SAND2012-1461 C

SAND2012-1461C

A Multithreaded Solver for the 2D Poisson Equation

Andrés Vidal
Alain Kassab
Daniel Mota
Department of Mechanical, Materials and Aerospace Engineering, University of Central Florida
4000 Central Florida Blvd, Orlando, United States
1-407-823-5778
Andres.Vidal@knights.ucf.edu

Damian Dechev[1]
Scalable and Secure Systems R&D Dept.
Sandia National Laboratories
Livermore, CA
ddechev@sandia.gov

Keywords: Multithreaded algorithm, distributed system, Poisson equation, linear system, Direct Numerical Simulation.

Abstract
A multithreaded solution for the 2D Poisson equation is presented. The proposed algorithm distributes the tasks between threads in Floating-Point Unit (FPU) intensive and non-FPU intensive. This technique also allowed us to make the communication between nodes asynchronous. Our approach of decoupling communication and computation allows for much greater scalability. This new multithreaded approach showed better performance in all multicore processors tested. In the case of the distributed systems tested, the proposed method had greater speed-up than the classical scheme. The technique Red/Black ordering was found to be effective only if data fit entirely in cache memory.

1. INTRODUCTION

The Poisson equation is perhaps the most popular differential equation used to explain many physical phenomena [7]. After discretization, the resulting linear system of equations is very stable numerically and it can be easily solved with any iterative procedure. This feature allows any non-expert programmer to code an algorithm to solve the problem.

In the case of Computational Fluid Dynamics (CFD), the most challenging problem is the simulation of turbulent flow with no physical model at all [6]. This technique, called Direct Numerical Simulation (DNS), and the cheaper counterpart Large-Eddy Simulation (LES), demands a huge amount of computing resources since the flow equations must be solved with all terms included and with a very high degree of detail. Since the idea of DNS is the study of the transient behavior of the flow, the related linear system must be solved hundreds of thousands of times until the data are suitable for statistical studies.

For large problems, the most frequent solution is a serial solver (iterating on a portion of the problem) in a parallel system with only one processing unit per node [2 and 9]. Many of the solutions proposed for parallel systems are devised by end-users, with little knowledge of parallel computer architecture or programming.

The purpose of this work is to present a numerical/computational iterative technique that gets the most out of the typical cluster configuration today: a distributed system with many multicore/singlecore processors in each computing node.

Our approach allows the programs run time to come much closer to perfect speed-up. Because the nodes are asynchronous it does not matter how large, slow, or complex the network of computers may be. This method allows for much greater scalability then is currently a[1]chieved. Our tests showed it to be several times faster than the traditional synchronized approach. Here we present an approach that offers the following contributions:

1. Method for programming a distributed system that uses Poison equations in a way that overlaps communication and computation

2. Increased speed of scientific computation

3. Allows supercomputers to be scaled to a higher number of nodes

4. Dividing code into Floating Point Unit (FPU) intensive and non-FPU intensive segments. Increasing efficiency on single processors

2. RELATED WORK

The most frequent solution found in the literature is a serial solver (iterating on a portion of the problem) in a parallel system with only one processing unit per node. For any of the most popular iteration methods, Jacobi, Gauss-Seidel or Successive-Over-Relaxation (SOR), the overall procedure is [9]:

- Every node recalculates all its points once.
- Communication is performed between nodes (or between nodes and the master node) to update boundary values.
- Process is repeated until convergence (usually monitored by a master or root node).

It is important to underline that the exchanging of information is done frequently with the Message Passing Interface (MPI) communication library. Additionally, most implementations use global reduction/scatter/gather operations that overload the network rapidly while increasing number of nodes. In other references, such as [2], the recommendation is to use libraries such as BLAS, LINPACK, EISPACK, LAPACK, ScaLAPACK, PETSc, etc.

For more specific flow problems, such as in [3, 4, 5, 7 and 10], the overall procedure is a serial calculation distributed in several nodes, with MPI communication done at the end of any iteration. In some cases there is an improvement in the way communication

[1] Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

is done in multi-domain flow problems, but no calculation is done during the communication stage.

The authors in [1] present a solution where partial updates are done through asynchronous MPI communication. This solution is closer to our method and improves performance with respect to the synchronous scheme, but it does not scale well.

In systems with multicore processors, the usual procedure is to execute one copy of the solver per processor and use MPI for the data exchange. The MPI communication library is very efficient and widely used. However it is not suitable for information exchange within a core as all processors on the same core have access to the much faster shared L2 cache memory.

3. CODING AND OPTIMIZATION

Independent of the methods or solutions that are going to be implemented and tested, the way a program is coded has important effects in its overall performance. For numerical computations, there are some basic and useful rules that can produce important improvements in computing time [9 and 11].

One way to improve performance is by reordering the way equations are calculated. Red/Black ordering organizes the equations like squares in a checkerboard (called *red* points and *black* points respectively) [9]. The direct data dependencies are eliminated since each recalculated point has exclusive access to its immediate neighbors. The resulting ordering scheme allows the calculation of the red points in parallel with the black ones. On the other hand, in a serial processor, if red points are updated first and then the black ones, cache memory will be better used since there is no data dependency between equations.

Other ways to improve performance is by the substitution of multiple real divisions with the product of the corresponding inverse [11], as well as the use of the integrated instruction $a + b * c$. In all implementations of the present work, the *g++* compiler was used with the optimization switch *–O3*.

4. DISTRIBUTED/MULTICORE SOLVER

An important observation in a DNS/LES simulation is the fact that more than 80% of the processing time is spent by the floating point unit. Another important issue to take into consideration is that many multicore processors have more threads than FPUs. In order to achieve an efficient solution, the main program (main-thread) may launch some additional solver-threads to iterate in a portion of the problem. The total number of solver-threads does not necessarily have to be equal to the number of FPUs. In this alternative, the main thread can perform the rest of the tasks that are not FPU intensive.

Our novel idea of separating the code into threads that are FPU intensive and threads that are not FPU intensive, all of them running at the same time, changes the way the calculation is going to converge. It is unlikely that the same problem will have the same convergence history in two different runs. Additionally, having threads using the FPU constantly will have all pipelines of the floating point unit full. This is the first time this technique has been applied and is one of the main contributions of our work.

With these considerations, our proposed algorithm can be stated as:

- Execute a predefined number of solver-threads. They iterate on a portion of the problem, calculating the maximum increment at the end of any iteration, and checking if the maximum numbers of iterations have been reached.
- Use the main thread to monitor convergence, coordinate communication between nodes, manage I/O operations, and to create restores points as necessary.
- All nodes perform the same operations, except the master node which additionally synchronizes all communication operations.

The basic idea of this solution is that the calculation is overlapped with communication and I/O operations, producing important savings in computation time. It is convenient to remember that MPI operations, even in high-speed networks, require a lengthy waiting/transfer time. This waiting time will become more significant as the number of nodes increases.

The tasks of any solver-thread can be stated as:

1. Get data of corresponding block.
2. Send signal to the main thread informing it that this thread is running.
3. Initialize solution in block.
4. Reset maximum increment and number of iterations.
5. For each point in block:
   - 5.1 Update solution.
   - 5.2 Compute maximum increment.
   - 5.3 Update number of iterations.
6. If *stop condition* is activated, quit.
7. Go to step 4.

In the second step it is necessary to create synchronization between the solver-thread and the main thread. In the Linux environments, when a new thread is created, it may not be running when control is returned to the calling process. This step is not necessary in Windows systems. After some initialization steps, any solver-thread will iterate continuously until the stop condition is activated. This separation of tasks will optimize the use of the FPU. Because any of the solver-threads may not be running at any moment, some solver-threads may iterate more than others and/or than the maximum predefined number of iterations (needed if calculation does not converge).

Finally, the tasks of the main thread are:

1. Start MPI.
2. Perform a general setup.
3. Start solver-threads.
4. Wait until all solvers are running.
5. While *stop condition* is not activated:
   - 5.1 Wait until first solver has done one iteration.
   - 5.2 Exchange data between nodes.
   - 5.3 If all solver-threads (in all nodes) have converged, activate *stop condition*.
   - 5.4 If all solver-threads (in all nodes) have reached/exceeded the maximum number of iterations, activate *stop condition*.
6. Write solution.
7. Finish MPI communications.

8. End computation.

The idea behind step 5.1 is to avoid the overload of the network from the constant exchanging of data. The main thread (in the root node) waits until the first solver-thread has done one iteration, while the other main threads just wait for the data. Updating the boundary values and recalculating the new ones at the same time will make convergence completely different. The solution of a linear system is the same no matter what iterative procedure is used, or what initial values are set or how the points are updated.

5. TESTING THE MULTICORE SOLVER

Before proceeding with the evaluation of the proposed solver, the multicore part of the solver will be tested first. The equation to be solved is:

$$\nabla^2 \phi = 0 \tag{1}$$

With the boundary conditions:

$$\begin{aligned} \phi(0, y) &= 10; \quad \phi(1, y) = 30 \\ \phi(x, 0) &= 40; \quad \phi(x, 1) = 20 \end{aligned} \tag{2}$$

The usual way to solve Equation (1) is by finite differences (or similar procedures), in which a discrete distribution of points is located in space, substituting all derivatives with finite difference expressions. In 2D, if all points are equally spaced, the resulting equation is:

$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}\right) \tag{3}$$

Using the iterative procedure SOR, equation (3) is re-written as:

$$\phi_{i,j} = \left[\frac{1}{4}\left(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}\right) - \phi^*_{i,j}\right]\omega + \phi^*_{i,j} \tag{4}$$

With $\phi^*_{i,j}$ the value computed in the previous iteration. The over-relaxation factor ω chosen is 1.5, which is not necessarily the optimum but enough to speed-up convergence. In all calculations, the tolerance was set to $1 \cdot 10^{-12}$. All internal points are initialized with the value 0.

To gain important experimental data on a variety of platforms, we evaluated our approach on seven systems:

S1: Computing node with 2 Intel Xeon, 3.20 GHz, 2 MB cache, hyper-threading disabled, Linux operating system and g++ compiler version 4.1.2.

S2: Computing node with 2 Intel i7 quadcore, 3.07 GHz, 12 MB cache, hyper-threading disabled, Linux operating system and g++ compiler version 4.1.2.

S3: Workstation with 1 Intel Xeon X5670, 6 cores, 2.93 GHz, 12 MB cache, hyper-threading disabled, Linux operating system and g++ compiler version 4.5.2.

S4: Computing node with 2 AMD Opteron 248, 2.20 GHz, 1 MB cache, Rocks 5.1 / CentOs 5.2 operating system and g++ compiler version 4.1.2.

S5: Front-end node with 4 Intel Xeon, 2.8 GHz, 1 MB cache, hyper-threading disabled, Linux operating system and g++ compiler version 4.1.2.

S6: PC with 1 Intel i7 quadcore, 2.80 GHz, 8 MB cache, enabled hyper-threading, Cygwin sub-system version 1.7.9 under Windows 7 and g++ compiler version 3.3.

S7: Computing node with 2 Intel Xeon, 3.06 GHz, 1 MB cache, enabled hyper-threading, Linux operating system and g++ compiler version 4.1.2.

The first parameter to be determined is the optimum number of solver-threads to be set in the multicore solver. This task was done by solving a linear system of 122x100 internal equations. Since the core idea of this work is the separation of the FPU operations and non-FPU operations, problem size should not affect the way solver-threads interact with each other. The optimum number of solvers was found to be respectively 1, 7, 5, 1, 4, 6 and 3. With the exception of system S6, the optimum number of solvers plus the main thread is equal to the number of physical threads. In the special case of the system S6 can be explained with the fact that the Windows 7 system is not designed for high performance computing. The rest of the systems are Linux implementations specially configured for intensive computations.

For example, the same code compiled with g++ 3.3 in the Interix sub-system always runs 20% slower than in Cygwin running in the same machine. Moreover, all codes compiled with Visual C++ 2010 run 20% slower than in Cygwin. This difference in time is very important in high performance computing.

Once the optimum number of solvers was determined, four different tests were performed for any of the systems already mentioned:

T1: Standard serial SOR solver. The classical algorithm was coded and it is not necessarily the best implementation. No comparison with other possible implementations was done.

T2: The same as test T1 but with Red/Black ordering.

T3: Uniprocessor multicore solver. The optimum number of solver-threads was determined previously. This is our proposed method.

T4: The same test T3 but with Red/Black ordering.

Table 1: Times for 96,000 equations, system S5 and test T1

Run	Iterations	Time (seconds)
1	16,997	32.361
2	16,997	32.663
3	16,997	32.342
4	16,997	32.642
5	16,997	32.255

Table 1 shows the times for the benchmark calculation (T1) done on the S3 system with 96,000 internal equations. As expected, the number of iterations needed in each run is exactly the same.

Table 2 shows the results of the solution of the same 96,000 equations, but using SOR with Red/Black ordering (T2). For this number of equations, Red/Black ordering speeds-up convergence considerably. This behavior is consistent with the ones reported in the literature [9].

The last column of Table 2 shows the maximum difference of the results obtained with SOR-RB and SOR alone. Even though the

maximum difference is about 10 times larger than the tolerance, the results are correct. We would like to emphasize that, when solving large linear systems, the maximum difference of the solution, in two consecutive iterations, decreases very slowly. For this problem, a tolerance of $1 \cdot 10^{-12}$ not necessarily means that the solution has 12 correct decimal digits. The opposite is true only for small linear systems.

Table 2: Times for 96,000 equations, system S5 and test T2

Run	Iterations	Time (seconds)	Max. $\Delta\phi$
1	16,997	19.043	$4.13 \cdot 10^{-11}$
2	16,997	18.011	$4.13 \cdot 10^{-11}$
3	16,997	18.736	$4.13 \cdot 10^{-11}$
4	16,997	17.057	$4.13 \cdot 10^{-11}$
5	16,997	17.527	$4.13 \cdot 10^{-11}$

The results with the multicore solver and 3 solvers are shown in Table 3, for the same 96,000 equations.

Table 3: Times for 96,000 equations, system S5 and test T3

Run	Iterations	Time (seconds)	Max. $\Delta\phi$
1	17,074-17,691	12.475	$2.38 \cdot 10^{-10}$
2	16,902-17,289	11.613	$1.10 \cdot 10^{-10}$
3	16,741-17,323	11.676	$1.41 \cdot 10^{-10}$
4	16,890-17,774	11.960	$2.03 \cdot 10^{-10}$
5	16,931-17,364	11.638	$1.23 \cdot 10^{-10}$

The most interesting aspect to underline from Table 3 is that the execution times for this new method are considerably lower than the tests T1 and T2. This difference leads us to the conclusion that, for this case, keeping the FPU full is an effective solution.

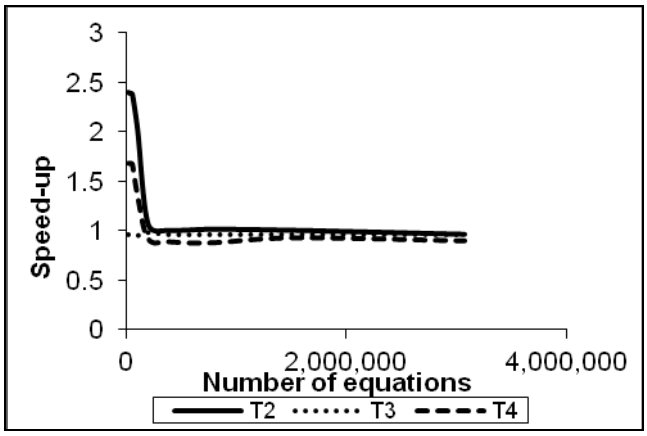

Figure 1: Speed-up in system S1 versus number of equations

Figures 1 to 7 show the results of the tests performed in all systems. The speed-up reported is calculated with respect to the test T1. In all figures, the values are the average of 5 calculations.

The results can be divided into two different groups. The first group consists in tests performed in systems with 2 single core processors and two physical threads. The second group of results is on systems with at least 3 solver-threads.

Figures 1 and 4 are the first group of results. There can be observed two different behaviors in the speed-up. When the problem size fits completely in cache memory, SOR with Red/Black ordering has the best performance.

There are several factors that come into play when determining if all the data will fit into the cache. There is the size of the cache, size of the data, and number of caches.

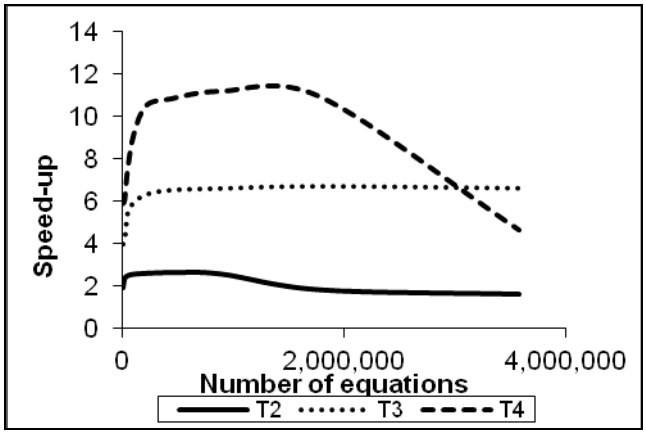

Figure 2: Speed-up in system S2 versus number of equations

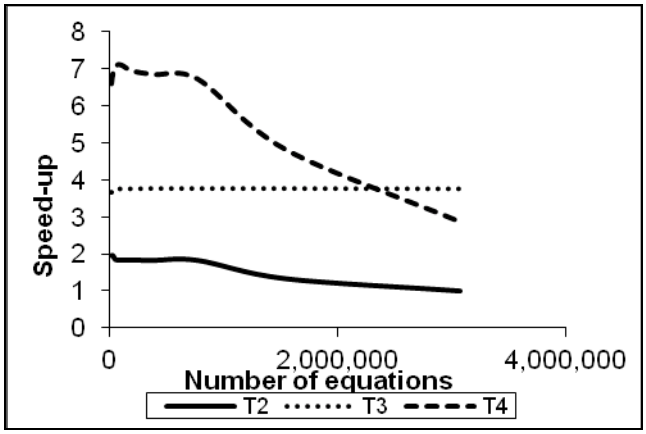

Figure 3: Speed-up in system S3 versus number of equations

Having all the data fit in cache legitimately is very rare. Given the sizes of scientific calculations it cannot be expected that there will be enough cache. Typically the way this will happen is a data set designed for a smaller system is used on a larger system that can further divide the data. Having data fit in cache when it previously didn't can cause super-linear speed-up and is a false comparison.

For this situation, the multicore solvers show poor performance, even if compared with the traditional SOR solver. The Red/Black ordering improves performance significantly but it is always below the SOR with RB. This is most likely because all of the threads in our multicore solver are constantly being swapped out

by the OS and losing time, while the traditional solver does not lose time to this overhead.

After the problem size exceeds the capacity of cache memory, performance of Red/Black ordering decays tremendously. This behavior has not been reported in the literature. When solving large linear systems, as in DNS/LES, it is not reasonable to think that all the data will fit in the cache memory.

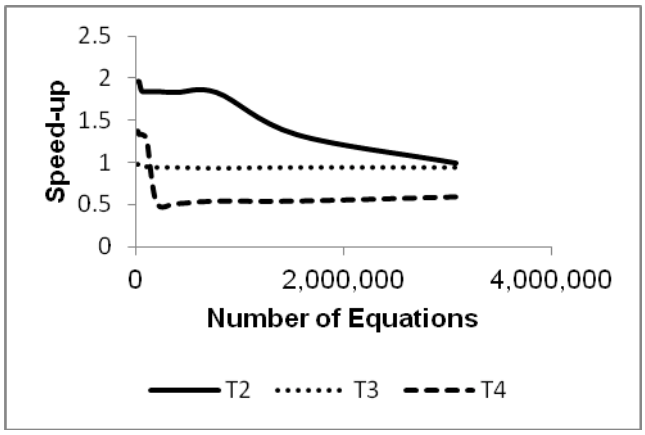

Figure 4: Speed-up in system S4 versus number of equations

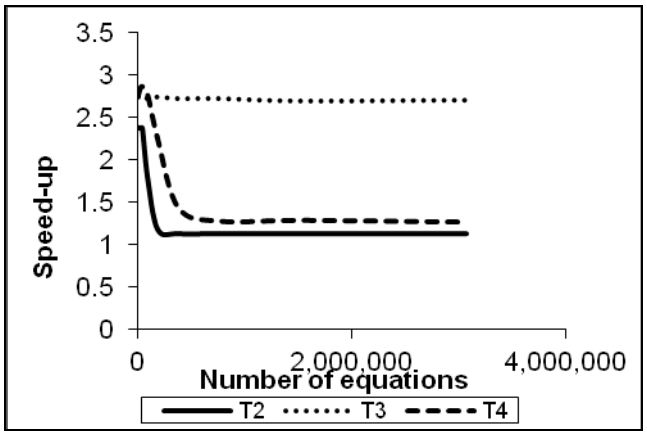

Figure 5: Speed-up in system S5 versus number of equations

The second group of results consists in Figures 2, 3, 5, 6 and 7. In all cases the tendency is similar and consistent. When the problem data fit in cache memory, the Red/Black ordering improves performance with respect to the traditional SOR. However, in this kind of systems, the multicore solver has better performance and it is always constant, no matter the problem size. It is interesting to note that Red/Black ordering boosts the performance of the multicore solver, but the speed-up collapses immediately as the problem data exceed the cache memory.

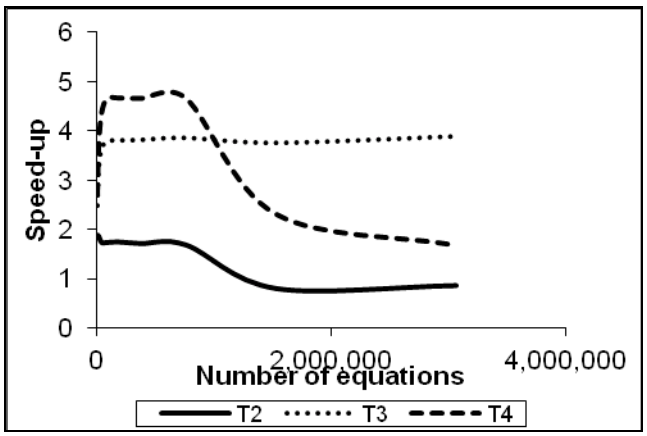

Figure 6: Speed-up in system S6 versus number of equations

From all these results it can be concluded that, for multicore/multithreaded systems, the optimum usage of the floating point unit is much more effective than trying to optimize cache accesses in a serial calculation.

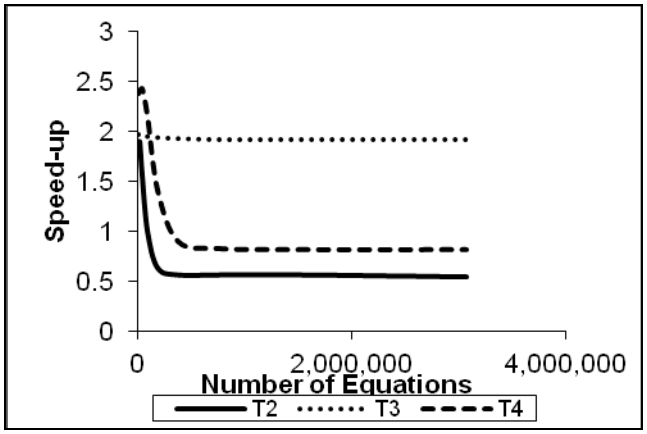

Figure 7: Speed-up in system S7 versus number of equations

6. TESTING THE GENERAL SOLVER

From the tests of the multicore solver it can be inferred that, in the evaluation of the distributed/multicore solver, the two most important aspects that will affect performance are the amount of equations per node and the communication overhead. The sub-domain method was selected for the parallelization of the problem since allows a complete distribution of data and work load between nodes.

Tests were performed in two different architectures:

Euler: Cluster with 64 nodes, each one with 2 AMD Opteron 248, 2.20 GHz, 1 MB cache, Rocks 5.1 / CentOs 5.2 operating system and g++ compiler version 4.1.2. MPI: Open MPI. Nodes/Processors available for tests: 50/100. Type of interconnection: GigabitEthernet (5)

Hilbert: Cluster with 64 nodes, each one with 2 Intel Xeon dual-cores, 3.00 GHz, 2 MB cache, disabled hyper-threading, Rocks 5.1 / CentOs 5.2 operating system and g++ compiler version 4.1.2. MPI: Open MPI. Nodes/Processors available for tests: 40/160. Type of interconnection: GigabitEthernet (5a)

Following the same procedure as done in the evaluation of the multicore solver, four distributed procedures were evaluated:

DP1: Standard serial SOR solver with communication performed at the end of any iteration. Number of copies of the same code per node is 2 in Euler and 4 in Hilbert

DP2: The same test as DP1 but with Red/Black ordering.

DP3: Distributed/Multicore solver. Number of solvers per node: 1 for Euler, 3 for Hilbert (plus the main thread)

DP4: The same test DP3 but with Red/Black ordering.

The first set of tests was to evaluate the performance as the number of nodes increases but keeping the same amount of equations per node. Each predefined number of nodes implies the solution of a different linear system, but it will help to understand how the previous different procedures perform. As done in the multicore solver tests, 5 calculations were performed in order to have representative values.

Table 4: Times (in seconds) for 96,000 equations per node in cluster Euler

Nodes	DP1	DP 2	DP 3	DP 4
2	23.681	17.298	25.842	19.855
4	28.989	22.579	27.595	22.847
8	45.895	39.515	28.082	26.036
16	63.610	56.643	28.386	27.800
32	92.534	86.042	28.540	29.434

Table 4 shows the average times for all tests done with 96,000 equations per node, and ranging from 2 through 32 nodes. From these results, two different readings can be done. The first is by observing the results horizontally. In this situation, the linear system to be solved is the same in all four tests. As predicted in the literature, Red/Black ordering improves the computing times in the whole range of nodes. When the number of nodes is small, the traditional approach is much better than the new method. On the other hand, when the number of nodes is increased, the overhead in communication takes an important amount of time in the traditional approach. It is very interesting to observe that the computing time almost does not change with the new solver. This can be explained because, in the new scheme, one processor is dedicated exclusively to send and receive data from the other nodes, while the other processor is updating values permanently.

The second reading that can be done to these results is vertically. It is true that the linear system solved is different (96,000*2, 96,000*4, 96,000*8, etc.) but the important increment of calculation time in the traditional solver is a good indicator of the overhead that the communication produces in the overall performance. It is convenient to remember that, in the traditional approach, no calculation is done while performing communications between nodes. For the case of 32 nodes, the new solver is quite faster than the traditional solver. This big difference, caused mainly by the overlap of communication and computation, is expected to produce important savings in computing time for a large number of nodes, used frequently in DNS simulations.

Table 5 shows the results for 384,000 equations per nodes. This number of equations is large enough to exceed the size of cache memory. These results are consistent with the ones shown in Table 4. Once again, with a small number of nodes, the traditional solver performs better than the new one but this tendency is reverted as the number of nodes increases. The slower memory speed makes the difference in time smaller, but it is expected that this magnitude should increase as the number of nodes increases.

Table 5: Times (in seconds) for 384,000 equations per node in cluster Euler

Nodes	DP1	DP2	DP3	DP4
2	66.492	88.116	108.926	195.906
4	71.005	97.242	109.458	199.931
8	90.756	111.339	109.848	204.168
16	113.419	136.368	111.987	204.527
32	142.855	161.765	115.748	203.245

By observing the results vertically, the overhead in communication is consistent in all cases. The new method has a small overhead in communication, producing important savings in computing times.

Table 6: Times for 192,000 equations per node in cluster Hilbert

Nodes	DP1	DP2	DP3	DP4
2	30.068	20.581	22.291	13.377
4	46.568	38.876	23.613	14.800
8	72.374	64.047	25.922	15.690
16	107.518	96.971	24.597	14.544
32	167.518	159.298	25.510	14.944

Table 7: Times for 764,000 equations per node in cluster Hilbert

Nodes	DP1	DP2	DP3	DP4
2	78.951	69.543	87.760	106.335
4	97.567	86.628	89.223	109.984
8	122.521	110.417	92.248	115.064
16	161.747	152.855	95.181	116.054
32	226.555	217.597	97.624	122.404

Looking at Tables 6 and 7, results on the cluster Hilbert have similar behavior. It is very interesting that in this quad-processor distributed system; there is a big difference in performance when the problem is in cache memory. These results confirm our hypothesis that having many threads accessing data in the same memory bank should produce better performance. In a similar manner, performance of our proposal is clearly superior as the number of nodes increases. It calls the attention that, in both

traditional approaches, communication overhead degrades performance considerably.

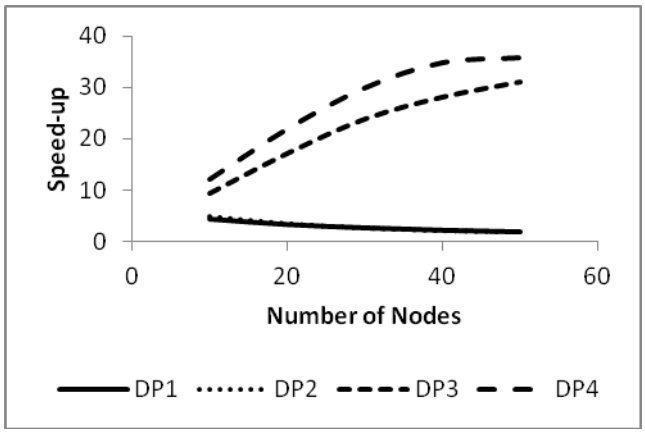

Figure 8: Speed-up for 720,000 equations on Euler cluster

Figure 8 shows the speed-up in the solution of 720,000 equations, ranging the number of nodes from 10 to 50. This linear system is small enough to fit entirely in cache memory in all cases. These results are very interesting because, due to the problem size, the calculation time is very small; relying performance mainly on communication between nodes.

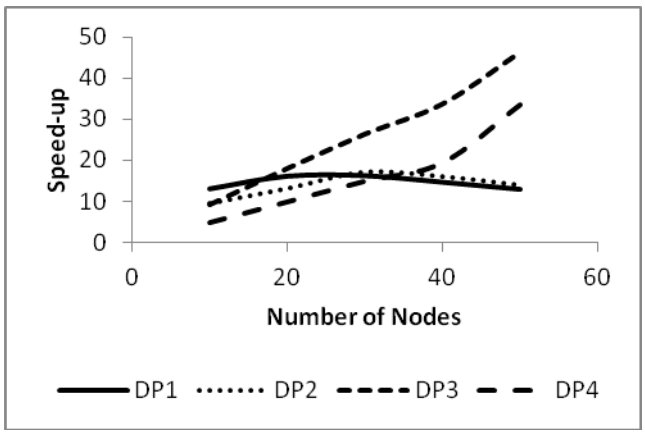

Figure 9: Speed-up 6,000,000 equations on Euler cluster

For the traditional solvers, even that this problem size is ideal for Red/Black ordering, there is no difference in speed-up between solution DP1 and DP2. On the other side, performance of both multicore solutions is very good and, eventually, speed-up stabilizes as the number of nodes reaches 50. As expected, in all these solutions that use global MPI communication operations, network speed will impose a limit in the amount of data that can be transferred.

On the contrary, the number of iterations for convergence increases permanently in both multithreaded solutions. This issue becomes more evident in the solution DP4 and 50 nodes, where the number of iterations required increases significantly. Due to the fact that, in method DP4, calculation and communication are done at the same time, this increment in iterations is basically more communication time. For both multicore solutions, the additional overhead has little effect on performance in comparison with the traditional schemes.

Figure 9 shows the comparison in speed-up for a linear system of 6,000,000 equations. For this problem size, for a number of nodes of 30 and less, data size is larger than L2 cache memory. For a number of nodes of 40 and 50, all data fit entirely in cache memory.

For this linear system, performance behavior of the classical approaches DP1 and DP2 is similar with reported in the literature; with an increasing speed-up and, when a maximum is reached, communication overhead produces a small degradation in performance. However, the situation is different for the multicore solutions DP3 and DP4. When the number of nodes is small, the traditional approach is superior but, the multithreaded approach DP3 has better performance as soon as the number of processors increases. The behavior of the solution DP4, the multicore solver with Red/Black ordering, is quite interesting because, for a small number of processors, this method has the worst performance of all solutions. The just described situation remains unchanged until the point in which the amount of nodes imposes an important load in the communication lines. Additionally, as soon as the problem size fits completely in cache memory, performance of the procedure DP4 is boosted considerably.

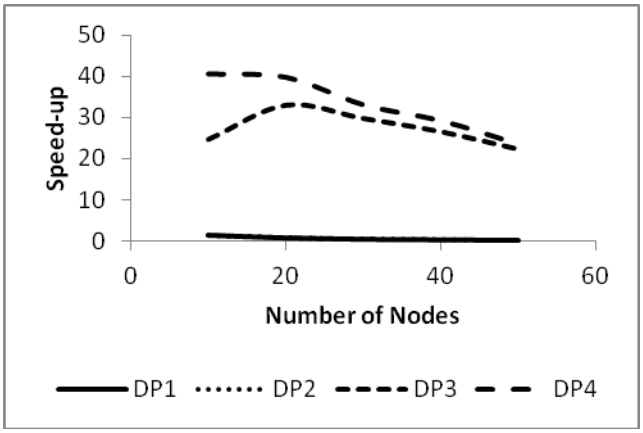

Figure 10: Speed-up for 720,000 equations on Hilbert cluster

For the multicore procedures DP3 and DP4, communication time is at most the same calculation time. Since in the multithreaded solutions, calculation and communications is done at the same time, and considering that the exchanging of boundary values is done at the end of the iteration of the first solver, calculation time is expected to be larger than system time.

Figure 10 shows the speed-up for the solution of 720,000 equations on the cluster Hilbert. For the traditional implemmentations DP1 and DP2, performance is quite similar to the ones obtained in the cluster Euler. It is very interesting that the performance of the new solver (DP3 and DP4) is clearly superior.

Even that the solution of a small system is not convenient with a large number of nodes, the big difference in performance confirms

that the separation of FPU/non-FPU operations is different threads takes the most out of the new architectures.

Finally, the case of 6,000,000 equations shows some interesting results. First of all, the difference in performance of the traditional approaches (DP1 and DP2) and the new ones (DP3 and DP4) is very big.

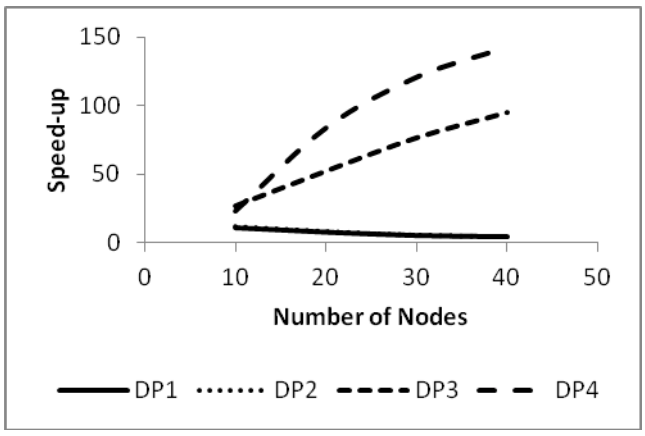

Figure 11: Speed-up for 7,200,000 equations on Hilbert cluster

In Figure 11 the performance of the standard solvers decays permanently, due basically to the communication overhead. For the new multithreaded implementations, the difference in performance is important between the dual-core cluster and the quad-core cluster. These results are consistent with the multicore tests; in which our proposal performs better in new architectures.

As mentioned previously, the sudden change in the way data are stored and accessed (from main memory to cache memory), makes the procedure considerably faster and cannot be considered as a property of the method. This super-linear behavior is a combination of multithreaded calculations, data stored in L2 cache memory and communication of data done at the same time; all this in a multicore cluster.

7. CONCLUSIONS

A distributed/multicore solver has been proposed to solve a 2D Poisson equation. The distribution of tasks in FPU intensive and non-FPU intensive threads produces an important improvement in performance in the new multicore architectures. The classical approach seems the best choice for uniprocessor systems.

For the distributed/multicore solver, the proposed distribution of tasks produces important savings in time as the number of nodes increases.

The Red/Black ordering technique was found to boost speed-up only if data fit completely in cache memory. This issue is very important because for very large linear systems, is more likely to have a big amount of data per node.

In this work we demonstrated that by having threads that are FPU intensive and non-FPU intensive, performance on a single processor can be increased. This technique also allowed us to make the communication between nodes asynchronous. Our approach of decoupling communication and computation allows for much greater scalability. Not only are these scientific simulations now scalable but they will also be much faster.

8. REFERENCES

[1] Chau, M., El Baz, D., Guivarch, R., Spiteri, P., 2007, "MPI implementation of parallel subdomain methods for linear and nonlinear convection-diffusion problems", Journal of Parallel and Distributed Computing, vol. 67, pp. 581-591.

[2] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A., 2003, *Source Book of Parallel Computing*, Morgan Kauffmann Publishers.

[3] Garbey, M., Vassiliesvky, Y.V., 2001, "A parallel solver for unsteady incompressible 3D Navier-Stokes equations", Parallel Computing, vol. 27, pp. 363-389.

[4] Henshaw, W.D., Schwendeman, D.W., 2008, "Parallel computation of three-dimensional flows using overlapping grids with adaptive mesh refinement", Journal of Computational Physics, vol. 227, pp. 7469-7502.

[5] Hsu, H.-S., Hwang, F.-N., Wei, Z.-H., Lai, S.-H., Lin, C.-A., 2011, "A parallel multilevel preconditioned iterative pressure Poisson solver for the large-eddy simulation of turbulent flow inside a duct", Computers & Fluids, vol. 45, pp. 138-146.

[6] Launder B., Sandham N., 2002, *Closure Strategies for Turbulent and Transitional Flows*, Cambridge University Press.

[7] Ng, K. F., Mohd Ali, N. H., 2008, "Performance analysis of explicit group parallel algorithms for distributed memory multicomputer", Parallel Computing, vol. 34, pp. 427-440.

[8] Press, William H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., 2007, *Numerical Recipes, The Art of Scientific Computing*, Third Edition, Cambridge University Press.

[9] Rauber, T., Runger, G., 2010, *Parallel Programming*, Springer.

[10] Sonzogni, V. E., Yommi, A. M., Nigro, N. M., Storti, M. A., *A parallel finite element program on a Beowulf cluster*, Advances in Engineering Software, vol. 33, pp. 427-443, 2002.

[11] Wadleigh, K. R., Crawford, I. L., *Software Optimization for High Performance Computing*, Hewlett-Packard Professional Books, 2000.