Accelerating Conjugate Gradient Solver: Temporal Versus Spatial Data

Korolija, Nenad G.; Milutinovic, Veljko; and Milosevic, Srdjan

***Abstract*—Simulation of an object in the wind tunnel is a long lasting process, and therefore an ideal candidate for making code run in parallel.**

Simulation complexity is still to high for today's computers. With a growing number of processes computation time is falling, but communication time is rising. Memory can also be the problem.

Existing solutions are based on one process being the master, and, as so, communicating with all other processes. That causes both time consuming communication while other processes wait for the master process and memory problem while one process holds all the data at one moment if no special technique is applied.

In this paper, another approach is described. Using load balancing, all processes became equal during the computation phase. That means that each one the n processes tends to hold approximately 1/n of the data, and works without waiting for communication to finish.

Numerical and computational analyses are done in order to show reader the major advantage of this approach.

As a result, in a real case, the speedup when switching from 64 to 128 processors is almost two.

Index Terms— *Master-slave parallelism – all of the processess working in parallel, non-zero block – block of matrix that has at least one non-zero value*

1. INTRODUCTION

BECAUSE making a car model and simulating its air resistance in the wind tunnel is both time and money expensive, the idea of making a simulation has became a reality. It is expected that the first BMW will be made without making a single model soon. In order to evaluate air resistance, one needs to discretize a volume, set PDE-s, and solve them. Solving PDE-s with huge number of unknowns is not possible without using mathematical algorithm, except in some special cases. By discretizing, a system with n linear equations and n unknowns is obtained. The most popular way of presenting these is using matrices. In this case, the matrix will be sparse. Conjugate Gradient method is a method for solving system of linear equations using induction. In each step, we are supposed to be closer to the exact solution.

This paper deals with optimizing code, for parallel execution of Conjugate Gradient algorithm without preconditioning with a sparse matrix that is a result of discretizing a volume and setting correspondent PDE-s. Code was tested on many multi-processors computer architectures, which were suitable for running MPI programs.

Dipl. Ing. Nenad G. Korolija, Dr. Veljko Milutinovic, Mr. Srdjan Milosevic, Scool of Electrical Engineering, University of Belgrade, Serbia

2. PROBLEM STATEMENT

When talking about simulations, where the volume is divided in small amounts of volume, it is obvious that by dividing on smaller pieces leads to the result with better precision. Of course, that also means longer execution time. Therefore, it is natural to try to run a simulation code in parallel.

While the majority of calculation in CG algorithm is matrix vector multiplication, the execution time is easy to calculate, and the multiplication can easily be divided on many processors.

Anyway, when the result should be spread to all of the processors, sometimes it is faster to run the whole simulation on single processor computer, then to run it on many processors, and then deal with the communications. Even if the communication lines are very fast, with every message passing interface one processor has to form a message head and body and send it, and the receiving part should do inverse operations, and the direct communication is not easy to establish, and usually not useful.

For example, with matrix vector multiplication divided on many processors, each of them needs to send the result to all of the others. In case of modeling the volume, the result matrix is sparse, which will be of great interest in further calculations.

3. PROPOSED SOLUTION AND WHY IT IS EXPECTED TO BE BETTER THEN OTHERS

The main idea is not presented by explaining each part of the algorithm/code, but instead, explaining each idea that lead to the final solution. Within each idea, main characteristics are given, a picture demonstrating what we have achieved by implementing it, and the problem to be solved by implementing next idea. But first, the serial implementation will be discussed.

The main ideas were:

- Dividing calculation onto many processes
- Reduced communication
- Master-slave parallelism

3.1 Serial implementation

The most important thing to notice when talking about making a simulation run on many processors is that there are approximately 200 iterations per one time step, where matrix-vector multiplication is the most processor demanding operation in iteration. Beside it, scalar vector product is calculated in all iterations, and the multiplication of the vector with a constant. The most promising thing to do is to split matrix vector multiplication on many processors. Other operations are not to be split at this stage because more time would be needed to send and receive the result than to calculate it on a single processor.

3.2 Dividing calculation onto many processes

Because of the nature of the problem, the matrix is divided onto non-zero blocks same sizes. The number of row blocks is divided by the number of available processors. Each processor is responsible for multiplication of approximately the same number of row blocks with appropriate vector. Figure 1 depicts a non-zero blocks marked as black circles, while the rest of blocks are zero blocks. The acceleration obtained by using this basic principle is obvious. Still, there is a problem to be solved. After each matrix-vector multiplication, a result should be collected, and then delivered to all processes. This approach requires sending and receiving huge amount of data.

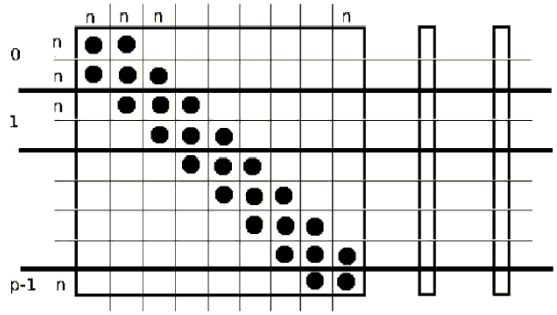

Figure 1 – dividing calculations onto p processors

3.3 Reducing communication

In order to make code more efficient, the structure of the matrix is examined. Figure 2 describes necessary vector data marked as tree dots for multiplication with the corresponding part of the matrix. It is easy to notice that if one processor multiplies any number of row blocks with the vector, it needs to receive only 2*n numbers, and send the same amount of data.

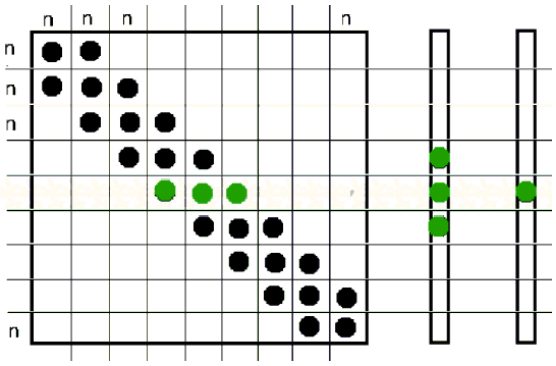

Figure 2 – determining necessary vector parts for multiplying with one row block

Comparing to the previous case, maybe it is not that obvious, but the communication necessary for matrix vector multiplication is reduced almost n/2 times, where n could be even 10000. Anyway, the problem is still a little bit covered, but easier to notice. The whole communication is done by the root process and each other process. In case we have a computer architecture made of equal nodes, each process would have to wait until all of the data has been received by the root process and sent to all other processes. If one process is run on single processor, all processors would have to wait for communication to finish.

3.4 Master-slave parallelism

Now that we know which part of the vector is necessary for which process in order to do the calculation, we can try do determine which process has got the requested data. Even if all of the processes are treated as equal when using MPI, we can mark process zero as the root, and all other processes as slaves. Similarly, we can force the process with any rank to work with corresponding row-blocks, and therefore know rank of the process that is working with any part of the vector. This way, as shown on figure 3, every process needs to send only n real numbers to the upper neighbor and n real numbers to the lower neighbor. Here, upper neighbor is the process whose rank number is less by one than the current process rank number, and lower is the one with the rank number higher by one. Similarly, it needs to receive same amount of data from same processes. While many computer architectures support parallel communication between processes, using master-slave parallelism could lead to almost n times less execution time compared to the one in previous case, where all the communication was done over the root process.

The last, but not least to say is that whole communication could be done in parallel to the calculation, which means that for big problem sizes, the communication time is around zero. This is achieved in tree stages. First is starting sending and receiving operation. Second is multiplying each row block that is independent

from other processes. Third is checking if the communication has been finished. Only in case of having small data sets, processors would have to wait. Otherwise, the remaining thing to do was multiplying the upper and the lower row block with correspondent vector parts.

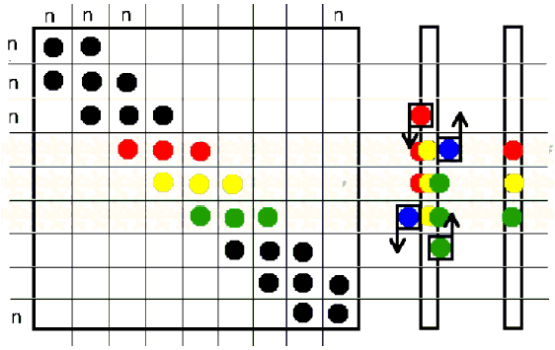

Figure 3 – communication of process with the neighbor processes.

3.5 Further development

For this kind of algorithm as CG is, the limit for making a code parallel as it can be is that in each iteration an scatter/reduce operation must be done. That includes sending a packet of data from each of the processes, than summing data from every process, and then delivering the result to all of the processes. By modifying CG algorithm for parallel implementation, a redundant communication could be obtained. Of course, algorithm should be very different in that case, while prediction of the sum of the data in next step should be made, and the next iteration could depend on the prediction, and not the real sum. Later, the correction would have to be done. Developing such algorithm would show if the result could be reached in a shorter time.

4. CONDITIONS AND ASSUMPTIONS OF THE RESEARCH TO FOLLOW

In this chapter, a brief introduction to the suitable computer architecture for this program is given. Testing was mainly done at cluster Mozart at SGS department on IPVS, in Stuttgart, Germany. It consisted of 64 nodes, each containing two processors with 1GB of RAM. Simulation can be run on any system having MPI installed on it, even single processor. Anyway, in order to obtain considerable less execution time than in case of running a serial version, cluster with big number of processors and good network is needed. Even if the network is not good, with the bigger program size, the process data exchange could be done in parallel to the calculation. Special sparse matrix was produced by Ionel Muntean's code, which was given in 9 vectors for the 2D case and 27 vectors for 3D case. These vectors represented non-zero data in the matrix. It is much more efficient to store only 9 or 27 vectors than to store whole non-zero blocks in the memory while most of the elements of the blocks would be zeros.

5. ANALYTICAL PERFORMANCE ANALYSIS

In this chapter, analysis is done considering memory and time aspects. For each of them, a comparison between serial and parallel version is given. In order to make paragraph more understandable, let n be the dimension of the matrix and the vectors, and p number of processes in parallel version of the program, which will be used in later text.

5.1 Memory usage

Memory usage will be considered with a size needed for representation of a real number defined as a minimum memory usage. For example, size n will represent 8*n bytes if one real number needs 8 bytes.

For serial version the memory size needed is determined in co ordinance with size of vectors and matrix. There are 4 vectors of size n, and matrix 5*n*n for 2D case, and 27*n*n for 3D case. In order to run the serial version of program that executes CG algorithm, all the memory must exist in single computer.

For parallel version, matrix size is 5*n*(n/p) for 2D case, and 27*n*(n/p) for 3D case. Size of each of 4 vectors is n/p. It is obvious that a parallel version is as parallel as it can be considering memory aspects, meaning that all the data that occupies most of the memory is spread over all p processes equally.

5.2 Time needed

Using CG algorithm without prediction, it is calculated that approximately 250 iterations is done in order to have solution enough close to the real system of linear equations result. Calculation time per iteration will be used as a minimum amount of time in order to make analysis more readable. Automatically, same calculation is valid for both 2D and 3D case.

For serial version all the calculation has to be done at single processor unit, meaning that time needed for execution is n.

For parallel version, it is good to define the time needed for data block sending and receiving. Anyway, while the transmitting of data is at least partially done in parallel to calculation of part of matrix and part of vector multiplication, the important thing is not to calculate the time needed for sending, but the difference between calculation time and that time. While this could make the analysis unnecessary complicated and example dependent, only graphics showing algorithm execution time for different problem sizes are given. At this point, it is important to notice that for the big problem sizes, but still real, the data sending/receiving is done completely in parallel to calculation even for considerably high number of processors. Therefore, the time needed to finish execution of parallel version of

code is: n/p plus time needed for reduce/broadcast operations, where every process sends/receives one real number. For big problem sizes, by doubling processor number, the execution time is reduced around twice! Therefore, it is obvious that a parallel version is as parallel as it can be considering processing time also, when the problem size is great enough to be reasonable to run it on more then one processor.

6. SIMULATION ANALYSIS

In this chapter, graphics and tables are given, for both 2D and 3D cases, in order to make possible for reader to realize the benefits of the proposed solution at the first glance. Figures 4 and 5 depict tables showing running time in seconds depending on the number of processors used for calculation in each row, and the problem size in each column. On figures 6 and 7, graphics are given for chosen problem sizes to show the dropping of the execution time with growing number of processors. Note that x-axis is exponential.

	20 x 20	50 x 50	100 x 100	200 x 200	400 x 400
np 1	0.014907	0.05483	0.202765	1.045567	4.306912
np 2	0.011452	0.037967	0.116468	0.653028	3.310686
np 4	0.014715	0.027751	0.067383	0.250178	1.761521
np 8	0.017923	0.024089	0.044449	0.121511	0.83695
np 16	x	0.025532	0.035872	0.074576	0.270791
np 32	x	x	0.03601	0.055549	0.147883
np 64	x	x	x	0.053953	0.112273
np 128	x	x	x	x	0.102522

Figure 4 – Measurements on cluster Mozart for 2D case (number of processes vs. problem size)

	20 x 20 x 100	30 x 30 x 100	40 x 40 x 100	30 x 30 x 600	60 x 60 x 1200
np 1	3.770388	8.015547	14.47042	49.212649	466.394529
np 2	2.786651	6.169195	12.845254	37.632147	555.831802
np 4	1.556983	3.684659	7.64594	20.757366	400.517335
np 8	0.788501	2.069098	4.260279	10.569032	204.146744
np 16	0.334108	1.229958	2.336778	5.592154	102.672306
np 32	0.212199	0.621971	1.647089	3.012153	51.750987
np 64	x	x	x	1.87647	26.920537
np 128	x	x	x	1.072528	14.734521

Figure 5 – Measurements on cluster Mozart for 3D case (number of processes vs. problem size)

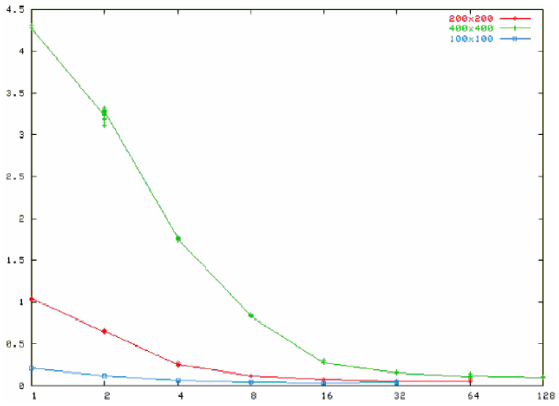

Figure 6 – Tree graphs for tree different problem sizes for 2D case (X-axis - number of processes, Y-axis – time given in seconds)

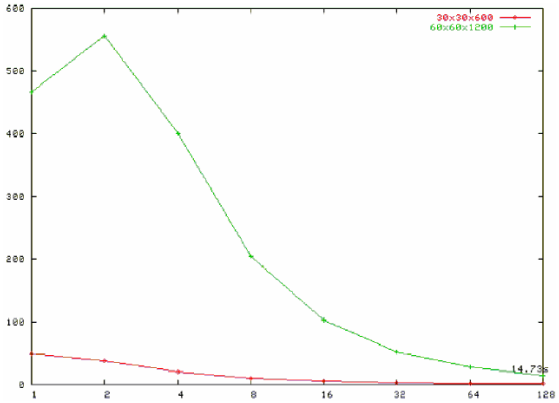

Figure 7 – Two graphs for two different problem sizes for 3D case (X-axis - number of processes, Y-axis – time given in seconds)

7. CONCLUSION

Even though most of today's computers are single processor computers, there are a lot of clusters and special purpose computers, and a lot of them still to come. Therefore, making computer programs for parallel execution on many processors is a very promising activity. Simulations are the best examples of the heavy computing programs, and as such, ideal candidates for making the code parallel. By making parallel version of CG algorithm, the time necessary for making a program was reduced in some cases even by 40 times. For achieving such a result one needs to be able to run a code on 128 processors architecture, like one in SGS department on IPVS in Stuttgart, Germany.

ACKNOWLEDGMENT

This work would not be possible without help of Prof. Joachim Bungartz and Ionel Muntean. Many thanks to them for inviting me on a three months practice in Stuttgart, and providing me access to the cluster Mozart, as well as for the great classes I had taken.

REFERENCES

[1] "A message passing standard for MPP and workstations", Jack J. Dongarra, Steve W. Otto, Marc Snir, Yorktown Heights, David Walker

[2] "Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster", Gabriele Jost, Haoqiang Jin, Dieter an Mey, and Ferhat F. Hatay

[3] "MPI: The Complete Reference", M. Snir, S.W. Otto, S. Huss-Lederman, D. W. Walker and J. J. Dongarra. Published by the MIT Press, 1995.

[4] "The Emergence of the MPI Message Passing Standard for Parallel Computing", R. Hempel and D. W. Walker, Computer Standards and Interfaces, Vol. 7, pages 51-62, 1999.

[5] "Redistribution of Block-Cyclic Data Distributions Using MPI", D. W. Walker and S. W. Otto, Concurrency: Practice and Experience, Vol. 8, No. 9, pages 707-728, November 1996.

[6] "MPI: A Standard Message Passing Interface", J. J. Dongarra and D. W. Walker, Supercomputer, Vol. 12, No. 1, pages 56-68, January 1996.

[7] "The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers", D. W. Walker, Parallel Computing, Vol. 20, No. 4, pages 657-673, April 1994.

[8] "Standards for Message Passing in a Distributed Memory Environment", D. W. Walker, Technical Report ORNL/TM-12147, Oak Ridge National Laboratory, August 1992.

[9] "On Characterizing Bandwidth Requirements of Parallel Applications", Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.

[10] Milutinovic, Veljko, “The Best Method for Presentation of Research Results”
IEEE TCCA NEWSLETTER, September 1996.

[11] Lectures from Prof. Dr. Hans-Joachim Bungartz, http://www5.in.tum.de/persons/bungartz.html

[12] Lectures from Ioan Lucian Muntean http://www.ipvs.uni-stuttgart.de/abteilungen/sgs/abteilung/mitarbeiter/ioan_lucian.muntean/de

[13] “Teaching High-Performance Computing on a High-Performance Cluster”, Ralf-Peter Mundani, and Ioan Lucian Muntean. IPVS, Universitat Stuttgart http://www.springerlink.com/index/YWAG0MWXGQ95HL21.pdf