Chapter 7

Domain Decomposed Additive Correction Multigrid

7.1 Introduction

In previous sections we have considered three approaches for the solution of algebraic equations arising from discretization of PDEs, but none satisfies all of our requirements:

1. ACM (section 3) is a robust, totally algebraic MG algorithm, without a long pre-processing stage, that is efficient as a sequential algorithm. However, it is not useful for loosely coupled systems as it is very fine-grained. According to [194], the efficiency of parallel ACM algorithms depends on the latency. This is confirmed by the poor speedups obtained in a fixed number of smoothing iterations on small meshes (see section 7.3).

2. DDV parallel MG, and the other parallel MG variants in section 5, are efficient and tolerant to relatively high latency networks but require discretization of the governing equations at the different levels (geometric multigrid). So it is not in principle compatible with an algebraic approach, as it would be needed for DPC code.

3. Schur complement (section 6) is totally algebraic and very efficient for constant left hand side problems such as pressure correction equations of incompressible flows, our main target (section 1.2), even on loosely coupled systems. However, it needs large amounts of memory so the maximum size of the problems to be solved is restricted.

Our goal now is to devise a way to combine the positive properties of the three methods, obtaining a reasonably efficient algebraic solver for loosely coupled systems, mainly to be used for pressure correction equations of medium and large scale problems.

The proposed algorithm, DDACM (Domain Decomposed Parallel Additive Correction Multigrid) is a MG algorithm based on the ACM correction equations [137], with both pre and post smoothing, without the need of halo update operations at the second leg, that uses Schur complement as a direct solver for the coarsest level and BILU (Block Incomplete LU factorization) as a smoother for the intermediate levels.

Here, the notation of section 3.2.2 is recovered. This is, finest level is labeled 1 and coarsest level M. As in the rest of the work, we assume for simplicity that a two-dimensional problem is to be solved. Like in DDV, a non-recursive formulation of the algorithm is used since it allows a better control of the communications. It is also divided into a first and a second leg.

DDACM, like DDV, relies on a direct solver for the coarsest level. However, it has a much more important role in DDACM. In DDV, as pre-smoothing operation is suppressed, the halos of all the levels can be updated simultaneously. Doing so, the communication time does not increase

substantially with the number of levels (as the size of the smaller levels is neglectable). Thus, the number of levels can be increased until the time to solve the smallest is neglectable and then use a sequential solver.

On the other hand, in DDACM pre-smoothing operation is not suppressed. The halos are updated level by level in the first leg of the algorithm. In spite of the neglectable size of the smaller levels, the communication cost increases due to latency. To avoid this problem, DDACM reduces the number of levels by means of using an efficient direct parallel Schur solver at an intermediate (relatively large) level.

Although they are mainly equivalent, the DDACM algorithm is easier to implement than the DDV algorithm (if the Schur complement subroutine is not considered). In particular, in DDACM the analysis of the data dependencies that allow to reduce the comunications is not as involved as in DDV. This is due to the lower order of the restriction and prolongation operators used in DDACM and to the use of CS instead of FAS.

7.2 Domain partition and halos

In section 5.2, the mesh disposition used in DDV algorithm was discussed. In DDACM, the problem begins after the discretization so we do not have to consider a mesh but a regular arrangement of unknowns in a two-dimensional pattern, like in sections 3.2 and 6. Each node is related to the neighbours only through the coefficients of the discrete equations. The underlying PDE is not needed. A two-dimensional domain decomposition is done, $P = P_x P_y$. Both the processor index p and its position in the processors mesh (p_x, p_y) are used in the algorithm description.

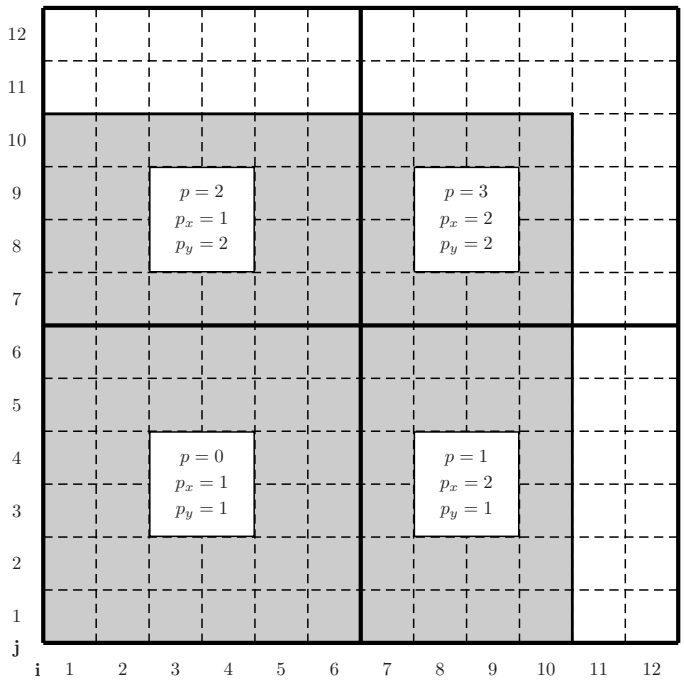

Figure 7.1: Schema of the domain decomposition used by DDACM algorithm.

A *disjoint* partition of the domain is done at the first level. This is, unlike in DDV algorithm (section 5.2), there are no nodes shared between two or more processors. A halo of J neighbouring points in each direction is added to the area accessible to each processor. To ease the implementation of the algorithm in a pre-existent code, it is advisable to do the partition outside the linear solution algorithm. Doing so, an arbitrary partition, not necessarily balanced from the linear solver point of view, can be imposed. This can be done according to other criteria, such as different amounts of

work per unknown in different regions of the domain in other stages of the algorithm.

An example of such an arrangement can be seen in Fig. 7.1, where a case with $N = 144, N_x = N_y = 12, P = 4, P_x = P_y = 2, J = 4$ is presented. The nodes accessible to $p = 0$ are filled. A gobal numbering scheme is used (according to the methodology discussed in section 4.6).

Like in DDV algorithm, if the domain is decomposed in two dimensions, halo update operation involves data exchange between each processor and its 8 neighbours (at N, S, E, W, NW, NE, SW, SE). To update the halos, each processor just replaces its halo values with data from the inner areas of its neighbours. No averaging operations are used.

7.2.1 Halo update

From the point of view of computing time, halo update operation is a critical aspect of parallel MG algorithms. The process (that was not described for the case of geometric parallel MG algorithms in section 5), can be divided into three stages: packing, communication and unpacking.

Packing

In the packing stage, that does not involve use of the network, each processor collects the data from its inner areas to be sent to each of its neighbours and stores them in a set of vectors, one per each neighbouring processor.

In DDV and DDACM, there are vectors of fields of different sizes to be updated in the same communication. They are packed correlatively in a single vector per neighbouring processor. The packing stage begins with a vector of fields (that can be unknowns, right hand sides, etc) and ends with a set vectors to be sent to the neighbouring processors.

For tightly coupled parallel computers, packing/unpacking time can be comparable to communication time (due to the access to non-correlative memory positions) so it is worth optimizing the process. This is not the case for loosely coupled systems, where it is a small fraction of the total communication cost.

Communication

In the communication stage, that involves use of the network, each of the vectors with halo data is transfered to the corresponding neighbour. This is done by means of a sequence of calls to the low-level point-to-point send and receive functions provided by the communications library (MPI in our case). There are many options to do so. The main requirements are to guarantee the absence of deadlocks ([155], section 3.4) and to have good performance.

Communication functions can be used for halo update or for any other purpose. According to the software engineering approach used in the implementation (section 4.6), they are part of the lower layer of the code. Thus, the general case in which each processor has to exchange data (receive, send or send-and-receive) with any other processor is considered here.

Three different functions have been implemented and benchmarked in JFF cluster:

- Mode 0: Totally asynchronous ([155], section 3.7). Each processor initiates all the send and receive operations and then waits until they have been completed. This is done using non-blocking communications, as shown in Alg. 7.1. $sb[p]$ and $rb[p]$ are the previously allocated send and receive buffers. The send start calls (MPI_Issend) initiate the sending operations but return before they are completed. The same holds for the receive start operations (MPI_Irecv). Each MPI_Wait call returns when the corresponding operation has been completed.

  This is, in mode 0, each processor exchanges data simultaneously, in both directions, with all its neighbours.

  As it is not necessary to specify an order in the sequence, this mode can be used safely (i.e., without risk of deadlocks) for arbitrary communication patterns. This has been the approach

```
Mode 0 halo update {
        for p in my_neighbours:
                MPI_Issend (sb[p], p)
        for p in my_neighbours:
                MPI_Irecv (rb[p], p)
        for (all operations started)
                MPI_Wait
}
```

Algorithm 7.1: Mode 0 halo update.

used in section 4.7, where any communication pattern (arising from different dispositions of the subdomains) can be imposed by the user. However, this mode might not be the optimal solution. An additional inconvenient is the necessity of allocating all the buffers simultaneously. This mode has been used for the pre-processing stage in our implementation of the Schur complement.

- Mode 1: Blocking communications, with simultaneously Send-Receive operation between pairs of processors. In order to avoid deadlocks, when blocking communications are used, a valid sequence of messages has to be defined. It has to guarantee that send and receive operations always match. For the halo update operation here under consideration, one of the possible sequences is outlined in Alg. 7.2.

```
Mode 1 halo update
    Lateral halos
        if ( p_x + p_y is even) {
                exchange_data (N)
                exchange_data (S)
                exchange_data (E)
                exchange_data (W)
        }
        else {
                exchange_data (S)
                exchange_data (N)
                exchange_data (W)
                exchange_data (E)
        }
    Diagonal halos
        if (p_x is even) {
                exchange_data (NE)
                exchange_data (NW)
                exchange_data (SE)
                exchange_data (SW)
        } else {
                exchange_data (SW)
                exchange_data (SW)
                exchange_data (NW)
                exchange_data (NE)
        }
```

Algorithm 7.2: Mode 1 halo update.

Where p_x, p_y are the coordinates of p in the map of processors. In mode 1, each of the

exchange_data operations is implemented with asynchronous communications, as in mode 0 for each Send-Receive pair.

```
exchange_data (p) mode 1 {
                MPI_Irecv( p )
                MPI_Send( p )
                MPI_Wait
}
```

Algorithm 7.3: Mode 2 halo update.

Thus, in mode 1, each processor exchanges data in both directions with only one of its neighbours. Inspection of LAM (http://www.osc.edu/lam.html) source code revealed that MPI_Sendrecv function is implemented as in the previous pseudo-code, so the LAM function was used.

- Mode 2: Blocking communications, with consecutive send and receive operation for each pair of processors. The same sequence as in mode 1 is used (Alg. 7.3), but for mode 2, communications between each pair of processors are not started simultaneously. Alg. 7.4 is used to do so.

```
exchange_data (p) mode 2 {
                if ( myrank > p ) {
                                MPI_Ssend( p )
                                MPI_Recv( p )
                } else {
                                MPI_Recv( p )
                                MPI_Ssend( p )
                }
}
```

Algorithm 7.4: Mode 2 data exchange.

Thus, in mode 2, each processor exchanges data (only sending or only receiving) with only one of its neighbours.

Unpacking

In the unpacking stage, the packing process is reversed. Each processor distributes the vectors of numbers received from each of its neighbours in the appropriated position of the halos of the different fields transferred. The network is not used.

7.2.2 Benchmarking halo update in JFF cluster

Time to do the halo exchange operation varies between calls. The experiment has been repeated 100 times, recording the time that processor 0 needs to do the communication (not including the time spent in packing and unpacking operations). Depending on the order chosen, results would be slightly different if measured in other processors. As our goal here is to measure the performance of the network, the packing times have not been included in the measure.

For modes 0 to 2, with $P = 16$ processors, for different mesh sizes, the minimum, maximum and average times can be seen in Figs. 7.2, 7.3 and 7.4.

Mode 1 was chosen as it is only slightly inferior to mode 2 for the larger meshes and has the best repeatability. According to the results of Fig. 7.5, the use of different modes as a function of the mesh size could be considered.

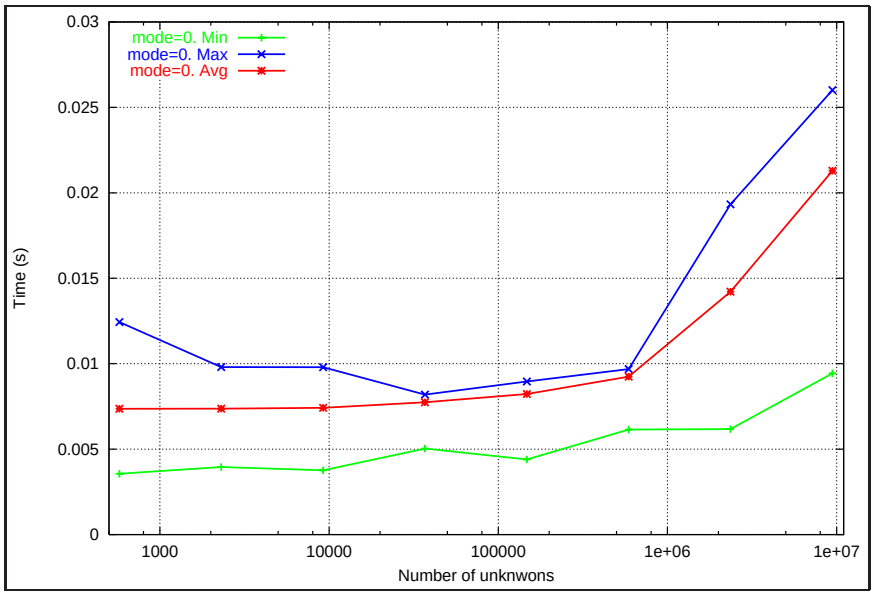

Figure 7.2: Minimum, maximum and average time to update a 4 nodes halo with $P = 16$ on the JFF cluster using mode 0.

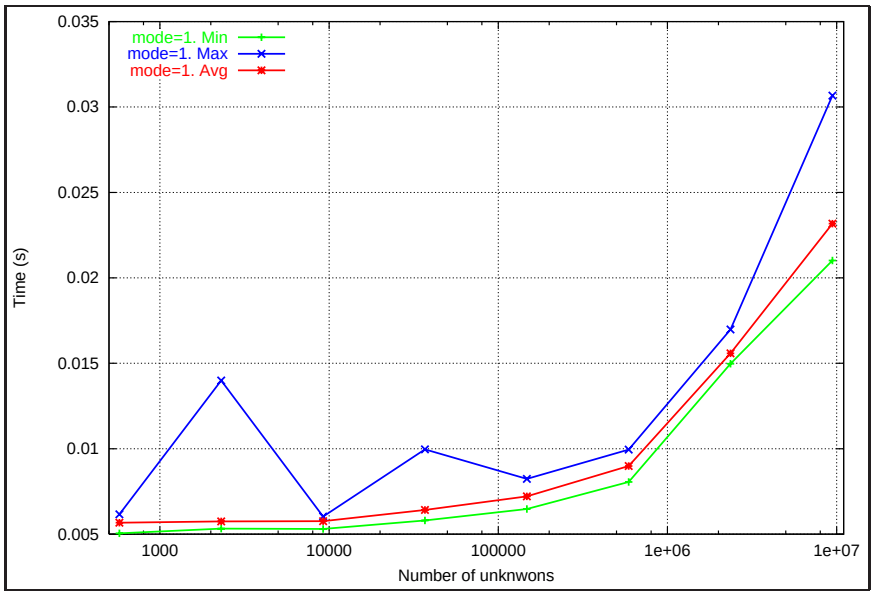

Figure 7.3: Minimum, maximum and average time to update a 4 nodes halo with $P = 16$ on the JFF cluster using mode 1.

Next experiment was to study the influence of the number of processors. In principle, halo update operation involves only the neighbours of each processor so it should be independent of P. Average and maximum times, for a sample of 100 halo updates can be seen in Figs. 7.6 and 7.7. The time depends on the number of processors. Surprisingly, maximum time is for $P = 9$. Variability of the maximum times is high, even using mode 1. Its causes are beyond the scope of this work. It has an adverse influence on the total computing time.

One of the key aspects of DDV and DDACM algorithms is their ability to update the halos of all the levels in a single communication episode, saving the time due to latency. To estimate the impact of this feature on the global performance of the algorithm, the time to update the halos of a set of levels has been measured, including packing and unpacking. On the JFF cluster, packing

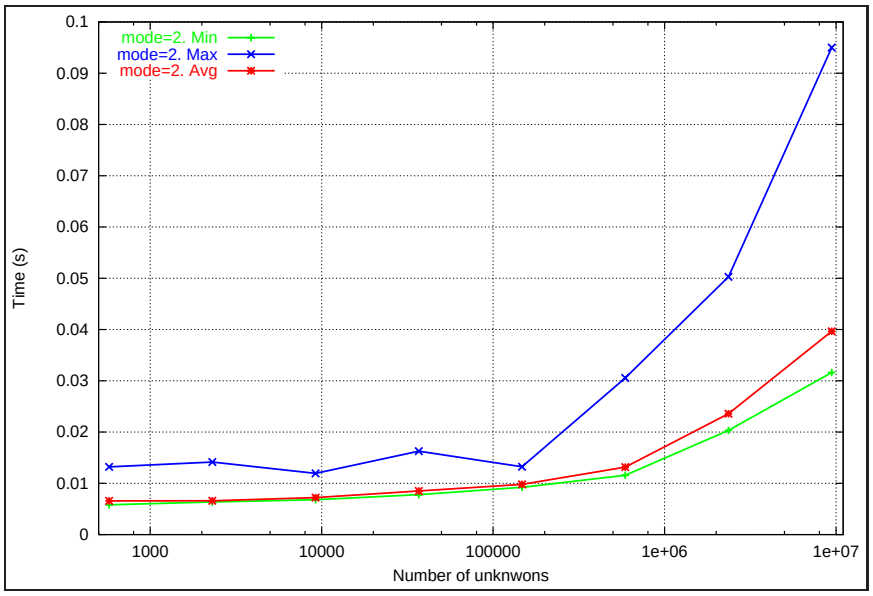

Figure 7.4: Minimum, maximum and average time to update a 4 nodes halo with $P = 16$ on the JFF cluster using mode 2.

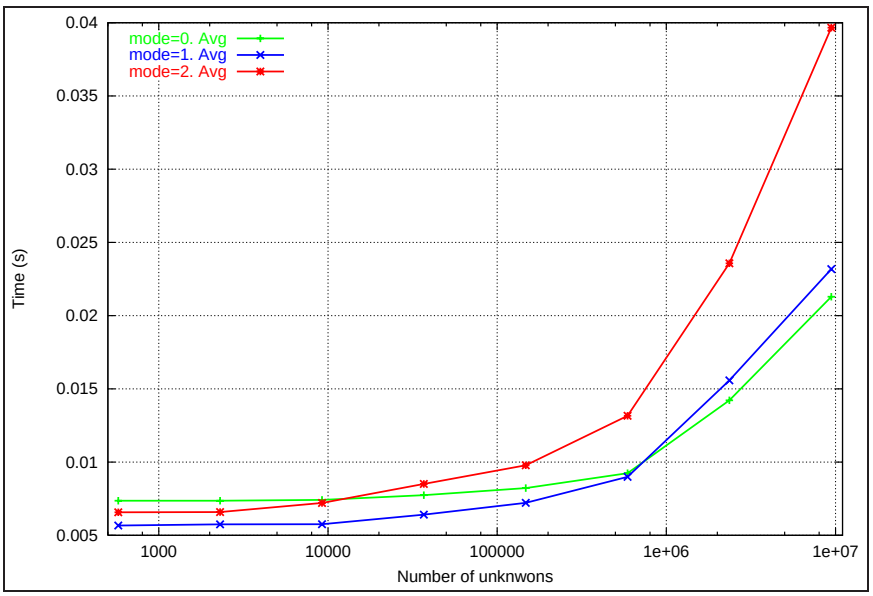

Figure 7.5: Comparison of average times to update a 4 nodes halo with $P = 16$ on the JFF cluster using the different modes.

time is small compared with communication time.

First, the halos were updated level by level and then in a single message. The experiment was done with the finest meshes of 768^2 and 3072^2 nodes. Times measured here include communication and packing. In our implementation, packing time is roughly equal when updating the halos simultaneously or level by level. Results can be seen in Fig. 7.8. As an example to clarify the figure, consider the situation with a 768^2 mesh. If the first level updated is $l = 5$, the levels updated are $l = 5$ and $l = 6$, with 348^2 and 768^2 nodes. As expected, the times are equal when only one level is updated. Times include communication and packing. The sample size was 100. Mode 1 was used with 16 processors.

As it can be seen, when updating the halos one by one, the problem is dominated by the latency.

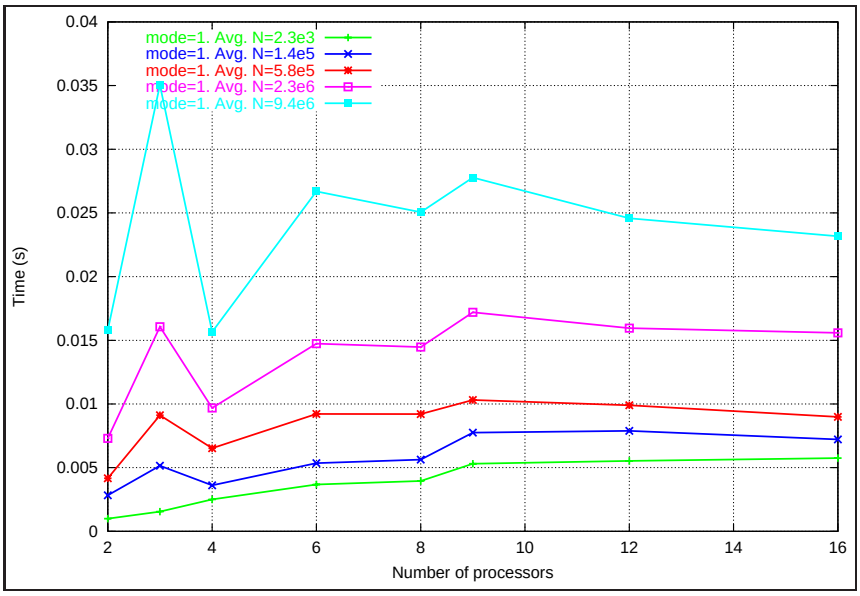

Figure 7.6: Average communication time to update a 4 nodes halo on the JFF cluster for different numbers of processors, using mode 1.

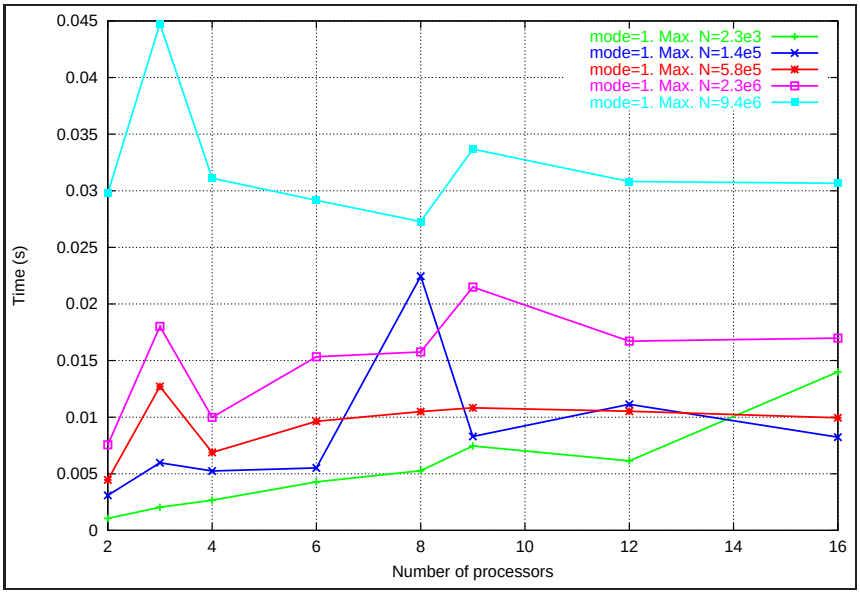

Figure 7.7: Maximum communication time to update a 4 nodes halo on the JFF cluster for different numbers of processors, using mode 1.

Thus, the cost of updating an additional level is almost independent of the number of unknowns in the level. However, if a single message is used, the number of levels has a smaller incidence on the total time.

7.3 Block Incomplete Lower-Upper Smoothing

Block Incomplete Lower-Upper decomposition (BILU) is a Jacobi algorithm (section 4.5.1) that uses an incomplete LU factorization (specifically, MSIP [108]) iterative solver for each block.

Using BILU (instead of RBGS as in the DDV solver), the smoothing stage is always *dependent*

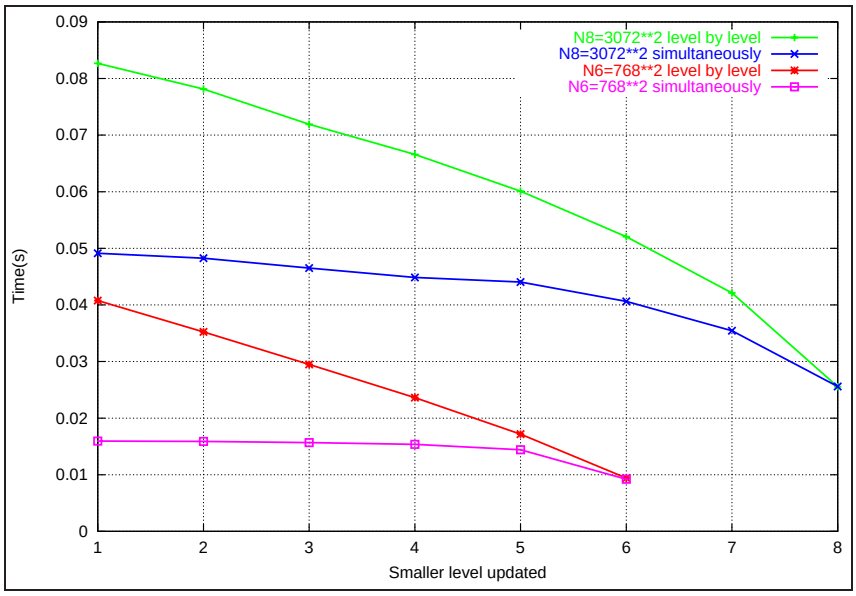

Figure 7.8: Average time to update a 4 nodes halo of a group of levels, simultaneously or level by level.

on the number of processors, no matter how big are the overlapping areas. However, according to previous experiments with GS and MSIP as smoothers for sequential ACM, the MSIP version is (in general) more efficient. Additionally, one of the main motivations of DDACM is to keep it as similar to the sequential ACM as possible, to guarantee an easy transition to the parallel DPC version.

Thus, DDACM, at least on this BILU-based first implementation, is dependent on the number of processors [1]. So there are no nodes strictly "correct" (in the sense of chapter 5), but many of the conclusions obtained in chapter 5 are still valid qualitatively.

As in the general Block-Jacobi algorithm of chapter 4, each processor divides its part of matrix A into a local A_{loc} plus a non-local part A_{nb}:

$$A = A_{loc} + A_{nb} \tag{7.1}$$

Then, the solution is obtained iteratively using:

$$A_{loc}x^{k+1} = b_{loc} = b - A_{nb}x^k \tag{7.2}$$

To solve for x in this *local* equation, each processor does ν_{loc} ILU iterations, using the approximate LU decomposition of A_{loc}, as in equation (2.74),

$$L'_{loc}U'_{loc}\Delta^{k'+1} \approx r^{k'}_{loc} \tag{7.3}$$

where k' is a local iteration counter and $\Delta^{k'+1} = x^{k'+1}_{loc} - x^{k'}_{loc}$ is the increment of the local vector. Each processor evaluates and stores the ILU decomposition of its A_{loc} matrix (equation 2.67), obtaining L'_{loc}, U'_{loc}. Alg. 7.5 is used in DDACM to do a total of ν iterations.

To evaluate the efficiency on the algorithm as a smoother on the JFF cluster, a fixed number of iterations has been done, with $\nu_{loc} = 1$. As BILU is dependent on the number of processors, the norm of the residual at the end of the iterations is a function of P.

The measured speedups, for different numbers of processors and mesh sizes have been represented in Fig. 7.9. As expected, the speedup increases with the mesh size. However, only for the larger

[1] Even with smoothers independent of the number of processors, MG algorithms tend to be dependent on the number of processors (section 7.4).

BILU {
  do $k = 1 \rightarrow \frac{\nu}{\nu_{loc}}$ {
  Update x halos
  Recalculate $b_{loc} = b - A_{nb}x$
  for $k' = 1 \rightarrow \nu_{loc}$ {
    Do a local iteration:
      $r_{loc} = b_{loc} - A_{loc}x$
      Solve $L'_{loc}U'_{loc} = \Delta\, r_{loc}$
      $x \leftarrow \Delta + x$
  }
}

Algorithm 7.5: BILU smoother used in DDACM.

meshes it is close to its theoretical value for $P > 9$. For meshes smaller than $N = 192^2$, we actually have $S < 1$: the CPU time *increases* due to the use of the parallel computer.

These results indicate that solvers based on BILU (or on any approach as small grained as BILU) are useless on loosely coupled computers, except for very large meshes. However, as MG is based on a hierarchy of levels, the time gained in the dense levels can easily be lost in the coarse levels, obtaining modest speedups (if any at all).

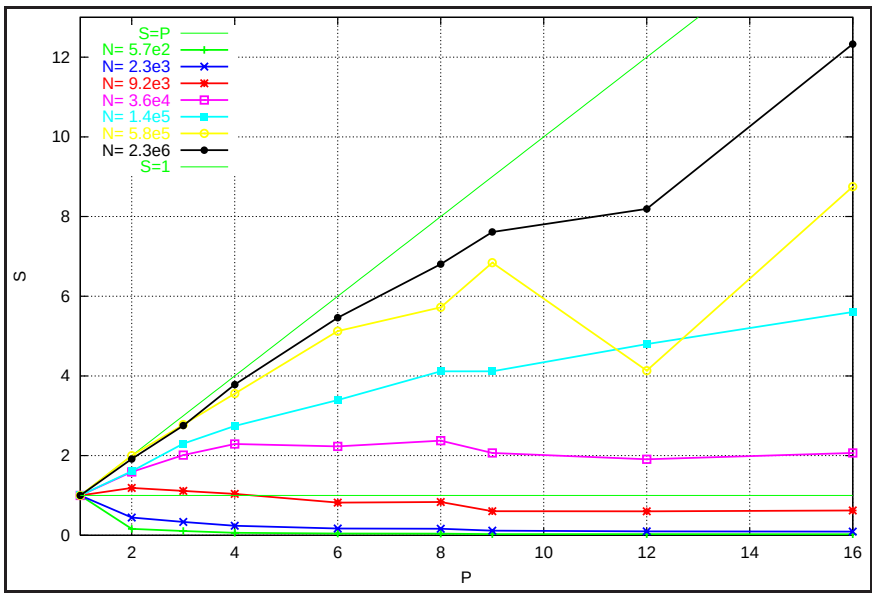

Figure 7.9: Speedup obtained in a fixed number of BILU iterations on the JFF cluster with different mesh sizes, with $J = 4$ and $\nu_{loc} = 1$.

7.4 Block partitions and MG equations

To do the block partition of the domain (as in section 3.2), care has to be taken to avoid joining in the same block nodes owned by different processors. Thus, "special" blocks of 2x1, 1x2 or 1x1 equations might be needed near the processors' boundaries.

A global numbering of the nodes has been used (section 4.6), so each processor has to know the initial index of its correction domain in both axes. Thus, when the block partition of each axis is done, processor q should know the number of blocks of the partitions done by processors $p < q$.

The algorithm to generate the block partitions has to ensure that they are consistent for all the processors, e.g., the blocks at the halo regions should be identical for the processors at both sides of the boundary. To solve these problems, considering that the CPU time involved in the block partition is small, in our implementation every processor generates the partition for all the domain in each of the axis.

Multidimensional block partitions are generated as the Cartesian product of the decomposition in each axis. As in the case of DDV algorithm, the number of nodes for each processor in each axis is restricted to be larger than the halo size J. Otherwise, communication schemes would be much more complex, involving the neighbours of the neighbouring processors.

An example of one dimensional domain decomposition, with $N_x = 18$, $P_x = 2$, $J = 4$, can be seen in Fig. 7.10. Each of the equations of a level is represented with a box. The total number of unknowns of each level, denoted N_x^l, is shown at right hand side. The levels are stacked, with $l = 1$ at the top and $l = M$ at the bottom. A thick vertical line is used to indicate the limits of the areas owned by the different processors. The unknowns accessible to processor $p = 0$ are dashed. As $J = 4$, there is a halo of 4 nodes owned by $p = 1$ that can be used by $p = 0$ to read/write temporary information. Similar conventions are also used in Figs. 7.11 and 7.12.

In the situation presented in Fig. 7.10, it would not be possible to use a third level as the number of nodes of $p = 1$ would be smaller than $J = 4$. This aspect is discussed in the context of DDV algorithm in section 5.4.4. Also, note how the block 5 of level 2 is formed joining only one node of level 1, to avoid the division between $p = 0$ and $p = 1$ areas.

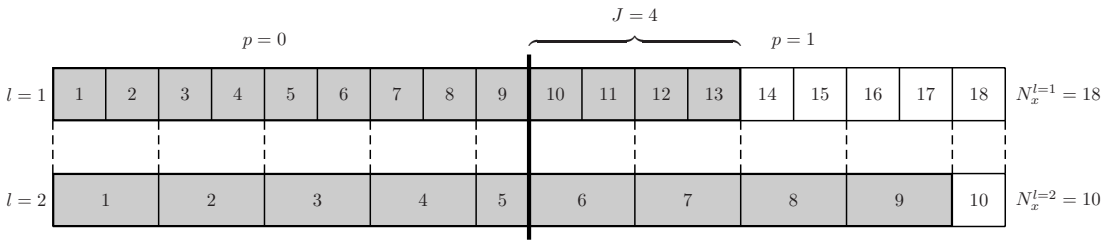

Figure 7.10: Block partitions in DDACM.

Due to the need of using special block sizes near the boundaries of the processors' regions, the correction equations depend on P unless the mesh sizes are carefully chosen as in DDV. This is shown in an example in Fig. 7.11, where a problem with $N_x = 18$, $J = 2$ is solved with $P = 2$ and $P = 3$ processors. The situation with $P = 3$ is presented at the top. As the number of nodes owned by each processor is even, all the blocks contain two equations. This is not the case for the situation with $P = 2$, at the bottom. Thus, level 2 equations are different for different numbers of processors. Also the maximum number of levels can be different, as in this example.

These small differences, that can not be avoided as the number of mesh nodes has to be fixed according to other criteria, cause parallel DDACM algorithm to be (slightly) dependent on the number of processors (section 4.3.2).

7.5 Algorithm

7.5.1 Pre-processing

Once the block partition is available, each processor can generate coefficients for the correction equations of its subdomain, using expressions (3.33), just like in the sequential ACM. Recall that this is done using only the matrix A of the initial level. However, like in DDV algorithm, to be able to relax the overlapping areas of the different levels, the corresponding coefficients must be available. The only solution is to obtain them from neighbouring processors. This can be done for all the levels together in a single message.

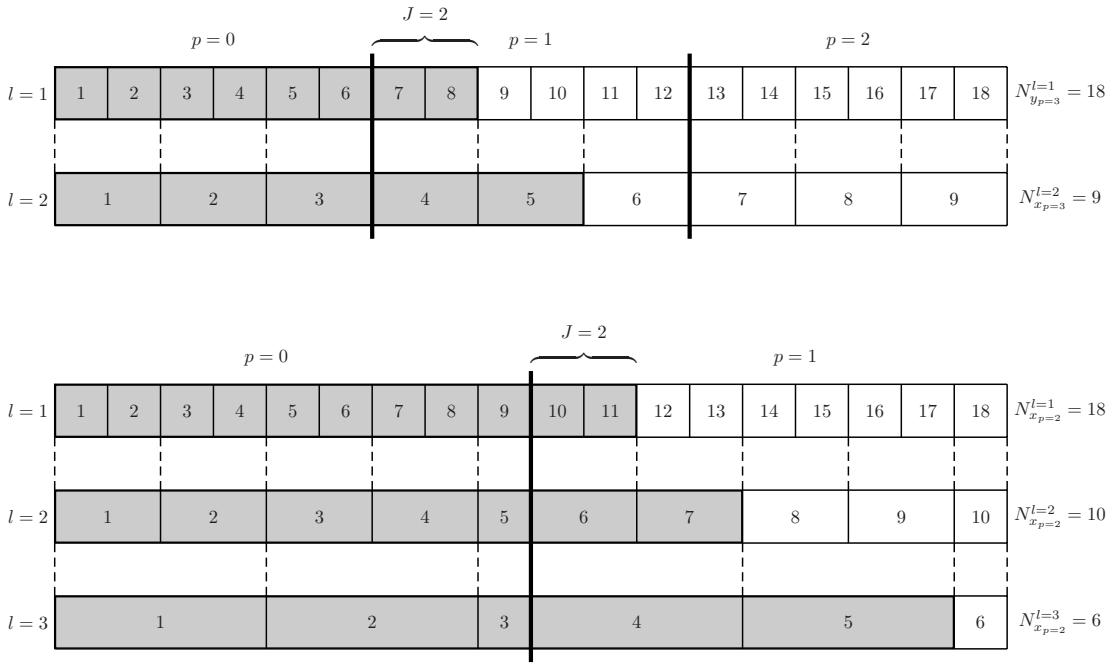

Figure 7.11: DDACM correction equations for different numbers of processors.

In the pre-processing stage, the matrices for the local problems of each level, from 1 to $M-1$, are evaluated and their incomplete LU decomposition is computed and stored. For the coarsest level M, Schur decomposition is evaluated and stored. The preprocessing algorithm is given in Alg.7.6:

DDACM Preprocessing (A, M) {
  l=1 $\rightarrow M-1$ {
    $A^{l+1} \leftarrow A^l$ with equation (3.33)
  }
  update halos of matrices A^1 to A^M
  l=1 $\leftarrow M-1$ {
    Form local matrix, $A^{l,p}_{loc} \leftarrow A^{l,p}$ (section 7.3)
    Evaluate ILU decomposition of $A^{l,p}_{loc}$ (section 2.7.3)
  }
  Evaluate Schur decomposition of L^M (section 6):
    $\tilde{L^M}^{-1}_{s,s} \leftarrow L^M$
}

Algorithm 7.6: DDACM algorithm. Preprocessing.

Here, $A^{l,p}_{loc}$ is the Block-Jacobi local matrix of processor p for level l. (A_{ii} in equation 4.13). In our implementation, the number of levels M is a parameter to be introduced externally to the algorithm.

7.5.2 Solution

After the preprocessing of matrix A, equations of the type

$$Ax = b \tag{7.4}$$

can be solved iteratively with the following DDACM algorithm:

```
Solve Ax = b with precision ε {
        update x^1, b^1 halos
        i=1 → n_i {
            first leg :
                l = 1 → M − 1 {
                    if l > 1
                        update b^l halo
                    do ν_1 local ILU iterations to improve x^l (Eq. 7.3)
                    at the inner nodes {
                        evaluate residual vector r^l
                        evaluate right hand-side of next level, b^{l+1} ← r^l
                    }
                    begin next level with 0 as initial guess, x^{l+1} ← 0
                }

            lower level :
                solve exactly A^M x^M = b^M using the available Schur decomposition
                update x^1, ··· x^M halos

            second leg :
                l = M − 1 → 1 {
                    correct x^l ← x^{l+1}
                    do ν_2 local ILU iterations to improve x^l (Eq. 7.3)
                }
                update x^1 halo

            residual control :
                if ||r^1|| < ε
                    break
        }
}
```

Algorithm 7.7: DDACM algorithm. Solution.

Remarks:

- First leg:
  - Halo update of x^l is not needed:
    - For $l = 1$, it is updated at the beginning of the algorithm and at the end of each iteration (after second leg).
    - For $l > 1$, update is unnecessary as the iterations begins with the initial guess $x^l = 0$ for the correction.
  - The ν_1 local iterations are done without halo updates. At the end, x^l depends on the number of processors. The same holds for r^l and b^{l+1}, evaluated from x^l. However, due to the use of overlapping regions that also relaxed, if the number of iterations is low, the differences are relatively small in the inner nodes.
  - Unlike in the correction levels, the vector $b^1 = b$ remains constant for each execution of the algorithm. Thus, halo update of b^1 is done just once at the beginning.
  - Halo update of b^l (for $l = 2 \cdots M - 1$) before the iterations is needed, otherwise the numerical efficiency is reduced and the total computing time increases.
  - The halo of b^M is not updated after the first leg, as all the operations needed to solve level M are done inside the Schur complement solver.

- Lower level:
  - During the first leg, the halos of x^l vectors have not been updated. Before starting the second leg, they must be updated as they contain the values obtained from the BILU relaxation. As in the DDV cycle (section 5), this is done with a single message after the solution of x^M.
  - If the Schur decomposition is not to be reused and/or the frequencies of the error are high (as in the case of transient convection-diffusion equations), it might be better just to do a few extra local ILU sweeps on M level equation. This aspect has not been considered in this work as our main target are pressure correction equations of incompressible flows.
- Second leg:
  - During the second leg, as in DDV algorithm, the values obtained after level l smoothing in a halo region of J nodes can be used to correct a halo region of $2J$ nodes in level $l-1$. A schema of this process can be seen in Fig. 7.12. As the overlapping area is J for all the levels, the J outer values of each level are not used for next level correction.
  - The halo of x^1 is updated at the end of the second leg because it is needed to evaluate $||r^1||$ and then to begin the local iterations in the next first leg.

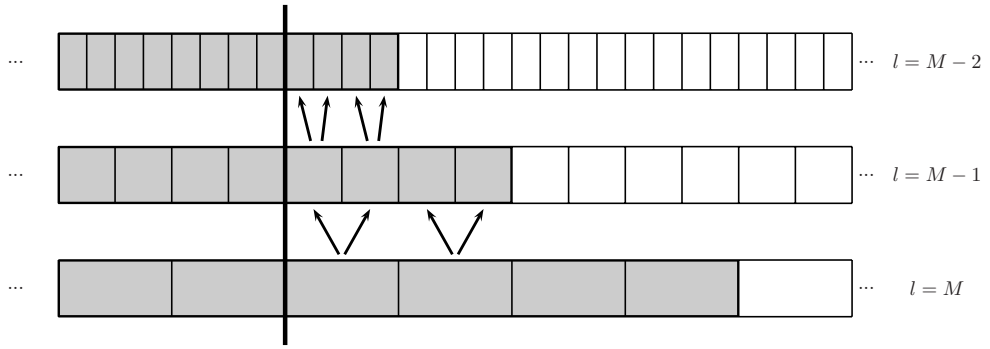

Figure 7.12: Second leg in DDACM algorithm.

Unlike in DDV, in DDACM the relaxation is dependent on the number of processors, but the two outer rings of nodes, where the difference is more important, are not used. Thus, communication in the second leg can also be suppressed, with very little effect on the convergence ratio.

The main guideline of DDV is to preserve the independence of the number of processors. This is not the case of DDACM, which depends on the number of processors. There are many different possibilities concerning where to update the halos of b and x vectors. In general terms, halo updates reduce the dependency on the number of processors but increase the cost per iteration. The algorithm presented needs $M+1$ each-to-neighbours communications per iteration, plus one global communication during the Schur solution. It was the best solution for the model problem (section 2.7.1) on the JFF cluster. DDACM can be easily tailored to other architectures.

7.6 Benchmarking DDACM

The problem model (section 2.7.1) has been solved with different number of processors and for different problem sizes. The main parameter of the DDACM solver is the number of levels, M. If M is reduced, the number of iterations decrease but the amount of RAM memory per node and the pre-processing time increase. In the limit, if $M = 1$, DDACM is a direct Schur complement algorithm.

To show its effect, all the possible M values have been used for each problem size. The lower limit of the possible M values is due to the RAM memory needed for the Schur decomposition, while the upper limit is due to the size of the smallest level, that has to be partitioned. In all the cases, the number of local iterations used was $\nu_1 = 1, \nu_2 = 1$, the MSIP coefficient $\alpha = 0.4$ (section 2.7.3), and as a criteria to stop the iterations, a reduction of the residual of the initial guess by a factor of 10^{-6}. The execution times on the JFF cluster are presented in Tables 7.1- 7.4. For each number of levels M, the computing time T in seconds and the number of iterations n are given. The minimum computing time is given at column Min.

	$M=2$		$M=3$		$M=4$		Min.
N	**t (s)**	n	**t (s)**	n	**t (s)**	n	**t (s)**
1296	2.57×10^{-2}	11					2.57×10^{-2}
5184	2.14×10^{-1}	14	2.67×10^{-1}	21			2.14×10^{-1}
20763	1.23×10^{0}	15	1.50×10^{0}	27	2.16×10^{0}	39	1.23×10^{0}
82944	8.41×10^{0}	16	9.05×10^{0}	31	1.32×10^{1}	50	8.41×10^{0}
331776			3.58×10^{1}	32	5.12×10^{1}	58	3.58×10^{1}
1327104					2.19×10^{2}	60	2.19×10^{2}

Table 7.1: Execution times and number of iterations of DDACM for different mesh sizes and number of levels, with one processor.

	$M=2$		$M=3$		$M=4$		Min.
N	**t (s)**	n	**t (s)**	n	**t (s)**	n	**t (s)**
1296	1.59×10^{-1}	13					1.59×10^{-1}
5184	2.46×10^{-1}	16	4.49×10^{-1}	26			2.46×10^{-1}
20763	5.72×10^{1}	16	9.83×10^{-1}	32	1.620×10^{0}	49	5.72×10^{-1}
82944	2.38×10^{0}	16	2.74×10^{0}	33	4.750×10^{0}	58	2.38×10^{0}
331776	1.40×10^{1}	16	1.40×10^{1}	33	1.974×10^{1}	62	1.39×10^{1}
1327104					6.450×10^{1}	62	6.44×10^{1}

Table 7.2: Execution times and number of iterations of DDACM for different mesh sizes and number of levels, with four processors.

	$M=2$		$M=3$		$M=4$		Min.
N	**t (s)**	n	**t (s)**	n	**t (s)**	n	**t (s)**
1296	2.63×10^{-1}	13					2.63×10^{-1}
5184	3.60×10^{-1}	15	7.36×10^{-1}	26			3.60×10^{-1}
20763	5.75×10^{-1}	17	1.19×10^{0}	32	2.11×10^{0}	51	5.75×10^{-1}
82944	1.27×10^{0}	16	2.25×10^{0}	33	4.12×10^{0}	60	1.27×10^{0}
331776	6.30×10^{0}	16	5.97×10^{0}	33	1.02×10^{1}	65	5.97×10^{0}
1327104	3.12×10^{1}	15	2.14×10^{1}		3.08×10^{1}	63	2.14×10^{1}

Table 7.3: Execution times and number of iterations of DDACM for different mesh sizes and number of levels, with nine processors.

In order to clarify the numerical efficiency of the algorithm, the number of iterations needed to achieve convergence has been represented against the number of processors in Figs. 7.13 and 7.14, for meshes with $N = 82944$ and $N = 1327104$, respectively. As it can be seen, it almost does not depend on the number of processors or on the problem size but only on the number of levels. If M is increased, the cost of the direct solution with the Schur algorithm decreases, but the number of iterations increases.

The speedup achieved for the three larger meshes considered has been represented in Fig. 7.15. It has been evaluated using the number of levels that yields the minimum time for each situation.

	$M=2$		$M=3$		$M=4$		Min.
N	**t (s)**	n	**t (s)**	n	**t (s)**	n	**t (s)**
1296	2.79×10^{-1}	12					2.79×10^{-1}
5184	4.07×10^{-1}	16					4.07×10^{-1}
20763	5.76×10^{-1}	17	1.20×10^{0}	33	2.12×10^{0}	51	5.76×10^{-1}
82944	1.06×10^{0}	17	1.92×10^{0}	35	3.88×10^{0}	63	1.06×10^{0}
331776	3.51×10^{1}	16	4.22×10^{0}	34	8.05×10^{0}	67	3.51×10^{0}
1327104	1.65×10^{1}	15	1.59×10^{1}	33	2.79×10^{1}	65	1.59×10^{1}

Table 7.4: Execution times of DDACM for different mesh sizes and number of levels, with sixteen processors.

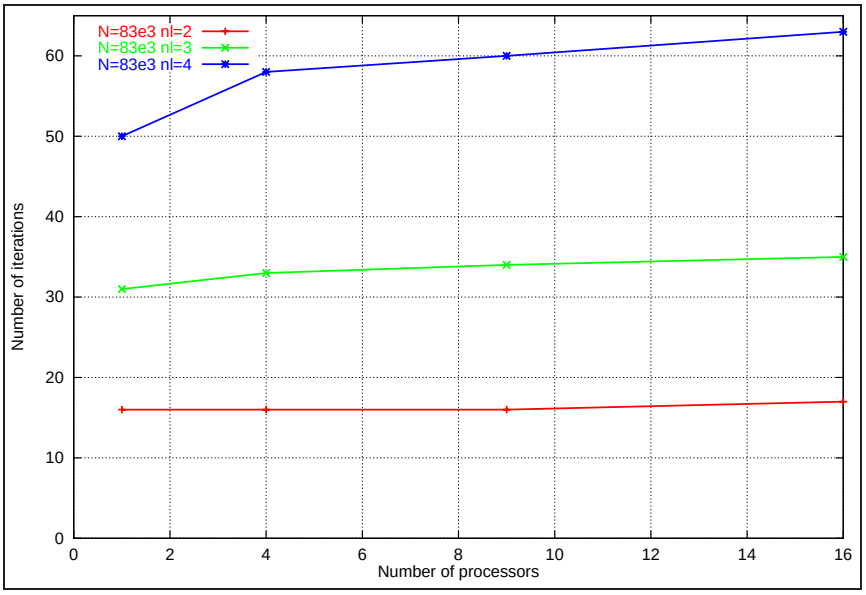

Figure 7.13: Number of iterations needed by DDACM to converge the problem model versus the number of processors, for a mesh with $N = 82944$.

An analysis of the cost of each operation has been carried out for the case of the 1152×1152 mesh, solved with different number of levels. A summary of the results can be seen in Fig. 7.16, where the fraction of the total computing time spent in the different parts of the algorithm is plotted versus the number of levels.

The cost of the Schur solver decreases with the number of levels and becomes almost neglectable for $M = 4, 5$. Except for the case with $M = 2$, the cost of the second leg is significantly lower than the cost of the first leg. Note also the relative importance of the residual control.

7.7 Final remarks

For the case of problems where a constant matrix has to be solved with different right hand sides, such as the pressure correction equation in the case of incompressible flows, DDACM algorithm appears to be a promising solver. As it uses an algebraic approach to construct the correction equations, it can be used as a black-box linear solver (like for instance Krylov subspace algorithms), allowing a clear separation of discretization and solution stages. It is not strictly independent of the number of processors but it has a high numerical efficiency. It is a combination of an iterative algorithm (ACM+BILU) and a direct algorithm (Schur complement) for the coarsest level, that uses techniques from the DDV multigrid cycle to reduce the number of communications. It allows to stop the iterations when the required level of precision has been reached. As shown in section 3.3.3,

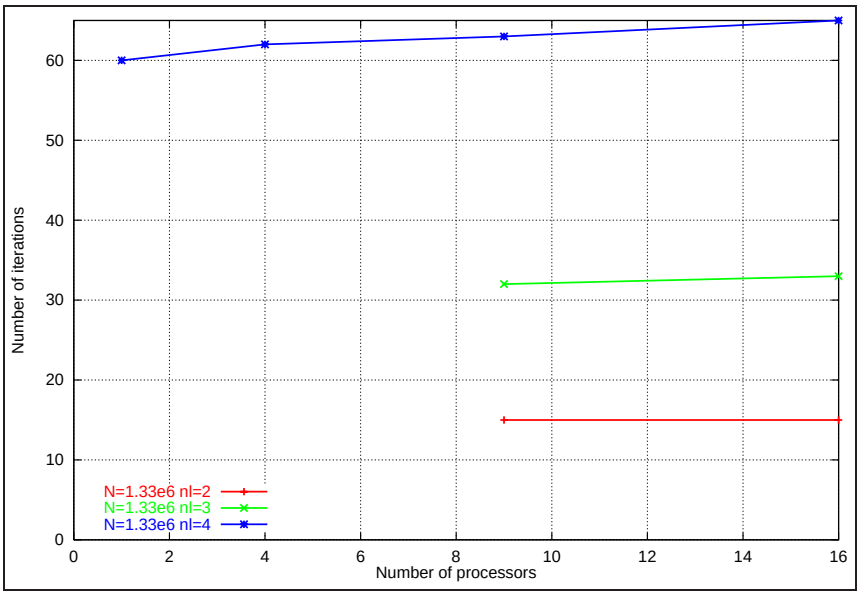

Figure 7.14: Number of iterations needed by DDACM to converge the problem model versus the number of processors, for a mesh with $N = 1327104$.

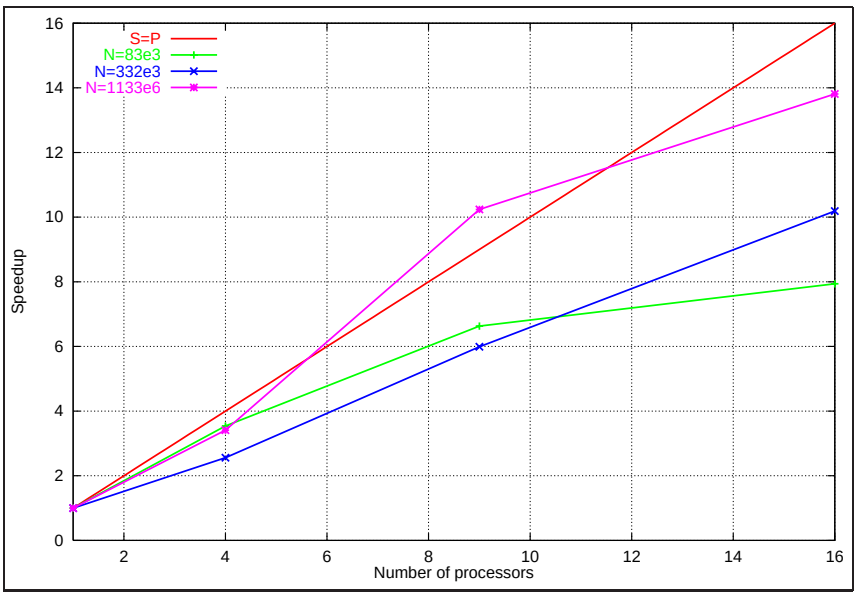

Figure 7.15: DDACM Speedup for different meshes.

this is an advantage for the case of pressure correction solvers.

As for any iterative solver, its efficiency depends not only on the matrix A but also on the properties of the right hand side b and the initial guess. Thus, in order to tune its parameters and provide realistic measures of its efficiency, it has to be implemented in a CFD code.

The extension of the DDACM algorithm, based on the Schur complement variant proposed here, to three-dimensional problems might require too much RAM memory. This means that a high number of levels would be required by DDACM, potentially leading again to network latency problems. A possible solution could be to use the current direct algorithm for the interface equation (based on the distributed evaluation and storage of the inverse of the interface matrix) but iterative solvers for the inner equations of each processor.

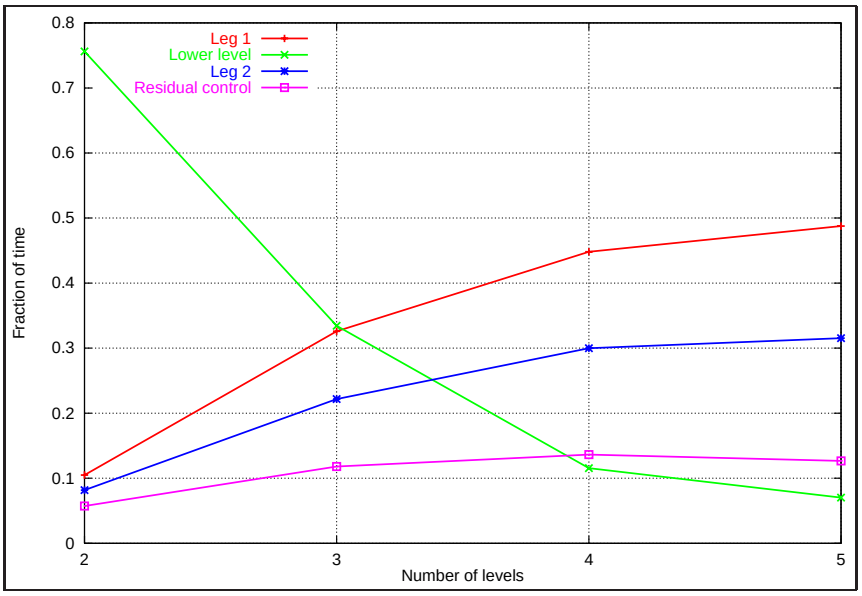

Figure 7.16: Breakout of computing time for different number of levels.

7.8 Nomenclature

A	matrix
b	right hand side
l	level
p, q	processors
P	number of processors
J	size of the halo region
L'_{loc}	lower triangular factor of matrix A_{loc}
M	number of levels
rb	receive buffer
sb	send buffer
U'_{loc}	upper triangular factor of matrix A_{loc}
px, py	position in the processors array
T	computing time
x	unknown vector

Greek symbols

ν	number of iterations

Subindices

loc	local
nb	neighbouring
x	horizontal axis
y	vertical axis
nb	neighbours

Superindices

l	level
p	processor