented language that is executed on top of a virtual machine. The user's program source-code is compiled into a byte-code, which is interpreted by the JVM (Java Virtual Machine) when the program runs over a specific architecture [12]. One of the benefits of the Java language is the support to multithread programming as a part of the language. In Java, each thread that runs in JVM is associated with an object of the Thread class [12].

4 Shared-Variable Parallel Programming

Shared-memory parallel architectures can use shared-variables for the communication between the application processes or threads. The effective use of parallel systems is a very difficult task because it involves the design of correct and efficient parallel applications. This fact results in several complex problems as process synchronization, data coherence and event ordering. There are some ways to using parallelism in order to provide some transparency to the programmer [6]. Regular modern operating systems (OSs) provide support to multiprocessor systems. In these systems, the parallel execution is activated on the creation of multiple threads or processes that run in parallel.

A thread (or a control thread) is a sequence of executing instructions. Each process has one or more threads. The threads belonging to a process share its address space, its code, most of its data and process descriptor information. The use of threads makes it easier for the programmers to write their concurrent and parallel applications in a transparent way [6].

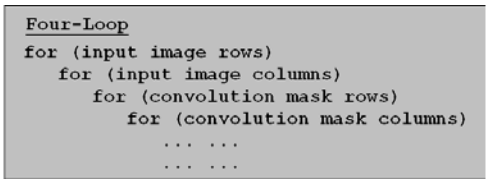
There are two main operating system specific multithread programming libraries. One of them is for Unix/Linux OSs (Pthread standard), and the other for Windows OSs (WinThread). Besides, there are the operating system and architecture independent Java Thread Class.

5 Image Convolution Implementations

In this section we present the image convolution implementations used in this work. They are: a sequential procedural C++, a sequential OO C++, a sequential Java (OO), a parallel procedural C++ (using WinThread), a parallel OO C++ (using WinThread), parallel Java (OO, using Java Thread Class). We used the Borland C++ Builder 5.0 compiler for the C++ implementations and J2SDK 1.4.0_01 for Java implementations.

5.1 Procedural Implementations

The sequential procedural (C++) implementation is based on a 4-level loop showed in Fig. 2.



```
Four-Loop
for (input image rows)
   for (input image columns)
      for (convolution mask rows)
         for (convolution mask columns)
            ... ...
            ... ...
```

Fig. 2. Basic 4-level loop convolution algorithm

These implementations use three different matrixes that store: the original image, the convolution mask and a temporary matrix. The last one is used to store the convolved pixels of the image. We initially load the image and the mask matrixes with the correct values and initialize other variables. Then the basic 4-level loop convolution is executed and the result of each convolved pixel is copied into the temporary matrix. Finally, the values of the temporary matrix (containing the convolved image) are copied into the image matrix.

Upon the sequential procedural implementation, we developed a parallel procedural version (in C++). The difference is that the whole image is divided into slices containing image rows. Each slice is convolved by a specific user level thread, whose code is similar to the basic 4-level loop (showed in Fig. 2). A thread parameter (argument) is the starting row of the slice that it will convolve. The three matrixes (image, mask and temporary) are shared among all threads. A vector of threads keeps the reference to each thread (function, arguments, etc).

After loading the image and convolution variables, we create the threads according to the number of desired processes. Each thread is created in a suspended mode and is referenced by a position in the vector of threads. After their creation, they are all started. At this point all threads are executed in parallel or concurrently. When all threads finish their jobs, the temporary matrix is copied into the image matrix.

5.2 Object-Oriented Implementations

The OO convolution implementations are based on the sequential one. An advantage of the OO programming paradigm is the code reutilization.

On the C++ and the Java sequential OO implementations we created a single class encapsulating all the objects used on the procedural implementation (e.g.: the image and the mask matrixes, variables and functions). On the main code we declared the object, allocated memory space for them, and called the functions that executed the operations. At the end of the execution we deallocated the space of the objects.

The parallel (with user level multithread programming) OO implementations (in C++ and Java languages) were developed through the construction of two objects. The first one was responsible for thread management (creation, destruction, synchronizations and accesses to the shared variables) and for keeping the image to be convolved. The second object implemented the thread's code. Each thread executed a part of the image convolution operation. In the Java implementation, this operation was located in the run method of the class. This method extends the Thread library and implements the main methods of the threads (run, join, start, stop). When the program started the threads were created and initiated, and then they executed the convolution operation sharing the matrix that contained the image.

Then the whole image was divided into slices that were convolved by each thread. The thread convolution code is also similar to basic 4-level loop (showed in Fig. 2). A thread parameter (argument) is the starting row of its slice (which it will convolve). So the threads calculate the finish row of the image slice.

6 Results

We executed all tests with the convolution implementations over an Intel Dual Pentium III 933MHz with a 768MB primary memory, and the Windows XP operating system. In each test, the image, the mask and the other convolution variables of each implementation were previously loaded in memory. We did this to eliminate the influence of the virtual memory on the performance results. Thus, the time had been measured for the computations that followed. The C++ implementations tests were executed using Prober, a functional and performance analysis tool [10]. The Java implementations tests were executed manually.

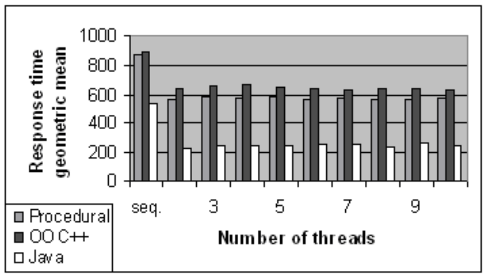
We executed each implementation ten times for each chosen image size (512x512, 1024x1024, 2048x2048 pixels) and mask size (3x3, 5x5, 7x7). We calculated both the arithmetic and the geometric means of the response times obtained in each execution (iteration) in order to analyze and compare the results. We show in Table 1 the arithmetic and the geometric means for 1024x1024 images with a 5x5 mask in the OO C++ implementation. The difference between the arithmetic and the geometric means was short for all implementations; namely, the system was steady and presented a deterministic behavior in all tests.

Table 1. Arithmetic and geometric means for the OO C++ Implementation (1024x1024 image, 5x5 mask)

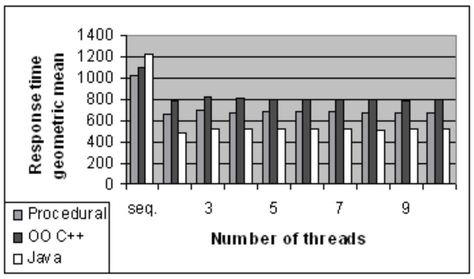
Threads	Arithmetic	Geometric
1	1093.800	1093.875
2	791.000	791.375
3	821.900	822.250
4	809.500	808.750
5	804.700	802.875
6	804.700	802.750
7	799.900	800.625
8	801.400	800.625
9	790.400	790.750
10	793.800	793.000

In all tests regarding the procedural implementations (with the C++ language), we observed that the response times obtained using pure parallelism were shorter than the response times obtained using parallelism combined with concurrency. In these cases, the use of concurrency associated with parallelism was not interesting because the achieved speedup was greater than those obtained from purely parallel executions, as we expected. This happened because, in the case of concurrency for processor and memory, the response time is increased by the time spent in context switching among the threads. In the tests regarding the OO C++ implementation, we observed that only few executions obtained better results (shorter response times) with pure parallelism than with both parallelism and concurrency. We obtained the shortest difference between the response times using pure parallelism and one using parallelism combined with concurrency, as the image and mask size increased.

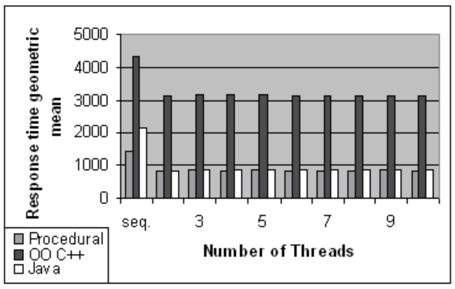
In the tests regarding Java (OO paradigm), the response times with pure parallelism were better than those with parallelism and concurrency, except for 512x512 images and 5x5 mask. The graphics 1, 2 and 3 show the response times of the implementations using pure parallelism (two threads, each one running over a processor of the test machine) and the response times of the implementations combining parallelism with concurrency (more than 2 threads), for all paradigms, languages and mask sizes, using 1024x1024 images.



Graphic 1. Geometric mean of the response times for a 1024x1024 image and a 3x3 mask



Graphic 2. Geometric mean of the response times for a 1024x1024 image and 5x5 mask



Graphic 3. Geometric mean of the response times for a 1024x1024 image and 7x7 mask

As it was expected, the response times of the OO implementations always presented larger response times than the procedural implementations in the C++ language (for the sequential and the parallel versions). Table 2 presents the response times obtained with the procedural and the OO C++ implementations for 2048x2048 images and a 7x7 mask. Observing the obtained results, we concluded that the use of multiple threads in the OO C++ implementation was not interesting in this case, because the maximum efficiency was smaller than for the other implementations (approximately 0,735 for 2048x2048 images and 3x3 masks). A possible reason for this is the fact that the thread management of the C++ Builder compiler was not optimized considering the context.

The programming and executing model of the procedural implementations fitted the hardware functioning model better than the model of the OO C++ implementation. For that reason the procedural versions presented the best speedup results. Table 2 shows the speedup obtained with the sequential and the parallel OO C++ versions.

The most of the response times regarding the sequential OO C++ implementation were better (shorter) than those obtained with the Java implementation. The worst response times were achieved in the executions of the OO C++ implementation. They involved: a 512x512 image using a 3x3 mask, a 1024x1024 image using 3x3 and 7x7 masks, and a 2048x2048 image using a 3x3 mask. On the other hand, the response times of the multithreaded versions were always better with the Java implementations.

Table 2. Geometric mean of the response times and the speedup for the procedural vs. the OO C++ (2048x2048 image, 7x7 mask)

N. of Threads	Procedural Response Times	Procedural Speedup	OO Response Times	OO Speedup
1	5820.375	1	5841.625	1
2	3296.875	1.766	4250.000	1.374
3	3383.000	1.713	4285.000	1.361
4	3304.500	1.761	4261.625	1.369
5	3326.125	1.748	4275.250	1.366
6	3308.625	1.759	4261.750	1.370
7	3318.125	1.754	4259.875	1.371
8	3310.375	1.758	4259.625	1.371
9	3306.750	1.760	4251.750	1.374
10	3302.875	1.761	4242.125	1.377

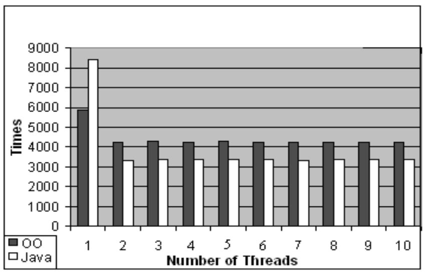
In Table 3 we show the response times obtained with OO C++ and Java implementations for 2048x2048 images and a 7x7 mask. For better visualization we show these response times in Graphic 4.

As we expected, the response times for the sequential Java were worse than those of the OO C++ because the first is interpreted. On the other hand, the parallel Java implementation performed better, presenting better response times. A possible reason for that is the fact the images and masks were already loaded on the JVM stack, which is an accessible memory area for all threads. Thus, the memory accesses and the context switches among Java threads are faster than the mechanisms used on the C++ threads. We are now evaluating and analyzing low-level metrics using PAPI and other performance monitoring tools as JFluid (Java) and Windows Performance Monitor.

Table 3. Geometric mean of the response times and the speedup for the OO C++ vs. the Java implementations (2048x2048 images, 7x7 mask)

Threads	OO C++	OO Speedup	Java	OO Speedup
1	5841.625	1	8408.250	1
2	4250.000	1.374	3322.000	2.532
3	4285.000	1.361	3378.875	2.484
4	4261.625	1.369	3347.500	2.510
5	4275.250	1.366	3363.375	2.501
6	4261.750	1.370	3373.125	2.493
7	4259.875	1.371	3339.875	2.519
8	4259.625	1.371	3359.125	2.503
9	4251.750	1.374	3353.375	2.506
10	4242.125	1.377	3357.375	2.504

As the execution environment was a dual processor computer, the size of the L1 cache was duplicated and the number of processors benefited all parallel versions. As a consequence, the cache hit ratio increased and the Java parallel versions obtained a higher speedup, possibly because of the memory stack sharing within JVM and its threads. These facts can explain the high speedup reached by Java implementations. Its speedups were greater than two (maximum expected speedup using two processors). In Table 3 we show the speedup for the OO C++ and the Java implementations with 2048x2048 images and a 7x7 mask.



Graphic 4. Response times for the OO C++ and the Java implementations (2048x2048 image, 7x7 mask)

7 Conclusions

For all tests with different images and mask sizes, the six parallel versions (using two threads) presented better results than their respective sequential versions, as we expected. And, we showed that the speedup for all Java parallel implementation was greater than 2. In the case of 512x512 image and a 3x3 mask the speedup reached 2,97. As we expected, the use of parallelism combined with concurrency did not provide the best results in any situation.

The procedural implementation's response times were better than the ones of the OO C++ implementations. This was also expected because the execution model of the procedural paradigm fitted the hardware's functioning model better than the OO paradigm.

As we expected, the response times of the sequential Java implementations was worse those yielded by the OO C++ versions. It occurred because the Java based code is interpreted. On the other hand, the response times of the parallel (multithreaded) Java implementations were better than the OO C++ ones for all image and mask sizes. Thus, we believe that the thread utilization (creation, manipulation, finalization, etc) is more optimized in the Java language through the use of the Thread class than the thread utilization in C++ Builder compiler. We intend make further tests in future researches using performance monitor tools to investigate this.

Other possible reasons for these results are: the fact that JVM's memory stack may cause the cache-hit ratio to increase due to the doubled cache size; and the fact that Java Threads present faster context switches than the C++ threads. For such reasons parallel Java implementations presented the best results.

In this work, our main contributions were the comparison between the procedural and the object-oriented programming paradigms, and the comparison between the C++ and the Java languages using multithread mechanisms applied on a digital image processing operation.

As future works we intend to investigate the reasons for the best performance obtained with the Java interpreter, as well as the worse performance obtained with multiple threads in the OO C++ implementation, using C++ Builder compiler. This can be made by running tests on other C++ compilers. We also intend to investigate the possibly better thread management of JVM, in order to explain the obtained results.

References:

[1] G. S. Almasi and A.A. Gottlieb, "Highly Parallel Computing", 2nd. Edition, Benjamim/Cummings, 1994.

[2] S. S. Mukhrjee, S. V. Adve, T. Austin, J. Emer and P. S. Magnusson, "Performance Simulation Tools", Guest Editors' Introduction, Computer, Vol. 2, IEEE, February, 2002.

[3] K. Hwang and Z. Xu. "Scalable Parallel Computing: Technology, Architecture, Programming", McGraw-Hill, 1998.

[4] G. Satir, and D. Brown. "C++: The Core Language", 1st. Edition, O'Reilly, October 1995.

[5] R. C. Gonzalez and R. E. Woods, "Digital Image Processing", 2nd. Edition, Addison Wesley Publishing Co., Massachusetts, 1987.

[6] D. O. Penha, J. B. T. Corrêa, and C. A. P. S. Martins, "Análise Comparativa do Uso de Multi-Thread e OpenMp Aplicados a Operações de Convolução de Imagem", III WSCAD (*Workshop de Sistemas Computacionais de Alto Desempenho*), Vitória, Brazil, 2002.

[7] D. O. Penha, J. B. T. Corrêa, L. F. W. Góes, L. E. S. Ramos, C. V. Pousa, and C. A. P. S. Martins, "Comparative Analysis of Multi-threading on Different Operating Systems Applied on Digital Image Processing", CSITeA (*International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications*), 2003.

[8] J. G. Brookshear, "Computer Science: An Overview", 5th. Edition, Addison-Wesley, Reading, MA, 1997.

[9] B. W. Kernighan, and D. M. Ritchie. C Programming Language. 2nd Edition. Prentice Hall PTR, March 1988.

[10] L. F. W. Góes, L. E. S. Ramos, and C. A. P. S. Martins. "Performance Analysis of Parallel Programs using Prober as a Single Aid Tool", 14th. SBAC-PAD (*Symposium on Computer Architecture and High Performance Computing*), 2002.

[11] D. Roman, M. Fischer, and J. Cubillo. "Digital image processing-an object-oriented approach", Transactions on Education, IEEE, Vol. 41, Issue 4, November, 1998.

[12] K. J. Gough. "Stacking them up a comparison of virtual machines", ACSAC, Proceedings 6th Australasian, January, 2001.

[13] H. J. Nelson, D. E. Monarchi, and K. M. Nelson. "Evaluating emerging programming paradigms: an artifact-oriented approach", System Sciences, 31st Hawaii International Conference, Vol. 6, January, 1998.