UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

Optimization of a parallel Monte Carlo method for linear algebra problems.

by

Diego Dávila

July 4, 2016

Master of Innovation and Research in Informatics
Specialization: High Performance Computing

Supervisor: Vassil Alexandrov
Barcelona Supercomputing Center

Tutor: Eduard Ayguadé
Computer Science Department

Contents

Abstract

Many problems in science and engineering can be represented by Systems of Linear Algebraic Equations (SLAEs). Numerical methods such as direct or iterative ones are used to solve these kind of systems. Depending on the size and other factors that characterize these systems they can be sometimes very difficult to solve even for iterative methods, requiring long time and large amounts of computational resources. In these cases a preconditioning approach should be applied.

Preconditioning is a technique used to transform a SLAE into a equivalent but simpler system which requires less time and effort to be solved. The matrix which performs such transformation is called the preconditioner [7]. There are preconditioners for both direct and iterative methods but they are more commonly used among the later ones.

In the general case a preconditioned system will require less effort to be solved than the original one. For example, when an iterative method is being used, less iterations will be required or each iteration will require less time, depending on the quality and the efficiency of the preconditioner.

There are different classes of preconditioners but we will focused only on those that are based on the SParse Approximate Inverse (SPAI) approach. These algorithms are based on the fact that the approximate inverse of a given SLAE matrix can be used to approximate its result or to reduce its complexity.

Monte Carlo methods are probabilistic methods, that use random numbers to either simulate a stochastic behaviour or to estimate the solution of a problem. They are good candidates for parallelization due to the fact that many independent samples are used to estimate the solution. These samples can be calculated in parallel, thereby speeding up the solution finding process [27].

In the past there has been a lot of research around the use of Monte Carlo methods to calculate SPAI preconditioners [1] [27] [10]. In this work we present the implementation of a SPAI preconditioner that is based on a

Monte Carlo method. This algorithm calculates the matrix inverse by sampling a random variable which approximates the *Neumann Series expansion*. Using the *Neumman series* it is possible to calculate the matrix inverse of a system A by performing consecutive additions of the powers of a matrix expressed by the series expansion of $(I - A)^{-1}$.

Given the stochastic approach of the Monte Carlo algorithm, the computational effort required to find an element of the inverse matrix is independent from the size of the matrix. This allows to target systems that, due to their size, can be prohibitive for common deterministic approaches [27].

Great part of this work is focused on the enhancement of this algorithm. First, the current errors of the implementation were fixed, making the algorithm able to target larger systems. Then multiple optimizations were applied at different stages of the implementation making a better use of the resources and improving the performance of the algorithm.

Four optimizations, with consistently improvements have been performed:

1. An inefficient implementation of the *realloc* function within the MPI library was provoking the application to rapidly run out of memory. This function was replaced by the *malloc* function and some slight modifications to estimate the size of matrix A.

2. A coordinate format (COO) was introduced within the algorithm's core to make a more efficient use of the memory, avoiding several unnecessary memory accesses.

3. A method to produce an intermediate matrix P was shown to produce similar results to the default one and with matrix P being reduced to a single vector, thus requiring less data. Given that this was a broadcast data a diminishing on it, translated into a reduction of the broadcast time.

4. Four individual procedures which accessed the whole initial matrix memory, were merged into two processes, reducing this way the number of memory accesses.

For each optimization applied, a comparison was performed to show the particular improvements achieved. A set of different matrices, representing different SLAEs, was used to show the consistency of these improvements.

In order to provide with insights about the scalability issues of the algorithm, other approaches are presented to show the particularities of the algorithm's scalability:

1. Given that the original version of this algorithm was designed for a cluster of single-core machines, an hybrid approach of MPI + openMP was proposed to target the nowadays multi-core architectures. Surprisingly this new approach did not show any improvement but it was useful to show a scalability problem related to the random pattern used to access the memory.

2. Having that common MPI implementations of the broadcast operation do not take into account the different latencies between inter-node and intra-node communications [25]. Therefore, we decided to implement the broadcast in two steps. First by reaching a single process in each of the compute nodes and then using those processes to perform a local broadcast within their compute nodes. Results on this approach showed that this method could lead to improvements when very big systems are used.

Finally a comparison is carried out between the optimized version of the Monte Carlo algorithm and the state of the art Modified SPAI (MSPAI). Four metrics are used to compare these approaches:

1. The amount of time needed for the preconditioner construction.

2. The time needed by the solver to calculate the solution of the preconditioned system.

3. The addition of the previous metrics, which gives a overview of the quality and efficiency of the preconditioner.

4. The number of cores used in the preconditioner construction. This gives an idea of the energy efficiency of the algorithm.

Results from previous comparison showed that Monte Carlo algorithm can deal with both symmetric and nonsymmetric matrices while MSPAI only performs well with the nonsymetric ones. Furthermore the time for Monte Carlo's algorithm is always faster for the preconditioner construction and most of the times also for the solver calculation. This means that Monte Carlo produces preconditioners of better or same quality than MSPAI. Finally, the number of cores used in the Monte Carlo approach is always equal or smaller than in the case of MSPAI.

Published work

Oscar A. Esquivel-Flores, **Diego Dávila**, Vassil Alexandrov, (2016) *Enhanced Monte Carlo Methods for Sparse Approximate Matrix Inversion*, Book of abstracts of the 3rd BSC International Doctoral Symposium.

Vassil Alexandrov, Oscar Esquivel-Flores, **Diego Dávila**, (2016), *Enhancing Monte Carlo Preconditioning Methods for Matrix Computations*, Journal of Computational Science, Elsevier (Under revision).

Acknowledgements

I would like to extend a special acknowledgement to:

The Barcelona Supercomputer Center and the Severo Ochoa foundation for the scholarship granted.

The Education and Training team at BSC, for all the support.

Vassil Alexandrov and Eduard Ayguadé, for the invaluable guidance.

Oscar Esquivel, for all the help and friendship.

Hannali Melendez, for being my motivation and my adventure partner.

Finally I would like to thank my family and friends for the unconditional support and the good wishes.

Chapter 1

Introduction, motivation and goals.

1.1 Introduction.

Many problems in science and engineering are represented by Systems of Linear Algebraic Equations (SLAEs). Numerical methods are used to solve this kind of systems and they are categorized, usually, as two types: *direct* and *iterative* methods.

In the general case, direct methods produce more precise results than iterative ones, at the cost of requiring larger amounts of time. Iterative methods are able to produce results with different precision by varying the number of iterations performed. The larger the number of the iterations, the more accurate the result and the larger the time needed.

The time and computational effort required, to find the solution of a given SLAE, depend on the size of the system as well as on other factors (characteristics) such as the system being diagonally dominant, the symmetry or the condition number of the system. this factors influence the convergence rate of a given method, therefore they must be taken into account to decide the best method to be applied. For example, a system with a higher condition number (ill-conditioned) is likely to require more time to be solved than other with similar characteristics and a lower condition number.

It is well known that modern science deals with a vast variety of problems that require great amounts of calculation. Despite the abundance of processing power of nowadays systems, some classes of scientific problems are able to exhaust those tools [27]. An example of this kind of systems are *clusters of computers* and *supercomputers*.

Given the parallel nature of the above mentioned systems, the benefits obtained by their use are tightly related to the parallel capabilities of the applications (*i.e. only parallel application can benefit from the use of parallel systems*). Furthermore the percentage of parallelism in a given application will define the potential speedup obtained by its parallel execution [18].

Preconditioners are used to reduce the time required, for a given method, to solve a SLAE. There exist a wide variety of preconditioners but we will focus only on those that use a SParse Approximate Inverse of the system (SPAI). SPAI preconditioners work under the assumption that it is possible to use the inverse of a system's matrix to find its solution.

In the past, it has been demonstrated that Monte Carlo methods can be used to efficiently tackle Linear Algebra problems [3] [10] . Given the stochastic approach of these methods, the computational effort required to find an element in the inverse matrix of a given SLAE, is independent from the size of the matrix. This allows to target systems that, due to their size, can be prohibitive for common deterministic approaches. [27]

Generic Monte Carlo algorithms posses certain properties that make them suitable for highly parallel architectures. They present an *efficient distribution of the compute data* and a *minimum synchronization* is required while computing. This two properties naturally lead to scalable algorithms [27].

Recently research has been focussed to show the advantages of these methods for solving Linear Algebra problems such as the calculation of a matrix inverse; results demonstrate that Monte Carlo algorithms are a good choice for SPAI preconditioning for general matrices but also implies, that further research into their scalability behaviour, is required [3] [4]

A variety of parallel Monte Carlo methods have been developed within the past 20 years. A comprehensive compendium of the Monte Carlo functions can be found in [29] [27] [10]. Various parallelization strategies and approaches were investigated since then [10] [27].

In this work we present an enhanced version of a SPAI preconditioner that is based on a Monte Carlo method. This new optimized version is compared against the previous one, as well as the *state-of-the-art* MSPAI, which is the main accepted deterministic algorithm for SPAI preconditioning. Our results show that Monte Carlo-based algorithm can be used instead of MSPAI to reduce the computation time and resource usage while producing results with similar or better quality.

Also a scalability analysis is carried out, showing that the random pat-

terns in the memory access have a strong influence in the performance of the algorithm. further research, to solve this issues, is proposed within the context of quasi-Monte Carlo Methods.

1.2 Motivation.

The use of preconditioners is very common nowadays and much research effort has been applied on this subject, given that linear systems, composed by millions of equations, are now commonly found in many applications.

Also "the solution of large sparse linear systems is central to many numerical simulations and is often the most time-consuming part of a computation" [8].

Design and optimization of preconditioners which are able to take advantage of parallel architectures are necessary to exploit current computational power such as HPC systems.

Given the stochastic nature of Monte Carlo-based algorithms, they are good candidates to tackle these kind of problems in an efficient way. These methods have the particular property of their computational complexity growing linearly with the size of the system and being highly parallelizable.

Efficient implementation of preconditioners will allow solving large SLAEs which may be of vital importance for the science in general. Also these implementations can be directly translated into savings of *execution time* for current problems.

1.3 Goals.

The general objective of this work is to enhance the current implementation of the Monte Carlo algorithm by: making it more stable and improve further its performance and scalability. Specific goals are listed below:

1. **Eliminate errors**. Stability in the application will be achieved by solving the know issues and applying general testing to find possible hidden errors within the implementation.

2. **Improve the performance**. Find the bottlenecks that affect the performance of the application, propose and implement efficient solutions to cancel or mitigate their effects.

3. **Analyse the scalability** An hybrid approach of MPI + OpenMP for an hybrid parallel architecture will be carried out to investigate whether it can provide improvements in the scalability and the performance.

4. **Compare with the state-of-the-art**. A performance comparison will be performed against the state-of-the-art MSPAI algorithm.

5. **Code refactoring**. Make the code more maintainable in order to make it easier for further optimizations.

Chapter 2

State-of-the-art.

2.1 Preconditioners.

Preconditioning is known as the transformation of a System of Linear Algebraic Equations (SLAE) into an equivalent one which requires less time to be solved. The matrix used for this transformation is called the **preconditioner** [7]. When carried out efficiently, the preconditioned system would be solved faster than the original one.

Solving Systems of Linear Algebraic Equations (SLAEs) is of great importance in many areas of study in science and engineering. *Numerical methods* are used to find the solution of these systems."The solution of large sparse linear systems is central to many numerical simulations and is often the most time-consuming part of a computation" [8].

There are two main categories of numerical methods: *Direct* and *iterative* ones. The main differences between them are the number of steps required for their execution and the precision of the calculated solution. When dealing with dense matrices, the complexity of direct methods, like Gaussian Elimination is $\mathcal{O}(n^3)$ and for the iterative ones like Jacobi is $\mathcal{O}(kn^2)$ [15].

When an precise solution is required *direct methods* are the preferred option. They are reliable, robust and due to their deterministic nature, they require a finite number of steps, for this reason they tend to require a predictable amount of resources (time, memory, storage, etc). The drawback is that most of them are difficult to parallelize and many of them do not scale well.

In the other hand *iterative methods* which are easier to parallelize and are a faster option [16]. The accuracy of these type of methods can be parametrized as it depends on the number of steps (or iterations) executed.

They are a good choice when an approximate solution is required.

It is well known that most of the direct methods have a poor scalability [8] (the time and resources needed grows exponentially with the size of the problem). Given that the majority of problems are only worth to solve within a limited time, iterative methods are sometimes the only available choice to solve large systems. Still when dealing with very large systems, iterative methods might fail to obtain a good enough approximate solution within a reasonable amount of time. It is in these cases where preconditioning becomes a necessity.

It should be noted that "systems with several millions of unknowns are now routinely encountered in many applications" [8]"

The general idea of preconditioning, within the context of iterative methods, is to modify the input (ill-conditioned) system in such a way that the iterative method converges faster. In general, a good preconditioner should be easy to compute (in terms of computational effort), to calculate and apply, (*i.e. the time saved due to preconditioning must be larger than the time invested in the preconditioner construction).* Although, resources constraints could be also a good reason to apply a preconditioning regardless of the time.

A good balance between quality and time of construction is the key for a good preconditioner. Notice that, when parallel systems are used to build a preconditioner, the parallel efficiency of the construction algorithm will play a very important role in this trade-off.

The use of preconditioners is very common nowadays and much research effort has been applied on this subject.

"Nothing will be more central to computational science in the next century than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly. For Krylov subspace matrix iterations, this is preconditioning" [28].

Several kinds of preconditioning methods can be found In the literature [8], In our research we will be focussed on SParse Approximate Inverse(SPAI) preconditioners.

2.1.1 SPAI preconditioners.

It is well know that some numerical methods are more adequate for matrices that feature certain properties. Preconditioning is designed to enhance these properties, for this reason preconditioners and numerical methods are commonly bonded together *(i.e. preconditioners are designed on base of*

numerical methods).

" An optimal general-purpose preconditioner is unlikely to exist" [8].

In this sense, there are many different types of preconditioners. This work is only focused in the SParse Approximate Inverse (SPAI) type.

As the name implies, SPAI preconditioning uses the approximate inverse of system as preconditioner. This is, given the SLAE (2.1) in wich A represents the matrix of coefficients, x is the unknown solution vector and b is the right-hand side vector. One could use the inverse of A (A^{-1}) in both sides of the system (2.1) to obtain (2.2). Then given the matrix properties $AA^{-1} = A^{-1}A = I$ and $Ix = x$ the expression in (2.3) can be obtained, which gives the solution for x.

$$Ax = b \tag{2.1}$$

$$A^{-1}Ax = A^{-1}b \tag{2.2}$$

$$Ix = A^{-1}b \tag{2.3}$$

Calculating the inverse of a SLAE could be as complex as finding its solution, this is the reason why SPAI precoditioners use an *approximation.* Then instead of having (2.2) we will have (2.4) and thus (2.5) instead of (2.3) (the symbol " ˆ " denotes an approximation). Even when (2.5) is not a solution to the system it is an equivalent and less complex expression of it, meaning that a given numerical method will require less time to calculate the solution for this new system than for the original one.

$$\hat{A}^{-1}Ax = \hat{A}^{-1}b \tag{2.4}$$

$$\hat{I}x = \hat{A}^{-1}b \tag{2.5}$$

Roughly speaking, SPAI preconditioners work in this way, the main difference between one method or another is the technique used to calculate the approximate inverse.

Minimination of the Frobenius norm.

A common approach, in SPAI deterministic methods is to calculate an approximate inverse based on the Frobenius norm (2.6) minimization [17].

$$\min_{M} \|AM - I\|_F^2 \tag{2.6}$$

The idea is to minimize the difference between de *identity matrix* (I) and the result of multiplying the preconditioner M by the initial system A. Notice that If $M = A^{-1}$ the difference in (2.7) will be zero.

As we have said before, calculating $M = A^{-1}$ will require as much effort as calculating the solution of A. For this reason an approximation of A^{-1} ($\hat{A}^{-1}$) is used instead.

When the preconditioner matrix M is very similar to matrix A^{-1}, AM will be also very similar to I and the preconditioned system will be simpler.

$$AM - I \tag{2.7}$$

Notice that the problem of minimizing the Frobenius norm, can be decomposed into n minimization problems. In this case n will represent the number of columns in the preconditioner matrix M. In this way each column of M can be calculated separately making the problem easy to parallelize.

This technique for calculating the inverse matrix is used by the algorithm MSPAI [21] which is further described in section (2.1.2).

Neumman series expansion approach

Another approach to approximate an inverse matrix is the use of *Neumann series expansion*. This states that the inverse of a matrix can be calculated with (2.8). For this to be true, the condition (2.9) must be satisfied.

$$(I - C)^{-1} = \sum_{k=0}^{\infty} C^k \tag{2.8}$$

$$\|C\|_p < 1 \tag{2.9}$$

For simplicity we will always consider (2.9) with the *infinity norm* ($p = \infty$) but in practice this condition applies for every norm.

For this approach it is necessary to find a Matrix C such that $A = (I - C)$ and the condition (2.9) is met.

It can be probed that if A is a *diagonally dominant* matrix (2.10), an equivalent matrix A' (2.11) can be used to find a matrix $C = (I - A')$ which meets the condition (2.9).

$$|a_{i,i}| > \sum_{j \neq i}^{n} |a_{i,j}| \quad \forall i \tag{2.10}$$

$$a'_{i,j} = \frac{a_{i,j}}{a_{i,i}} \quad \forall i, j \tag{2.11}$$

The parallelization of this technique is not as straightforward as in the previous case. This approach is used in the algorithm described in section 2.2 and it is used in combination with a Monte Carlo method on which the parallelism relies.

Regardless of the method used to calculate the approximate inverse it is important to maintain a high sparsity of the preconditioner. A well conditioned system may reduce the number of iterations needed, but when the resulting system is dense, iterative methods tend to spend more time per iteration. A good balance between the quality of the inverse and its sparsity must be always considered and maintained.

2.1.2 MSPAI.

Given the inherent parallelism of SPAI preconditioners, in the past there have been different approaches of parallel implementations of this kind of preconditioners. Recently a class of Frobenius norm minimizations that has been used in the original SPAI implementation [17] was modified and provided in a parallel software package.

This approach was carried out by the original authors of the SPAI implementation and it is called Modified SParse Approximate Inverse (MSPAI) [21]. The word "Modified" comes from the inclusion of certain properties of a family of preconditioners known as "Modified preconditioners" such as Modified ILU, Modified Cholesky, etc.

The main contribution to the SPAI implementation is the probing approach. This way MSPAI combines the advantages of classical probing,

application of modified preconditioners and the Frobenius norm minimization.

Further, this package also provides implementation improvements such as the use of a dictionary to avoid redundant calculations, and a dynamic load balancing which allows for a better workload distribution among the parallel resources.

2.2 Monte Carlo-based algorithm.

This section describes an algorithm that constructs a SPAI preconditioner based on *Neumann series expansion*. The algorithm is a Monte Carlo Markov Chain based one, which is used to reduce the density as well as the computational effort needed to produce the matrix inverse, thus calculating an sparse approximation of it.

This work is focused on the optimization of this algorithm's implementation which is presented later in this section.

2.2.1 Monte Carlo methods

Monte Carlo methods are probabilistic methods, that use random numbers to either simulate a stochastic behaviour or to estimate the solution of a problem. They are good candidates for parallelization due to the fact that many independent samples are used to estimate the solution. These samples can be calculated in parallel, thereby speeding up the solution finding process. [27]

2.2.2 The algorithm

The algorithm can be roughly explained within the following 5 phases. Notice that phases 1 and 5 are only necessary when the initial matrix is not *diagonally dominant*.

1. The initial matrix is transformed into a *diagonally dominant* matrix.
2. A transformation is carried out in order to make the *diagonally dominant* matrix suitable for the *Neumann series expansion*.
3. A Monte Carlo method is applied on top of *Neumann series expansion* to calculate a sparse approximation of the inverse matrix.
4. Given the transformation carried out in 2, it is necessary to calculate the inverse of the diagonally dominant matrix parting from the resulting matrix of 3.

5. Similar to previous step, a recovery process is applied to make up for the transformation in 1.

Algorithms 1 and 2 [10] are presented below. Algorithm 2 is based upon algorithm 1 and it is used when the initial matrix is not *diagonally dominant*. In the other hand, Algorithm 1 can be used directly when the input matrix is *diagonally dominant*

Further explanation about the algorithm's details are explained later in this section.

Algorithm 1 *Monte Carlo Algorithm for Inverting Diagonally Dominant Matrices [10]*

Step 1. *Read in matrix B*

1: *Input matrix B, parameters ε and δ*

Step 2. *Calculate intermediate matrix (B_1)*

1: *Calculate $B_1 = diag(B)$*

Step 3. *Calculate matrix C, A and $\|A\|_\infty$*

1: *Compute the matrix $C = B_1^{-1}B$ and $A = (I - C)$*

2: *Compute $\|A\|_\infty$ and the Markov chains $N = \left(\frac{0.6745}{\epsilon(1-|A\|_\infty)}\right)^2$*

Step 4. *Calculate matrix P*

1: *Compute the probability matrix, P.*

Step 5. *Calculate matrix C^{-1}, by MC on A and P*

1: *For $i = 1$ to n:*

1.1: *For $j = 1$ to N*
Markov Chain Monte Carlo Computation.

1.1.1: *Set $W_0 = 1$, $point = i$ and $SUM[k] = \begin{cases} 1 & if \quad i = k \\ 0 & if \quad i \neq k \end{cases}$*

1.1.2: *Select a* nextpoint*, based on the transition probabilities in P, such that $A[point][nexpoint] \neq 0$*

1.1.3: *Compute $W_j = W_{j-1}\frac{A[point][nextpoint}{P[point][nextpoint]}$*

1.1.4: *Set $SUM[nexpoint] = SUM[nexpoint] + W_j$*

1.1.5: *if $|W_j| \geq \delta$ set* point = nextpoint *and goto 1.1.2*

1.2: *Then $c_{ik}^{-1} = \frac{SUM[k]}{N}$ for $k = 1, 2, ..., n$*

Step 6. *Calculate B^{-1}*

1: *Compute the Monte Carlo inverse $B^{-1} = B_1^{-1}C^{-1}$*

Algorithm 2 *Monte Carlo Algorithm for Inverting General Matrices*

Step 1. *Read in matrix B*

1: *Input matrix B, parameters ε, δ and α*

Step 2. *Calculate diagonal dominant matrix $\hat{B}$.*

1: *Calculate $\|B\|_\infty$*

2: *Calculate $\hat{B}$ such that* $\hat{b_{i,j}} = \begin{cases} b_{i,j} & if \quad i \neq j \\ b_{i,i} + \alpha\|B\|_\infty & if \quad i = j \end{cases}$

Step 3. *Apply Algorithm 1 with $B = \hat{B}$ to obtain $\hat{B}^{-1}$*

Step 4. *Recovery of B^{-1} from $\hat{B}^{-1}$*

1: *Compute $S = \hat{B} - B$*

1:1 *Let S_i for $i = 1, 2, \ldots, n$ where each S_i has just one of the non-zero elements of the matrix S*

1:2 *Set $B_n^{-1} = \hat{B}^{-1}$*

1:3 *Apply $B_{i-1}^{-1} = B_i^{-1} + \frac{B_i^{-1} S_i B_i^{-1}}{1 - trace\left(B_i^{-1} S_i\right)}$ for $i = n, n-1, \ldots, 1$*

2: *Then $B^{-1} = B_0^{-1}$*

Phase 1. Ensuring diagonal dominance

In 2.1.1 we talked about the relation between the restriction in the *Neumann series expansion* and the diagonal dominance of the matrix. Knowing that, we want to force this property in the initial matrix B by applying the following transformation:

$$\hat{b_{i,j}} = \begin{cases} b_{i,j} & if \quad i \neq j \\ b_{i,i} + \alpha\|B\|_\infty & if \quad i = j \end{cases}$$

This is, the off-diagonal elements of $\hat{B}$ are the same as those of B and the diagonal elements of $\hat{B}$ are defined as $b_{i,i} + \alpha\|B\|_\infty$ where α is an arbitrary *input parameter* with a value > 1 and recommended to be ≤ 2.

$$\|B\|_\infty = \max_{1 \leq i \leq n} \quad \sum_{j=1}^{n} |b_{i,j}| \tag{2.12}$$

Notice that the diagonal dominance of the matrix ensures the convergence of the *Neumann series expansion* in the system but a further transformation is needed to strictly comply with (2.13), such transformation is performed in the next phase.

Phase 2. Make the matrix suitable for *Neumann series*

From the Neumann Series expansion approach described in 2.1.1 we know we want to find a matrix M such that $\hat{B} = I - M$. This way we will have $\hat{B^{-1}} = \sum_{k=0}^{\infty} M^k$ due the *Neumann series* (2.8).

Taking into account the condition (2.9) and that fact that $M = I - \hat{B}$, we need (2.13) to be true. Knowing that $\hat{B}$ can have any arbitrary value, we need to perform a transformation that guarantees that this condition is always met.

$$\|I - \hat{B}\|_p < 1 \tag{2.13}$$

Given the diagonal dominance of $\hat{B}$, we know that adding the absolute values of all the non-diagonal elements, in any given row, will be less than the absolute value of the diagonal element in that row (2.10). Then if we multiply all the elements in a row by the reciprocal of the diagonal element of that row (*i.e. all elements in the row are divided by the diagonal element*), we will end up having a 1 in the diagonal and the addition of all non-diagonal elements will be less than 1.

The above means that performing (2.14) will make (2.13) to be met. Of course we will have to change $\hat{B}$ in (2.13) by $\hat{\hat{B}}$.

$$\hat{\hat{B}} = (diag(\hat{B}))^{-1} * \hat{B} \tag{2.14}$$

Now we can calculate a matrix $A = (I - \hat{\hat{B}})$ and $\hat{\hat{B}} = (I - A)$ such that $\hat{\hat{B^{-1}}} = \sum_{k=0}^{\infty} A^k$

Phase 3. Monte Carlo and Markov chains.

Until this point we have a matrix A that meets the *Neumann series* condition. And the matrix $\hat{\hat{B}} = (I - A)$. For simplicity, from now and then we

will call matrix $\hat{\hat{B}}$ matrix C, (*i.e.* $C = \hat{\hat{B}}$)

In this step the operation (2.15) is carried out. We know that for the inverse matrix to be accurate we need $k >> 1$, this would imply k matrix-matrix multiplications which are known to be expensive operations. Also this would lead us to calculate a very dense inverse matrix.

$$C^{-1} = \sum_{k=0}^{\infty} A^k \tag{2.15}$$

This is where the Monte Carlo method plays its role. Using a *random variable* whose mathematical expectation is the desired solution (2.15), a sampling is performed and results are used to construct the inverse matrix (C^{-1}). Applying stochastic sampling instead of computing complete k matrix-matrix multiplications leads to a sparse matrix obtained with much less operations rather than a dense one requiring much more operations.

The Monte Carlo process can be expressed as (2.16) and the random variable as (2.17) [2].

$$c^{-1}_{r,r'} \approx \frac{1}{N} \sum_{s=1}^{N} \left[\sum_{(j|s_j=r')} W_j \right] \tag{2.16}$$

$$W_j = \frac{a_{r,s_1} a_{s_1,s_2} \quad ... \quad a_{s_{j-1},s_j}}{p_{r,s_1} p_{s_1,s_2} \quad ... \quad p_{s_{j-1},s_j}} \tag{2.17}$$

In (2.16) $(j|s_j = r')$ means that W_j is only summed when $s_j = r'$ being r' the column index in C^{-1} and s_j a certain state in the Markov chain. Notice that all the values of a row can be calculated using the same Markov Chain if different r' are used to store the different results (in Algorithm 1, the SUM variable is used for this end).

A Markov chain is based on a *random variable* sampling. A pair of states, within the chain, represent a specific element in matrices A and P. Consider the following Markov chain:

$$S = S_0 \rightarrow S_1 \quad ... \quad S_j \rightarrow \quad ...$$

with:

$$j \in \{1, 2, ..., n\}$$

Exemplified by:

$$S = 1 \rightarrow 5 \rightarrow 7 \rightarrow 1 \rightarrow 2 \quad ...$$

The first state($S_0 = 1$) identifies the row we are calculating, it is variable (r) in (2.16) and (2.17). By the second state ($S_1 = 5$) we know the first element chosen is ($a_{1,5}$), the next element would be ($a_{5,7}$) and so on. The variable (r') would be (5) and (7) for the last two elements respectively, this means that the value of W_j obtained by those states using (2.17), would be accountable only for elements ($c_{1,5}^{-1}$) and ($c_{1,7}^{-1}$) respectively.

The Markov Chain is constructed based on a probability matrix P. There are two different approaches used to build this matrix, the first and simplest one is based on a Uniform distribution and is called *Uniform Monte Carlo*(UM), the second one is called *Monte Carlo Almost Optimal*(MAO) and, theoretically, leads to more accurate results [5] but, when implemented, it requires significantly more computational resources, not only for its construction but for its use. See section 4.3.3 for more details.

The number of different states that could be found at any chain is equal to the matrix size. The size of the chain *(i.e. the number of states in the chain)* is bounded by the *input parameter* δ. The Markov chain is stopped (cut) when W_j becomes smaller than δ.

The number of chains (N in (2.16)), used to calculate an entire row, is calculated using the *input parameter* ϵ and the infinity norm of matrix A within the formula 2.18 [10].

$$N = \left(\frac{0.6745}{\epsilon(1 - |A\|_\infty)} \right)^2 \tag{2.18}$$

Notice that the complexity of this method is $\mathcal{O}(nNL)$. This is linear to the matrix size (n) given that the process to calculate an entire row is not bounded by (n) but by N an L. Here (N) is the number of chains and (L) is the size of the Markov Chain.

Phase 4. Calculate the inverse of the diagonally dominant matrix.

In phase 2, we calculated matrix $\hat{\hat{B}}$ from matrix $\hat{B}$ and called C, for the sake of simplicity. Then, in phase 3 we calculated the inverse of that matrix. Now we want to calculate $\hat{B}^{-1}$ from $\hat{\hat{B}}^{-1}$.

Having (2.14), we know we can multiply $diag(\hat{B})$ in both sides of the equation and obtain (2.19), then we can invert all the terms in the equation and get (2.20).

$$diag(\hat{B}) * \hat{\hat{B}} = \hat{B} \tag{2.19}$$

$$(diag(\hat{B}))^{-1} * \hat{\hat{B}}^{-1} = \hat{B}^{-1} \tag{2.20}$$

Notice that only a matrix-vector multiplication is necessary to obtain $\hat{B}^{-1}$ from $\hat{\hat{B}}^{-1}$

Phase 5. Calculate the inverse of the initial system

Due to the transformation carried out in Phase 1, a recovery process needs to be performed in order to calculate the inverse B^{-1} of the original input matrix B parting from $\hat{B}^{-1}$. Remember that phase 1 and 5 are only necessary when the input matrix is not *diagonally dominant*.

In [10] the following iterative process is described for that purpose:

$$B_k^{-1} = B_{k+1}^{-1} + \frac{B_{k+1}^{-1} S_{k+1} B_{k+1}^{-1}}{1 - trace(B_{k+1}^{-1} S_{k+1})} \tag{2.21}$$

Where k goes from n-1 to 0 ($k = n-1, n-2, ..., 0$), having that $B_n^{-1} = \hat{B^{-1}}$. S_i is all zero except for the {ii}th component, which is from the matrix $S = \hat{B} - B$.

In [10] it is claimed that the complexity of this algorithm is lower than it seems at first glance, due to the simplicity of matrix S. Even though, it is easy to notice that the complexity of the process grows polynomially, with the main term requiring $\mathcal{O}(n^4)$ operations.

This process can be very costly when big matrices are being used. It is also difficult to parallelize. Good results still can be obtained if this phase is skipped.

2.2.3 Implementation details (Original code).

For this chapter a minimal knowledge about parallel programming models is required. If the reader is not familiarized with terms like *HPC, Distributed memory, Shared memory, Single-core, Multi-core and MPI* it is recommended to read chapter 2.3 first.

This algorithm was originally designed for a HPC cluster composed of single-core compute nodes. It is written in C and uses the MPI library. It also makes use of the Bebop sparse matrix converter [20] to translate the input matrix format into a CSR format.

2.2.4 Parallelization details.

The first two phases described in section 2.2.2, are executed sequentially by the *MPI Master process (rank 0).*

Matrices A, B_1 and P are calculated during those phases. Notice that $A = (I - C)$ and $B_1 = diag(\hat{B}))$

Then a procedure is called by all the processes in which the partitioning of the matrix A is carried out. The distribution of the work is done evenly when the number of rows is divisible by the number of processes. In the opposite case, the remaining rows are distributed among the smaller MPI processes (without including the Master process).

After that, matrices A, B_1 and P are *broadcast* using MPI_Broadcast(), so that all the remaining processes can have a copy of them. Then the Monte Carlo process (phase 3) is started in parallel by all MPI processes.

During the Monte Carlo phase, each MPI process will calculate a piece of the inverse matrix of C (C^{-1}), using matrix A; remember that $C = (I - A)$. Each resulting row will be then multiplied by matrix B_1^{-1}, to get their respective part of $\hat{B}^{-1}$ (phase 4).

After finishing the Monte Carlo process and phase 4, each process will send its part of the matrix ($\hat{B}^{-1}$) to the master process by calling MPI_Send(). The master process will perform a corresponding MPI_Receive() and will merge the received parts with its own.

Given a concatenation issue due to the CSR format (explained in the following subsection), the Send-Receive process has to be ordered, having to receive first the data from process 1, then process 2 and so on.

Finally the last phase (5) is optionally executed by the master processes on matrix ($\hat{B}^{-1}$) to calculate B^{-1}. A *flag* called RECOVER is set/unset in the makefile to indicate whether to execute this process or not.

In the past, an unsuccessful attempt to parallelize this process has been carried out. Its iterative nature made that difficult to achieve; iterations cannot be executed in parallel given that each of them depends on the previous one. On the other hand, using an approach in which each iteration is executed in parallel, would imply a high increment in the communications given that, a synchronization would be required at each iteration.

2.2.5 CSR format details.

The CSR format is well known to be a very efficient option when dealing with sparse matrices, given that only non-zero values are stored. This allows the use of big sparse matrix within a reduced amount of memory.

The bad thing about using CSR format is that the size of any given matrix varies when operations are performed on it. This way many matrix operations become more complex when this format is used. A simple sparse matrix-matrix addition, for example, is very likely to result in a matrix which is bigger that its operands, this would not allow to reuse one of the operands matrices to store the result.

The following example shows a matrix-matrix addition performed in a CSR format, notice how the vectors *val* and *col* in the resulting matrix are bigger than those in the operands.

$$\begin{pmatrix} 1 & 0 & 5 \\ 0 & 0 & 4 \\ 0 & 8 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 4 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 5 \\ 0 & 1 & 4 \\ 0 & 8 & 3 \end{pmatrix}$$

$$\begin{array}{ccc} val = [1,5,4,8] & val = [4,1,3] & val = [1,4,5,1,4,8,3] \\ col = [1,3,3,2] & col = [2,2,3] & col = [1,2,3,2,3,2,3] \\ row = [1,3,4,5] & row = [1,2,3,4] & row = [1,4,6,8] \end{array}$$

An example of this within the implementation of the algorithm can be found in the matrix-matrix addition carried out in the phase 5. See (2.21).

Furthermore, when a CSR matrix is being constructed, and the number of total non-zero values is unknown *a priori*. If an estimate is not able to be calculated, several memory reallocations must be carried out to store the values; in the worse case(when memory is a constraint) a memory reallocation will be carried out for each new element.

This worse case-approach was used in the original implementation for the construction of matrix A and other intermediate matrices.

Similarly for a concatenation to be performed, if the number of non-zero values of each of the sub-matrices remains unknown, the operation must be carried out in order; otherwise the position within the final memory structure, of any given matrix, could not be calculated.

The above affects the algorithm execution in the sense that, after the Monte Carlo phase, if a process has finished with anticipation, it has to wait for all previous processes to finish to be able to communicate its results to the master process. The communication cannot be carried out in advance and the process would transition to an idle state.

2.3 Parallel programming models

Within the HPC context, the objective of *parallel computing* is to increase the performance of applications. This is achieved by the explicit division of application's workload among the available resources within a parallel environment, all this with the aim of diminishing the execution time.

This explicit division is carried out through the use of parallel programming models. There exist several of these models which target different architectures. The underlying architecture in which a application is thought to be executed is, therefore, the main aspect to consider when choosing a parallel programming model.

Depending on the communication model parallel architectures can be roughly categorized in 3 groups:

1. *Distributed memory*: In this type of architecture, several machines (called compute nodes) are connected through a network, memory is kept private for each machine (distributed) and the only way to read/write other's machine memory is through the network. MPI is the most common programming model used for these type of architectures.

2. *Shared memory*: In this architecture the main memory can be directly

accessed by all the processors/cores in the system. The most representative model for this approach is called OpenMP.

3. *Accelerators*: Also referred as *many-core* architectures, in this approach is an external device, containing "many cores", communicates with the CPU commonly through the PCIe port.

   It can also be considered a *shared memory* approach given that all the cores, within the device, have access to a common memory. This memory is local to the device and should not be confused with the system's main memory. The most popular programming model of this kind is CUDA.

Nowadays HPC systems are built using a combinations of all these 3 parallel architectures. A common example it is a cluster of computers connected via a high performance network, each computer has two or more processors which in turn has two or more cores, all these cores within a single machine, share the system's main memory, but this memory is kept private from other's machines within the cluster. Finally some of these nodes have an accelerator in addition.

The above description corresponds to an hybrid parallel architecture. In order to take full advantage of these kind of architectures a mixture of parallel programming models has to be used. A very common hybrid model is the one which includes MPI and OpenMP.

Further details on programming models as well as a performance an a qualitative survey can be found in [23].

2.3.1 MPI.

The Message Passing Interface is the main *programming model* used for *distributed memory*. In the general case it works as follows.

When an MPI application is launched several processes are created and they all execute the exact same application. The number of processes is specified by the user at the moment of the execution.

Within the application, MPI routines are used to control the flow of each one of the processes. One of the most important ones is MPI_Comm_rank() which gives a unique numerical identifier for each of the processes. This identifier is called *rank* and is used to distinguish among the different processes.

There exist several MPI routines used to communicate data from one process to another, called *point-to-point* and others which involve more than 2 processes, called *collectives*.

The MPI_Send() function is an example of a *ponit-to-point* communication, at the name implies it is used to send data from one process to another, the size and type of the data, as well as the receiving process identifier must be specified within the call. Its counterpart MPI_Recv() is used to receive the data in the other extreme, similarly the size and type of data, as well as the sending identifier must be specified.

Within the algorithm, these functions are used to send/receive the pieces of the inverse matrix from each process to the master process.

An example of a *collective* operation is the MPI_Broadcast() function, this function is used when data is needed to send to all or a certain group of processes. The call must be performed by all the processes involved in the operation, the group identifier (called communicator) must be provided.

This function is called by all the processes in order to get a copy (from the master) of the matrices needed within the Monte Carlo proccess.

Collective functions are used together with *communicators* which can be seen as logical groups of MPI processes. At the beginning only one communicator exist and it encompasses all the processes, it is identified by the macro MPI_COMM_WORLD. User can define further *communicators* accordingly to the needs of the application, this way collective operations can be performed on a sub-set of MPI processes.

2.3.2 OpenMP.

Open Multi-Processing is a set of compiler directives, environment variables and callable runtime library routines used to express shared-memory parallelism [13].

The work unit for this model is the *thread* which is an execution instance of a given parallel section of the code. Variables within a thread can be defined to be private or shared with all the other threads.

The number of threads is automatically set to the number of cores in the system and can be overwritten by the *environment variable* OMP_NUM_THREADS.

Programmers uses compiler directives(pragmas) to wrap portions of code that have a parallel connotation.

A simple example of the use of OpenMP can be observed with the use of the following directive: *#pragma omp parallel{}*

This directive specifies that all the code within the curly brackets is executed by all the available threads. A call to the function omp_get_thread_num() will return a unique identifier for each of the threads. This id can be used

to control the execution flow of each thread independently.

This directive is used in section 4.4.1 to execute the Monte Carlo process in parallel.

Other directives like *#pragma omp parallel for* are used to distribute the iterations of a given loop among the available threads.

When using this kind of models, programmers must be very careful to avoid unwanted behaviours. Examples of this are *race conditions* and *false sharing*.

A *race condition* happens when a variable end up with different values in different executions under the same conditions, due to the variable interleaving of the threads accesses to the given variable.

false sharing occurs when two private variables from different threads are tight together within the same *cache line* (the minimal memory transfer unit), for this reason every time a thread modifies its variable, a synchronization is carried out to keep the *cache line* coherent for both threads. This synchronization is logically unneeded and subtracts performance execution.

Together with the above, other important drawback of this approach is the *overhead* implicated in the thread creation and all the synchronization needed to maintain the memory coherence among the threads.

2.3.3 Hybrid approach(MPI + openMP).

Nowadays it is very common to find HPC applications that use more than one parallel programming model. One of the most common combinations is the use of MPI and OpenMP to target a cluster environment of Multi-core machines.

The common approach is to create one MPI instance on each machine and then one OpenMP thread for each of the available cores in the machine. This is the *naive* way to exploit the "best of two worlds" but in practice, different combinations of number of MPI processes and number of threads are tested to find the best combination which is application dependent.

Actually there are some applications which perform better when only MPI (unified MPI) processes are used in both, the distributed and the shared environment. [11]

2.3.4 MPI for shared memory.

Even though MPI is designed for *Distributed memory* architectures, it is also possible to use it within a *shared memory* environment. In this case memory is kept private to each MPI process and communications are not carried out through the network but directly as memory operations (Ex. Sending a given value from process 1 to process 2 is carried out by copying the value from the private memory of process 1 into the private memory of process 2).

In [24] it is demonstrated that it is difficult to obtain significant and consistent improvements when using OpenMP intead of MPI within shared memory architectures.

An example of the above can be those applications with a random memory access pattern. One very important benefit of using OpenMP, comes from the memory bandwidth increment given the shared memory approach. This benefit vanishes when memory is randomly accessed (data fetched by one thread is not likely to be used by other threads.)

In [22] it is explained why random accesses to memory decrease the memory bandwidth.

Chapter 3

Methodology.

Having that the current implementation present some execution errors, the first thing to do is to *find and fix* those error in order to have a stable and comparable version of the software.
Then we will *define a baseline* to be able to compare and measure the performance of different versions of the code.
After that a general analysis of the fixed implementation will be carried out, and different approaches, to *improve the Monte Carlo-based algorithm*, are going to be performed in an iterative fashion.

Each new optimization approach will be implemented and validated. In the case of successful implementations, a comparison with the previous version will be performed to show the particular improvements of the approach.

Finally, the fastest achieved version will be compared with the *state-of-the-art* MSPAI. Different aspects, such as the time to compute the preconditioner, the time needed to compute the solution of the preconditioned system and the resource usage will be measured and presented.

Experiments will be carried out in Marenostrum III supercomputer at Barcelona Supercomputing Center (BSC). It currently consists of 3056 compute nodes equipped with 2 Intel Xeon 8-core processors and 64GB of RAM interconnected via InfiniBand FDR-10.

The Monte Carlo-based algorithm is written in C and it uses the open MPI-1.8.0[1] implementation of MPI. The intel-13.0.1 OpenMP will be used for an hybrid approach implemented later as a part of this work. The solver used to measure the time needed to calculate the solution for the preconditioned system, is the paralution-1.1.0[2] implementation of GMRES.

[1] www.open-mpi.org/
[2] www.paralution.com/

Below a more detailed and structured methodology is described.

1. **Find and fix errors**. Minimal modifications will be carried out to avoid the current crashes of the system.

   1.1. A set of small matrices is going to be used to find the patterns related to the application errors.

   1.2. A Comparison of the current implementation and the mathematical description will be performed to find possible silent errors. "Testing do not guarantee the absence of errors".

   1.3. Step-by-step execution using a system with a known solution will be analysed.

2. **Define a baseline**. Together with the fixed version (obtained in the previous step), a set of matrices and a process to measure the quality of the preconditioner, will be proposed to conform to the baseline that will be used to compare the new implementations.

   2.1. Select a representative set of matrices with different sizes and characteristics to be used to compare different executions.

   2.2. Define a process that will be used to quantify the quality and efficiency of the preconditioners.

3. **Improve the Monte Carlo-based algorithm.**

   3.1. Use BSC tools to locate bottlenecks and analyse the behaviour of the existing code and further versions.

   3.2. Analyse the algorithm description to find new sources of parallelism and room for improvement.

   3.3. Implement new proposals following common good practices of software development.

   3.4. Validate the new implementations using the baseline described before.

4. **Compare with MSPAI.** Speedup and scalability analysis will be performed.

   4.1. Use the baseline from step 2 to compare Monte Carlo-based and MSPAI algorithms in terms of *quality*, *efficiency* and time required to build the preconditioner.

Chapter 4

Development of the work.

4.1 Finding and fixing errors.

Using a set of small matrices, the original code was tested to find implementation errors. Two important errors were found during this process:

1. A validation was carried out, after reading the initial matrix, to ensure the number on non-zero elements was equal or smaller than the size of the matrix (Ex. $nz <= dimension^2$). This process did not take into account that, when large systems are used, the operation ($dimension^2$) can reach a value greater than the maximum value held by an integer variable, causing an overflow and the program's execution to stop. This and other validations were removed. Now it is assumed the initial matrix is correct from the beginning.

2. The execution of a process, called "Recover", that was not affecting the preconditioner construction, was causing execution crashes so it was disabled.

After these issues were fixed, the original program was able to execute all the matrices defined within the baseline. See 4.2.

Once the initial testing was done it was proceeded to a deeper analysis of the implementation taking the mathematical description of the algorithm as a guide. Two things were noted within the algorithm:

1. The last element in the last row of the initial matrix was never used, it was trimmed from the memory structure.

2. There is a part within the Monte Carlo process, where each of the preconditioner's rows is being calculated. There was a case when a pointer was randomly set to an empty row causing the algorithm to behave

wrongly and the number of elements in the final matrix (the preconditioner) to be smaller than it should be. There is no description, within the algorithm, about what has to be done when this happens so it was decided the algorithm should pick, randomly, a different not empty row.

4.2 Defining a Baseline.

We have selected a matrix set from 3 different sources: The Matrix Market [9], The University of Florida Sparse Matrix Collection [14] and some real-life problems from our collaborators.

The following set is composed by matrices of different sizes, sparsity and symmetry.
In all the cases matrices are non-diagonally dominant (4.1).

$$|a_{ii}| <= \sum_{j \neq i} |a_{ij}| \quad \forall i \tag{4.1}$$

Matrix	Dimension	Non-zeros	Sparsity	Symmetry
Appu	$14,000 \times 14,000$	1,853,104	0.95%	non-symmetric
Na5	$5,832 \times 5,832$	305,630	0.46%	symmetric
Nonsym_r5_a11	$329,473 \times 329,473$	10,439,197	0.01%	non-symmetric
Rdb2048	$2,048 \times 2,048$	12,032	0.29%	non-symmetric
Sym_r3_a11	$20,928 \times 20,928$	588,601	0.13%	symmetric
Sym_r4_a11	$82,817 \times 82,817$	2,598,173	0.04%	symmetric

Table 4.1: Matrix set.

The next step was to generate a process to quantify the quality of the resulting preconditioner when applied to each of the matrices in the set. This is done by multiplying the approximate inverse (*i.e the preconditioner*) ($\hat{A^{-1}}$) in both sides (4.3) of the original system (4.2) in order to generate an equivalent but simpler system (4.4), then solving this new system using a solver and use the resulting vector, which is an approximation of x ($\hat{x}$), back in the original system to measure its fit (4.5).

$$Ax = b \tag{4.2}$$

$$\hat{A^{-1}}Ax = \hat{A^{-1}}b \tag{4.3}$$

$$\hat{I}x = \hat{A^{-1}}b \tag{4.4}$$

$$Error = \frac{1}{n}\sum_{i=0}^{n} |A_i\hat{x}| - |b_i| \tag{4.5}$$

4.3 Improving the Monte Carlo-based algorithm.

In this chapter all the optimizations applied to the Monte Carlo-based algorithm, are described.

4.3.1 Broadcast slowdown.

The main problem reported by the users of the current program was the non-sufficient scalability of it, thus this was the first problem analysed.

Using the original version of the code (with minimal modifications to avoid crashes) and different number of cores configuration, all the matrices in the set were executed and the algorithm's behaviour analysed.

Some of these scalability results are shown in Figures 4.1 and 4.2. In all cases an inflexion point can be observed when going from 16 to 32 cores, this could be due to the overhead introduced by MPI when going from 1 compute node (16 cores) to more than one (32 cores).

BSC performance tools Extrae [12] and Paraver [26] were used to test the last hypothesis. Results are shown in Figures 4.3 and 4.4. Note how the broadcast operation gets highly increased, going from 0.45% to 62.24% of the total execution time.

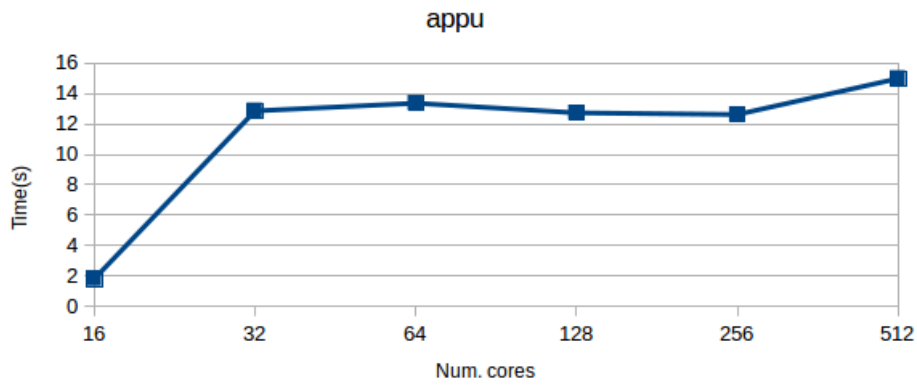

Figure 4.1: Scalability test in matrix Appu: Time to calculate the preconditioner for different number of cores (16 - 512).

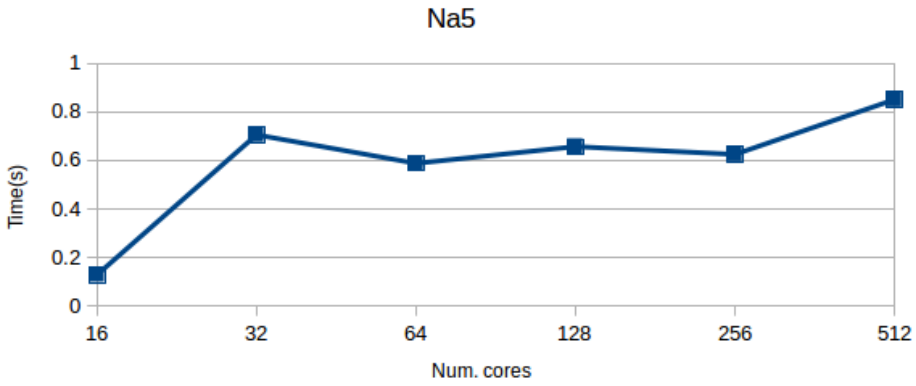

Figure 4.2: Scalability test in matrix Appu: Time to calculate the preconditioner for different number of cores (16 - 512).

Even when an increased overhead, when using more than 1 compute node, is common in parallel applications, the increment seems bigger than what one would expect. A deeper analysis was done in order to find other issues related to this behaviour.

Outside MPI	MPI_Send	MPI_Recv	MPI_Bcast	MPI_Comm_rank	MPI_Comm_size
96,19 %	-	2,68 %	0,45 %	0,52 %	0,00 %

Figure 4.3: Percentage of used by MPI operations in the execution of Appu matrix within 1 compute node (16 cores).

Outside MPI	MPI_Send	MPI_Recv	MPI_Bcast	MPI_Comm_rank	MPI_Comm_size
37,08 %	-	0,37 %	62,24 %	0,13 %	0,00 %

Figure 4.4: Percentage of time used by MPI operations in the execution of Appu matrix within 2 compute nodes (32 cores).

By analysing paraver's traces (Figures 4.5 and 4.6) we can note how the broadcast operation is actually divided in several parts, look at the green flags on the top of the trace, they mark the beginning of each MPI operation (in this case the different broadcast).In the case of the 32-cores execution, the first broadcast is taking way longer that the remaining ones, while in the 16-cores case, the first broadcast takes a shorter time compared with the consecutive ones. Yellow zones represent the broadcast operations, while the blue ones represent the computation not related to MPI.

Figure 4.5: Broadcast operations in the execution of Appu matrix within 1 compute node (16 cores).

Figure 4.6: Broadcast operations in the execution of Appu matrix within 2 compute nodes (32 cores).

This observation enabled us to identify that the increment in time was not related to the piece of data being broadcasting but to the moment in which this operation takes place. This hypothesis was easily proved by switching the positions of the first and second broadcast operations within the code, and observing the exact same behaviour (The first operation was always the longer one).

A further analysis of the code showed that the slowdown in the broadcast operation was related to several memory reallocations (*i.e. calls to the realloc function*). This problem is discussed in [6], stating that when using the OpenMPI library ,the OS apply a "lazy" policy regarding to the release of not-used memory after a memory reallocation. This makes the system running out of memory and force it to use swap memory which is known to be very slow.

Replacing the *realloc()* calls was not a trivial task, given that in some processes the resulting size of a memory structure remains unknown until you actually require that memory (See section 2.2.5). What has been done is to make an estimation of the memory required, adding an extra margin, and make just one memory allocation.

The evaluation of this modification can be found in 5.1.

4.3.2 Memory usage.

It is well known that "application runtimes today are increasingly dominated by memory operations" [30] and it is not only about the quantity of those operations but also because they are know to be expensive in terms

of execution time. In order for a program to be efficient, programmers must minimize the number of memory access. A major change applied within the Monte Carlo process was carried out within this context.

In the original code when a row, for the final preconditioner matrix, its being calculated the algorithm first allocate memory for an empty row that is as big as the matrix's size. Then this row is randomly populated with only few values ($<<$ matrix size). Finally the row is accessed item by item to locate non-zero values which will be used, along with their respective indexes, in the next procedure.

Accessing the whole row, knowing that we are looking only for few items its an inefficient approach. In the new version an extra memory structure is created, this way we are able to save both the random value and its index. This values are stored in a consecutive order (like in a COO format). Then the algorithm will have to access the memory structure only as many times as items it has.

Notice that during the row population, more than 1 random value can end up within the same row position in which case they must be summed, This represented a problem in the new version, given that each value has a separated slot. To solve this new problem a further procedure is used ensure that values within the same column index are summed.

To exemplify the memory access reduction let's analyse the case of executing the Monte Carlo-based algorithm with the matrix Appu. This matrix's size is $14000 * 14000$ and if we look at the resulting preconditioner obtained using default parameters ($\epsilon = 1 \times 10^{-1}$ and $\delta = 1 \times 10^{-1}$) we will see that the number of non-zero elements is ≈ 46500, this give us a sparsity of 0.023% (4.6). This means that each row has a average of 3 elements ($3 \approx 14000 * 0.00023$)) so instead of performing 14000 memory accesses (original version) the new version will only do 6 (3 values and 3 indexes).

$$sparsity = 100 * (non - zero\_elements/size^2) \tag{4.6}$$

This approach is evaluated in section 5.2, results and speedup metrics can be found there.

4.3.3 Reducing the broadcast data

There are two methods, described in the algorithm, to produce the probabilistic matrix P which is used during the construction of the Markov Chain:

1. Almost Optimal (MAO). In which the probability to select each of the non-zero values in the matrix A, is relative to the size of its value. This is: the bigger the value, the more chances it has to be selected. See (4.7).

2. Uniform (UM). In this case the probability of all non-zero values in A is the same. See (4.8).

$$p_{ij} = \frac{|a_{ij}|}{\sum_{j=0}^{n} |a_{ij}|} \quad \forall a_{ij} \neq 0 \tag{4.7}$$

$$p_{ij} = \frac{1}{\sum_{j=0}^{n} |a_{ij}|} \quad \forall a_{ij} \neq 0 \tag{4.8}$$

Experiments carried out in [5], shows that "UM needs about 6-10 times more chains to reach the same precision of MAO" but our experiments, carried out in section 5.3, have shown that there is not a significant difference, while using one method or the other when rough precision is used to calculate the estimators ($\epsilon \geq 1 \times 10^{-1}$ and $\delta \geq 1 \times 10^{-1}$).

Using the Uniform distribution (UM) has the advantage that the P matrix can be drastically reduced. It goes from a size of nxn to only n given that when using the MAO approach a probability value must be stored per each non-zero value in the matrix A while, in the case of UM, only 1 probability element is needed for each row (the probability on all the values within the same row is the same).

The reduction of matrix P impacts directly the overall performance given that this matrix is included within the broadcast data and the broadcast process is one of the most time-consuming procedures (See figures 4.7 and 4.8).

Results on the application of this approach can be found in section 5.3.

4.3.4 Merging sequential functions

In the initialization phase of the algorithm there were 4 procedures which needed to access the whole initial matrix B, to calculate matrices A, P, B_1 and other intermediate values. A brief description of those procedures is listed below:

1. calculate_norm: Calculates the norm of the initial Matrix.

2. calculate_bhat: Calculates intermediate matrices B_1 and B_2.
3. calculate_A: Calculates matrix A.
4. calculate_P: Calculates matrix P.

Given the simplification of the matrix P, after applying the modification described in section 4.3.3, and a code analysis carried out, these procedures were able to be merged into 2 larger functions which reuse the memory accesses pattern.

This optimization calculate matrix A directly from B and B_1, making matrix B_2 not longer necessary.

After the code merge this is how the work, carried out in the removed functions, was re-assigned:

1. calculate_norm: *Calculates the norm of the initial Matrix* and part of matrix B_1.
2. calculate_bhat: Calculates the remaining part of B_1 *together with matrices* A *and* P.

After this modification, this sequential stage is not longer worth to parallelize given that overhead induced (broadcasting the initial matrix and merging the results back) would be higher than the time needed for the sequential execution.

The improvements of this approach are shown in section 5.4

4.4 Scalability analysis

The communication overhead induced by MPI plays a very important role in this algorithm, it accounts for a big part of the total execution time, specially the broadcast operation which prevents the algorithm to scale beyond 16 cores (*i.e. 1 compute node*). Even after the modifications described in sections 4.3.1 and 4.3.3.
The time spent within this operation is very significant and it grows with the number of compute nodes used. (See figures 4.7 and 4.8).

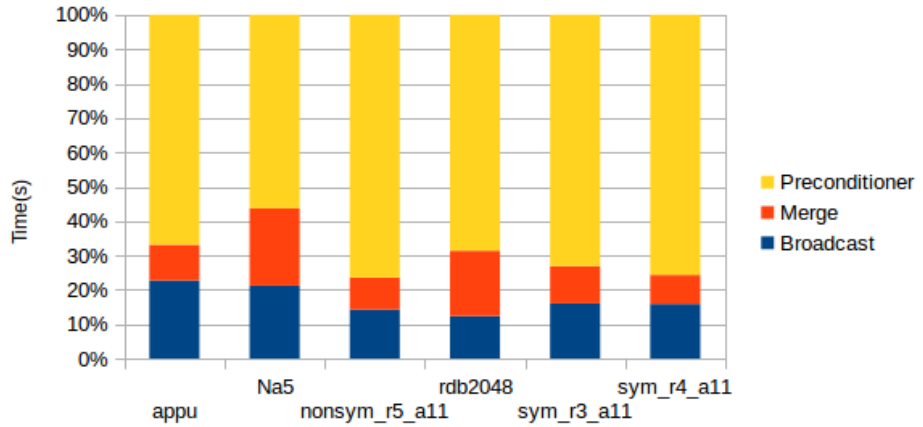

Figure 4.7: Execution time breakdown in a 16 cores execution.

One may think that the reduction in the time for the preconditioner given the parallelism, makes the effect of an increase in time for the broadcast

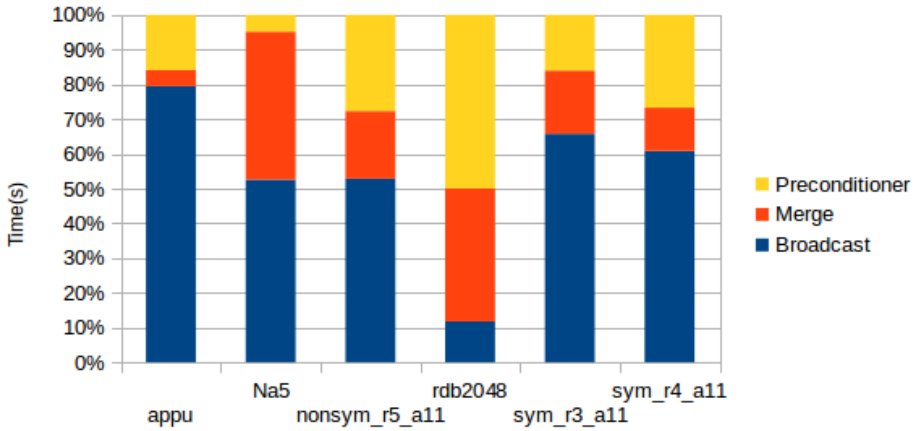

Figure 4.8: Execution time breakdown in a 256 cores execution.

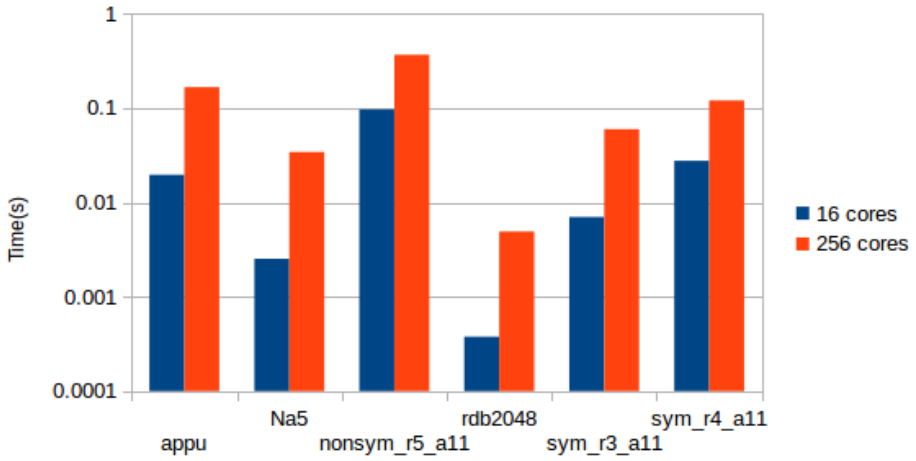

Figure 4.9: Execution time breakdown in a 256 cores execution.

operation, but this is not the case as it can be observed in figure 4.9, where the actual time of the broadcast is shown.

The following approaches are presented as part of the scalability analysis carried out. It is important to mention that they do not provide consistent improvements, reason why are not included in the final implementation.

First, an interesting hybrid(MPI + OpenMP) approach is described and the unexpected performance is explained.
Then a latency-aware broadcast is proposed and its benefits are shown for a specific large case.

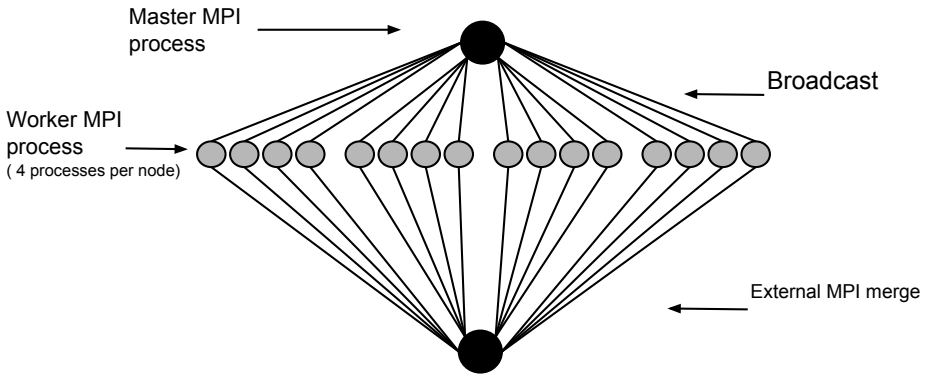

Figure 4.10: Original MPI aproach

4.4.1 An hybrid approach (MPI + OpenMP)

Considering the great overhead induced by the broadcast operation, an hybrid approach (section 2.3.3) of MPI + OpenMP makes a lot of sense. This way the algorithm could only broadcast data to 1 MPI process per node and within the node all the cores will share this data.

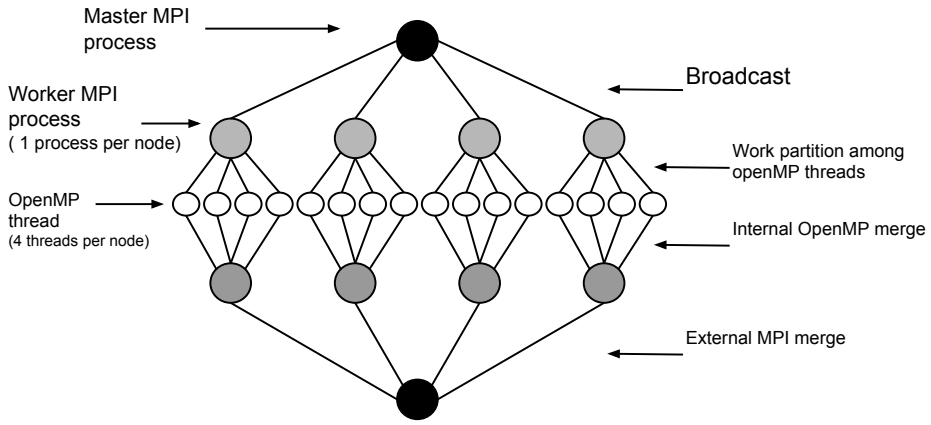

Figure 4.11: Hybrid approach MPI + OpenMP

Notice that in both cases (figures 4.10 and 4.11) the number of cores used remains the same, the difference relies on the number of workers involved in the broadcast operation (less workers are involved, within the broadcast operation, in the hybrid version).

The implementation is based into a *outside to inside* approach in which the initial workload is divided among the MPI processes, then each MPI

process creates its own set of threads and further divide its assigned part of the workload among them. Each of these threads will perform the Monte Carlo process on its workload independently. When all the threads are done with their work they will merge their results into a shared memory structure which will be sent back to the master MPI process. Finally this master process will merge all the results into the final matrix. See (Figure (4.11).

The evaluation of this implementation has shown that the execution time is longer than in the previous version (only MPI) meaning that the overhead (section 2.3.2) implied is greater than the broadcast savings. For this reason we have decided not to continue on this path.

In section 2.3.4, it was mentioned that memory accesses performed in a random way could prevent an algorithm to obtain the common benefits from a shared memory model. This seem to be the case for our Monte Carlo algorithm (its random nature is likely to produce this kind of behaviour).

In figures 4.12, 4.13 and 4.14 we can observe the *miss ratio* (number of cache misses for each 1000 instrucctions) in the L3 cache (which is shared among the different cores) for the Monte Carlo process within three different configurations:

1. 1 MPI process + 1 OpenMP thread: This is a sequential execution (only 1 core is used) this set the baseline for the following cases.
   Miss ratio: (0.38 - 0.56).

2. 1 MPI process + 2 OpenMP threads: Here we can see a great increment in the miss ratio when 2 cores are used within a shared memory context.
   Miss ratio: (1.03 - 1.18).

3. 2 MPI process + 1 OpenMP thread: We can see that using 2 cores, like in the previous case, within a distributed memory context, keeps the miss ratio in the same level as in the sequential case.
   Miss ratio: (0.37 - 0.56).

There has been noted that the great increment observed in the miss ratio, within the shared memory context, provoke an IPC degradation in such a way that adding more threads to the execution does not adds any improvement in the application's performance.

Based on the observations of [22] and the previous experiment, we believe that avoiding the random pattern in the memory accesses can help to boost the performance of this algorithm.

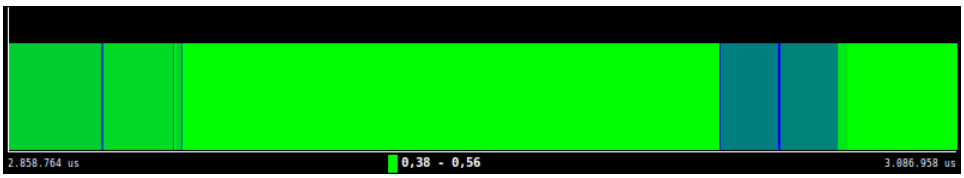

Figure 4.12: L3 cache miss ratio within the Monte Carlo process for 1 MPI process and 1 OpenMP thread. (Miss ratio: (0.38 - 0.56)).

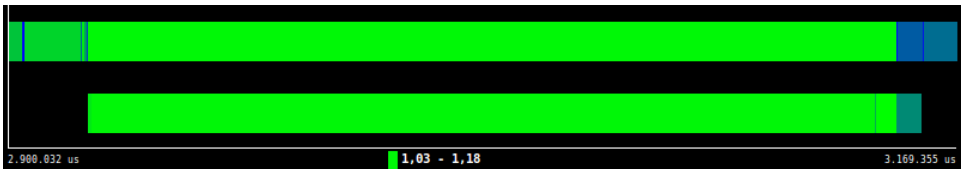

Figure 4.13: L3 cache miss ratio within the Monte Carlo process for 1 MPI process and 2 OpenMP threads.(Miss ratio: (1.03 - 1.18)).

4.4.2 Two-step broadcast

There are several broadcast algorithms that can be implemented by MPI (basic linear, chain, pipeline, split binary tree, binary tree and binomial tree). *"A runtime decision module can be used to select the best algorithm and tuning parameters, according to message size, communicator size, and other input variables"*. Unfortunately these collective operations have been designed to fit single-processor cluster [25]. This means that they do not take into consideration the intra-node and inter-node communication different latencies, which play an important role in the broadcast performance.

During the experimental phase it has been observed that when reducing the number of MPI processes while keeping the same number of compute nodes *i.e. having less processes per node*, the broadcast time is reduced.

Taking into consideration the observations of the last two paragraphs, a new implementation of the broadcast operation has been designed. Instead of having a unique broadcast, this operation is now carried out in two steps:

- *First step.* The master node broadcast the initial data only to 1 process in each node.
- *Second step.* The process receiving the data in the previous step will further broadcast this date to the remaining processes within its node. This is actually a memory copy operation given that all processes are located in the same compute node.

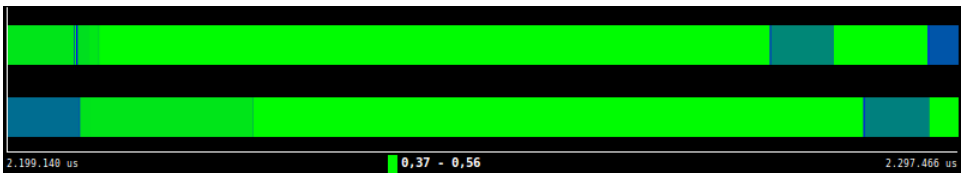

Figure 4.14: L3 cache miss ratio within the Monte Carlo process for 2 MPI process and 1 OpenMP thread. (Miss ratio: (0.37 - 0.56)).

In order to perform different broadcast operations involving different groups of workers, two extra MPI communicators must be created.

1. Head communicator. It groups all the representative workers within each of the nodes. *It includes 1 worker per compute node.* Only one communicator of this type is created, its size is equal to the number of compute nodes involved in the execution.

2. Local communicator. Includes all the workers which belong to a same compute node. There will be as many local communicators as compute nodes. The size of a local communicator is equal to the number of workers per node.

These communicators are created at the beginning of the execution. The broadcast operation is then executed first by those processes belonging to the head communicator. After that, all processes execute the broadcast operation using local communicators.

In Figures 4.15 and 4.16 we can see how this approach is performing better for large number of cores, but consistently worse in the optimal cases (the shortest time).

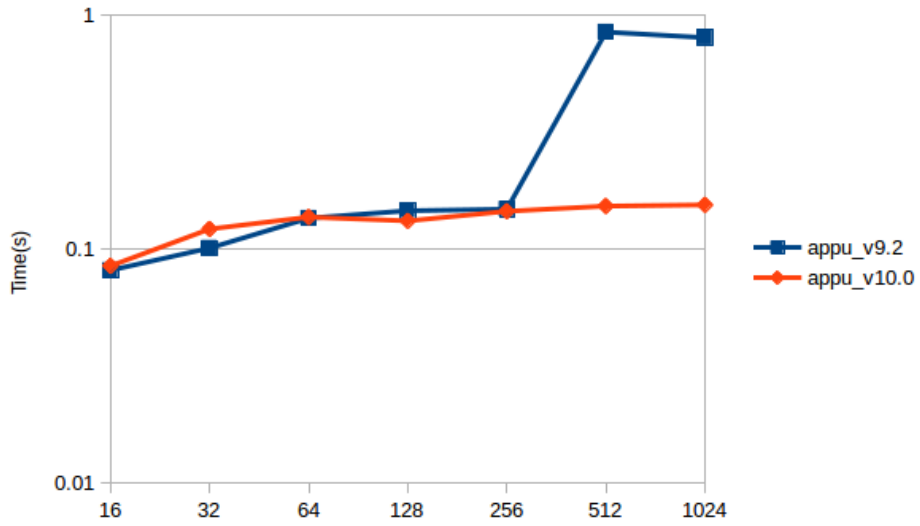

Figure 4.15: Scalability comparison for the two-step broadcast(v10.0) in the appu matrix.

In Figure 4.17 we have used a matrix which does not belong to the matrix set (Table 4.1) only to observe the behaviour in large cases. In this case the fastest time is achieved by the two-step broadcast.

This approach was discarded given that, for the optimal case, within the defined matrix set(Table 4.1), it does not provide a consistent improvement.

Nevertheless, we think this algorithm could be useful if used with a matrix set of large matrices.

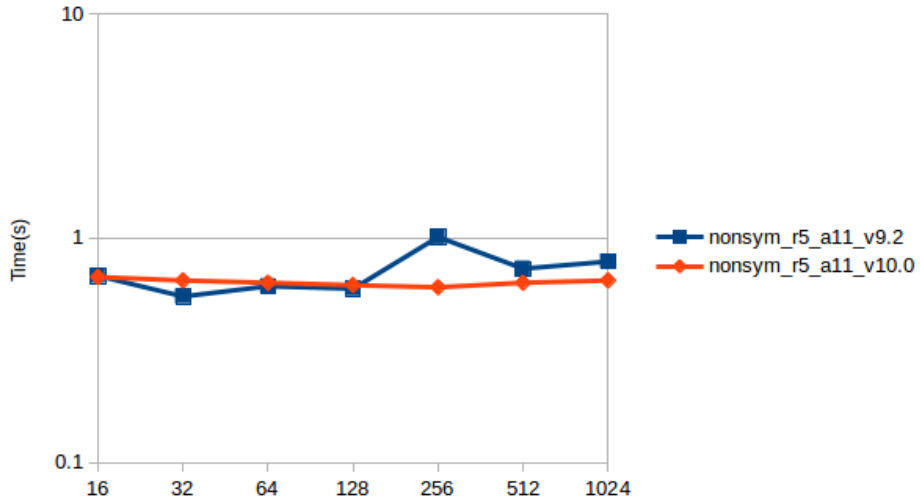

Figure 4.16: Scalability comparison for the two-step broadcast(v10.0) in the nonsym5_r5_a11 matrix.

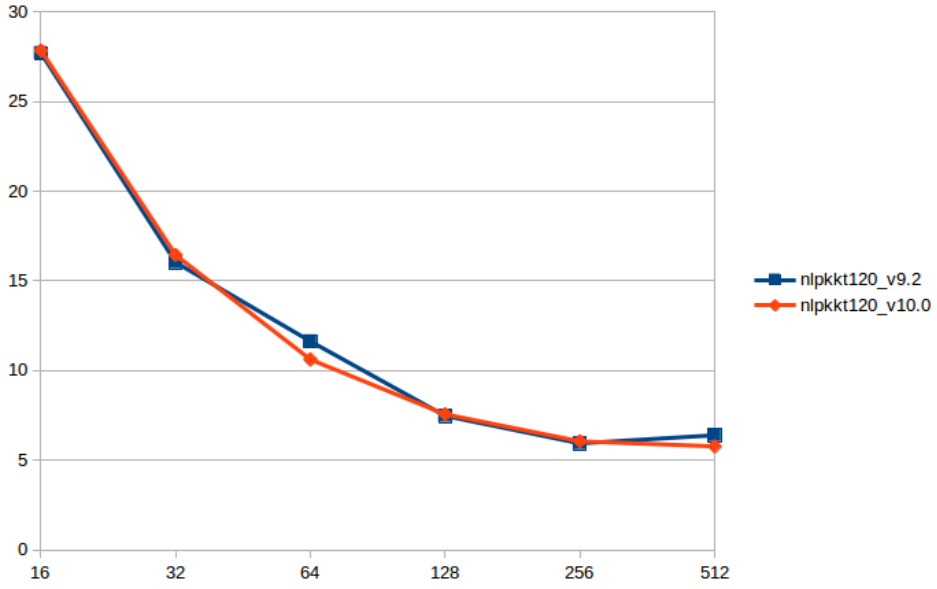

Figure 4.17: Scalability comparison for the two-step broadcast(v10.0) in a very big matrix (3.5M x 3.5M)

Chapter 5

Results and evaluation of the work.

In this chapter we will show the results obtained during the experimental phase. The objective is to demonstrate the improvement evolution of the code after applying the modifications previously described.

It is important to mention that the improvements are shown in a consecutive fashion (*i.e. each new version is compared with the previous one.* This way the reader could appreciate the particular improvements of each modification applied. In section 5.5 a comparison between the best version and the original code is presented. In section 5.6 a comparison between the best Monte Carlo version and the MSPAI application is carried out.

In order to get comparable executions between different versions of the Monte Carlo algorithm, the random generation is configured in such a way that Markov Chains (section 2.2.2) are unique for different matrices but not for different executions of the same matrix. Also the following *input parameters* have been used in all the executions, but the one compared with MSPAI.

$$\epsilon = 1 \times 10^{-1} \quad \delta = 1 \times 10^{-1} \quad \alpha = 5 \times 10^{1}.$$

The time presented always corresponds to the algorithm execution and the communications. The time needed to read the initial matrix as well as the time for writing the preconditioner into a file, are never included in the results.

5.1 Broadcast slowdown evaluation.

In section 4.3.1 a problem within the broadcast operation and its relation to the realloc() function is described. Here a comparison between an execution, before and after the realloc() calls removal is done to show the impact of this approach within the broadcast time and the total execution time.

An important aspect noted during the evaluation of this approach, was the fact that single-node executions were also affected by this problem. By comparing figures 5.1 and 5.2 it can be noted the improvement, due to this modification, in a single node execution.
Notice that both figures have the same time scale. This means that the execution in figure 5.2 finishes within 3/4 of the time required by the execution depicted in figure 5.1. (See marker **2** in figure 5.2).

The main reduction in time in this case is observed during the initial stage, when the intermediate matrices are being created and the calls to the *realloc* function take place. (See marker **1** in figures 5.1 and 5.2).

Figure 5.1: Single node(16 cores) execution of appu matrix **before the optimization**. Only the master and the first worker process are shown.

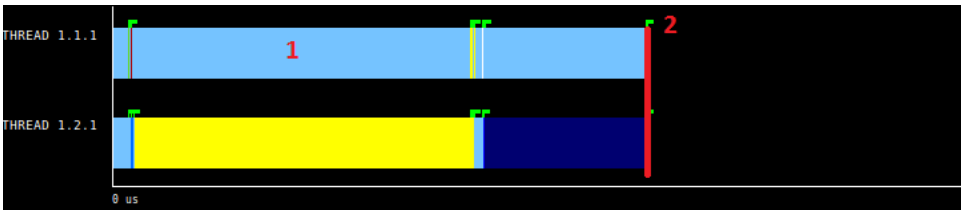

Figure 5.2: Single node(16 cores) execution of appu matrix **after the optimization**. Only the master and the first worker process are shown

When going beyond 16 cores, meaning that more than 1 compute node is involved, we can notice how the problem gets magnified. See Figures 5.3 and 5.4. Notice that both figures have the same time scale. In this case the main difference is observed in the broadcast operation. See marker **1** in figures 5.3 and 5.4.

Figure 5.3: Execution of appu matrix in 2 compute nodes(32 cores) **before the optimization**. Only the master and the first worker process are shown.

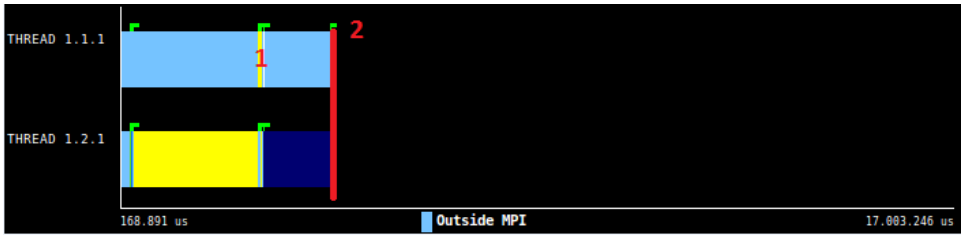

Figure 5.4: Execution of appu matrix in 2 compute nodes(32 cores) **after the optimization**. Only the master and the first worker process are shown.

Finally in Figure 5.5 the improvements for all the matrices in the set are shown. Our experiments show an average speedup of **7.8x** with the previous version.

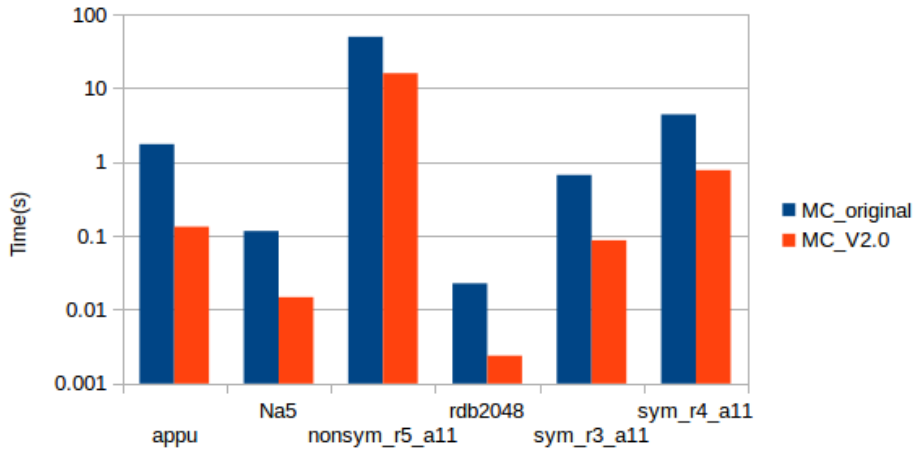

Figure 5.5: Execution time in 16 cores before / after removing realloc calls (MC_original / MC_V2.0).

5.2 Memory usage evaluation.

In this section, time improvements resulting from the memory usage modification applied to the code, are demonstrated. This modification is described in detail in section 4.3.2.

Figure 5.6 shows the reduction in time for the new implemented version (v6.0). But also, cases without improvements are observed (matrices rdb2048 and Na5).

At the end of section 4.3.2 we talked about the overhead induced by this new implementation. We think that this is the reason for the results obtained on matrix Na5 and rdb2048. Also it has been observed that matrix rdb2048 is the one which produces the densest preconditioner. Dense preconditioners are bad for this new approach which is based on the low sparsity observed in the resulting preconditioner.

An important aspect to observe in figure 5.6 is how the improvement grows with the size of the matrix, regardless of (sym_r3_a11), we can notice how the smallest case (rdb2048) has a significant negative speedup, then the next case (Na5) present a smaller negative speedup, but from that point all the following cases present an incremental improvement.

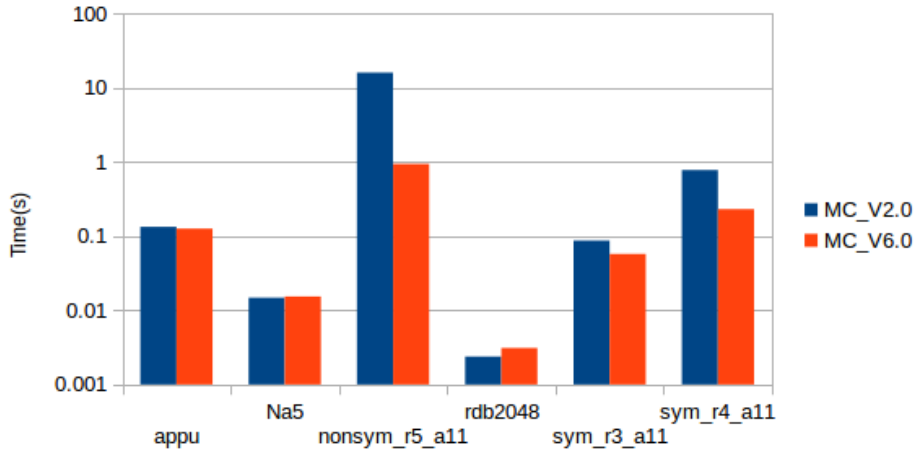

Figure 5.6: Execution time in 16 cores, before / after applying the memory usage modification (MC_V2.0 / MC_V6.0) .

At the end an average speedup of **4.12x** was achieved with this new version in comparison with the one at the previous section.

5.3 Reducing the broadcast data evaluation.

The broadcast data at the beginning of the execution has a big influence in the total execution time. In section 4.3.3 we have discussed an approach to reduce the amount of broadcast data.

This new approach relies on the fact that using any of the two different distributions (Uniform and Almost Optimal), in order to generate the random behaviour in the Monte Carlo process, has little impact on the algorithm's number of computations needed, precision attained and ultimately results, but it has a big impact on the amount of data to be broadcast. Figure 5.7 shows the difference, in terms of the error(4.5), of using one distribution or the other.

Given the logarithmic scale used, it can be observed that the difference between using one distribution or the other is very small. The largest difference happens with matrix sym_r4_a11 and it is only about 1.8×10^{-4}.

With the last observation done, it has been proposed to set the Uniform distribution as the default one, thus allowing a significant reduction in the broadcast data. This reduction will influence directly the total execution time.

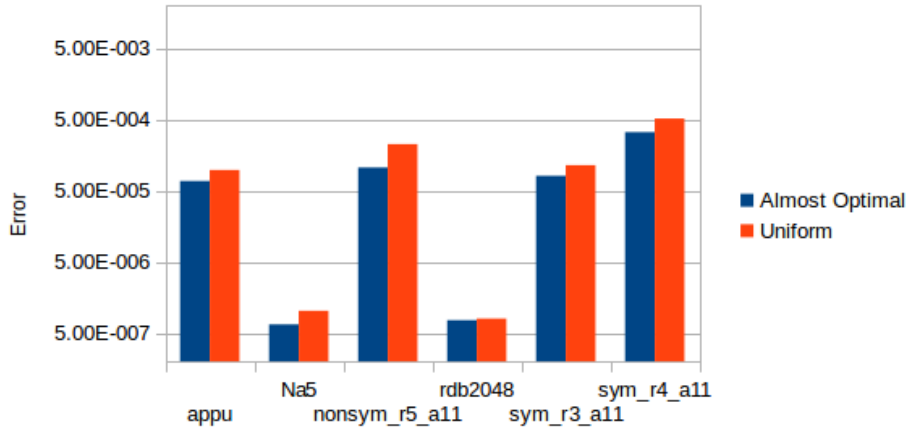

Figure 5.7: Error calculation when using Uniform and Almost Optimal distributions in MC_V6.0 with 16 cores.

Figure 5.8 shows a consistent improvement in the new version when the reduction in the broadcast data is applied, getting an average improvement of **1.14x** over the previous one.

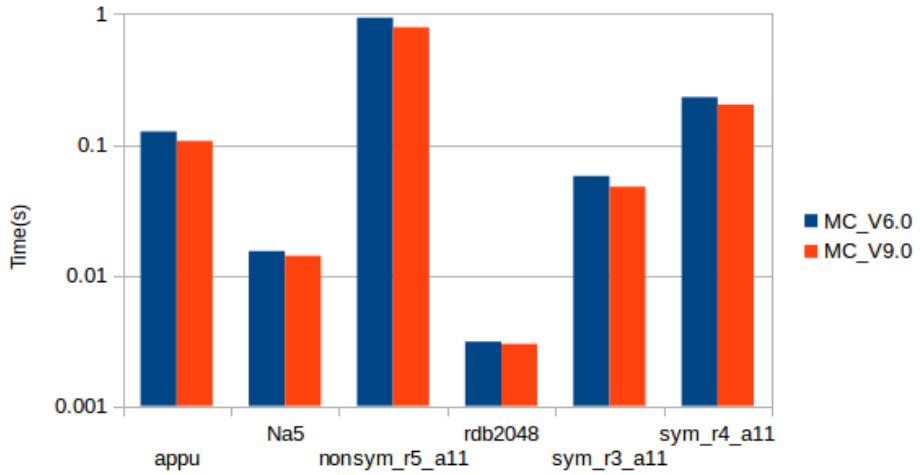

Figure 5.8: Execution time in 16 cores before / after the broadcast reduction (MC_V6.0 / MC_V9.0), both using an Uniform distribution.

5.4 Merging sequential functions evaluation

This section presents the results of applying the optimization described in section 4.3.4.

The improvements in time observed in figure 5.9 are the result of merging some of the initialization functions with the objective of reducing the amount of memory accesses.

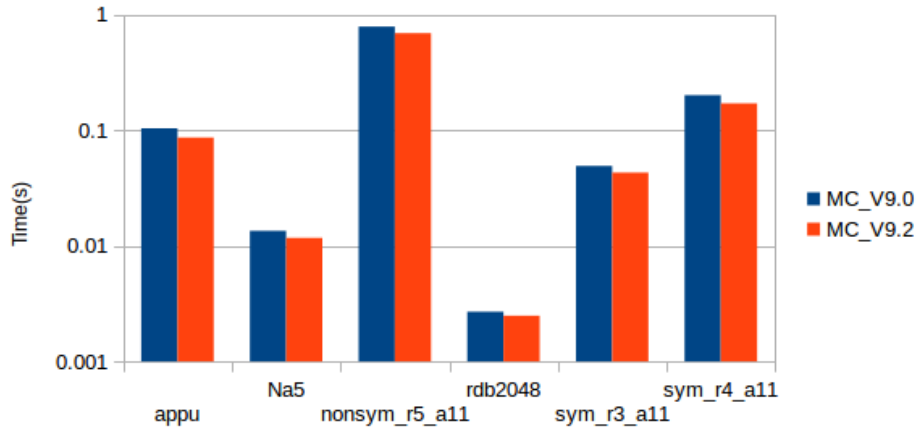

Figure 5.9: Execution time in 16 cores before / after merging sequential functions (MC_V9.0 / MC_V9.2) .

With this modification in the algorithm, a consistent average improvement of **1.15x** has been obtained.

5.5 Total improvement

In previous sections the improvement of each of the modifications applied to the algorithm has been shown separately, comparisons have been carried out for consecutive versions and now it is time to present how these accumulative improvements are compared with the original version of the code.

In Figure 5.10 it can be observed the total improvement achieved by all the optimizations applied to the implementation.

An average of **25x** improvement is achieved. It can be noted that the improvement grows with the size of the matrix, having an impressive **70x** improvement for the biggest matrix (*nonsym_r5_a11*) in the set. (See table 4.1).It is important to emphasize that both versions run under the exact same conditions and use the same amount of computational resources, the

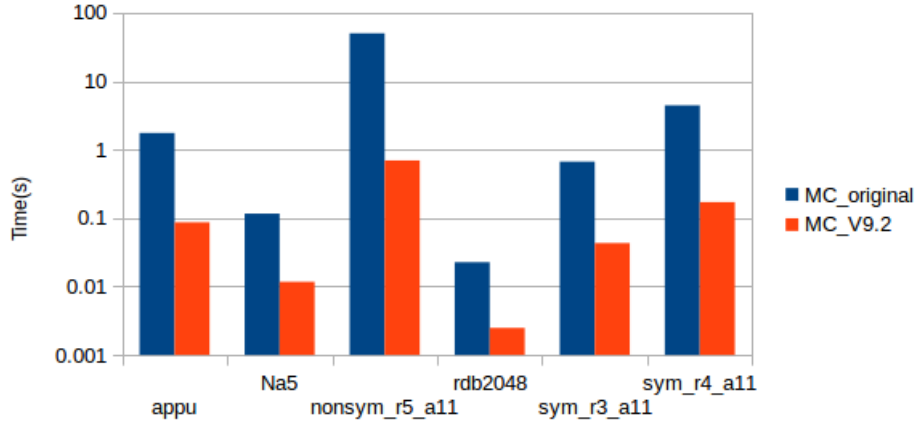

Figure 5.10: Execution time comparison of the original_MC and version 9.2 using 16 cores.

differences are only in the algorithm implementation.

The numerical results presented in this section can be found in table 7.1 at the Annex 1.

5.6 MSPAI comparison

In this section the optimized Monte Carlo algorithm, is compared with the latest version of the MSPAI application. (section 2.1.2).

Parameters of the Monte Carlo implementation have been adjusted accordingly ($\epsilon = 7 \times 10^{-1}$ $\delta = 1 \times 10^{-1}$ $\alpha = 5.$) to produce comparable results with those obtained with MSPAI's default configuration.

5.6.1 Scalability comparison.

Scalability plots (Figures 5.11 to 5.16) are presented to show the behaviour of both algorithms when the number of cores is scaled. In all these plots a metric called *MC_nocomm* is shown to provide an insight of the scalability of the Monte Carlo process itself (regardless the communication overhead and the initialization time).

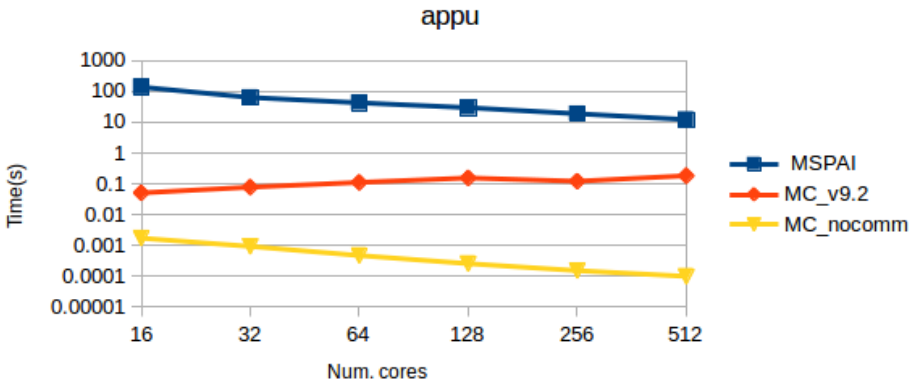

Figure 5.11: Scalability comparison MSPAI and MC for matrix appu.

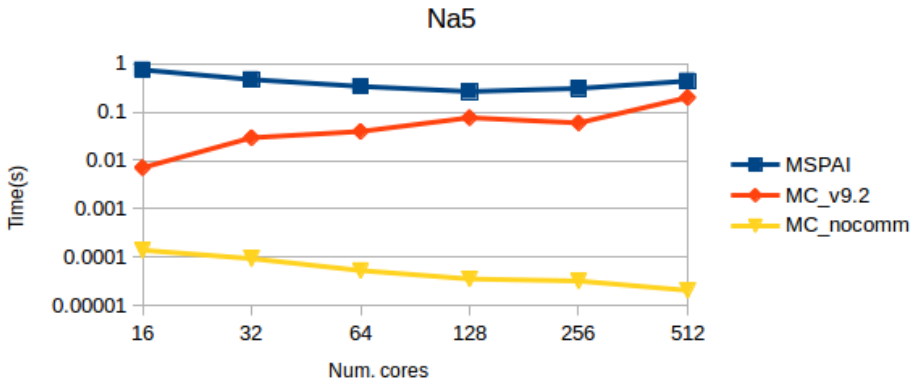

Figure 5.12: Scalability comparison MSPAI and MC for matrix Na5.

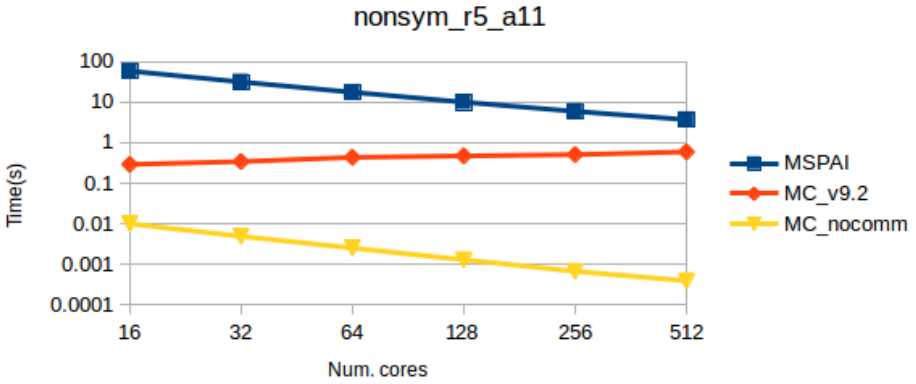

Figure 5.13: Scalability comparison MSPAI and MC for matrix nonsym_r5_a11.

It is quite obvious that the Monte Carlo algorithm performs several times faster than MSPAI, despite the communication hampering the scalability, which is mainly affected by the 3 following aspects:

1. The communication overhead, (discussed in sections 4.4.1 and 5.3).
2. The optimizations applied, reduce the computational effort needed, leaving less room for scalability.
3. The reduction in the number of iterations, to target a sparse and efficient preconditioner, diminishes the computation needed, producing the same effect that the previous point.

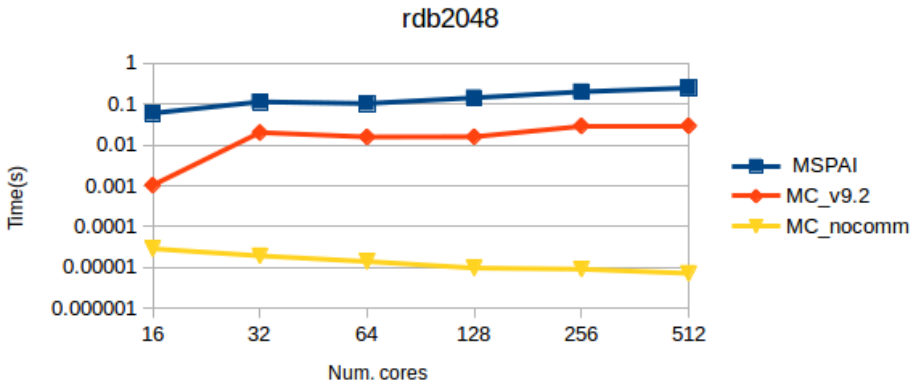

Figure 5.14: Scalability comparison MSPAI and MC for matrix rdb2048.

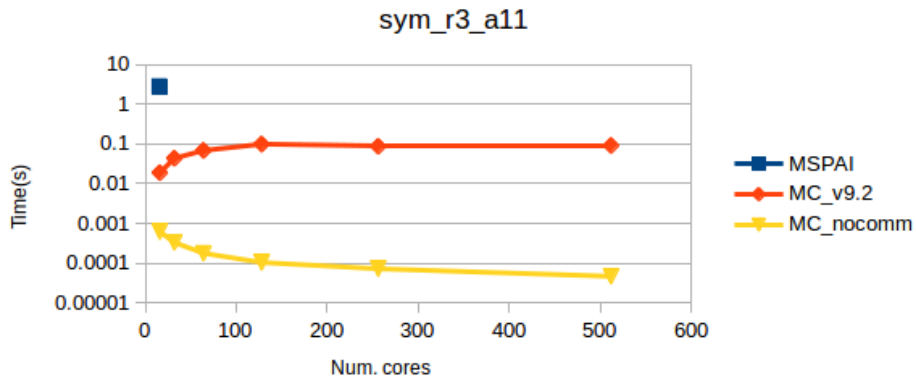

Figure 5.15: Scalability comparison MSPAI and MC for matrix sym_r3_a11.

In the case of MSPAI, scalability issues regarding to small matrices (figure 5.14 and 5.12) can be also found. This together with that, discussed above, are clear examples of *Amdahl's law*.

"Even when the fraction of serial work in a given problem is small, say, s, the maximum speedup obtainable from even an infinite number of parallel processors is only l/s. [18]"

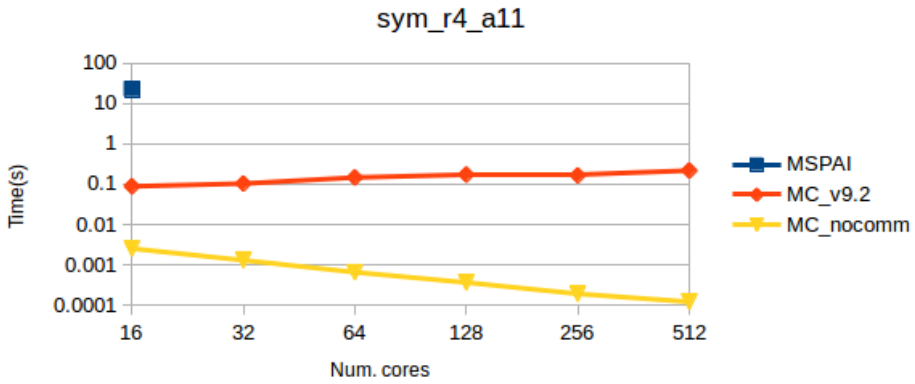

Figure 5.16: Scalability comparison MSPAI and MC for matrix sym_r4_a11.

Besides the large difference in performance, for the symmetric matrices, it can be observed that MSPAI does not converges when more than 16 cores are used.

Figure 5.17 summarizes the scalability plots by showing the *shortest time* achieved, for the preconditioner calculation, by each of the algorithms. The number of cores used to obtain this *shortest time* is used later as a metric in figure 5.20.

The numerical results presented in this section can be found in table 7.2 at the Annex 1.

5.6.2 Quality and efficiency comparison.

Recalling section 2.1, *"A good balance between quality and time of construction is the key for a good preconditioner"*.

We have already measured the time of construction, now we will evaluate the quality of the preconditioner.

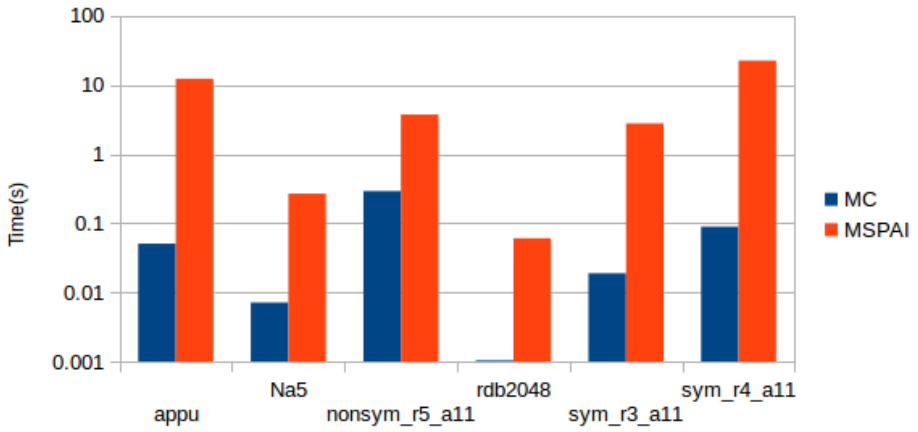

Figure 5.17: Shortest execution time achieved during the preconditioner calculation.

A metric must be selected to measure the quality of the preconditioner but also its efficiency. A good choice of such a metric is the time needed by the solver to find the solution for the preconditioned system, such metric is shown in figure 5.18.

Other metrics like the number of iterations required by the solver to calculate the solution of the preconditioned system, reflect the quality but not the efficiency. In that sense it has been observed that in many cases the number of iterations required by Monte Carlo preconditioned systems is smaller than in the case of those preconditioned by MSPAI.

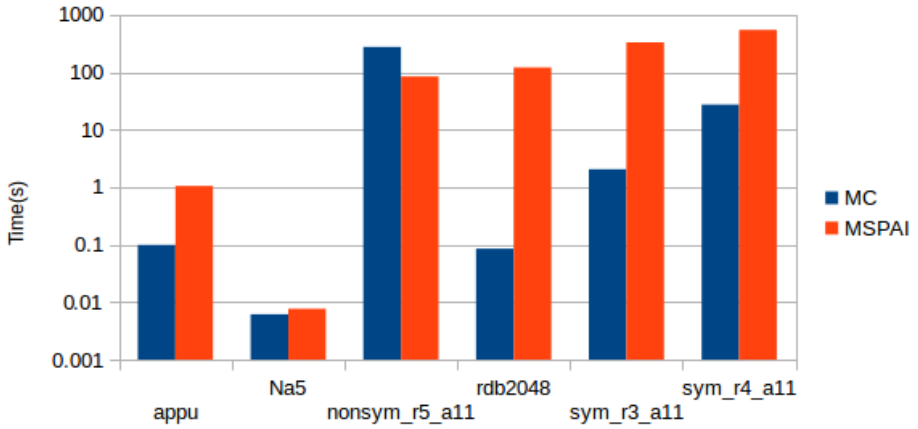

Figure 5.18: Time required by the solver to find the solution for the preconditioned system.

To provide a view of the overall time, we provide figure 5.19 in which the times of the preconditioner construction and the time needed by the solver (figures 5.17 and 5.18) are added. Here we can see that sometimes the time invested in the quality of the preconditioner can be compensated with a reduction in the solver execution. An example of this is the case of matrix *nonsym_r5_a11* which takes longer for the construction but in the overall it is faster.

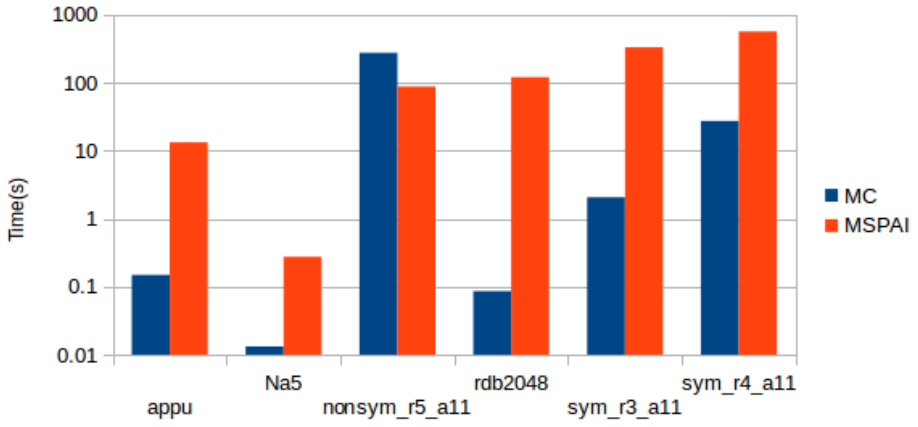

Figure 5.19: Total time = Preconditioner construction time + Solver execution time.

Observe that MSPAI only performs better in the case of a non-symmetric matrix and Monte Carlo performs much better than MSPAI for the symmetric matrices as well as in most cases for non-symmetric matrices too. Furthermore the errors obtained by MSPAI for the symmetric matrices are 3 orders of magnitude larger than the common values which are always smaller than 6×10^{-4}.

Finally an observation is done in terms of *resource usage* that directly translates into *energy efficiency*.

Taking the fastest executions for the preconditiner construction (figure 5.17) we show the number of cores used for such executions. Notice that the largest difference is of 32 times more cores, this means that MSPAI is using 512 cores (32 compute nodes) while Monte Carlo uses only 16 cores (1 compute node).

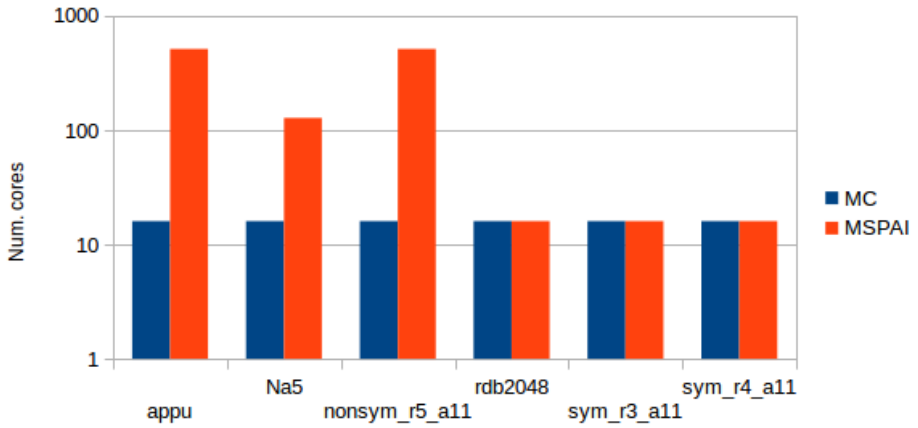

Figure 5.20: Number of cores used for the fastest execution.

After observing the previous analysis, we can say that the Monte Carlo algorithm is a great candidate for calculating SPAI preconditioners given the following considerations:

- It can deal with diverse matrices (symmetric and non-symmetric).
- Produces reliable results.
- Take considerable short time, even for large matrices.
- It is energy efficient.

The numerical results presented in this section can be found in tables 7.3 and 7.4 at the Annex 1.

Chapter 6

Conclusions

The research carried out in this Msc. Dissertation has achieved the stated goals as well as the major objectives outlined in section 1.3. In more details:

- An improved and enhanced Monte Carlo algorithm that produces a SParse Approximate Inverse-based preconditioner has been presented and described with great detail. This fulfils the overall objective of the dissertation.
- The implementation, of this method, has been *optimized* by a factor of 25x in average, and 70x in the best case. These gains in performance were obtained directly by improvements within the algorithm. It is important to highlight that the exact same conditions (computational resources) were used to compare the original and the enhanced version. This achievement fulfils the second goal of this work (Improve the performance).
- Known issues as well as other silent errors, were found, corrected and described in detail. This meets the first goal (Eliminate errors).
- The approach of an hybrid version of MPI + OpenMP was developed in order to adapt the code to contemporaneous architectures. Results of the evaluation of this version made evident the need for further research in quasi-Monte Carlo methods like [1] which could be used to boost the performance by improving the memory access patterns.
- Scalability issues related to the broadcast operation were analysed using a two-step approach. The usability of this method was demonstrated for cases in which very large matrices are used. This together with the previous point, accomplish the third goal of this work (Analyse the scalability).
- A detailed comparison was carried out between the Monte Carlo algorithm and the *state-of-the-art* MSPAI one known as the main accepted SPAI deterministic preconditioner. Results have shown that in

general the parallel Monte Carlo algorithm outperforms the MSPAI approach. This meets the fourth goal (objective) (Compare with the state-of-the-art)

- Comparison between different approaches was carried out with precision given the methodology proposed to quantify the error for a given solution.
- Functions within the code are now used to identify the different stages of the algorithm, making the program easier to understand and modify. Finally this complies with the last goal of this work (Code refactoring).

Chapter 7

Future work

The communication overhead, observed within the Monte Carlo algorithm, needs to be mitigated. Further analysis of the merge stage of the algorithm is needed and a more sophisticated implementation such as [25] or [19] can be applied within the broadcast stage.

Further investigation on quasi-Monte Carlo methods [1] and other techniques to reduce the effect of random patterns within the memory access, are necessaries to make the algorithm able to benefit from *shared memory* models.

The application of other programming models like OmpSs or CUDA are also encouraged, always having in mind the random memory access nature of the algorithm.

Finally, the recover mechanism (2.21) need to be modified to make it (if possible) more suitable for parallelization.

Bibliography

[1] V Alexandrov, O Esquivel-Flores, S Ivanovska, and A Karaivanova. On the preconditioned quasi-monte carlo algorithm for matrix computations. In *International Conference on Large-Scale Scientific Computing*, pages 163–171. Springer, 2015.

[2] Vassil Alexandrov, Emanouil Atanassov, Ivan Dimov, Simon Branford, Ashish Thandavan, and Christian Weihrauch. Parallel hybrid monte carlo algorithms for matrix computations. In *International Conference on Computational Science*, pages 752–759. Springer, 2005.

[3] Vassil Alexandrov and Oscar A Esquivel-Flores. On efficient monte carlo preconditioners and hybrid monte carlo methods for linear algebra. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, page 6. ACM, 2015.

[4] Vassil Alexandrov and Oscar A Esquivel-Flores. Towards monte carlo preconditioning approach and hybrid monte carlo algorithms for matrix computations. *Computers & Mathematics with Applications*, 70(11):2709–2718, 2015.

[5] VN Alexandrov. Efficient parallel monte carlo methods for matrix computations. *Mathematics and computers in Simulation*, 47(2):113–122, 1998.

[6] Josh Aune. openmpi realloc() holding onto memory when glibc doesn't., Aug 2007.

[7] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[8] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[9] Ronald F Boisvert, Roldan Pozo, Karin Remington, Richard F Barrett, and Jack J Dongarra. Matrix market: a web resource for test matrix

collections. In *Quality of Numerical Software*, pages 125–137. Springer, 1997.

[10] Simon Branford. *Hybrid Monte Carlo methods for linear algebraic problems*. PhD thesis, Reading University, 2008.

[11] Franck Cappello and Daniel Etiemble. Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12. IEEE, 2000.

[12] Barcelona Supercomputing Center. Extrae user guide, November 2015. www.bsc.es/sites/default/files/public/computer_science/performance_tools/extrae-3.2.1-user-guide.pdf.

[13] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[15] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[16] L Grigori. Sparse linear solvers: iterative methods, sparse matrix-vector multiplication, and preconditioning. 2015.

[17] Marcus J Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[18] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[19] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[20] Mark Hoemmen, R Vuduc, and R Nishtala. Bebop sparse matrix converter. *University of California at Berkeley. Web*, 2011.

[21] Thomas Huckle, Alexander Kallischko, Andreas Roy, Matous Sedlacek, and Tobias Weinzierl. An efficient parallel implementation of the mspai preconditioner. *Parallel Computing*, 36(5):273–284, 2010.

[22] Alex Hutcheson and Vincent Natoli. Memory bound vs. compute bound: A quantitative study of cache and memory bandwidth in high performance applications. 2011.

[23] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *IFIP International Conference on Network and Parallel Computing*, pages 266–275. Springer, 2008.

[24] Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM, 2003.

[25] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack J Dongarra. Hierknem: an adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 970–982. IEEE, 2012.

[26] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. mar, 1995.

[27] Janko Straßburg. *On hybrid and resilient Monte Carlo methods for linear algebra problems*. PhD thesis, University of Reading, 2014.

[28] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[29] Fathi Vajargah et al. *Parallel Monte Carlo algorithms for matrix computations*. PhD thesis, University of Reading, 2001.

[30] Jonathan Weinberg, Michael O McCracken, Erich Strohmaier, and Allan Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 50. IEEE Computer Society, 2005.

Glossary

BSC Barcelona Supercomputing Center.

COO Coordinate Format.

CUDA Compute Unified Device Architecture.

GMRES Generalized Minimal Residual.

HPC High Performance Computing.

IPC Instructions Per Cycle.

MI Matrix Inverse.

MIMD Multiple Instruction Multiple Data.

MM Matrix Market Format.

MPI Message Passing Interface.

OmpSs An extension to OpenMP developed at BSC.

OPENMP Open Multi-Processing.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Thread.

SPAI SParse Approximate Inverse Matrix.

SPMD Single Program Multiple Data.

Annex 1

Matrix	**MC_original**	**MC_V9.2**	**Speedup**
appu	1.7519	0.0866	20.2318
Na5	0.1159	0.0117	9.8939
nonsym_r5_a11	49.8880	0.6937	71.9111
rdb2048	0.0227	0.0025	9.1301
sym_r3_a11	0.6690	0.0431	15.5150
sym_r4_a11	4.4287	0.1712	25.8675
Average speedup			**25.42**

Table 7.1: Speedup analysis between MC (original version) and MC (version 9.2). Parameters = epsilon: 1e-1, delta: 1e-1 and alpha= 5

Matrix	**Program**	**16**	**32**	**64**	**128**	**256**	**512**
appu	MSPAI	140.94376	64.37142	43.32118	29.99135	19.09491	12.26772
appu	MC_v9.2	0.05055	0.07774	0.11099	0.15488	0.12078	0.18272
appu	MC_nocomm	0.00169	0.00091	0.00046	0.00025	0.00015	0.00010
Na5	MSPAI	0.75103	0.47326	0.34128	0.26775	0.31033	0.44103
Na5	MC_v9.2	0.00710	0.02962	0.03987	0.07690	0.05995	0.20181
Na5	MC_nocomm	0.00014	0.00009	0.00005	0.00004	0.00003	0.00002
nonsym_r5	MSPAI	59.23146	31.96763	17.72123	10.09388	5.97983	3.73357
nonsym_r5	MC_v9.2	0.29207	0.34284	0.43543	0.47438	0.50910	0.59549
nonsym_r5	MC_nocomm	0.00993	0.00485	0.00250	0.00128	0.00066	0.00039
rdb2048	MSPAI	0.06037	0.11456	0.10437	0.14382	0.20245	0.25051
rdb2048	MC_v9.2	0.00104	0.02020	0.01580	0.01596	0.02943	0.02965
rdb2048	MC_nocomm	0.00003	0.00002	0.00001	0.00001	0.00001	0.00001
sym_r3	MSPAI	2.77443	-	-	-	-	-
sym_r3	MC_v9.2	0.01887	0.04361	0.06849	0.09903	0.08731	0.09151
sym_r3	MC_nocomm	0.00062	0.00033	0.00018	0.00010	0.00007	0.00005
sym_r4	MSPAI	22.56154	-	-	-	-	-
sym_r4	MC_v9.2	0.08899	0.10398	0.14710	0.17396	0.16548	0.21878
sym_r4	MC_nocomm	0.00252	0.00129	0.00065	0.00036	0.00019	0.00012

Table 7.2: Preconditioner construction scalability times (seconds) for: MSPAI (using default parameters), MC (epsilon: 7e-1, delta: 1e-1 and alpha: 5) and MC_nocomm(MC without communications)

Matrix	Min prec.	Solver	Total	Error	Cores
appu	0.0506	0.0987	0.1493	9.82E-05	16
Na5	0.0071	0.0062	0.0133	1.03E-06	16
nonsym_r5	0.2921	274.6150	274.9071	2.26E-04	16
rdb2048	0.0010	0.0851	0.0861	8.09E-07	16
sym_r3	0.0189	2.0528	2.0717	1.15E-04	16
sym_r4	0.0890	27.3321	27.4211	5.17E-04	16

Table 7.3: Shortest time (seconds) of MC (epsilon: 7e-1, delta: 1e-1 and alpha: 5)

Matrix	Min prec.	Solver	Total	Error	Cores
appu	12.2677	1.0521	13.3198	4.48E-07	512
Na5	0.2678	0.0077	0.2754	8.96E-07	128
nonsym_r5	3.7336	84.4349	88.1685	1.00E-06	512
rdb2048	0.0604	120.7250	120.7854	1.79E-06	16
sym_r3	2.7744	329.9780	332.7524	7.96E-01	16
sym_r4	22.5615	544.1050	566.6665	8.79E-01	16

Table 7.4: Shortest time (seconds) of MSPAI (default parameters)