Hybrid Algorithms for Matrix Multiplication on Multicore Clusters

Fabiana Leibovich, Marcelo Naiouf, Laura De Giusti, Fernando G. Tinetti and Armando De Giusti

Instituto de Investigación en Informática LIDI (III-LIDI), Facultad de Informática, Universidad Nacional de La Plata, 50 y 120 2do piso, La Plata, Argentina.

Abstract - Hybrid programming (through messages and shared memory) has gained importance since the appearance of multicore cluster architectures, fruit of the technological advance of processors and the physical limitations imposed by traditional architectures. This new programming paradigm allows exploiting the new memory hierarchy offered by the architecture. The purpose of this work is to carry out a comparative analysis between two hybrid algorithms that use two different parallel programming libraries for shared memory (PThreads and OpenMP). Both algorithms use MPI as message passing mechanism. For the experiments, the classic matrix multiplication problem is used, which is the basis for numerous applications. The test architecture used for the experimental analysis is a multicore cluster. The performance obtained and programming tools are compared.

Keywords: parallel architectures, hybrid programming, cluster, multicore, message passing, shared memory.

1 Introduction

Parallel architectures have evolved to offer better response times for applications. As part of this evolution, clusters, then multi-cores, and currently multi-core cluster architectures, can be mentioned. The latter are basically a collection of multi-core processors interconnected through a network.

Multicore clusters allow combining the most distinctive features of clusters (use of message passing in shared memory) and multicore (use of shared memory). Also, they introduce modifications in memory hierarchy and further increase computer system capacity and power.

Taking into account the popularity of this architecture, it is important to study new parallel algorithms programming techniques that efficiently exploit its power, considering the hybrid systems in which shared memory and distributed memory are combined [1].

When it comes to implementing a parallel algorithm, it is very important to consider the memory hierarchy (Figure 1) available, since this will directly affect algorithm performance. Figure 1 below shows the evolution.

Memory hierarchy performance is primarily determined by two hardware parameters: memory latency (time elapsed from the moment a piece of data is required and the moment it becomes available) and memory bandwidth (the speed with which data are sent from the memory to the processor). In the case of traditional clusters (both homogeneous and heterogeneous), there are memory levels in each processor (processor register and cache levels L1 and L2), but a new level is also included: network-distributed memory.

When considering a multi-core architecture, there are, in addition to register and L1 levels corresponding to each
core, two memory levels: cache memory shared by pairs of cores (L2) and memory shared among the cores of the multi-core processor [2]. Currently, the new architectures include one additional memory level, L3 cache.

In particular, multi-core clusters introduce one additional level to the traditional memory hierarchy. In addition to the cache memory shared between pairs of cores and the memory shared among all cores within the same physical processor, there is the distributed memory that is accessed through the network. This can also be seen in Figure 1.

This paper is organized as follows: Section 2 presents the objectives, then, Section 3 analyzes the parallel programming libraries used for the experiments. In Section 4, the existing mapping techniques used in this paper are discussed, while Section 5 presents the study case selected for testing and the solutions implemented. Finally, Section 6 discusses the results achieved, and Section 7 details the conclusions and future work.

2 Objective

The purpose of this paper is to carry out a comparative analysis of two hybrid solutions [3][4][5] that use the two shared memory libraries most commonly used nowadays in parallel computing: Pthreads and OpenMP.

Even though there is a large number of parallel applications in various areas, one of the most traditional and widely studied in parallel computation, and used in this paper, is matrix multiplication. It allows analyzing application scalability in two ways: by increasing the size of the problem and increasing the number of execution cores.

3 Parallel Programming Libraries

3.1. Pthreads

Over time, hardware manufacturers have implemented their own versions for thread management and administration. These versions are very different from each other, which makes it difficult for programmers to develop multithread applications that are portable. For this reason, in 1995 the IEEE established the POSIX (Portable Operating System Interface) standard. Its last version is IEEE Std 1003.1, 2004 Edition [6].

Pthreads is a library that implements the POSIX standard defined by IEEE, and is composed by a set of types and calls to procedures in programming language C that includes a header file and a thread library that is part, for example, of the libc library, among others. It is used for programming parallel applications that use shared memory.

The subroutines that form the API in Pthreads can be classified in four large groups: thread management, mutex (routines that handle synchronization and mutual exclusion), condition variables, and synchronization.

3.2. OpenMP

OpenMP is a programming interface (API) defined by a set of hardware and software manufacturer including the following: Sun Microsystems, IBM, Intel, AMD, among others [7].

It provides a portable and scalable interface for the developers of parallel applications that use shared memory.

This API supports C/C++ and Fortran in multiple architectures, including LINUX and Windows NT. It provides several builders and directives to specify parallel regions, shared work, synchronization, and environment variables.

It is formed by the following three components:

1. Compilation directives
2. Running time routine library
3. Environment variables

3.3. OpenMPI

MPI is a message passing interface that defines both the syntax and the semantics of the set of routines that can be used in the implementation of programs that use message passing. It was created to solve the issues that appeared after each hardware manufacturer defined its own communication interface, which in general were incompatible with all others. The purpose of MPI is to solve this problem by defining a standard [8].

MPI is a library that can be used to develop programs that use message passing (distributed memory) and uses the programming languages C or Fortran.

One of the implementations of this standard is OpenMPI, which is used in this paper, because it provides manual mapping mechanisms to assign processes to cores for their execution. This is discussed in Section 4.

3.4. Pthreads vs. OpenMP

It should be noted that OpenMP generates a pool of n threads (n is defined by the programmer) that is going to be re-used as needed in a very efficient manner. Thus, there is no need to create and remove threads when entering/leaving each parallel section. This is extensively exploited when the algorithm has several parallel sections.

In the case of Pthreads, threads would have to be created and destroyed for each of these sections, or a pool of threads would have to be created and kept from the algorithm. Also, if the algorithm required synchronization among threads, or more complex
parallelizations, such as shared variables reduction operations, combination of critical and non-critical sections, etc., OpenMP would in theory yield better results than Pthreads, since the library itself provides specialized and optimized mechanisms to do so, whereas Pthreads would require a manual implementation using semaphores or condition variables.

For this reason, one of the advantages of Pthreads over OpenMP is that the programmer has greater control over thread creation, destruction and behavior, since thread management is at a lower level than in OpenMP.

4 Thread and Process Manual Mapping

The traditional operating systems map threads and processes to processors and cores using various scheduling techniques.

However, there are mechanisms that allow application programmers to manually map these threads and processes. In the case of threads, the code of the application has to be modified, whereas in the case of process mapping, it can be done while it is running by using directly the binary.

Based on the architecture used in this paper, that is, multicore cluster, both mapping alternatives were used and combined to obtain a better global system performance. This can be done because the algorithms being studied use a hybrid model, combining shared memory and message passing.

4.1. Process Mapping

Open MPI is an open code project that implements the MPI standard and offers the added functionality of providing directives for explicitly mapping processes to processing cores. To do so, it requires two files called rankfile and hostfile.

The hostfile file defines the number of cores available in the system and the name of the machine in the network. The format of this file is the following:

```
hostNameX slots = number of cores
hostNameY slots = number of cores
```

The rankfile file defines an input for each process, as shown below:

```
rank N = hostNameX slot = number of CPU
rank M = hostNameY slot = number of CPU:number of core
```

Figure 2 shows a diagram explaining the parameters number of CPU and number of core.

These two files must be passed as parameter upon execution.

The advantage of this alternative is that the source code is untouched, there is no need to modify it to add this functionality, and as a consequence, the mapping process does not consume execution time. The disadvantage is that mapping is static and cannot be dynamically modified because it is passed as parameter upon execution.

4.2. Thread Mapping

In shared memory systems there is no specific function to assign a thread to a given processing core. Instead of this, there is a function that allows defining the affinity for each task.

Affinity allows specifying which, of all existing cores in a system, can be assigned a certain process or thread. To be able to assign these processes or threads to any given specific core, the core in question must be set as the only schedulable processor/core for such assignment.

Unix-based operating systems offer a system call that allows explicitly defining affinity. This should be used in shared memory environments, since it can manage the cores and processors in the machine on which the operating system is being run. The heading of this function is as follows:

```
sched_setaffinity(pid_t pid, unsigned long cpusetsize, cpu_set_t *mask)
```

Where:

- **pid**: is the process identifier for which affinity is defined. If its value is 0, it represents the process being run. It should be noted that the threads belonging to a same process share the same PID; however, the affinity belongs to each separate thread and is stored in the thread context.

- **cpusetsize**: this is the length (in bytes) of the data noted by the mask parameter.
- mask: this represents the affinity mask. It is a bit mask where each bit represents a (logical) processor in the system. Bits are sorted from the less significant one, corresponding to the first logical processor, to the most significant one, corresponding to the last logical processor in the system.

  If bit = 0, the processor is non-schedulable

  If bit = 1, the processor is schedulable

By default, when a thread/process is created, it is schedulable for all existing cores in the system. To map a process or thread to a specific processor, such processor must be set as the only schedulable processor.

One of the advantages of this scheduling technique is the possibility of dynamically modifying (during the execution) the schedule of any given thread/process. This is possible because the system call can be included in the application code.

5 Study Case

Given two matrixes A and B having dimensions m*p and p*n elements respectively, the multiplication of both matrixes consists in obtaining matrix C of dimension m*n elements (C = A*B), where each element is calculated with equation (1):

\[ C_{i,j} = \sum_{k=1}^{p} A_{i,k} \times B_{k,j} \]  

(1)

The implemented solutions can be classified in two classes: traditional matrix multiplication (I), and block-based matrix multiplication (II) (matrix C is calculated in blocks). In both cases, the implemented solutions are hybrid, combining shared memory with message passing, and use the master/worker interaction pattern. These were both developed using C language – in the case of message passing, the OpenMPI library [8] was used, while Pthreads [6] was used in one case of shared memory and OpenMP [7] was used in the other.

The testing architecture used should be noted, since it affects the implementation of the algorithms. The hardware used to carry out the tests was a Blade with 16 servers (blades). Each blade has 2 quad core Intel Xeon e5405 2.0 GHz processors; 2 Gb of RAM memory (shared between both processors); 2 X 6Mb L2 cache shared between each pair of cores by processor. The operating system used is Fedora 12, 64 bits [9][10].

5.1. Traditional Matrix Multiplication

There is one process per blade and each blade has 8 execution units – 7 threads are used for processing activities, added to the processing activities from the process itself that acts as a worker (one thread per core). A master/worker structure is used, with one of the processes acting as master, dividing the rows equally among all processes. Once this is done, it generates the corresponding processing threads (acting as worker). The other worker processes act in a similar way and send their results to the master process.

The algorithm can be summarized as follows:

**Master process:**
- It divides the matrix into blocks of n rows/number of blades used for processing
- It communicates the corresponding rows from matrix A and all of matrix B to worker processes.
- It generates the threads and processes its own block
- It receives results from worker processes.

**Worker processes**
- They receive the corresponding rows from matrix A and all of matrix B.
- They generate the threads to process the data.
- They communicate the results to the master process.

Figure 3 shows an explanatory illustration. The numbers placed on matrix C indicate the number of the thread processed by each cell.

![Figure 3 Classic matrix multiplication](image)

5.2. Block Matrix Multiplication

Matrix C is calculated in blocks. To do so, each process receives the rows from matrix A and the columns from B required for calculating the block of matrix C that was assigned to it. The number of blocks into which matrix C is divided is divisible by the number of processes. The algorithm uses a master/worker-type interaction, where the master works both as coordinator and as worker. It divides matrix C into blocks to be processed and then generates processing phases. The same as in the traditional solution, each process is assigned to a blade and each thread to a core within the corresponding blade. Figure 4 shows an explanatory illustration. The numbers placed on matrix C indicate the number of the thread processed by each cell.
They communicate the results to the master process.

6 Results obtained

The following tables show the execution times (expressed in seconds), the difference in time between both implementations, and the percent difference, (based on *Pthreads*) obtained for solution (I) explained in the previous section for 16 (Table 1) and 32 (Table 2) cores, and the results obtained by solution (II) for 16 (Table 3) and 32 (Table 4) cores.

Also, two charts are shown comparing solution execution times for traditional multiplication (Figure 5) and block multiplication (Figure 6).

![Figure (4) Block Matrix multiplication](image-url)

Since all processors have the same computation power and all blocks to be processed are the same size, they will all be processed at (approximately) the same speed. Thus, for each processing phase, the master distributes the rows from matrix A and the columns from matrix B based on the block corresponding to each worker, including a block for itself. The master then processes its block and receives all the other results to move on to the next processing phase. The number of processing phases is calculated as follows: if b is the number of blocks that have to be processed and w is the number of workers (including the master, who also operates as a worker), the number of phases is b/w.

It should be noted that each process must store the rows from matrix A to be processed, the columns from matrix B, and the block from matrix C that it generates as a result.

In each phase, once the master distributes the blocks to the workers, it generates the corresponding threads to process its own block, dividing it into rows for each thread to process a subset of rows. The other worker processes act in a similar way, receiving data and sending their results to the master process.

The algorithm can be summarized as follows:

Master process:

- It divides matrix C into blocks.
  - For each phase:
    - It communicates the corresponding rows from matrix A and the corresponding columns from matrix B to worker processes, based on the block assigned to each of them.
    - It generates the threads and processes its own block.
    - It receives results from worker processes.

Worker processes:
  - For each phase:
    - They receive the data to be processed.
    - They generate the threads and process the data.

![Table (1)]

<table>
<thead>
<tr>
<th>Size</th>
<th>Pthreads</th>
<th>OpenMP</th>
<th>Difference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.32</td>
<td>0.32</td>
<td>0.0073</td>
<td>2.29</td>
</tr>
<tr>
<td>2048</td>
<td>2.28</td>
<td>2.34</td>
<td>0.0527</td>
<td>2.30</td>
</tr>
<tr>
<td>4096</td>
<td>17.19</td>
<td>17.51</td>
<td>0.3115</td>
<td>1.81</td>
</tr>
<tr>
<td>8192</td>
<td>133.18</td>
<td>139.59</td>
<td>6.41</td>
<td>4.81</td>
</tr>
<tr>
<td>9600</td>
<td>216.73</td>
<td>213.11</td>
<td>-3.61</td>
<td>-1.66</td>
</tr>
</tbody>
</table>

![Table (2)]

<table>
<thead>
<tr>
<th>Size</th>
<th>Pthreads</th>
<th>OpenMP</th>
<th>Difference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.22</td>
<td>0.23</td>
<td>0.015</td>
<td>7.14</td>
</tr>
<tr>
<td>2048</td>
<td>1.40</td>
<td>1.42</td>
<td>0.014</td>
<td>1.06</td>
</tr>
<tr>
<td>4096</td>
<td>9.56</td>
<td>9.76</td>
<td>0.198</td>
<td>2.07</td>
</tr>
<tr>
<td>8192</td>
<td>69.91</td>
<td>74.15</td>
<td>4.23</td>
<td>6.06</td>
</tr>
<tr>
<td>9600</td>
<td>113.48</td>
<td>113.51</td>
<td>0.027</td>
<td>0.02</td>
</tr>
</tbody>
</table>

![Table (3)]

<table>
<thead>
<tr>
<th>Size</th>
<th>Pthreads</th>
<th>OpenMP</th>
<th>Difference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.16</td>
<td>0.17</td>
<td>0.0072</td>
<td>4.39</td>
</tr>
<tr>
<td>2048</td>
<td>1.59</td>
<td>1.60</td>
<td>0.0047</td>
<td>0.29</td>
</tr>
<tr>
<td>4096</td>
<td>11.97</td>
<td>11.87</td>
<td>-0.0981</td>
<td>-0.81</td>
</tr>
<tr>
<td>8192</td>
<td>104.14</td>
<td>104.37</td>
<td>2.9637</td>
<td>2.92</td>
</tr>
</tbody>
</table>

![Table (4)]

<table>
<thead>
<tr>
<th>Size</th>
<th>Pthreads</th>
<th>OpenMP</th>
<th>Difference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.17</td>
<td>0.17</td>
<td>0.0037</td>
<td>2.16</td>
</tr>
<tr>
<td>2048</td>
<td>1.15</td>
<td>1.12</td>
<td>-0.0286</td>
<td>-2.47</td>
</tr>
<tr>
<td>4096</td>
<td>7.37</td>
<td>7.37</td>
<td>0.0069</td>
<td>0.093</td>
</tr>
<tr>
<td>8192</td>
<td>55.47</td>
<td>58.13</td>
<td>2.6632</td>
<td>4.800</td>
</tr>
</tbody>
</table>
Conclusions and future work

As it can be observed, the difference in the times obtained by both implementations is not significant—it is never greater than 10%—but in most of the cases for this problem, Pthreads yields slightly better results than OpenMP.

According to a report published by Intel, OpenMP has an initial overhead in its primitives as in the case of #pragma omp parallel for. This overhead will be significant or not depending on how much it represents, percentage-wise, in relation to the total execution time of the algorithm. It should be noted, however, that this overhead is initial and is not related to the operations executed later on [11].

In algorithms that efficiently exploit optimized operations of OpenMP, the difference between running the same algorithm with Pthreads or with OpenMP will in principle be more significant in favor of the latter [12].

However, from a programming effort standpoint, OpenMP markedly simplifies algorithm implementation thanks to the directives it provides. It allows parallelizing a sequential solution in only a few steps, given its high abstraction level. In fact, a solution that has been parallelized with OpenMP allows running the algorithm as if it were the original sequential solution just by disabling the library. The abstraction level provided by OpenMP greatly facilitates the learning and use of the library. Also, the programming directives themselves document the parallelization of the application.

As future line of work, we will work in adapting the algorithms to be run on a heterogeneous architecture, both from the point of view of processor speed and memory hierarchy. Thus, the algorithms will have to respond to new challenges such as load balancing and dynamic work distribution, among others.

8 References


