An Analysis of Parallel Approaches for a Mobile Robotic Self-localization Algorithm

Priscila Tiemi Maeda Saito[1], Ricardo José Sabatine[2], Denis Fernando Wolf[1], and Kalinka R. L. J. C. Branco[1]

[1]*Computer Systems Department, Institute of Mathematics and Computer Science*
University of Sao Paul, São Carlos, SP – Brazil
[2]*Computational Physical Department, Institute of Physics of São Carlos*
University of Sao Paulo, São Carlos, SP - Brazil
{psaito, denis, kalinka}@icmc.usp.br
sabatine@ursa.ifsc.usp.br

Abstract

Self-localization is a fundamental problem in mobile robotics. It consists of estimating the position of a robot given a map of the environment and information obtained by sensors. Among the algorithms used to address this issue, the Monte Carlo technique has obtained a considerable attention by the scientific community due to its simplicity and efficiency. Monte Carlo localization is a sample-based technique that estimates robot´s pose using a probability density function represented by samples (particles). The complexity of this algorithm scales proportionally to the number of particles used. The larger the environment, the more particles are required for robot localization. This fact limits the use of this algorithm in large size environments. In order to improve the efficiency of the Monte Carlo technique and allow it to be used in large environments we propose a parallel implementation of it. Our implementation is based on OpenMP and MPI message passing interface. Experimental results are used to show the efficiency of our approach.

Keywords: *Performance Attributes – parallel implementation of algorithms.*

1. Introduction

Mobile robotics is a research area that has been obtaining considerable attention by the scientific community. The main challenge in this field is to develop robots that can interact with the environment, learn, and make decisions in order to successfully execute specific tasks.

Self-localization is a main issue in mobile robotics. It consists of estimating the robot´s pose based on a previous acquired environment information (map) and data obtained from sensors like video cameras, sonars, and laser range finders. Most applications for mobile robots depend on correct localization to be accomplished, like path planning and autonomous navigation. The main difficulty to obtain correct localization is the inherent imprecision in the information obtained by sensors and maps. Almost all solutions to this problem are based on probability theory, which has been successfully used to handle the uncertainty caused by sensors inaccuracy [9].

The problem of mobile robot localization can be divided in local and global. For local localization or tracking, the initial pose of the robot is known a priori. The solution for this

problem consists of handle the uncertainty of the sensor information and track the robot in the environment. In the global localization, there is no a priori information about the robot´s initial pose and global search must be performed in the environment, which requires more complex techniques to be used. Several algorithms have been proposed in the literature in the last years. One of the more robust approaches to solve this problem is the Monte Carlo location algorithm (MCL), that has been proposed by [10].

The Monte Carlo (also known as particle filter) localization algorithm is based on keeping several hypothesis about robot position in a given environment. Each possible pose is represented by a unity called particle. During the initialization process, as the system does not have any a priori information about robot´s pose, particles are distributed all over the environment. As the robot moves, the particles are propagated following the motion of the robot. As the robot obtains information from sensors, this information is matched to the information that each particle would get from its pose in the environment. Particles that have a good information match obtain a high score. Particles with a poor matching receive a low score. The chance of a particle survives for the next iterations of the algorithm is proportional to its score. As a result of this process, only particles with a good sensor/environment information matching survive and, given enough particles, this algorithm is proven to converge to the robot's correct pose.

One of the major issues of the MCL algorithm is that its processing time is proportional to the number of particles used. There is no formal study that precisely calculates the number of particles necessary to guarantee the convergence of the algorithm. However, the larger is the environment, the more particles are required. Therefore, the utilization of MCL in real time is restricted by the size of the environment. One manner to deal with this limitation and make possible robot´s pose estimation in large environments is using distributed parallel processing techniques. In this work we use Open MP and MPI to obtain a parallel implementation of the MCL technique in order to improve its computational efficiency and allow real time robot localization in large environments.

2. Related Work

The works by [1][2][3][5] use parallel techniques to improve efficiency in control of robotic manipulators. In [4], a learning algorithm is implemented using PVM and applied to mobile robotics. In the work by [6] parallel techniques have been used for robot navigation and path planning.

The Monte Carlo technique is also used in solving problems, even in parallel version. In [11] the authors present the use of the Monte Carlo technique in parallel to assess the reliability of power systems. This work used a distributed environment consisting of 10 machines with processors power2 Risk 6000. As a result, the work presents an improvement in response time for the assessment of reliability in large systems. This demonstrates the feasibility of parallelization of the Monte Carlo technique.

Smith and Kent [12] present a study of the Monte Carlo Technique. They develop a code of Quantum Monte Carlo code in OpenMP [7] and compare this code with the MPI code. They provide a code that could mix the utilization of OpenMP and MPI parallelization to exploit the advantages of both parallelism approaches. This study presents that the code scales well with OpenMP threads to 32 processors and only slightly lower than with MPI processes. Above 32 processors the scaling is worse than with MPI processes, tailing off considerably

above 64 threads. It shows that the total time to develop a working OpenMP version was significantly less than the time taken to develop the original MPI code and that the majority of the time was spent in debugging and performance optimization rather than implementation.

3. Monte Carlo Localization

Self-localization (or just localization) is a major problem in robotics. Correct pose estimation is a requirement for most applications of mobile robots. One of the most efficient solutions to this problem is the Monte Carlo approach, proposed by [10]. It basically consists of estimating the position and the orientation of a robot given: sensor information, motion information, and a description of the environment (map).

As MCL is a iterative technique, the estimation is recalculated for every new motion and observation performed by the robot. Basically, the algorithm can be divided in three steps: motion (also known as prediction), observation (or update) and re-sampling.

The motion step consists of propagate the particles in the map based on robot´s motion information. Most robotic platforms are provided with encoders in the wheels, which generate odometric information (dead-reckoning) as the robot moves in the environment.

Figure 1. Robot and sensors used in the experiments

Unfortunately, the odometric data obtained by the encoders have some inherent uncertainty due to several factors, such as: mechanical imprecision, slippage, and difference in the pressure of tires. In order to compensate for the imprecision in the odometric information, a random error is added to the motion of each particle, which also increases the uncertainty in the robot´s pose estimation.

Table 1. MCL Algorithm

MCL Algorithm

For n = 1 to N

Propagate the particle n based on the robot motion

```
End For
For n = 1 to N
   Calculate the score for particle n
   Add the score of particle n to the total score sum
End For
For n = 1 to N
   Sort a random number between 0 and total score sum
   Add the correspondent particle to the particle set used for the
   next iteration
End For
End Algorithm
```

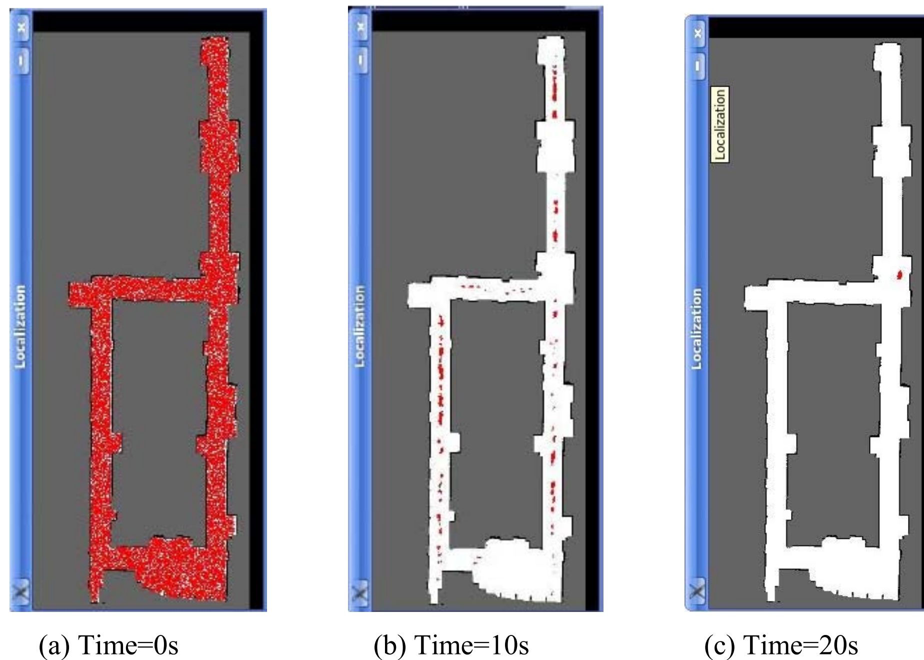
The observation step consists of matching the sensor information obtained by the robot to the environment around each particle. Usually, distance sensors like sonars or laser range finders are used for the observations (Figure 1). These sensors provide the relative position of the obstacles around the robot and this information is compared to the obstacles that each particle should detect at its position. A good matching means that a particle is a good candidate to represent the true position of the robot in the environment, resulting in a high score to that particle. A poor matching indicates that a particle is not likely to be in the correct position of the robot, therefore it results on a low score.

Finally, the last step is the re-sampling, which consists in keeping the particles with a high likelihood of representing the true position of the robot (high score) and eliminate the low score particles.

Assuming that there are N particles in a particular MCL implementation, every particle receives a score after the observation step. The re-sampling step will randomly choose new N particles from the previous set of particles. Each specific particle can be selected more than once. The probability of a particular particle being selected is proportional to its score. High scored particles have more chance to be chosen to the next iteration. Low scored particles tend to be eliminated. Usually, it is created a vector of size N with the partial sum of the scores. The value of the last position of the vector corresponds to the total sum of the particle scores. The selection of the particles that will be used in the next iteration is obtained generating N random numbers between zero and the total score sum. For each random number it is necessary to find the correspondent particle in the vector. This is usually, implemented using a binary search. This technique guarantees that high scored particles will have more chance to be selected.

Figure 2 shows an example of the MCL execution over time with data obtained by a real robot. The white parts of the environment correspond to empty spaces, or spaces that can be occupied by the robot. Black parts correspond to obstacles like walls, doors and other objects in the environment. The spaces cannot be occupied by the robot. The grey dots in white areas represent the particles (or the possible positions of the robot). In Figure 2(a), at time 0s, there is no a priori information about robot localization; therefore, the particles are spread all over

the environment. At time 10s (Figure 2(b)), the robot already obtained sensor and motion information and the particles have grouped in specific regions in the environment. At time 20s (Figure 2(c)), the particles converged to the right position of the robot in the environment.



(a) Time=0s (b) Time=10s (c) Time=20s

Figure 2. Monte Carlo localization. Robot´s pose estimation evolution in time.

4. Parallel Processing

There are several computer architectures that are referenced as parallel or multiprocessor architectures. All parallel architectures have as a main goal improving the computational power by increasing the number of processing elements.

In this paper we exploit the parallelism in two manners, a shared memory parallel computing and a distributed parallel computing. The first one is based on OpenMP, a directive-based method to invoke parallel computations on many shared-memory multiprocessors and the second manner consists of using the Massage Passing Library MPI (Message Passing Interface).

Distributed computing systems have been showing through the years its advantages over centralized systems, achieving a prominence place in the computing scenery in a considerably short time. This type of system has been continually improved by researches in order to provide better performance, at a rather low cost. The application of distributed systems to parallel computing allows a favorable cost/benefit for parallel computing. These systems provide computational power to appropriately execute parallel applications that do not require a massively parallel machine, but on the other hand, require more computational power than a sequential machine can offer.

In order to execute parallel computing over distributed systems, a managing software is usually required. There is also need for information passage between the various machines that compose the platform. There are specific libraries for the treatment of inter-processes communication and synchronization in the literature to perform this task. According to Seinstra & Koelma [20], parallel programming based on the massage passing requires the programmer to have the control over the distribution of data, in addition to the explicit specification of parallel execution code between the different processors. This task requires a high degree of knowledge and control over the system when compared to sequential programming.

The performance of parallel-distributed platforms is directly related to the process scheduling strategy. An effective scheduler improves the efficiency in resources utilization, thereby increasing its performance. More specifically, an efficient scheduling consists of an adequate assignment of loads or process to the computers in the system (load balancing).

Some of the most used libraries for massage passing are MPI (Message Passing Interface) [16] and PVM (Parallel Virtual Machine) [18]. These libraries provide routines to initiate and configure the environment, as well as send and receive data between the processing elements of the system. There are many implementations of MPI and PVM applications developed in languages such as Fortran, C and C + +. Java is another type of application that can be cited the mpiJava [15] and JPVM (Java Parallel Virtual Machine) [14].

The advantage of OpenMP is the possibility of using shared-memory systems, as it is illustrated in Figure 3.

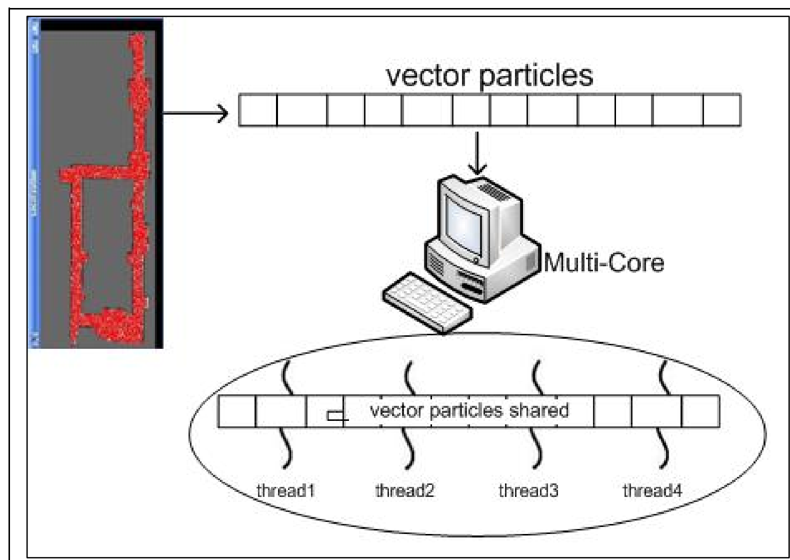


Figure 3. Strategy of parallelization using OpenMP

The basic requirements of parallel processing techniques, such as Monte Carlo, are an infrastructure that allows one to efficiently perform algorithms of low, medium, or high granularities. This infrastructure is composed mainly of communication and distribution of appropriate data. Techniques like MCL require high processing power, as in most cases they

handle large blocks of data to be processed. In this context, they can take advantage of the concepts of parallelization.

When it comes to mobile robots localization, the importance of parallelism is further highlighted, as the number of particles limits the execution of the algorithms in real time. The parallel approach proposed in this paper is a technique to efficiently divide the data into smaller blocks distributed among processors.

The data parallelism is the most adequate technique for the proposed algorithm because it is possible to execute the same code in different processors with different data. In this manner we have a single flow control - SPMD (Single Process Multiple Data). This strategy is shown in Figure 4.

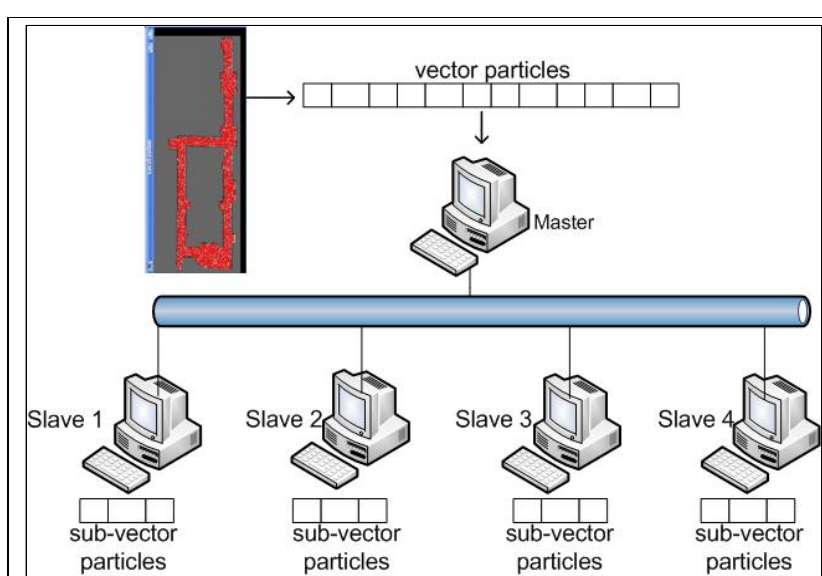


Figure 4. Strategy of parallelization using MPI

4.1. Parallel Implementation

The strategies presented in the previous section can be better illustrated with the real primitives used in the parallel implementation of the algorithms. The next subsection presents the OpenMP and MPI implementation.

Table 2. MCL Algorithm. OpenMP primitives in bold

MCL Algorithm with OpenMP strategy

#pragma omp parallel shared() private(n)

{

#pragma omp for

For n = 1 to N

```
    Propagate the particle n based on the robot´s motion
End For
}
For n = 1 to N
    Calculate the score for particle n
    Add the score of particle n to the total score sum
End For
For n = 1 to N
    Sort a random number between 0 and total score sum
    Add the correspondent particle to the particle set used for the
    next iteration
End For
End Algorithm
```

4.1.1 . OpenMP Implementation

It is possible to write parallel programs for multiprocessors in MPI but we can obtain better performance in same cases using a programming language tailored for a shared-memory. OpenMP is an application interface (API) for parallel programming on multiprocessors, which consists of a set of compiler directives and a library of support functions. Table 2 presents the primitives (in bold) used in the method presented in this paper.

This modification in the code allows the algorithm execute in parallel and obtain the advantages when the total of particles are considerably large.

Table 3. MCL Algorithm. MPI primitives in bold

MCL Algorithm with MPI strategy

If Master do {

  Sub-Vector generation

  SEND (Sub-Vector, N)

  RECEIVE (Sub-Vector-processed)

}

Else {

    RECEIVE (Sub-Vector, N)

    For n = 1 to N

```
        Propagate the particle n based on the robot´s motion
        SEND (Sub-Vector-processed)
        End For
}
For n = 1 to N
    Calculate the score for particle n
    Add the score of particle n to the total score sum
End For
For n = 1 to N
    Sort a random number between 0 and total score sum
    Add the correspondent particle to the particle set used for the
    next iteration
End For
End Algorithm
```

4.1.2. MPI Implementation

The strategy using MPI provides sub-vectors of particles that are sent from the master to the slaves. The slaves work in the data set of particles and than return a processed sub-vector to the master. The master receives all the results and gives the final results to the entire application.

The motivation to study both cases (shared and distributed memory) is the possibility to combine them. Most commercial multicomputers with hundreds of CPU are actually collections of centralized multiprocessors, and we can explore both capabilities of the system and use the entire platform. The literature demonstrated that programs using hybrid programs that use MPI and OpenMP execute faster than programs using only one of them.

5. Experimental results

The experimental results presented in this paper demonstrate the performance improvement obtained with parallel programming to solve problems in different areas.

Our tests have been performed with data collected with a real robot in an indoor environment. Three experiments have been executed in order to validate the proposed approach.

The first results were obtained in a parallel homogeneous distributed environment. The first homogeneous setup consisted of 5 homogeneous Dual Core AMD Athlon 64x2 with 4GB of memory. All the machines were interlinked by a network Ethernet Gigabit and used Linux Kernel 2.6 (Debian 5.0). The third setup is a SMP machine. It is a DualCore Pentium each core with 1,66GHz, with 2Gb of memory and Linux Kernel 2.6 (Ubuntu 8.0).

In order to evaluate the performance of the different implementation, the execution times obtained from sequential and parallel versions of the algorithms have been compared. The case studies show the response time of the algorithm for the same number of iteration with 40.000 and 400.000 particles, for both parallel and sequential implementations.

The results presented in Figure 5 were obtained from an average of 30 executions for sequential application and for each parallel implementation (for five hosts and five process; five hosts and six process; five hosts and seven process and five hosts and eight process). The performance tests were done in parallel with five machines and increasing the number of process in each machine providing the comparison of the performance of each possible combination. All results were statistically evaluated to demonstrate their level of significance.

Figure 5 presents the results of the sequential implementation and the executions of the same algorithm in parallel on 5 machines with 5, 6, 7 and 8 processes respectively. It is possible to notice that the execution time of sequential algorithm is statically worst than the parallel, using different numbers of process. The best case is when we use, with 400.000, five machines and 6 processes. Based on these examples we can prove that the use of parallel programming could be used when we have a big amount of particles.

Figure 6 shows the results obtained with OpenMP. Analyzing the graph, we can notice that the results obtained with this technique are worse than the results obtained with a sequential implementation. This is probably caused by the race condition presented in the OpenMP code. However, modify the original code to use OpenMP is easier than using MPI. An alternative solution to OpenMP is the Pthreads (Posix Threads Programming) [19].

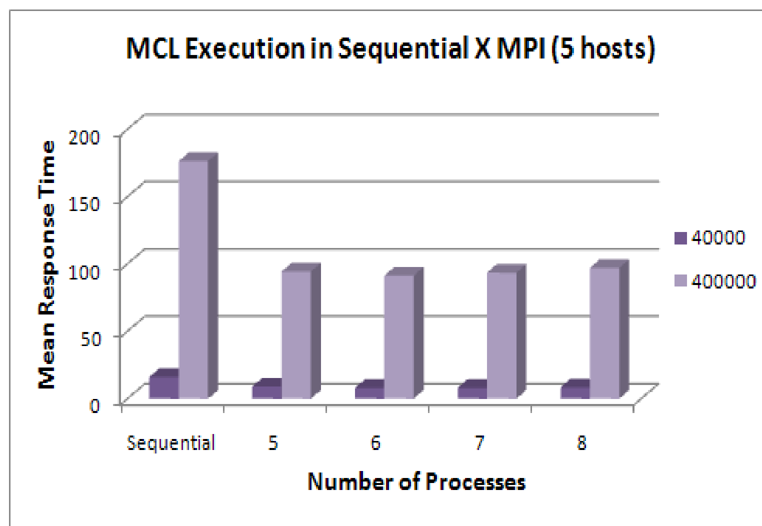


Figure 5. Graph illustrating the response time in execution of parallel and sequential MCL with MPI

As it can be seen in Figures 5 and 6, the results obtained with MPI were better than the ones obtained from OpenMP. Although it has not been tested in this work, combining both approaches may lead to even better results. A massive parallelism could be achieved through the use of multiple processors and multiple threads simultaneously.

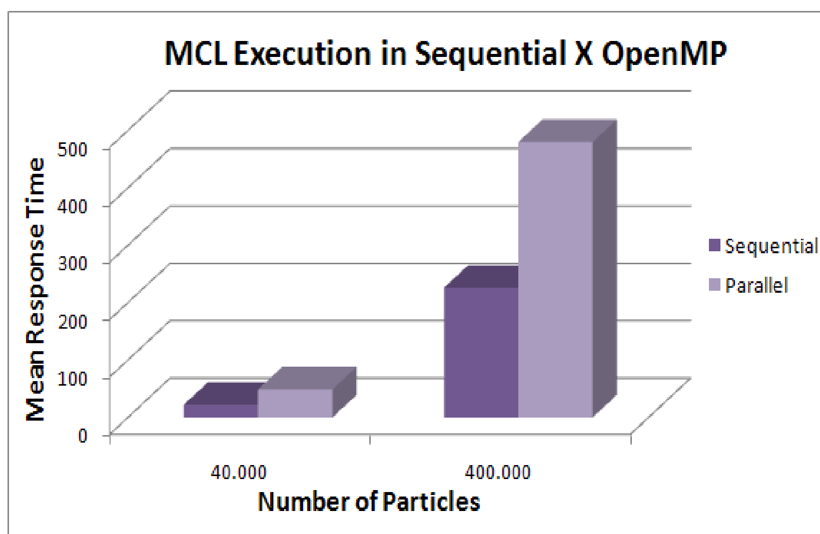


Figure 6. Response time in execution of parallel and sequential MCL implementations using OpenMP

The results presented in Figures 7, 8 and 9 were also obtained from an average of 30 executions for sequential application and for each parallel implementation (for three, five and seven hosts and three, four, five, six, seven and eight process). The performance tests were done in parallel with three machines and increasing the number of process in each machine. They demonstrate a performance comparison of each possible combination, with a increasing number of machines. All results were statistically evaluated to demonstrate their level of significance and the mean times are given in milliseconds.

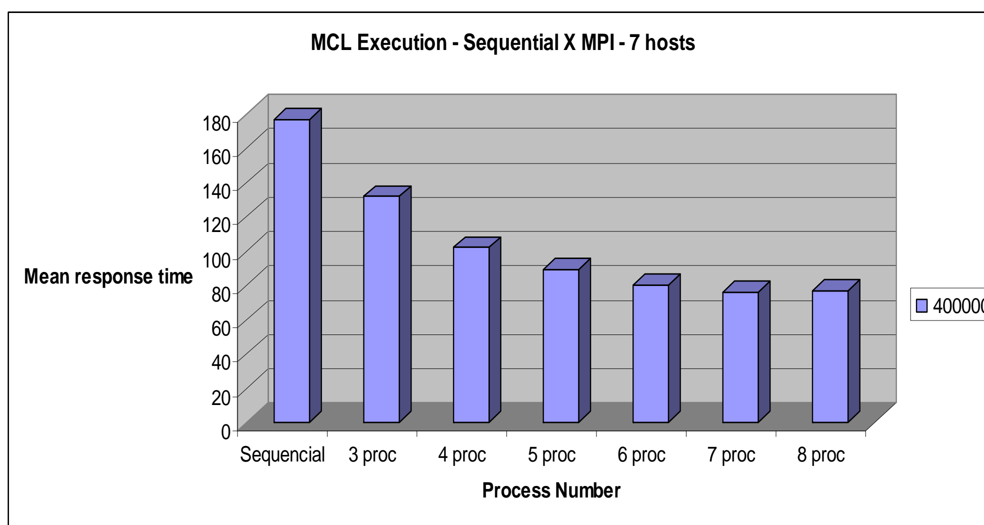


Figure 7. Graph illustrating the response time in execution of parallel and sequential MCL with MPI using 7 hosts

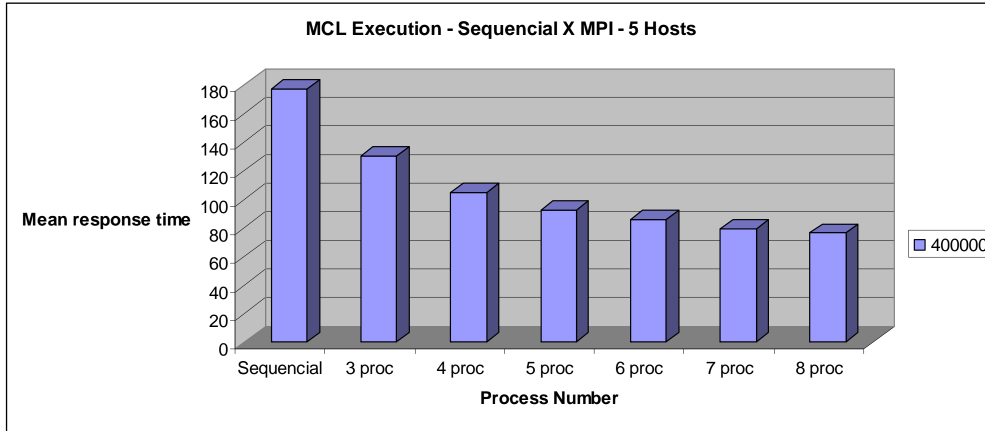


Figure 8. Graph illustrating the response time in execution of parallel and sequential MCL with MPI using 5 hosts

Figure 9 presents the results of the sequential implementation and the executions of the same algorithm in parallel on 5 machines with 3, 4, 5, 6, 7 and 8 processes respectively, where the best results have been obtained. It is possible to notice that the execution time of sequential algorithm is statically worst than the parallel, using different numbers of process. The best case occurs when 7 machines and 7 processes are used, and it is obtained a performance gain of 43%. We can also observe that the best gain in all case is when we have until two processes in each machine. This probably happens because the large amount of particles used in the tests, which benefits from using primitives or functions that explore the parallelism in each core. Based on these examples we can notice that the use of parallel programming could be used when we have a large amount of particles.

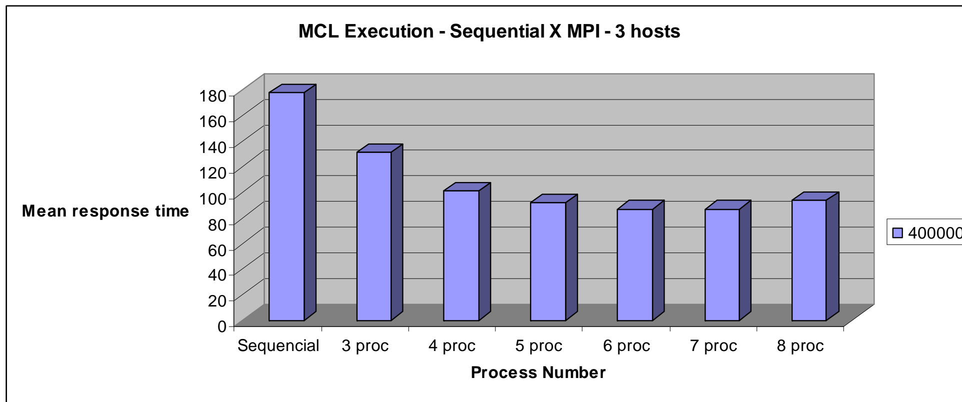


Figure 9. Graph illustrating the response time in execution of parallel and sequential MCL with MPI using 3 hosts

Figure 10 shows the results obtained with OpenMP. Analyzing the graph, we can notice that the results obtained with this technique are worse than the results obtained with a sequential implementation. This is probably caused by the race condition presented in the

OpenMP code. However, modifying the original code to use OpenMP is easier than using MPI. An alternative solution to OpenMP is the Posix Threads Programming (Pthreads) [19].

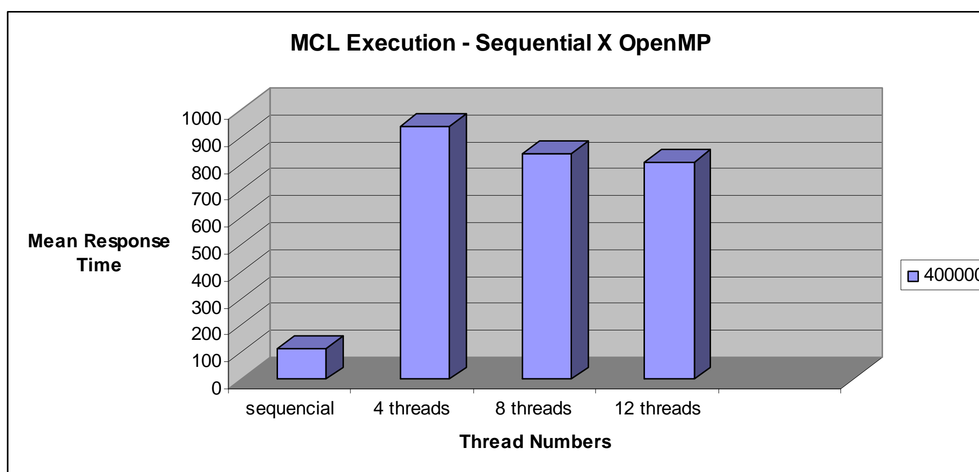


Figure 10. Response time in execution of parallel and sequential MCL implementations using OpenMP

As it can be seen in Figure 10, the results obtained with MPI were better than the ones obtained from OpenMP. Although it has not been tested in this work, combining both approaches may lead to even better results. A massive parallelism could be achieved through the use of multiple processors and multiple threads simultaneously.

6. Conclusion and Future work

Localization is a fundamental problem in mobile robotics. One of the most used techniques to solve this problem is the Monte Carlo localization algorithm, which use a sample-based strategy to estimate the robot´s pose. The computational complexity of the algorithm is proportional to the number particles used in the process, which limits its use in very large environments that demand a high number of particles.

This paper has presented a parallel implementation of the MCL algorithm based on Open MP and MPI. The results obtained from our experimental tests demonstrate the efficiency of our approach, increasing the number of particles that can be used for real time localization.

Based on the results achieved, it can be observed that there is a considerable benefit on using parallel computing to estimate the localization of a robot, i.e., there is a significant decrease in the response time when the proposed parallel approach is compared to a regular sequential implementation.

As a future work we plan to mix the OpenMP with MPI using the available cores in the machines and organizing the data application to get the best of the both words: the distributed-memory and the shared-memory parallel computing using the hybrid strategy making a comparison and attesting the real advantages in use hybrid programs.

The case studies presented in this paper were performed in homogeneous platforms. For future work we plan to investigate the use of the heterogeneous environment in the computation of MCL. We can also use load balancing strategies to improve the response time and to use the Monte Carlo Technique to estimate robot location in real time in a large

environment. The use of load index, performance index and mobile agents could also improve the load balancing and give better results to problem addresses in this paper.

Acknowledgements

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology - Critical Embedded Systems - Brazil), processes 573963/2008-9 and 08/57870-9.

References

[1] M. Yamakita, Y. Hoshino, K. Morimoto, and K. Furuta. Parallel Implementation of Newton- Euler Algorithm with One Step Ahead Prediction. In Proceedings of the 1991 IEEE International Conference on Robotics and Automation, pages 1842{1849, Sacramento, CA, 1991.

[2] H.Zhangand R.P.Paul. A Parallel Inverse Kinematics Solution forRobot Manipulators Based on Multiprocessing and Linear Extrapolation. In Proceedings of the 1990 IEEE International Conference on Robotics and Automation, pages 468{474, Cincinnati, OH, 1990.

[3] M. Vukobratovic, N. Kircanski, and S.G. Li. An Approach to Parallel Processing of Dynamic Robot Models. International Journal of Robotics Research, 7(2):64{71, 1988.

[4] A.M.Printista, M.L.Errecalde, C.I.Montoya "A Parallel Implementation of Q-Learning Based on Communication with Cache" Journal of Computer Science & Technology, Vol.6, 2002.

[5] C.S.G. Lee and P.R. Chang, Efficient parallel algorithm for robot inverse dynamics computation. IEEE Trans. Syst. Man Cybernet. 16 4 (1986), pp. 532–542.

[6] Maja J Matarić, "Parallel, Decentralized Spatial Mapping for Robot Navigation and Path Planning", Parallel Problem Solving from Nature, Hans-Paul Schwefel and Reinhard Maenner, eds., Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991, 381-385.

[7] L.A. Smith and P. Kent. Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code. Proc. of the 1st European Workshop on OpenMP, pages 6-9, Sept. 1999.

[8] Esselink, K.; Loyens, L. D. J. C.; Smit, B., Parallel Monte Carlo simulations, Physical Review E , Vol.51(2), pp.1560-1568, 1995.

[9] Thrun, S., Burgard, W., Fox, D., Probabilistic Robotics, MIT Press, 2005.

[10] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte Carlo localization for mobile robots. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 1999.

[11] Borges, C.L.T., Falcão, D.M. "Avaliação da Confiabilidade de Sistemas de Potência em Paralelo Usando Simulação Monte Carlo Seqüencial", Revista Controle & Automação. Vol. 11 nº2/Mai, Jun, Jul, Agosto 2000, 94-99.

[12] Smith ,L., Kent, P. "Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code" in Concurrency: Practice And Experience, 2000; 12:1121–1129.

[13] Ferrari, A. J.; (1998) JPVM: Network parallel computing in Java, In ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, February 1998. Concurrency: Practice and Experience.

[14] Ferrari, A. J. (2007) JPVM The Java Parallel Virtual Machine. Available at: http://www.cs.virginia.edu/~ajf2j/ jpvm.html. Last access June 2009.

[15] Baker, M. et al. (1998) mpiJava: A Java Interface to MPI. Submitted to First UKWorkshop on Java for High Performance Network Computing, Europar.

[16] Quinn, M. "Parallel Programming in C with MPI and OpenMP". McGraw-Hill Science/Engineering/Math; 1 edition, June 5, 2003.

[17] Snir, M. Otto, S. M., Huss-Lederman, S., Dongarra, J. (1996) MPI: The Complete Reference, The MIT Press.

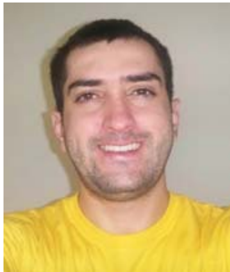
[18] Beguelin, A., PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing, The MIT Press, 1994.

[19] Pthreads, https://computing.llnl.gov/tutorials/pthreads/ last access in June 2009.

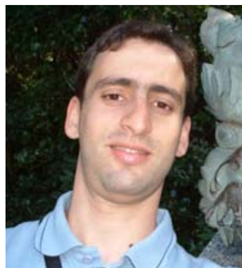
[20] SEINSTRA, F.J.; KOELMA, D. User Transparency: A Fully Sequential Programming Model for Efficient Data Parallel Image Processing, Concurrency and Computation: Practice and Experience, vol. 16, no. 6, pp. 611-644, May 2004.

Authors

Priscila Tiemi Maeda Saito received the BSc degree in computer science from Euripides Soares da Rocha University of Marilia (UNIVEM), Brazil, in 2007. She is currently a M.Sc. student in computer science at the Mathematics and Computer Science Institute of the University of Sao Paulo (USP) at Sao Carlos, Brazil. Her research interests include parallel computing and distributed systems.



Ricardo Jose Sabatine received the BSc degree in computer science from Euripides Soares da Rocha University of Marilia (UNIVEM), Brazil, in 2007. He is currently a M.Sc. student in computational applied physics at the Institute of Physics at Sao Carlos (IFSC) - University of Sao Paulo (USP), Brazil. His research interests include grid computing, parallel computing and distributed systems.



Denis F. Wolf has a degree in Computer Science from Federal University of Sao Carlos (1999), MSc. from University of Sao Paulo (2001), and Ph.D from University of Southern California (2006), also in Computer Science. He is currently an Assistant Professor at University of Sao Paulo – Sao Carlos where he works in the Computer Systems department since 2007. He is one of the founders of the Mobile Robotics Laboratory at USP and his research interests include mobile robotics, machine learning, and embedded systems.

Kalinka R. L. J. C. Branco has degree in Technology in Data Processing from Paulista Foundation of Technology and Education (1995), MSc. in Computer Science from University of São Paulo (1999) and Ph.D. also in Computer Science from University of São Paulo (2004). She is currently Assistant Professor in the Institute of Mathematics and Computer Science - ICMC - USP, working in the department of Computer Systems. She has experience in Computer Science, with emphasis on Distributed Computing Systems and Parallel Computer, working mainly in the following areas: distributed systems, computer networks, security, performance evaluation and processes scheduling. She is member of Brazilian Computer Society.