Appendix B: Sequential Processing Performance of Computers

This appendix is dedicated to the description of computer performance obtained in the application field of linear algebra problems, especially those that can be expressed in function of matrix multiplications.

Computer performance characterization is a very well known and studied problem for several reasons, and is essentially used to estimate the type of problems that can be solved in terms of size and running times.

Although it is important to characterize computer performance, this Appendix also describes how code optimization affects performance. In addition, this appendix shows how this optimization has varying effects depending on each computer's hardware characteristics.

In the specific case of parallel applications, it is essential to know the precise sequential performance in order to characterize, with the same precision, the gain obtained by parallel processing. Similarly, it is possible to estimate - or at least we count with the minimum and necessary data to estimate - the cost-benefit relation of parallel computers, in general, and of computers networks processing in parallel, in particular.

B.1 Introduction

Computer performance characterization has been used for several purposes, among which we can mention [7]:

- Estimation of problem solution capacity, both in reference to the *size* of problems to be solved and the necessary *running times*.
- Verification of computers. costs, not only in terms of hardware but also of the necessary base and application software.
- Choice of the most adequate computer for the problem or type of problem to be solved. In this case the performance rate is implicitly used as a comparing parameter of the computers to be used.

Traditionally, the numerical computing capacity of a computer has been characterized by the quantity of floating point operations per time unit (Mflop/s: millions of floating point operations per second) or by a number identifying it univocally (SPEC: [6] [15]). Two general lines have also been traditionally adopted for the computation of this performance rate:

1. Processing hardware analysis: floating point unit/s, design of floating point unit/s (pipelines, internal registers, etc.), cache memory/ies (levels, sizes, etc.), main memory capacity, etc.
2. Execution of a program or a set of specific computing programs called *benchmarks*.

Processing hardware analysis generally gives rise to what is known as *peak performance*, or maximum *theoretical* performance of the computer. This characterization line of performance has normally been adopted by computer manufactures, and the fact that it is rather unlikely to be obtained by a specific application is already accepted.

Benchmarks use has become daily due to the existing gap between the peak performance and the real performance normally obtained by applications when running in computers. It is really difficult to choose a set of programs that meat the characteristics of, or represents, *the complete* range of possible programs that can be run over a computer. Consequently, there exist many used benchmarks and there are even more that are proposed.

When the type of specific applications over which computers are to be used is defined, characterization in this application field without using the most general benchmarks is still really useful. This is the case of those applications defined in terms of matrix multiplications [2] [4], and thus the very matrix multiplication performance is what can more accurately be obtained within this field and what is going to be taken as reference *benchmark* for experimentation.

Using such a specific benchmark - and so closer to an application to be solved - has, in the context of parallel applications, one more advantage: it accurately defines computers relative speed for local processing. Even though this index (computing relative speed) is not so necessary or important in the context of parallel computers with homogeneous processing elements, it is essential for parallel computing with heterogeneous processing elements. Without this type of information, it is really difficult to reach a computational load balance.

B.2 Computers Used

Choosing computers to characterize sequential performance, and trying to represent with this characterization *all* the possible cases, is almost as difficult as choosing the set of programs that allow comparing computer performance. In the case of parallel computing over installed local networks, it is clear that the computers interconnected in a local network must be analyzed, though the variety of possibilities is really wide.

The three local networks which we will work on are those described in detail in Appendix A:

- CeTAD: Center for Analog-Digital Techniques (*Centro de Técnicas Analógico-Digitales*), Electrontechnics Department, Faculty of Engineering, National University of La Plata. It is the eldest network installed, and its computers are used for several purposes.
- LQT: Theoretical Chemistry Laboratory (*Laboratorio de Química Teórica*), CEQUINOR, Chemistry Department, Faculty of Exact Sciences, National University of La Plata. This network - installed several years ago - aims at solving numerical problems; sequential and parallel works developed with PVM and Linda are run.
- LIDI: belongs to the Laboratory of Research and Development in Computer Sciences (*Laboratorio de Investigación y Desarrollo en Informática*), Faculty of Computer Sciences, National University of La Plata. It is dedicated to the teaching of parallel programming and research. It can be directly considered as of Beowulf-type, although not belonging to the most expensive in terms of quantity of machines and interconnection network.

Both the CeTAD network and LQT have several interesting characteristics for the study of performance with the aim of using them for parallel computing:

- Preexistence: they were not built to be studied but to be used. In this sense, they are *real* and have evolved as local networks do with the updating and addition of computers.
- Heterogeneity: it can be considered as a consequence of its preexistence to the present research work, but it is important to notice that both computer networks have quite different machines at least in terms of computing speed and, in the case of the CeTAD network, even at the level of computers's architecture.
- They are meant for sequential computing: even though parallel programs are run in both networks, at least when installed, machines were dedicated to sequential computing. In fact, several of the computers of the CeTAD local network exist for strictly sequential tasks and have been adapted for parallel computation by installing the necessary software.
- Standard and low cost hardware: both computers and interconnection networks are widely known and have low costs. In fact, more than one computer used can be considered as ready to be dropped out.

LIDI network provides a standard research framework of parallel processing in computers networks. More specifically: the computing and communication hardware is homogeneous and the interconnection network can be considered as the proper one since it coincides with the basic ideas of a Beowulf installation.

In Appendix A, the three local networks are described with the most outstanding characteristics of each computer. On the other hand, it is important (and in a way, it is one of the most important objectives of this Appendix) to identify the general performance characteristics in computers networks beyond particular details of each machine. Consequently, we will try to come up to more general performance conclusions about heterogeneous computing hardware.

B.3 Description of Experiments

Three large groups of experiments were carried out, depending on the way of codifying-designing- implementing the solution to the problem:

1. Non-optimized code. Matrix multiplication is codified with the three classical iterations, without any effort neither on the part of the implementation nor the compiler.
2. Code optimized by compiler. In this case, the code is compiled without optimizing (the three iterations) with all the available optimization options of the compiler used for the computer processor.
3. Optimized code at the source program level. In this case, the code optimization effort is carried out at the source code level (without recurring to the machine code); besides, the compiler's optimization options can be optionally used, though this is considered as unnecessary.

For each type of optimization, the performance obtained is computed for several sizes of possible matrices. The variation of the problem size to be solved (in terms of memory and computing requirements) is usual in this type of *benchmarks* and aims at identifying possible dependences of a computer.s sustained performance on the application size.

In the following subsections, we describe:

- The most important characteristics together with the objective/s of each of the optimization types, always within the context of sequential processing and that of numerical problems, in general, and of matrix processing, in particular.
- The sizes of matrices chosen to estimate computer performance, and the criteria by which they are chosen.
- The most important characteristics of the experimentation in each computer, which in a certain way is a view of a higher level of abstraction for the two previous points.

B.3.1 Code without Optimization

The performance obtained with the matrix multiplication without optimizing is not, in fact, of great use as performance index since, without any effort on the part of the programmer, we can obtain something better properly using the available compiler's options. In this case, it is necessary to know well only the compiler and it is very useful to know the characteristics of the processor.

The performance obtained with the non-optimized code will be used with two objectives:
1. As gain reference in each computer for the other two cases of optimization: of the compiler and of the source code.
2. As reference to compare the optimization effects in heterogeneous computers.

B.3.2 Compiler Optimization

In the case of the compiler optimizations, there are no disadvantages at the conceptual level. As already explained, we should know the compiler and the processor characteristics related to the arithmetical operations and/or design of the cache memory/ies. However, it is always convenient to take into account the fact that the best compiler for each computer is usually that provided by the manufacturer. According to the evolution in the development and commercialization of computers, the compiler was initially provided by the manufacturer company without additional costs. From many years now, the compiler is optional and the license must separately be bought from the manufacturer (of the executable code and the owned libraries). This situation has now at least two consequences:
1. Many of the installed computers do not have the (most appropriate) compiler of the computer.s manufacturer company.
2. Free-use compilers. utilization has become even more popular; e.g., gcc/gccegcs for C language, which can be obtained via Internet in binary and / or source code.

From the beginning it has been established that computers are meant to be used such as they are installed and, in any case, with the minimum software (particularly, compilers) installation additional cost. In consequence, not only it is advisable to use computers with free-use compiler, such as gcc/gcc-egcs, but it is also necessary to quantity its efficiency in each machine. The fact of characterizing computer performance with a compiler in particular is also characterizing, in a way, the compiler's performance in itself. Beyond particular conclusions arrived at with this respect, which are not in principle a topic of research of this thesis, there are other considerations to take into account:
- Generally, the compiler installed in each machine will be that used. It is assumed that it is the *best* or, at least, that with minimum cost.
- The used computer-compiler combination may not be the best and, thus, it is useful for characterizing the computer in particular but it is not for comparing machines in general (e.g., at the level of computers models).

According to the latter consideration, the norm of making reference to computers by their names (*hostnames*), without any reference to trademarks and/or models, is adopted in all the performance characterization graphics (in this Appendix and all of those of the experimentation). Even though the names of the machines do not count with any information (*a priori*) related to the performance - this can obscure the interpretation of the results -, it is not proper to characterize the performance at the level of trademarks . models of computers when the compiler used is not the *best* (assuming that the *best* is that provided by the machine's manufacturer) that can be used in each computer. On the other hand, Appendix A gives *all* the details of each computer, and in the interpretation of the results we will mention explicitly each of those considered as important.

B.3.3 Source Code Optimization

It is very difficult to assure *a priori* that the source code optimization made is effective. However, in the field of numerical computing in general, there exist many experiences (and publications) in relation to the ways of taking the utmost advantage of the processing hardware characteristics. Some of the optimizations already considered as standard at the level of source code are [3] [1] [10] [11] [5]:

- Intensive use of internal registers dedicated to numerical data.
- Block processing in order to take advantage of the data assigned in cache memory/ies.
- Maximum use of the running units implemented with pipelines (identification of hops and dependencies).
- Striping of command that operates with floating point data and integer data, for the maximum possible use of superscalar processors.
- Identification and *isolation* of operations with data dependencies that reduce superscalar processor performance.

In the particular case of matrices multiplication, there are already available several optimization sources and several information sources, even at the level of source code available in Internet. On the one hand, we can even expect that each processor will have its own set of numerical computing routines, such as the particular case of PentiumIII and Pentium 4 of Intel, which set out several available [9] routines in source code (though in assembler language, at the level of processor.s commands), which can be obtained via Internet [13].

Even though libraries optimized by the processors manufacturer companies themselves can be really attractive (and of free use), two drawbacks are still present:

- Not all the processors have such type of code-library available.
- The source code usually depends on a compiler in particular; in the previous case of Intel, owner compilers are necessary (whose utilization depends on buying a license) both of C language and assembler language.

On the other hand, there are many libraries already available in Internet which generate optimized code in general, normally in C language or FORTRAN in order to be compiled locally in computers independently of the installed compiler [3]. One of the most significant projects, and under development, is that called ATLAS (Automatically Tuned Linear Algebra Software) [12] [14]. Like most of the packages in the field of linear algebra. ATLAS was initially and exclusively dedicated to matrix (sequential) multiplication and, subsequently, was extended to all Level 3 LAPACK [4]. This alternative is satisfactory for several reasons:

- It is of free use, the only cost is that of the installation. This normally implies installing code obtained in Internet and executed by a set of programs (scripts) in order to identify the processing hardware and optimize the code according to what is estimated by these programs. Finally, a set of libraries containing the optimized routines and that can be used from the programs are available.

- Optimization concepts are clear enough to be implemented independently of a library or distribution in particular. This means that it is not even necessary to use a library available in Internet but an appropriate code can be developed with its further cost.
- It is quite independent of the processor, since it can be done in source code, such as C or FORTRAN, and it can thus be locally compiled in each computer.

Since the code is optimized without making a specific use of a compiler, the optimization options of the compiler do not usually improve the running code. Once more, it is necessary to recall that these optimizations are appropriate for this type of matrix processing and cannot necessarily be made in all the cases or for all the applications in general.

The source code optimization will also be referred to as *complete optimization*, since it is generally the best that can be done in order to obtain optimal or near-optimal performance of each computer.

B.3.4 Sizes of Matrices to be Multiplied

The sizes chosen of the matrices to be multiplied (square matrices are assumed, of *nxn* elements) are related to the characteristics of the matrix processing and of each computer.s memory subsystem. As example: if matrices are small enough to be completely assigned in the first level of cache memory (L1 Cache), the sustained performance will be really satisfactory and with values near the processor.s theoretical optimum. If, on the contrary, matrices cannot be assigned in any of the levels of cache memory, the performance will depend almost directly on the data access pattern to be processed.

Given the heterogeneity of computers with which we are experimenting (Appendix A) and, in particular, the different sizes and levels of the computers. cache memory, several relatively small sizes of matrices with respect to the size of the main memory were taken as reference: matrices of order n = 100, 200, 400. Numerical data are represented as floating-point numbers of simple precision (norm IEEE 754 [8]) of four bytes. In consequence, for n = 100, the quantity of necessary data to feed a matrix will be of 100^2x4 bytes, slightly less than 40 KB of data.

Two values were taken as representative of the matrix sizes that can be handled in a 32 MB main memory: matrices of order n = 800 and of order n = 1600. With matrices of 800x800 elements, the quantity of required memory to contain the three matrices intervening in a multiplication (C = A x B) is of approximately 7.3 MB (approximately, 22.8% of the total of 32MB of main memory). In the case of matrices of 1600x1600 elements, the approximate quantity of memory that is required is 29.3 MB, which represents the 91.6% of the total of 32 MB of main memory.

In the case of computers with 32 MB of main memory, the size of problem with n = 1600 can be considered as enough, or at least n = 1600 is big enough so that the cache memory size will not contain a relatively big part of the problem. However, taking into account that there are machines with 512 MB of main memory, values of *n big enough* were looked for in order to fill the main memory completely.

In consequence, in all the computers, the experiments were carried out with square matrices of order *n* = 100, 200, 400, 800 and 1600. In the case of the computers with main memory of 64 MB or 512 MB, and in order to have reference values to be used in the speedup computation, experiments with bigger matrices were carried out. In addition, since there always exists the tendency to use computers to the maximum of their capacities, experiments with the maximum possible in terms of matrices sizes were also carried out. As expected, this depends not only on the size of the main memory installed but also on the swap space set up in the system.

For computers of 64 MB of main memory, the sizes considered as representatives of the problems that require a good part of or all the main memory correspond to the values of matrix orders *n* = 1900, 2000, 2200 and 2400. These sizes of matrices imply the approximate percentages of memory requirements (assuming 64 MB in the whole) of: 65%, 72%, 87%, and 103%, respectively. Recall that it is possible to experiment with the values near and higher than the 100% of main memory requirements of the set up swap memory size.

In computers of 64 MB of main memory, the maximum size with which the matrix multiplication could be carried out is for *n* = 3200, and, as reference, experiments with *n* = 3000 were also carried out. These matrices sizes imply the approximated percentages of memory requirements (assuming 64 MB on the whole) of 183 % and 161%, respectively. As previously mentioned, the maximum sizes of the problem depend on three aspects:

- installed main memory
- swap memory set up
- operating system, since it is the one which, as a last resort, decides when to cancel a process due to lack of memory.

And these three aspects coincide, at least, in the fastest machines of the CeTAD and LIDI networks.

For computers of 512 MB of main memory, the sizes considered as representatives of the problems requiring a good part or all of the main memory correspond to the values of matrix orders *n* = 4000, 5000, 6000 and 7000. These sizes of matrices imply approximated percentages of memory requirements (assuming a 412 MB on the whole) of: 36%, 56%, 80%, and 110% respectively. Recall that it is possible to experiment with the values near and higher than the 100% of main memory use in order to store data depending on the memory size set up swap.

In computers of 512 MB of main memory, the maximum size with which the matrix multiplication could be carried out is for *n* = 9000, and, as reference, experiments with order *n* = 8000 were also carried out. These matrices sizes imply the approximated percentages of memory requirements (assuming 512 MB on the whole) of 181% and 143% respectively.

In brief, in the fastest computers of each local network, we have experimented to the limits of the total memory capacity. In all the cases, when we show each computer performance, we also show the maximum matrices size that can be solved without *swapping* memory pages.

B.4 Matrix Multiplication Performance

The following subsections show the running times and also the performance expressed in Mflop/s obtained in each of the computers which solves a matrix multiplication according to the type of optimization used. When necessary, some comments explaining the values of the computer performance indexes are also included. Since the eight computers of the LIDI local network are equal, all the data of the experimentation are shown only for one of them. Finally, another subsection is also included in order to compare performance among the different types of optimization chosen.

B.4.1 Performance without optimization

Fig. B.1 shows the running times (in seconds) obtained for the (square) matrix multiplication of different sizes in the CeTAD computers.

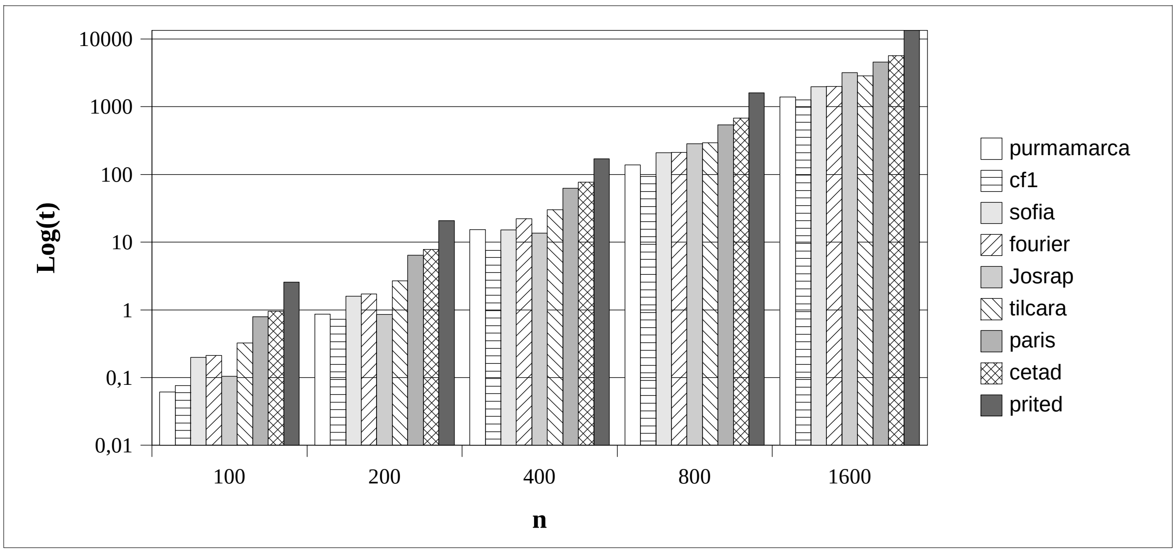

Figure B.1: Running Times in CeTAD without Optimization.

Fig. B.2 shows the running time in the LQT computers. The axis **x** of the graphic shows the value of *n* (matrices size) and the axis **y** shows the running time in logarithmical values.

Even tough Fig. B.1 and Fig. B.2 allow us to have an approximated idea of the necessary running times in each of the computers, the logarithmical scale and the very performance index defined as the running time may complicate the results interpretation. All the same, we can easily identify that for a defined matrix size, the computing speed differences are noticeable.

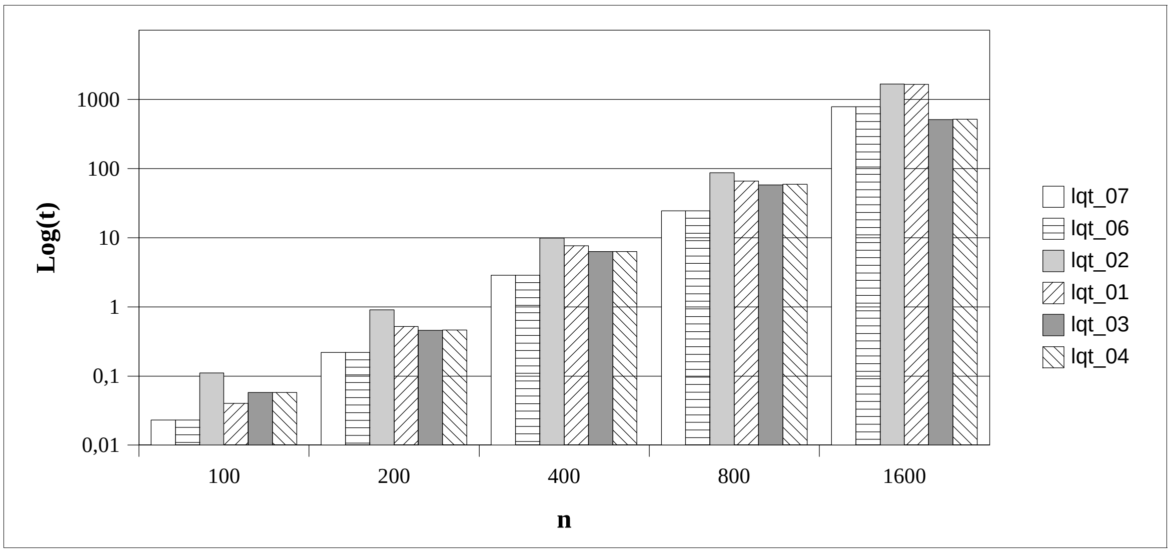

Figure B.2: Running Times in LQT without Optimization.

Fig. B.3 as well as Fig. B.4 show the same experiments though identifying the Mflop/s obtained in the computer. What can be easily identified are the differences in each machine computing speeds, which are quite masked in the case of the time graphics with logarithmical scale.

In all the computers, it is worth to note the effect the cache memory size and the problem size to be solved have on the performance. In all the cases, as the quantity of data increases, the likelihood of reusing a datum assigned in cache memory decreases since no access pattern to data is established a priori so as to take advantage of the memory hierarchy with one or more levels of cache memory.

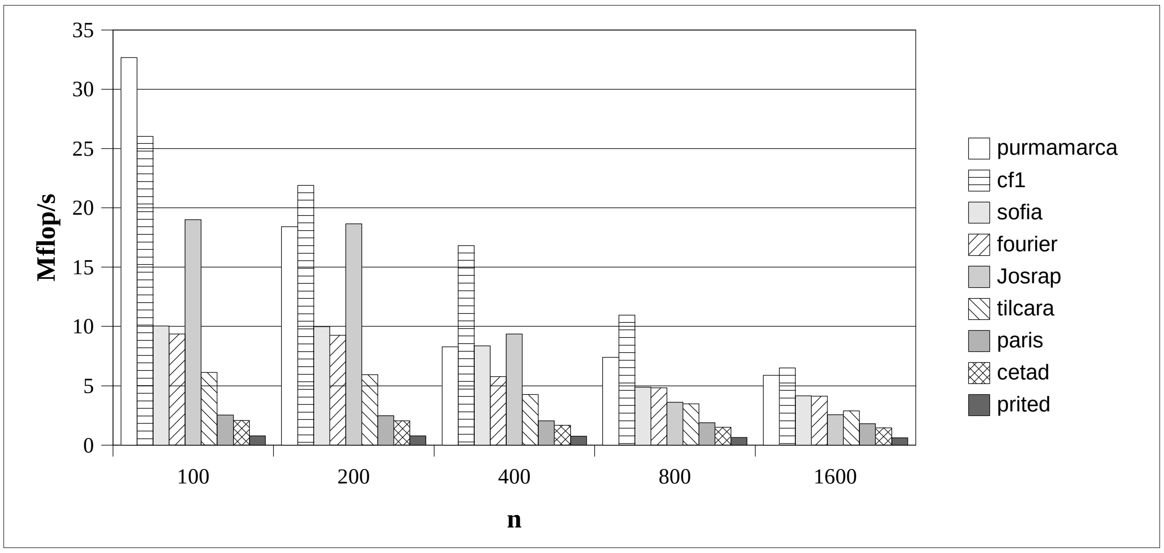

Figure B.3: Mflop/s in CeTAD without Optimization.

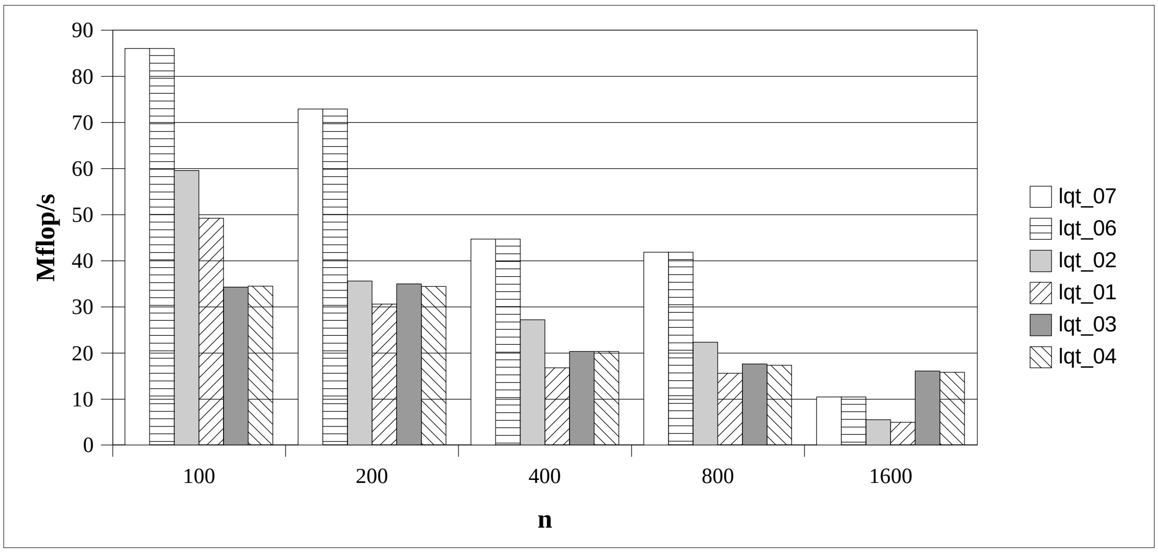

Figure B.4: Mflop/s in LQT without Optimization.

It is interesting to notice that, as processors become much faster, the impact on the decrease of performance when the problem increases is greater, since the relative memory and processing speed difference increases proportionally. For instance, **lqt_01** passes from almost 50 Mflop/s with matrices of 100x100 to 5 Mflop/S with matrices of 1600x1600, which implies using only a tenth part of the possible performance, such like it was measured for the problems with matrices of order *n* = 100. The differences of relative speed between access to memory and processing generate, in turn, relative speed differences among those computers that depend on the size of problem to be solved. For instance, for *n* = 100, the computer called **paris** (Fig. B.3) has almost twice the computing power of **Josrap**, and for *n* = 1600, **Josrap** has a better performance than **paris**. Even though for sequential computing this gives a different idea of relative speed for different sizes of problems, in the context of parallel computing, it has a direct impact on the computational load balance, which should be solved by the application in function of the data processing problem size.

B.4.2 Performance with Compiler Optimization

As previously explained, generally application programs have the minimum optimization degree provided by the compiler for the computer (more specifically, for the processor). In all the reported machines, the compiler used is *gcc/gcc-egcs* and, in consequence, the options do not vary significantly in the different machines. However, we should take into account that, in a heterogeneous environment, the variety of compilers can be equal to the quantity of machines that are being used and, thus, knowing the compilers. details (processors optimizations) is equally complex.

Fig. B.5 and Fig. B.6 show the running times of each of the machines of CeTAD and LQT respectively. Like in the previous figures in which the running times can be observed, the axis **x** of the graphic corresponds to the different sizes of matrices (from matrices of order *n* = 100, to matrices of order *n* = 1600), and the axis **y** of the graphic shows the running time with logarithmical scale.

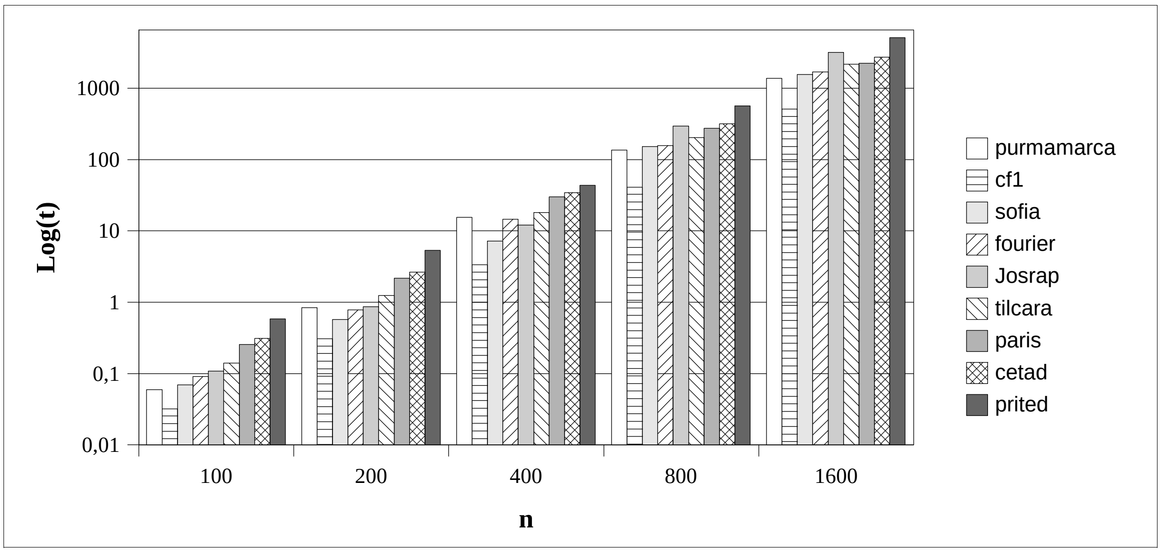

Figure B.5: Running Times in CeTAD with Compiler's Optimization.

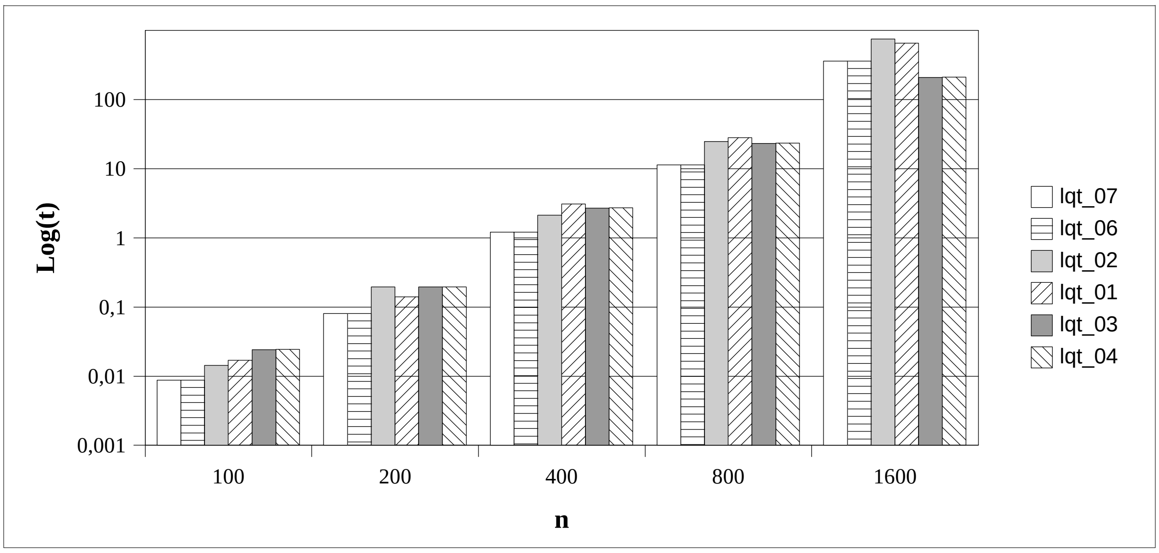

Figure B.6: Running Times in LQT with Compiler's Optimization.

Comparing Fig. B.5 with Fig. B.1, it can be noticed that, in many computers, the total running time decreases for all the sizes of matrices with which the experiments were carried out. Similarly, Fig. B.6 can be compared to Fig. B.2. Once more, it is hard to identify with precision the relative speed differences due to the logarithmical scale shown by the running times.

Fig. B.7 and Fig. B.8 show the performance of each computer in Mflop/s, where the differences in terms of running without any optimization can be better identified. Depending on the computer and the size of the problem that is being solved, the performance improves in some cases more than the 100%: **lqt_01** passes from almost less than 50 Mflop/s for *n* = 100 (Fig. B.4) to less than 120 Mflop/s for the same size of problem (Fig. B.8).

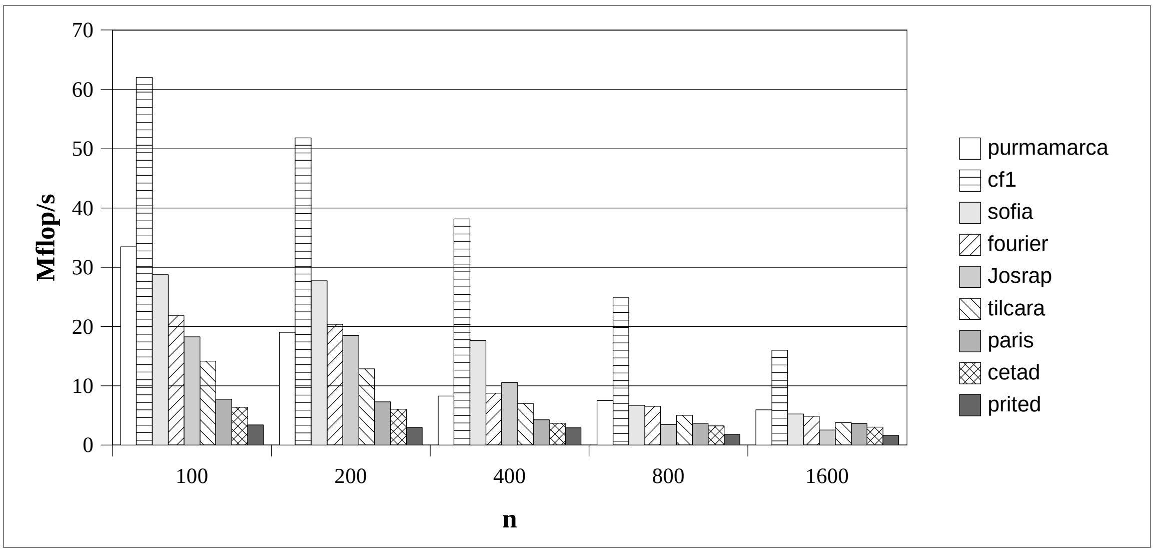

Figure B.7: Mflop/s in CeTAD with Compiler.s Optimization.

Even though the performance notably improves in some cases, such as in the previously mentioned, and generally improves in all the computers, the weight of the problem size in the performance obtained is still important. Similarly, the performance decrease of machines as the problem size increases has different characteristics in each of them and, thus, the differences in the machine.s relative speed are still dependent on the size of problem to be solved.

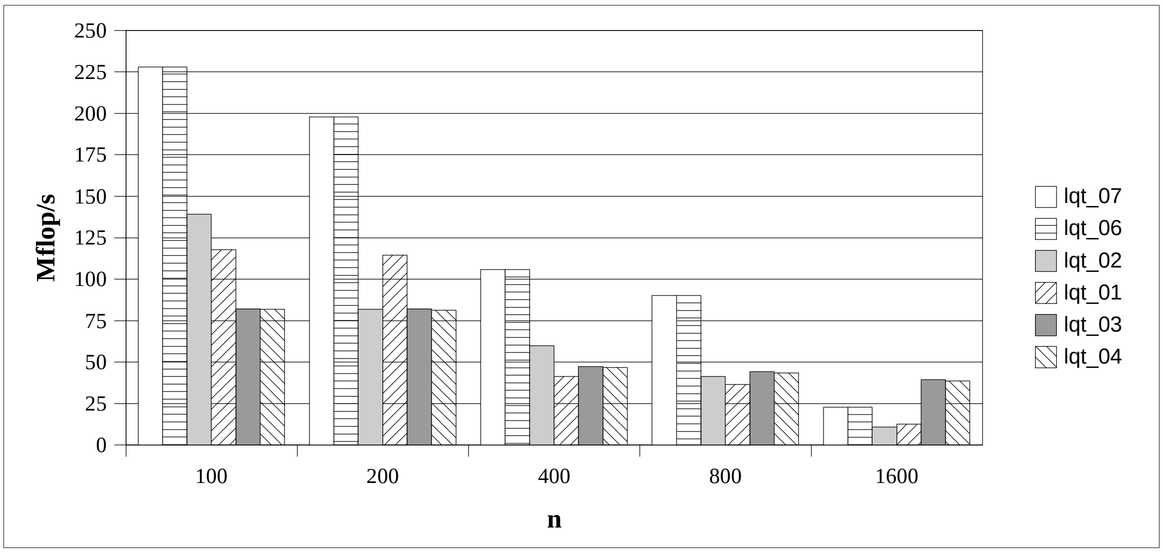

Figure B.8: Mflop/s in LQT con with Compiler.s Optimization.

Since it is very difficult that a compiler could perform all the possible optimizations [12], it is very important to count with a code specially optimized for the computers that are being used. Even more beneficial is to take advantage of the available code specially optimized for the computers. The following subsection shows the experiments carried out with this type of code and the very important differences at the level of performance increase and at the conceptual level of performance, in general, of machines running processing code in order to carry out linear algebra operations.

B.4.3 Performance with Source Code Optimization

Fig. B.9 and Fig. B.10 show the running times of the CeTAD and LQT computers, respectively, for each of the matrix sizes when the source code is optimized to obtain the best possible performance.

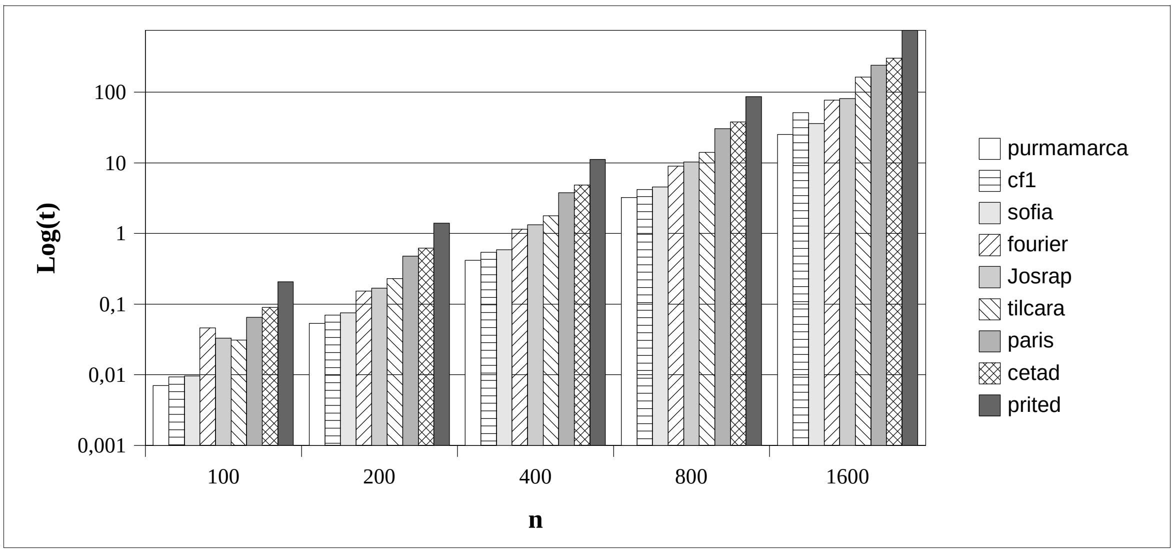

Figure B.9: Running Times in CeTAD with Fully Optimized Code.

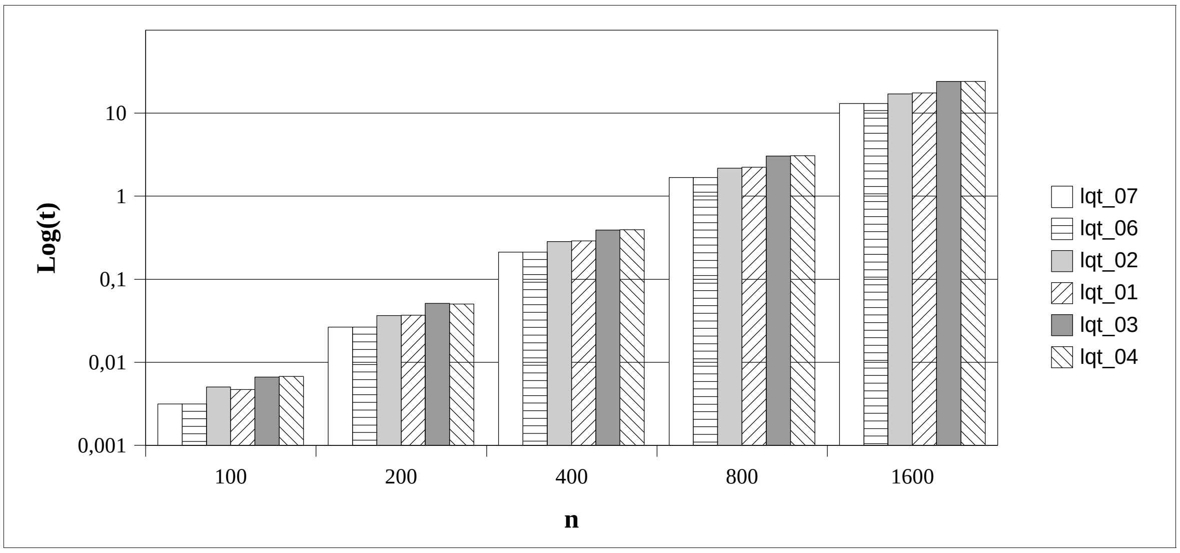

Figure B.10: Running Times in LQT with Fully Optimized Code.

Comparing Fig. B.9 with Fig. B.5, and Fig. B.10 with Fig. B.6, it is possible to verify a really important and general decrease in the total running time (in all the computers and for all the sizes of problems) when using source code optimization. As an example, Fig. B.5 shows that the computer **purmamarca** (in the CeTAD LAN) uses approximately one second to solve a 200x200 element matrix multiplication. Fig. B.9 shows that the same problem in the same computer is solved in less than a tenth of a second. On the other hand, Fig. B.9 shows that the computers of LQT **lqt_06** and **lqt_07** use much more than a hundred of seconds to solve a 1600x1600 element matrix multiplication. Fig. B.10 shows that the same problem in the same machines is solved in slightly more than ten seconds.

Each machine's performance expressed in Mflop/s is shown in Fig. B.11 for the machines of the CeTAD and in Fig. B.12 for the machines of LQT.

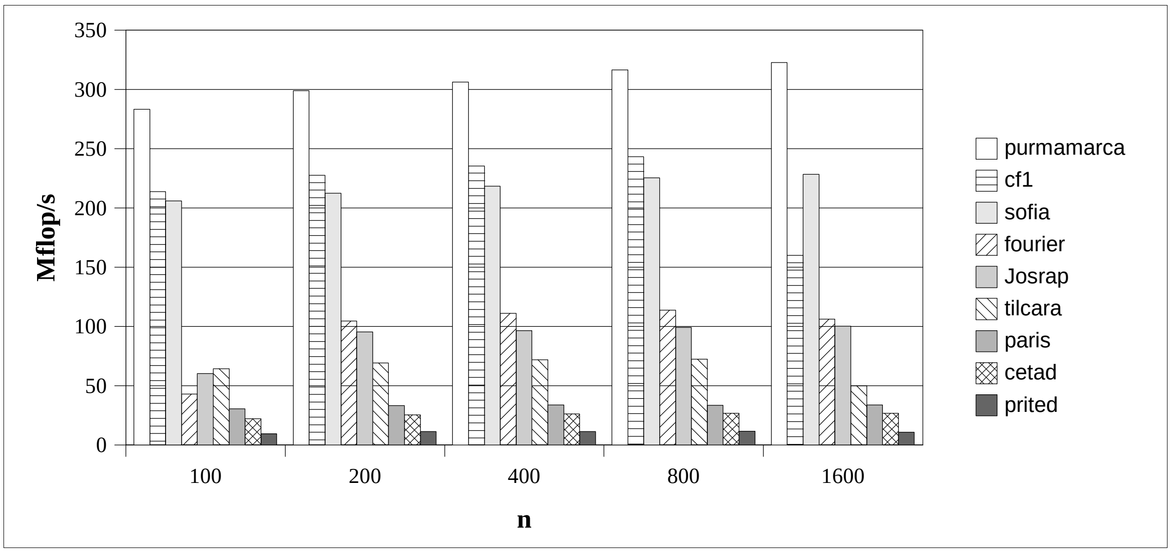

Figure B.11: Mflop/s in CeTAD with Fully Optimized Code.

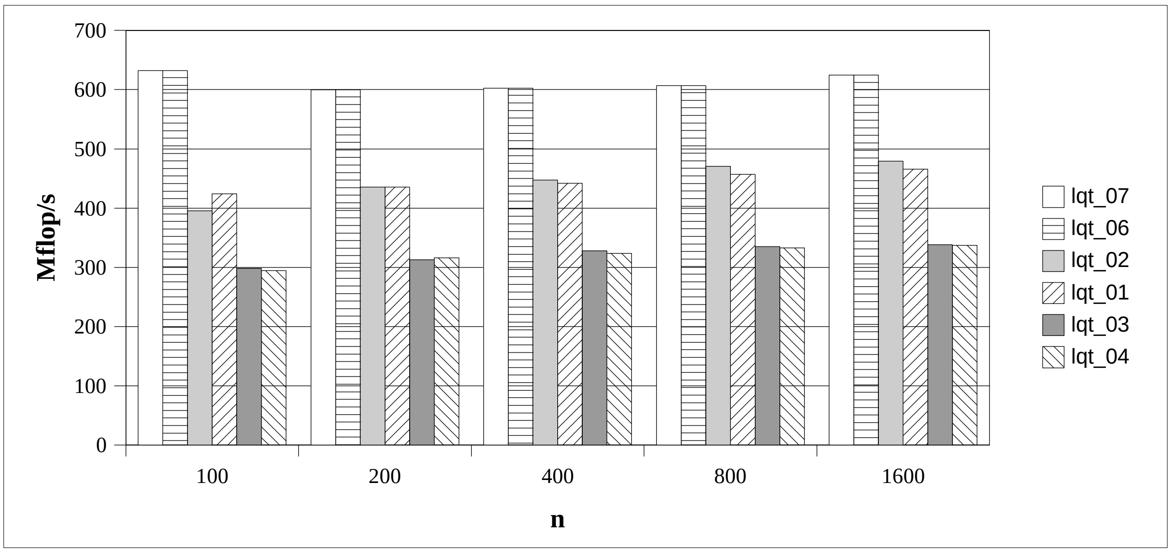

Figure B.12: Mflop/s in LQT with Fully Optimized Code.

If the performance values of the machines of the CeTAD of Fig. B.11 (fully optimized code) are compared to performance values of Fig. B.7 (compiler-optimized code) and with those of Fig. B.3 (non-optimized code), the performance differences are really noticeable. Not only these differences can be verified in terms of absolute values -which is similar to what happens with the running time-, but also in terms of performance variations depending on the size of problem that is solved and the relation between each machine's computing relative power.

From these two last figures, the characteristics that are almost as important as the increase of the performance absolute values of all the machines can be identified:

Performance does not decrease as the size of problem increases: the two figures show it clearly for almost all the computers. Despite the high heterogeneity, the performance follows this "rule" regardless processors or clock cycles. Exceptions that can be identified appear in Fig. B.11, and are the computers **tilcara** and **cf1**. In both cases, the performance degrades noticeably for the value of n = 1600. In the case of **tilcara**, it passes from slightly more than 70 Mflop/s for n = 800 to 50 Mflop/s (decrease of almost the 30% of the performance) for n = 1600, and **cf1** passes from slightly more than 240 Mflop/s to approximately 160 Mflop/s (decrease of almost 35%) for the sizes n = 800 and n = 1600, respectively. In both cases, this is due to the relation between the size of the main memory and the size of the problem. For n = 800, the amount of required memory for the matrices is slightly more than 7 MB and, thus, there is enough space in all the machines; whereas for n = 1600 the amount of required memory is more than 29 MB. Both in **tilcara** and in **cf1** (**cetadfomec1**) something similar also happens: the operating system activity report shows that they have to recur to the swap memory space, even though their main memory is of 32 MB and, in principle, they can contain all the processed data. It should be recalled that the main memory must contain the operating system kernel (Linux), parts of the memory that are not possible to send to swap space (operating system buffers, for instance), and a minimum code of the very application. In addition, since the computing power of the **cf1** processor (Celeron 300 MHz) is more than three times the power of **tilcara** (Pentium 133 MHz), it is expected that the impact on the performance decrease will be greater.

Performance is almost constant or slightly increases when the problem is bigger. It can be proved in both figures with the previously mentioned exceptions, for each of the computers (once more, independently of the processors and the machines. heterogeneity). The basis for following this rule is the block processing carried out over data. Each time a datum is assigned in cache memory, it is reused to the maximum, since the data access pattern (in memory) is specially codified with this end. This implies that increasing the amount of data does not increase the number of failures in cache memory and, what's more, increasing the amount of data in the case of matrix operations, and, in particular, of matrix multiplications, implies that it is possible to increase the number of accesses to cache memory with the same data. In this way the performance increase when the size of matrices increases can be explained, since the data assignment in cache is still an access to main memory but, once in cache memory, it can be reused more times (thus increasing the number of "hits" of cache memory) because each datum has more operations to carry out.

The relation between computers computing speeds is almost constant, without

variations higher than 10%. This characteristic can be identified as a consequence of the almost constant performance or of the slight increase of the performance previously explained, though within the context of parallel applications that are solved in heterogeneous computers, it is really important and it should be necessarily be analyzed. For instance, in Fig. B.12 it can be noticed that computer **lqt_02** has a computing power of approximately more than a third of that of lqt_03 in the 100x100 matrices processing, and both this and this speed relation are kept approximately constant independently of the matrices sizes that are processed.

From the point of view of parallel applications, this last characteristic of the experimentation implies a noticeable simplification in the way the computational load balance is solved. For instance, if a 10% computation load unbalance is accepted, the way the load is distributed can be implemented independently of the problem size that should be solved in each computer, and thus, a tuning parameter is eliminated or at least the variation range is remarkably diminished.

B.4.4 Bigger Matrices in Computers with the Highest Computing Power

Many of the intensive computing applications normally tend to take up a large quantity of memory (processing of large volumes of data). On the other hand, it is also expected that the applications will take advantage of all (or most of) the computers' resources, in particular the main available memory. This is why experiments were carried out in order to characterize machines performance when the problem to be solved takes up all the memory and even when the requirements are higher than the main installed memory and computers have to recur to the use of the configured swap space.

The machines chosen to carry out these experiments were those with highest computing power of each network, i.e. **purmamarca** (Pentium II 400 MHz, 64 MB) of CeTAD and **lqt_07** (Pentium III 1 GHz, 512 MB) of LQT. Experiments were designed with two purposes, using fully optimized code:

- Characterization of the performance for the biggest size of problem that can be contained by the main memory.
- Characterization of the performance for the biggest size of problem that can be solved (including swap memory space).

Fig. B.13 shows the performance of **purmamarca** for different matrix sizes, including those implying the use of swap memory during the processing of the matrix multiplication C = AxB. From matrices of order n = 1600 inclusive, the proportion of approximate main memory that is required to contain all the problem data is shown. For example, for matrices of order n = 1900, the 65% (approximately) of the main memory is used to contain matrix data.

Fig. B.13 also shows the highest value for n, for which all the data can reside in main memory (n = 2000) without recurring to swap memory during computations. That is, from n = 2200 the computing must recurring to swapping memory pages in order to solve the problem. In this way the performance decrease for values of *n* = 2200, 2400, 3000, and 3200 with respect to the performance obtained with *n* = 2000 is explained.

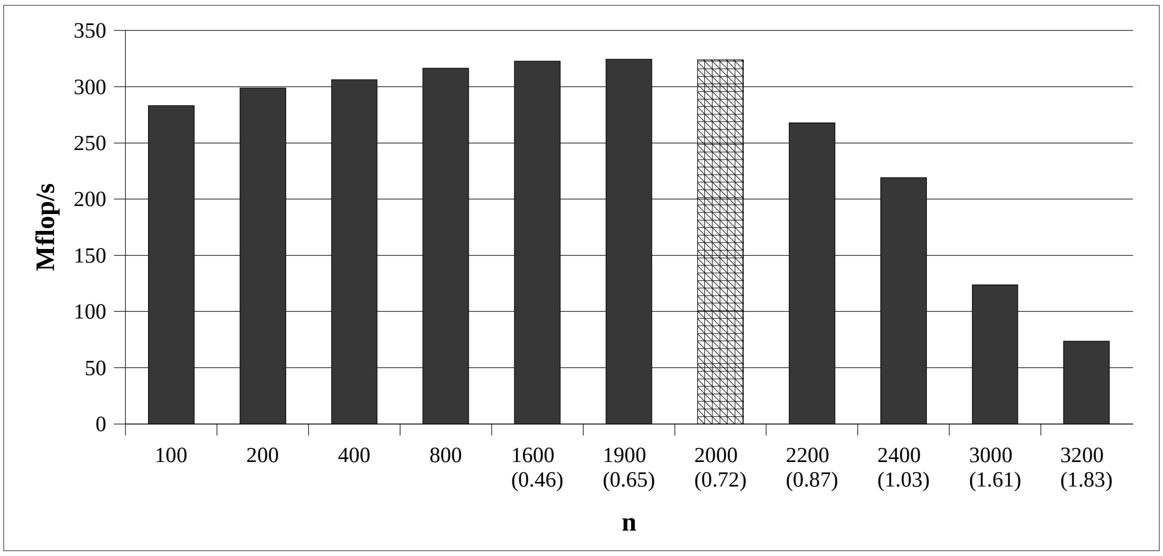

Figure B.13: Mflop/s in **purmamarca**.

Fig. B.14 shows the performance of **lqt_07** for different matrix sizes that are multiplied. It also shows the biggest size that can be completely contained in main memory (*n* = 5000) and, from *n* = 4000 inclusive, the approximate proportion of necessary main memory to contain all the problem data.

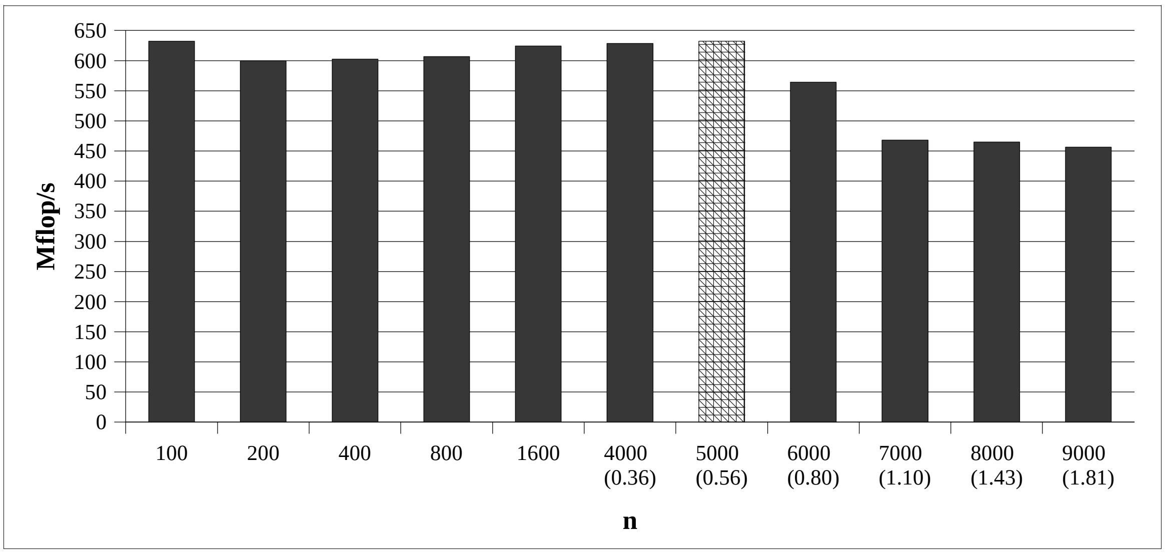

Figure B.14: Mflop/s in **lqt_07**.

In both computers the performance follows the same tendency: it increases with the increment of the problem data quantity while the main memory is capable of containing all the data to be processed, and it decreases as a higher proportion of swap space (in disk) is used. However, variations in **lqt_07** are lesser, both in relation to the increase and decrease of performance. The most significant fact might be that the performance does not decrease as much as in **purmamarca** from the use of swap memory during computations. Even though the speed of the disks used may be quite influencing, it is more likely that the two characteristics will be combined in order to keep the performance relatively high in **lqt_07** though swap memory space is used. Both

- the size of matrices, and
- the block processing

are combined in such a way that, on the one hand, the quantity of floating point operations is much higher because matrices are bigger and because the processing requirements are $O(n^3)$, and on the other, block processing makes each data in main memory to be used to the maximum and thus the page faults frequency implying the page swapping from memory to disk is reduced.

B.5 LIDI Computers Performance

Fig. B.15 shows LIDI (homogeneous) computer performance depending on the code optimization level for matrix multiplication of order *n* = 100, 200, 400, 800 and 1600. Code performance without any type of optimization is shown as "No opt.", code performance optimized by the compiler is referred to as "Comp." and the program performance optimized at source code level for matrices multiplication is shown as "Full".

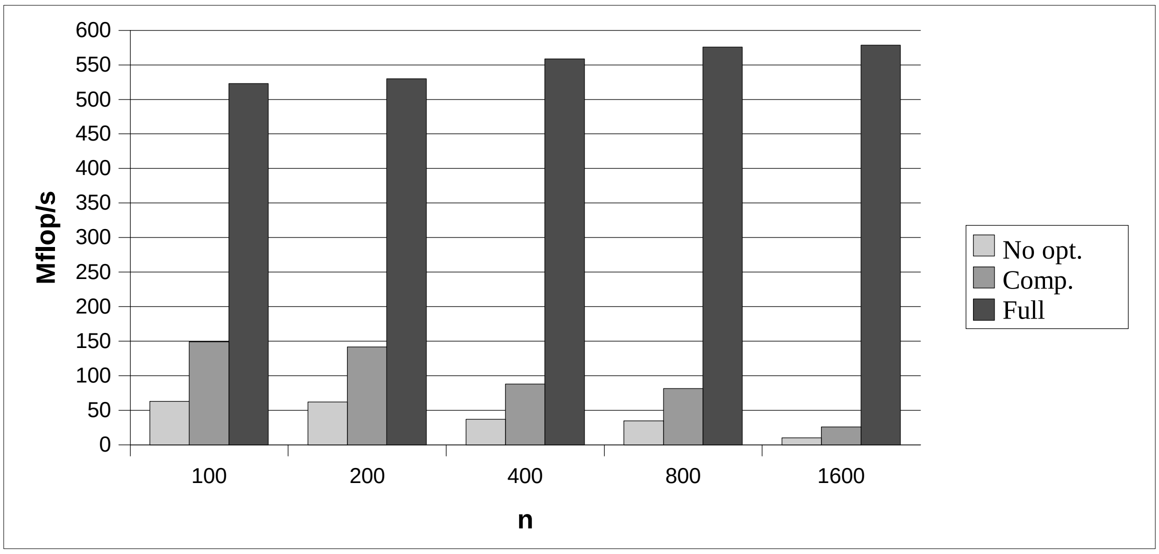

Figure B.15: Performance in Mflop/s Depending on Optimization in LIDI.

In fact, Fig. B.15 summarizes what happens in the rest of the computers of CeTAD and LQT, though with the very absolute value of LIDI computers. The worst performance is that corresponding to the code without any kind of optimization and, in the best of the cases, is of slightly more than 50 Mflop/s. From the point of view of performance, the code optimized by the compiler is approximately three times better than the non-optimized code. In other words, with the code optimized by the compiler, the same computer solves the same problem in a third of the time required with the non-optimized code. The performance with both the non-optimized code and the complier optimized depend on the problem size and is considerably reduced as the amount of data to be processed increases. The code fully optimized is even better in terms of performance than the compiler optimized and is quite independent of the size of problem, with variations that do not overpass the 10%. In addition, as the size of problem increases, the performance increases as well, all of which is a considerable advantage with respect to the code without optimization and the compiler optimized code.

Fig. B. 16 shows the performance with fully optimized code of LIDI computers for different sizes of matrices that are being multiplied. Like for the computers with higher computing power of CeTAD and LQT, the biggest size that can be completely contained in main memory (*n* = 2000) is shown as well as, from *n* = 1600 inclusive, the approximate proportion of main memory necessary to contain all the problem data.

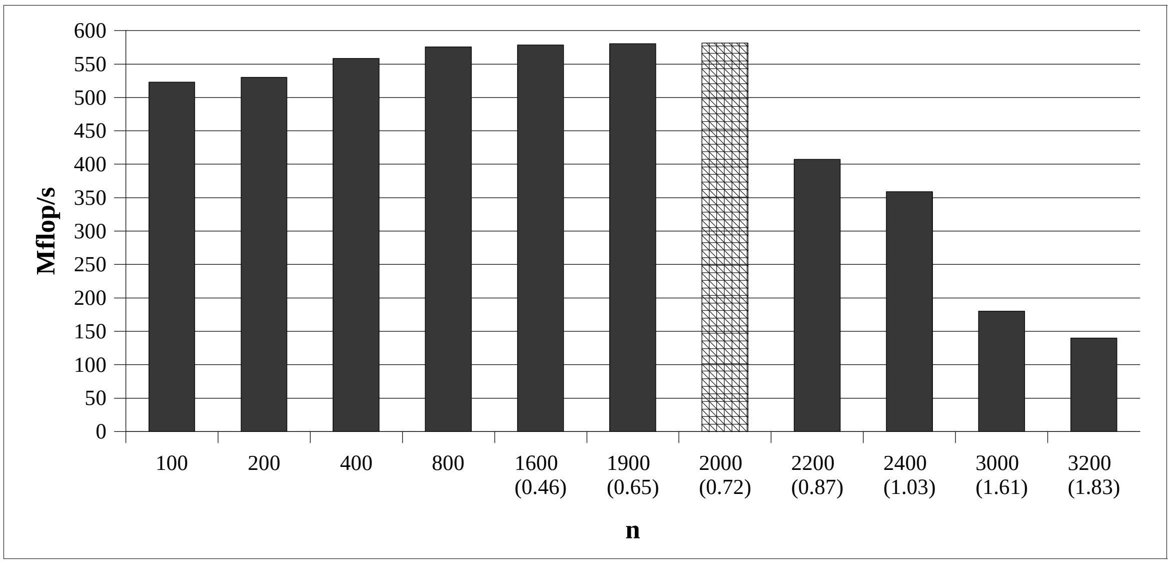

Figure B.16: Mflop/s in the Computers of LIDI.

Beyond the differences in absolute values, the performance variations of LIDI computers is similar to the variation of the machine with highest computing capacity of CeTAD, **purmamarca**, which Fig. B.13 shows. In fact, the large decrease of performance as swap memory is more frequently used is similar.

B.6 Conclusions

From the experimentations carried out, we do not only count with the precise performance values related to

- The resolution of the elemental operation of matrix multiplication in each computer, necessary to implement the computation load balance for parallel computing;
- The performance values to use individual performance metrics of each computer of each

local network, and also to be used in the *speedup* computation obtained with the resolution in parallel.

But also the code to be used in each computer should be completely optimized for at least two reasons:

1. Performance is several times better, which improves the total running time that is required significantly (proportionally). Sequential performance not only is crucial for the effective computers. use (to their maximum capacity) but also highly important for the correct computation of speedup values that are obtained with parallel processing.
2. The computers' speed relation is independent (or with slight variations) of the size of problem to be solved, which highly simplifies the way in which the computational load balance is solved.

References

[1] Alpern B., L. Carter, J. Ferrante, “Space-limited procedures: A methodology for portable high-performance”, International Working Conference on Massively Parallel Programming Models, 1995.

[2] Anderson E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide (Second Edition), SIAM Philadelphia, 1995.

[3] Bilmes J., K. Asanovi , C. Chin, J. Demmel, “Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology”, Proceedings of the International Conference on Supercomputing, Vienna, Austria, July 1997, ACM SIGARC.

[4] Dongarra J., J. Du Croz, I. Duff, S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 14 (1), March 1988.

[5] Golub G. H., C. F. Van Loan, Matrix Computation, Second Edition, The John Hopkins University Press, Baltimore, Maryland, 1989.

[6] Henning J. L., SPEC CPU2000: Measuring CPU Performance in the New Millenium, Computer, IEEE Computer Society, July 2000.

[7] Hennessy J. L., D. A. Patterson, Computer Architecture. A Quantitative Approach, Morgan Kaufmann, San Francisco, 1996.

[8] Institute of Electrical and Electronics Engineers, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1984, 1984.

[9] Intel Corporation, Streaming SIMD Extensions - Matrix Multiplication, AP-930, Order Number: 245045-001, June 1999.

[10] McCalpin J. D., M. Smotherman, “Automatic benchmark generation for cache optimization of matrix algorithms”, Proceedings of the 33rd Annual Southeast Conference, R. Geist and S. Junkins editors, Association for Computing Machinery, New York, March 1995.

[11] Saavedra R., W. Mao, D. Park, J. Chame, S. Moon, “The combined effectiveness of unimodular transformations, tiling, and software prefetching”, Proceeding of the 10th International Parallel Symposium, IEEE Computer Society, April, 1996.

[12] Whaley R., J. Dongarra, “Automatically Tuned Linear Algebra Software”, Proceedings of the SC98 Conference, Orlando, FL, IEEE Publications, November, 1998.

[13] http://developer.intel.com/design/pentiumiii/sml/245045.htm

[14] http://www.netlib.org/atlas

[15] http://www.spec.org