Chapter 5

Parallel geometric multigrid

5.1 Introduction

Chapters 5 and 7 are devoted to parallel multigrid algorithms. A quote from [185] seemed appropriated to begin this chapter, "It is quite clear that the (highly) parallel implementation, however efficient, of poor sequential algorithms cannot provide the breakthrough in performance required for the increasing demands on Computational Science. What is needed is a combination of both *parallel and numerical* efficiency. A natural candidate to deliver such combined efficiency is multigrid, known to be sequentially optimal for a wide class of applications ...".

In sections 1 and 2, the need of an efficient linear solver for incompressible flows has been exposed and it has been shown (section 3) that ACM is a feasible option. The main difficulties associated with parallel linear solvers for loosely coupled computers have been presented in section 4.

Our goal now is to develop a new ACM solver suitable for low cost, loosely coupled parallel computers. It should preserve the versatility and robustness of ACM (needed by DPC code), allowing an easy transition to parallel systems. The development of this algorithm will be done in three steps:

- This chapter, the first step, is devoted to Domain Decomposed V cycle (DDV), a parallel (geometric) multigrid algorithm, developed by Brandt and Diskin. In the original paper [20], DDV is described but not implemented on a parallel system. Here it will be implemented and benchmarked on a cluster of PCs, the platform where it should be more advantageous. It will be compared with conventional parallel MG algorithm. The different DDV features will be considered separately to elucidate their effect. Although DDV is a geometric MG algorithm, that is not of direct application for our context, the information provided by its analysis will be useful in our development.

- Chapter 6, the second step, presents a specific implementation of a direct parallel solver (Schur complement), that will be used as a DDACM component.

- Finally, in chapter 7, the third step, all the elements developed in the work are combined to produce DDACM.

5.1.1 General aspects of parallel MG algorithms and DDV

A first idea to parallelize MG could be to assign a level or a group of levels to each processor. This is not possible as only a few processors could be used, and it would cause load imbalance and massive data transference between processors. Additionally, there is a data dependence between the operations done in the different levels.

A domain decomposition approach is probably the only valid option. The finest level domain is decomposed into P regions assigned to the different processors. Each processor is in charge of

all the levels in its region of the domain. For each level, a *halo* region of nodes is defined with the purpose of the temporary storage of data from the regions assigned to other processors.

All the components of the MG algorithms are directly parallel, except the smoother. Stationary iterative solution algorithms are used for sequential MG, such as line-TDMA, incomplete LU [108] or SCGS [99] for coupled algorithms. In parallel MG, block-Jacobi relaxation schemes [104] are used. Each processor relaxes its subdomain using explicit values of the neighbours, updated after each iteration. A difficulty of this approach is that the number of smoothing iterations to reach a given norm of the residual increases with the number of domains, as shown in section 4.5.1.

However, the main difficulty of parallel MG algorithms is that they are extremely fine-grained algorithms. For the coarsest level, the total number of unknowns to be relaxed can be of the same order as the number of processors. Thus, the number of operations that each processor can perform between communication episodes is very low. The computational costs are dominated by network latency. Using conventional multigrid algorithms, tightly coupled systems are needed to achieve good speedups [186]. A possible solution [187], is to combine the pairs of neighbouring processes when proceeding to coarser levels; but this can not solve the latency problem.

The ideas used in the DDV multigrid cycle [20] to solve the difficulties of parallel MG are:

1. A direct solver is used for the coarsest level (as used by other parallel MG methods such as [188]).
2. Red-Black Gauss-Seidel [189] is used as smoother. The result of the smoothing operations is independent of the number of processors.
3. The domains assigned to the processors are augmented with overlapping areas that are also relaxed. If this is done, it is possible to perform a group of smoothing iterations without updating the halos, while keeping the smoother independent of the number of processors.
4. Pre-relaxation stage is suppressed. In this conditions, it is shown that only one or two each-to-neighbours messages per iteration are needed.

An additional technique to suppress the communications in the finest mesh has not been exploited in our implementation.

5.2 Notation

The aim of our implementation is to be exactly as described in [20]. In particular, the same terminology (except minor details) and mesh disposition are used here.

Consider equation (3.1), defined in Ω, a two-dimensional domain,

$$Lu = f \tag{5.1}$$

To solve it, a sequence of grids $\Omega^1, \Omega^2, .., \Omega^M$ will be used. To be consistent with the notation used in [20], here M is the **finest** mesh or *target level* and Ω^1 is the **coarsest** mesh contains only few points. Please note that this is exactly the opposite as in sections 3 and 7.

For our example, a regular mesh is used. The distance between two adjacent nodes in level k is h_k, with $h_k = h_{k-1}/2$. The position of the nodes is $(x_i, y_j) = (ih_k, jh_k)$, with i and j from 0 to N_i^k and N_j^k, respectively. The nodes of the different levels are aligned: the position of (i, j) in Ω^k is coincident with the position of $(2i, 2j)$ in Ω^{k+1}.

A geometric MG approach is used (section 3.1.3). On each grid Ω^k, a discretization is assumed to be available:

$$L^k u^k_{(i,j)} = f^k_{(i,j)} \qquad (i, j) \in \Omega^k \tag{5.2}$$

For each level k, the mesh Ω^k is decomposed into P rectangular subdomains, assigned to different processors. Unlike other formulations, the decomposed meshes Ω^k_p, with p from 0 to $P-1$ are not

disjoint: the boundaries are *shared* by the processors. So, for a two dimensional problem a node can either be owned just by one processor, shared by two neighbouring processors or shared by four neighbouring processors.

A *halo* of J^k neighbouring points in each direction is added to each subgrid Ω_p^k. Processor p can access to this area to read and write values owned by other processors. In the algorithms considered here, $J^k = J$ for all k.

The *genuine value* of a node is defined as the average of all the values of the processors that share the node.

If we consider processor p, the nodes in the domain can be classified into four groups according to the level of ownership of p:

1. Inner nodes, exclusively owned by p.

2. Inner nodes, exclusively owned by p but lying in the halos of one or more of p neighbours, so they have to be sent to them in halo update operations.

3. Boundary nodes that are shared by p and its neighbours.

4. Halo nodes that are not owned by p but are laying in its halos, so p can read or write them, but not contribute to their genuine value.

5. Nodes not owned by p or lying in its halos, so p can not access them.

5.2.1 Genuine values and halo updates

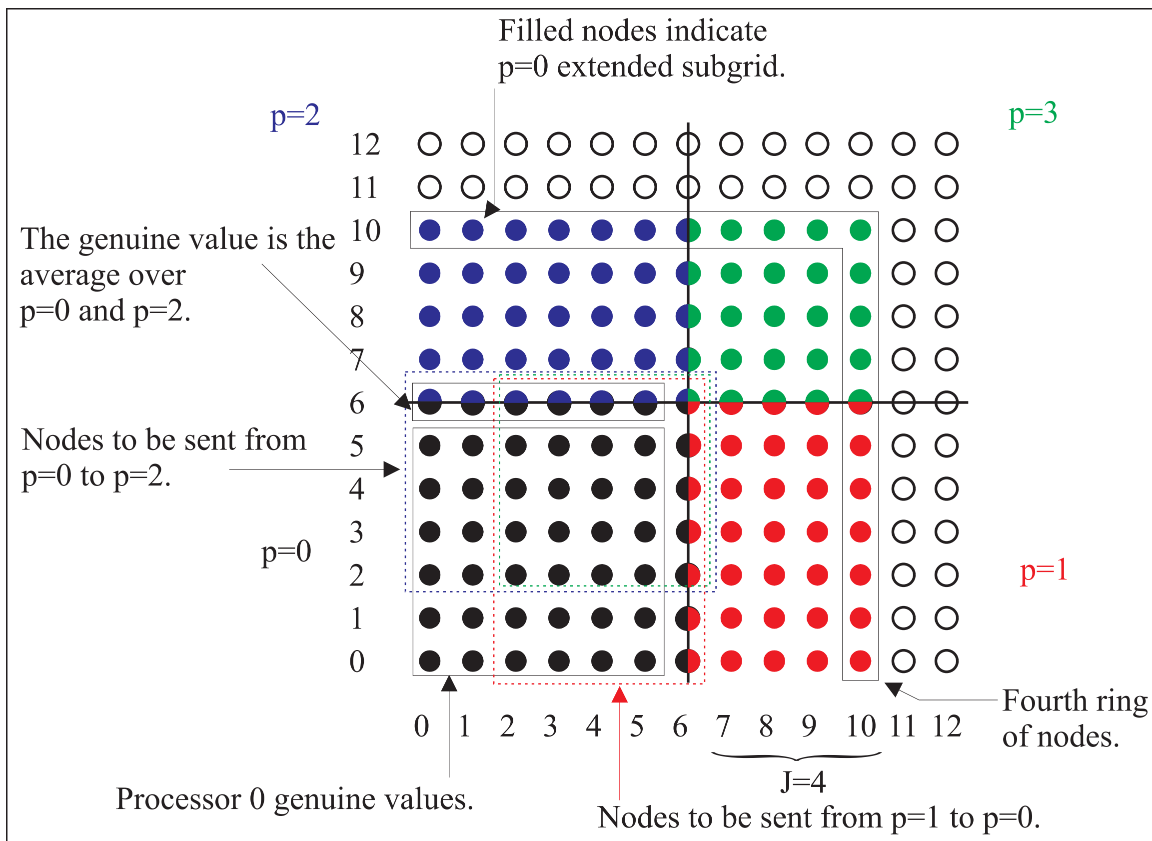

Figure 5.1: Domain decomposition used by the geometric multigrid algorithms.

We say that the halos are *updated* if they contain genuine values in all the points. To update the halos, an all-to-neighbour communication has to be done[1]. Then, the values of the boundary nodes are averaged. A schema of the algorithm used is presented in Alg. 5.1. Note that the node in the intersection of two boundaries is divided by four.

```
Update u halos:
        For all (i,j) in my halo (but not in my boundaries)
                u_(i,j) = 0
        For each of my neighbours nb {
                Pack the data from my domain that nb needs
                        to update his halo and boundary
        }
        Exchange data with neighbours
        For each of my neighbours nb {
                Add the data from nb to my own data
        }
        For all (i,j) in my horizontal boundary
                u_(i,j) <- u_(i,j)/2
        For all (i,j) in my vertical boundary
                u_(i,j) <- u_(i,j)/2
```

Algorithm 5.1: Parallel geometric multigrid. Halo update.

To clarify the mesh disposition and decomposition, consider Fig. 5.1. A situation with $N_i = N_j = 12$, $J = 4$ and $P = 4$ has been considered. In the figure, the point of view of processor $p = 0$ is adopted: all the nodes accessible to it are filled. Some of them are owned exclusively by one processor: black nodes are owned only by $p = 0$, red nodes by $p = 1$, blue nodes by $p = 2$ and green nodes by $p = 3$. Boundary nodes, owned by two or four processors, are represented accordingly. The boundaries have been represented using solid lines. The nodes that are to be transferred form $p = 0$ to its neighbours in halo update operation are indicated in the figure. Note that, as the genuine value at the boundaries is defined as the average of all the processors that share the node, not only inner nodes but also boundary nodes are to be communicated.

In our implementation of the algorithm (see section 4.6), processors use a global numbering (i.e., from $(0, 0)$ to (N_i, N_j) to access the nodes. For instance, the node $(6, 6)$ is at the intersection of both boundaries for all the processors. A mapping technique is used so that each processor uses the global numbering but only stores its part of the domain (including the halos).

To simplify the description of parallel algorithms, the halos are considered as formed by *rings*. Ring 0 is the boundary. Ring 1 is formed by the nodes next to the boundary, within the halo. In general, ring k is formed by the nodes next to the ring $k-1$. Rings with negative indices are formed by inner nodes.

For one-dimensional domain decompositions, the rings are rows or columns at both sides of each subdomain, while for two-dimensional decompositions, they are concentric frames surrounding the subdomains. As an example, the fourth ring of processor $p = 0$ is indicated in Fig. 5.1.

[1]All the parallel algorithms presented in this work assume that a message-passing programming model (section 4.2.2) is used, i.e., explicit messages are needed to transfer data between the processors.

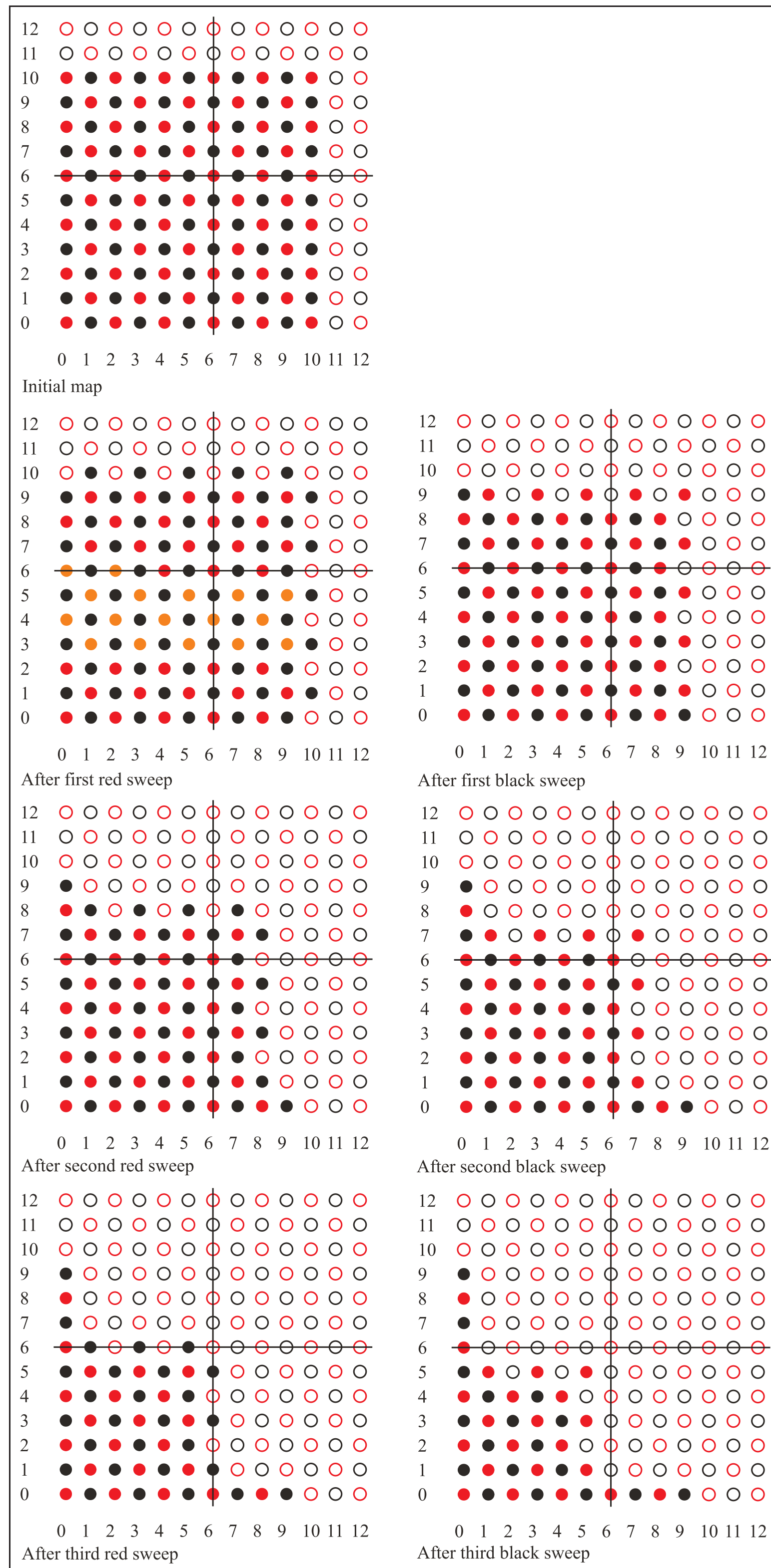

Figure 5.2: Propagation of erroneous values after each Red-Black Gauss-Seidel iteration.

5.3 Parallel smoothing

In our implementation (as in [20]), Red-Black Gauss-Seidel relaxation ($RBGS$) is used. In the first half of each sweep, the "red" equations (with even $i+j$) are relaxed, and in the second half the rest of them, the "black" ones. More information about parallel Red-Black Gauss-Seidel can be found for instance in [189].

Assume now that we want our relaxation process to be independent of the number of processors P (i.e., $E^{num} = 1$). We should determinate the number of sweeps that we can do for a given halo size J.

First we do ν iterations with a single processor. Then, starting with updated halos, we do ν parallel iterations with P processors. In the parallel iterations each processor can only relax the nodes accesible to it: inner nodes, boundaries and halos (except the outer ring where non-available neighbouring nodes would be needed).

We compare both results and mark as *erroneous*[2] the nodes where the value is different (this is, it depends on the number of processors); the others are referred as *correct*. If erroneous nodes are only in the *halo* regions, we can update them again and go on with the iterations, keeping the algorithm independent of the number of processors.

The evolution of the correct areas during $RGBS$ iterations, with $N_i = N_j = 12$, $J = 4$ and $P = 4$ has been represented in Fig. 5.2, which reminds of a cellular automata. Correct nodes (in the area accessible to $p = 0$) are filled and erroneous nodes are void. Initially, all the nodes are correct (top left). The nodes in the outer ring of the halo (i.e., $i = 10$ or $j = 10$), can not be relaxed by $p = 0$ as nodes lying outside the area accessible by $p = 0$ would be needed. The nodes at the column $i = 0$ and at the row $j = 0$ remain correct as they contain the boundary condition. As it can be seen, if two iterations are done, erroneous values do not reach the area owned by $p = 0$. So, using $J = 4$, we can do two consecutive iterations without updating the halos. In general, the number of *local*[3] iterations that can be done, keeping the algorithm independent of the number of processors is:

$$\nu_{loc} = \frac{J}{2} \tag{5.3}$$

To simplify the description of parallel computations like the previous, γ will be used to designate the number of halo rings where *all* the nodes are correct for *all* the processors. For a given J, $\gamma \leq J$. If $\gamma = 0$, only the boundary is correct. If $\gamma = J$ the halo is totally correct. If $\gamma < 0$, there are erroneous nodes in the inner part of the subdomain. As an example, in Fig. 5.2, after the second red sweep, $\gamma = 1$. After ν iterations

$$\gamma = 4 - 2\nu \tag{5.4}$$

If the number of inner nodes to be relaxed by each processor is large enough, the overcost of relaxating the halo regions is neglectable and by increasing J we can reduce the number of messages, but not the total information transfered, saving the time due to network latency. Alg. 5.2 is a parallel relaxation algorithm to reduce the norm of the residual up to ϵ, allowing a maximum number of $\nu \cdot \nu_{loc}$ iterations.

A global communication is needed to evaluate $||r||$. Depending on the problem size and on the parallel computer, the cost of this operation can be significant compared with the rest of the operations. The same holds for any dynamic criteria to stop the iterations. It is convenient to simplify the code, doing a fixed number of iterations, as shown in Alg. 5.3.

As an example, the time needed to complete a total number of $\nu = 20$ iterations on a tightly coupled system, the Cray T3E, with $\nu_{loc} = 2$ and $J = 4$, has been represented against the problem size in Fig. 5.3. As expected, it grows almost linearly. The differences for small problems are due to

[2] Meaning that their value is different of the value obtained with a single processor. An equivalent name could be "Nodes where the value depends on the number of processors".

[3] I.e., without updating the halos.

```
Parallel RBGS smoother:
        Update halos
        i1=1 → ν {
                Do ν_loc RBGS iterations
                Update halos
                Evaluate r = f − Lu
                if ||r|| < ε break
        }
```

Algorithm 5.2: Red-Black Gauss-Seidel smoother.

```
Parallel RBGS smoother (without residual evaluation):
        Update halos
        i1=1 → ν {
                Do ν_loc RBGS iterations
                Update halos
        }
```

Algorithm 5.3: Red-Black Gauss-Seidel smoother for a parallel geometric multigrid.

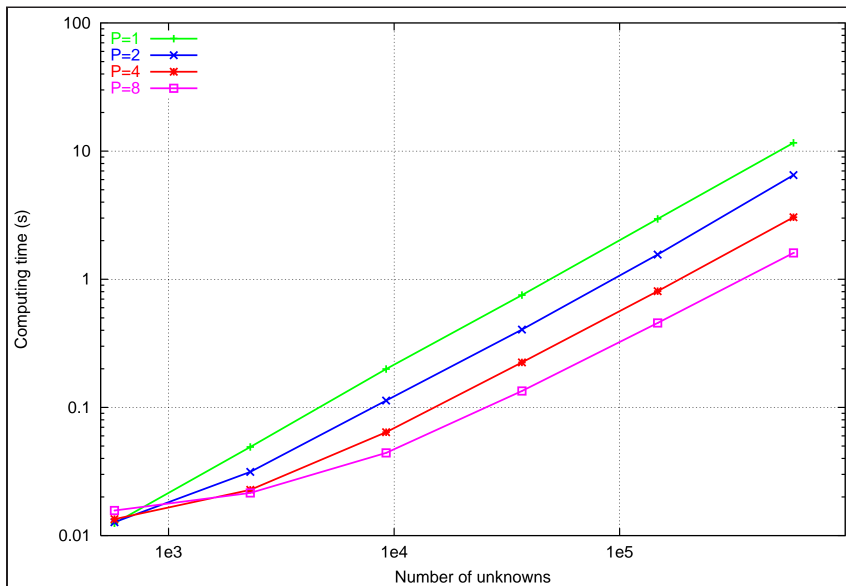

Figure 5.3: Time to do a fixed number of Red-Black Gauss-Seidel iterations on the Cray T3E.

the extra work needed to relax the halo regions. Fig. 5.4, representing the parallel efficiency (E^{par}) is more eloquent. For small domains, the efficiency is very poor. As expected, it is higher for a lower number of processors (except a decrease in efficiency for the largest mesh in the case with two processors that is not specially relevant). For the case of the smaller meshes, it can even be *speed down*, this is, $t_8 > t_1$. As the size of the halo regions and the communication overheads becomes neglectable, the efficiency grows. In the limit, we should expect $E^{par} = 1$.

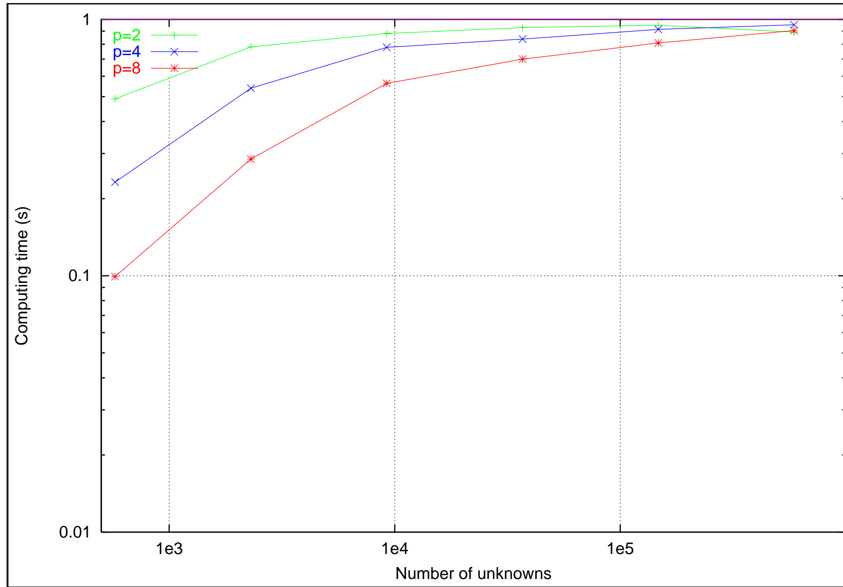

Figure 5.4: Parallel efficiency of Red-Black Gauss-Seidel iterations on the Cray T3E.

5.4 Parallel V-Cycle. Standard approach

5.4.1 Algorithm overview. FAS equations

A parallel FAS V cycle has been implemented using the concepts exposed in previous paragraphs. Consider an equation of the form (5.2). An initial approximate solution u^k is given. The FAS cycle $FAS_k(\nu_1, \nu_2; \kappa)$ for improving this approximation is recursively defined as:

- **Pre-relaxation**

  Improve u^k by means of ν_1 *RBGS* iterations (here ν_1 is the total number of iterations, done in packets of ν_{loc}). As we know from section 5.2.1, it is possible to do so while keeping the algorithm independent of the number of processors.

- **Formation of level $k-1$ FAS correction equation**

$$L^{k-1}u^{k-1} = f^{k-1} \tag{5.5}$$

  where the right-hand side of level $k-1$ is:

$$f^{k-1} = L^{k-1}\left(\hat{I}_k^{k-1}u^k\right) + I_k^{k-1}r^k \tag{5.6}$$

  with the standard residual vector,

$$r^k = f^k - L^k u^k \tag{5.7}$$

  (The inter-grid transfer operators $\hat{I}_k^{k-1}$ and I_k^{k-1} are defined in section 5.4.2)

  Note that equations (5.5) and (5.6) are a transcription of conventional FAS equations (3.22) and (3.23), using the notation from [20].

- **Solve correction equation (5.5)**

  If $k-1=1$, as Ω^1 is small enough, use a direct algorithm. Otherwise, evaluate the following initial guess:

$$u^{k-1} = \hat{I}_k^{k-1}u^k \tag{5.8}$$

and invoke κ times the algorithm $FAS_{k-1}(\nu_1, \nu_2; \kappa)$ to solve it. The parameter κ is the cycle index. Only V-Cycles, with $\kappa = 1$, have been considered in this section.

- **Coarse grid correction**

  Correct u^k using u^{k-1},

$$u^k \leftarrow u^k + I_{k-1}^k \left(u^{k-1} - \hat{I}_k^{k-1} u^k \right) \tag{5.9}$$

- **Post-relaxation**

  Improve u^k by means of ν_2 *RBGS* iterations.

5.4.2 Inter-grid transfer operators

The inter-grid transfer operators used are:

- $\hat{I}_k^{k-1}$: Solution restriction operator (from fine to coarse mesh). *Injection* is used:

$$u_{(i,j)}^{k-1} = \left(\hat{I}_k^{k-1} u^k \right)_{(i,j)} = u_{(2i,2j)}^k \tag{5.10}$$

- I_k^{k-1}: Residual restriction operator (from fine to coarse mesh). *Full weighting* is used:

$$\begin{aligned} r_{(i,j)}^{k-1} = \left(I_k^{k-1} r^k \right)_{(i,j)} = \frac{1}{4} r_{(2i,2j)}^k &+ \\ \frac{1}{8} \left(r_{(2i+1,2j)}^k + r_{(2i-1,2j)}^k + r_{(2i,2j+1)}^k + r_{(2i,2j-1)}^k \right) &+ \\ \frac{1}{16} \left(r_{(2i+1,2j+1)}^k + r_{(2i+1,2j-1)}^k + r_{(2i-1,2j+1)}^k + r_{(2i-1,2j-1)}^k \right) & \end{aligned} \tag{5.11}$$

- I_{k-1}^k: Correction interpolation operator (from coarse to fine mesh). Linear interpolation is used.

5.4.3 Non-recursive formulation of the algorithm

If no special attention is paid to efficiency, parallel multigrid algorithms can be formulated using standard recursive formulation (as in previous paragraphs or in section 3.2). However, if a detailed control of the halo update operations is to be done, it is convenient to rewrite the previous steps without using recursion, as in Alg. 5.4. The convenience of this change will be clear in next paragraphs. Its parameters are: J is the size of the halo regions; ν_1 and ν_2 are the number of pre-relaxation and post-relaxation iterations; the parameter Δ controls the treatment given to level 1: if $\Delta = 1$, a direct solver is used while if $\Delta = 0$ it is treated as the other levels.

Using this notation, we can appreciate better the three stages of the algorithm:

1. First leg. Descends from the target level k to the coarsest level $1 + \Delta$.

2. Direct solver. Level 1 is solved using a direct algorithm. Solution values u^1 are propagated to all the processors.

3. Second leg. Goes back from the coarsest level 2 up to the target level k.

As if $\nu_{loc} = J/2$, the algorithm is independent of the number of processors.

```
V − Cycle (J, ν1, ν2, Δ)
    do {
        l=M → 1 + Δ {
            Update f^l and u^l halos
            i = 1 → ν1/νloc {
                Do νloc = J/2 RBGS iterations in level l
                Update u^l halos
            }
            Evaluate level l − 1 right hand side f^(l−1), Eq. (5.6)
            Evaluate level l − 1 initial guess u^(l−1), Eq. (5.8)
        }
        if Δ = 1
            Solve level 1 equation using a direct linear solver
        l=2 → M {
            Correct u^k using u^(k−1). Eq. (5.9)
            i = 1 → ν2/νloc {
                Do νloc = J/2 RBGS iterations in level l
                Update u^l halos
            }
        }
        Evaluate ||r^M||
    } while ||r^M|| < ϵ
```

Algorithm 5.4: Parallel V cycle.

5.4.4 Direct solver for level 1

In the previous algorithm, the typical values for ν_1 are 1,2,4. Unless the size of level 1 is very small, more *RBGS* iterations are needed to solve it with acceptable precision and not deteriorate the convergence rate of the cycle. This is not an inconvenient for sequential computers. There are two possible solutions:

- Do more iterations for level one.
- Reduce the size of level one until ν_1 suffice.

However, these solutions are not acceptable for a loosely coupled parallel computer:

- It is inneficient to do more iterations for level 1 as their cost is dominated by latency for the smaller levels. I.e., their cost is not at all neglectable.
- The smallest level can not be arbitrary small, as it is restricted due to the halo sizes

Respect to the second problem:

Halo update operation involves only the three adjacent processors (for one-dimensional domain decomposition) or eight (for two-dimensional domain decompositions).

Thus, the halo regions of each processor must lie in the inner regions of its immediate neighbouring processors. This restricts the minimum size of level 1 to be:

$$\frac{N_x^1}{P} \geq J \tag{5.12}$$

for one-dimensional domain decompositions and

$$\frac{N_x^1}{\sqrt{P}} \geq J \tag{5.13}$$

for two-dimensional domain decompositions.

If these condition are not satisfied, each processor would need to exchange data with an exceedingly large number of neighbours. Of course, such a halo update operation is possible to implement, but would be inneficient.

For the case of the V cycle, it could be possible to implement a special function for level 1, but not for the DDV cycle as it updates the halos of all the levels in a single message.

The only valid option is to use a direct solver for level 1. In this paper, we assume that $N_1 << N_M$, so the cost of solving it is neglectable compared with the rest of the levels, and we can use a *sequential* direct algorithm. In our context this assumption is acceptable.

There are different options to implement the direct sequential solver:

- One processor solves $L^1u^1 = f^1$:
  1. Each processor p transfers its part of f^1 to $p = 0$ (all-to-one communication).
  2. $p = 0$ solves $L^1u^1 = f^1$, while the others wait for the answer.
  3. The global solution u^1 is sent back to all the processors (one-to-all communication)[4].
- All the processors solve the same problem $L^1u^1 = f^1$:
  1. Each processor p transfers its part of f^1 to all the processors (all-to-all communication).
  2. All them solve their copy of the problem and go on with the rest of the algorithm without more data transference.

Our implementation uses the second option. If we assume that the complete L^1 operator is available to all the processors, no other data transference is needed. In particular, as we use a direct method, we do not need an initial u^1 map. The algorithm used for the direct solution is band LU decomposition (section 2.7.2).

5.5 Parallel V-Cycle: DDV approach

The DDV cycle is a simplification of the conventional V cycle that allows a reduction in the number of halo update operations. To do so, the pre-smoothing is suppressed ($\nu_1 = 0$) and the number of post-smoothing iterations is reduced. All the halo information of the different levels is grouped in one (or two) packets and transfered together before the first leg. Data transference is totally suppressed during the second leg.

DDV cycle is equivalent to a conventional V cycle without pre-smoothing. However, in order to be able to supress the level-by-level halo update operations, the DDV implementation has to be intricate, specially for two-dimensional domain decompositions. The goal of this section is to show how can the DDV cycle be implemented and why are two communications needed for two-dimensional domain decompositions.

5.5.1 Data dependencies

The discrete operators used in the previous MG algorithm can be grouped into four functions:

- *RBGS* $\left(u^k, f^k\right) \rightarrow u^k$
- *down* $\left(u^k, f^k\right) \rightarrow f^{k-1}$; $f^{k-1} = L^{k-1}\left(\hat{I}_k^{k-1}u^k\right) + I_k^{k-1}r^k$
- *guess* $\left(u^k\right) \rightarrow u^{k-1}$; $u^{k-1} = \hat{I}_k^{k-1}u^k$.

[4] A variant of this approach would be to sent only the part of the solution that each processor needs, but as N_1 is small, this is more expensive as involves P consecutive communications.

- *up* $(u^k, u^{k-1}) \to u^k;\ u^k = u^k + I_{k-1}^k \left(u^{k-1} - \hat{I}_k^{k-1} u^k\right)$.

Assume that we want our parallel algorithm to be independent of the number of processors. If we want to produce a given set of correct[5] output nodes, we need to determinate the set of correct input nodes that we need.

For *RBGS* function, this question has already been considered in section 5.2.1. Function *guess*, involving only $\hat{I}_k^{k-1} u^k$, is a part of function *down*. So we only have to elucidate the data dependencies of operators *down* and *up*. To do so, we proceed like in Fig. 5.2 but in the reverse order, *starting from the output* data to see what is the input data that we need.

The process needed to evaluate *down* has been represented in Fig. 5.5. Output nodes (level $k-1$) are black and input nodes (level k) are red. We assume that we want to produce a set of 3×3 nodes with correct values of f^{k-1}, filled at the top of the figure. We want to determinate what input nodes are needed to produce them.

The last operation to be done to evaluate $I_k^{k-1} r^k + L^{k-1}\left(\hat{I}_k^{k-1} u^k\right)$ is the addition, so the correct nodes of both terms to be added have to be available at the 9 filled nodes. Next, we consider for instance the left branch of the figure, where we look for the nodes involved in the evaluation of $I_k^{k-1} r^k$. To evaluate $\left(I_k^{k-1} r^k\right)_{(i,j)}$ using equation (5.11), the red node $(2i, 2j)$ in level k and its 8 neighbours are needed.

Thus the correct region of r^k that we need, represented using filled red dots, grows one point in all directions. This process goes on until the primitive values of u^k and f^k are obtained. The correct area of u^k in the central branch is bigger than the correct area in the right-hand side branch, so their values prevail.

As a **short summary**: to proceed to a coarser level, we need an area of input nodes larger that the area of output nodes that we want to obtain.

The same process has been done for *up* operator in Fig. 5.6. There is an important difference between them: *up* can produce an area of u^k as large as the input. This is, as the space between the nodes is doubled in each level, if we have an area of γ correct nodes at level $k-1$, we can apply *up* operator to generate 2γ correct nodes at level k.

The information about the data dependencies of the MG operators used in our algorithm will be used now to reduce the number of halo update operations (which are expensive in our computing context) as much as possible. To do so, it is convenient to reverse the order of the algorithm and begin with the second leg.

5.5.2 Avoiding data transfer in the second leg

Consider the instant in the second leg of the Alg. (5.4), at level l, when we are going to use *up* operator to correct u^l map with u^{l-1} values, using equation (5.9). As discussed in section 5.5.1, if u^{l-1} is correct up to $\gamma = 2$, and u^l is correct up to $\gamma = 4$, we can apply *up* operator to generate a map of u^l correct up to $\gamma = 4$.

A schema of this process can be seen in Fig. 5.7. After the correction, if one *RBGS* iteration is done in level l, u^l will still be correct up to $\gamma = 2$. This is, we can call again *up* operator and generate u^{l+1} correct up to $\gamma = 4$. This process can proceed indefinitely, without any communication, no matter how many levels are to be relaxed, leaving an area of $\gamma = 2$ correct nodes at the finest level, that will be very important for the next leg one.

With $J = 4$, it is not possible to do more than one iteration without communications. To see why, consider the instant after the sequential solution of level 1. Then, u^1 is correct up to $\gamma = 4$, so we can generate an initial u^2 map correct up to $\gamma = 4$. If next we do two *RBGS* iterations at level 2, our boundaries would still be correct (as it can be seen in Fig. 5.6), but not the halos (this is, $\gamma = 0$) so we would not be able to generate correct u^3 values for all the inner nodes. Thus, unless the number of levels is one or two, our algorithm would not be independent of the number of

[5] In the sense of section 5.2.1.

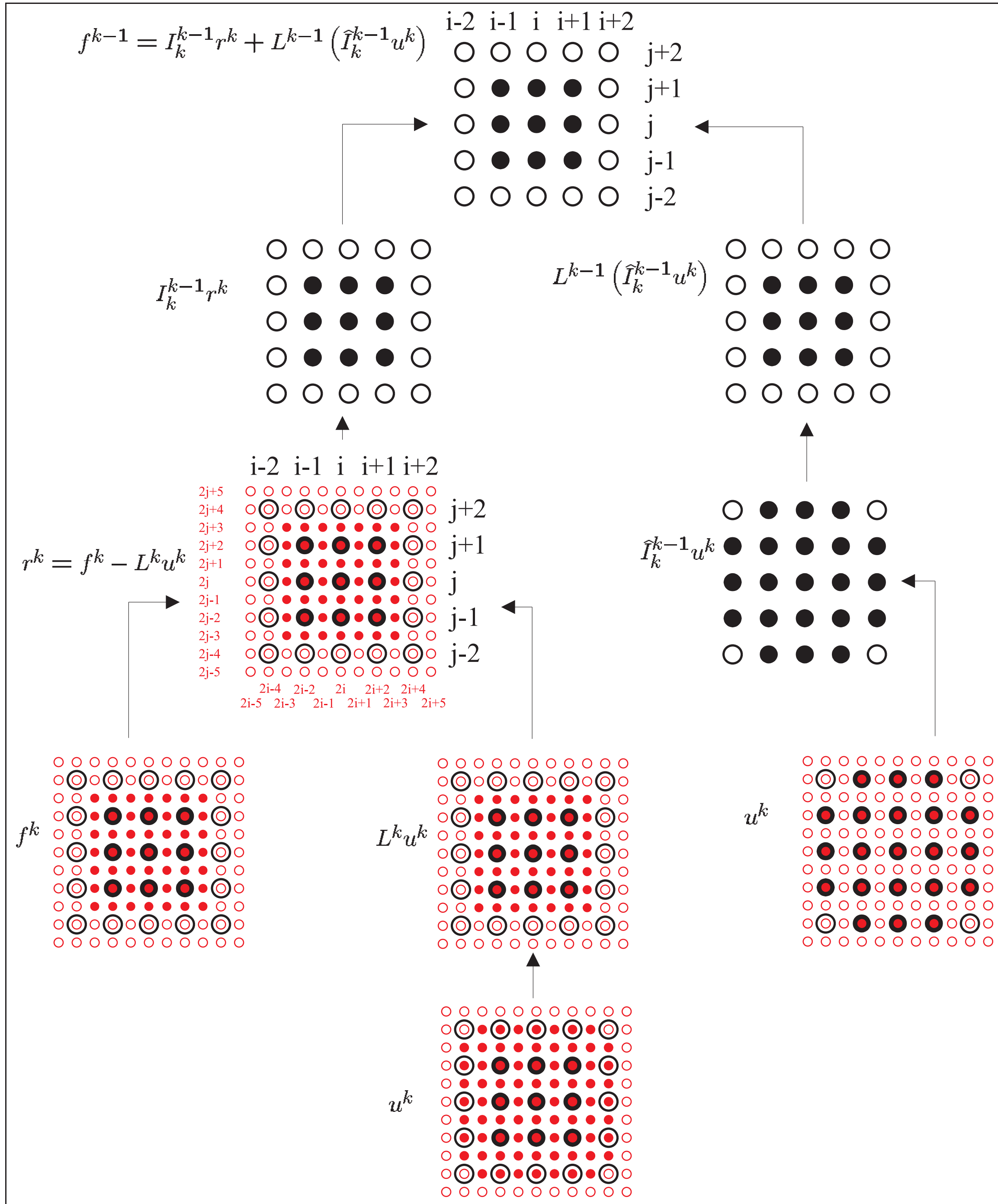

Figure 5.5: Data dependencies of operator $down\left(u^k, f^k\right) \rightarrow f^{k-1}$.

processors.

5.5.3 Avoiding data transfer in the first leg

As we have seen, after the second leg if we still assume that $J = 4$ and $\nu_2 = 1$, we have u^l maps updated up to $\gamma = 2$, for $l = 2 \cdots k$. Assume now that we decide to suppress pre-smoothing iterations ($\nu_1 = 0$). We will determinate which of the data needed to do another second leg (u and f values

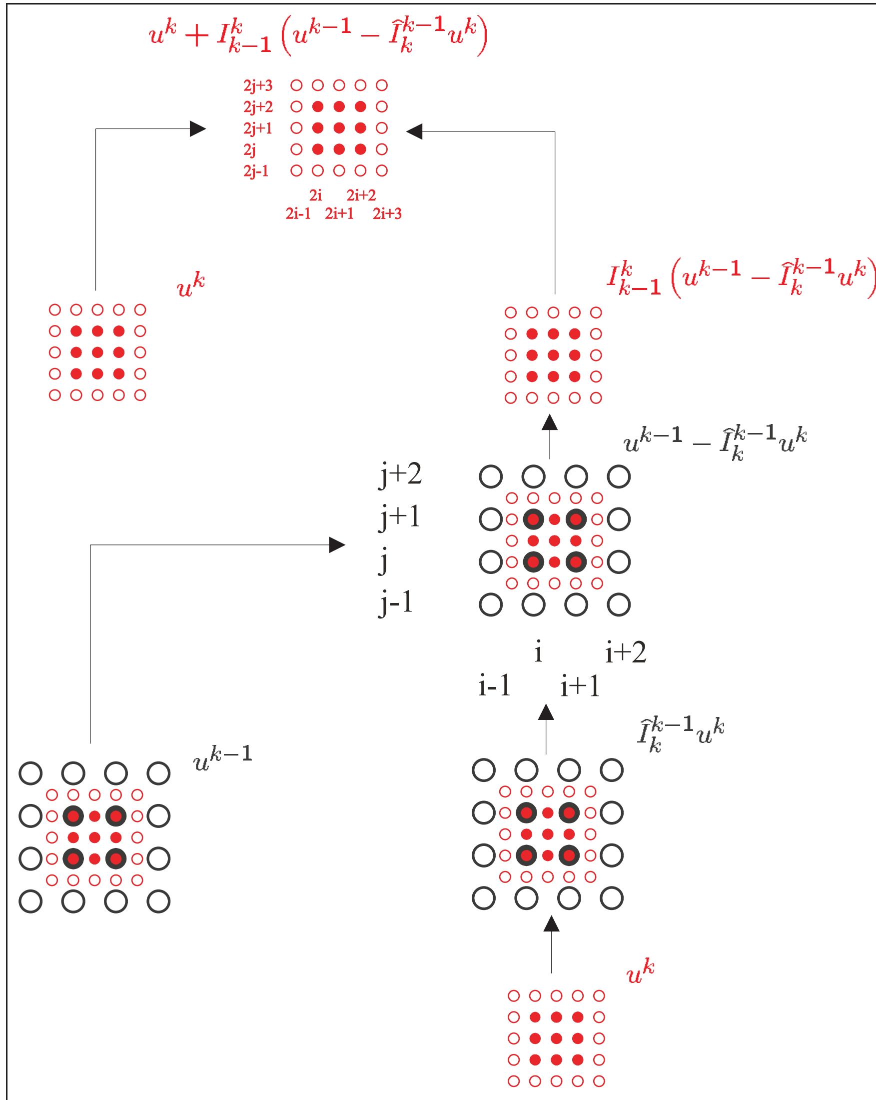

Figure 5.6: Data dependencies of operator $up\left(u^k, u^{k-1}\right) \to u^k$.

for the different nodes) can be generated without communications:

- Inner nodes. From the analysis of Fig. 5.5, we can see that if u^M and f^M are updated up to $\gamma = 2$, we can evaluate the inner nodes for f^{M-1}. In this way, each processor can proceed to the other levels, generating f^l for $l = M-1 \cdots 1$, and also the initial u^l maps at the inner nodes, without any communication.

- Boundary. We begin from the finest level $l = M$. In order to generate f^{M-1} at the boundary, we need u^M and f^M up to $\gamma = 1$. This is not a problem, as f^M is available (it is the right hand side of the original problem, that we updated at the beginning of the iterations). The problem arises at the next level. To generate f^{M-2} at the boundary, u^{M-2} correct up to $\gamma = 2$ and f^{k-1} correct up to $\gamma = 1$ would be needed. The first is available but not the second, as in the previous level it was generated only in the inner nodes.

This is, f can be evaluated **only** in the inner nodes (but not in the boundaries or in the halos). Then, **using a single message**, each processor can transfer to its neighbours the f and u values of its inner nodes, **for all the levels using a single message**.

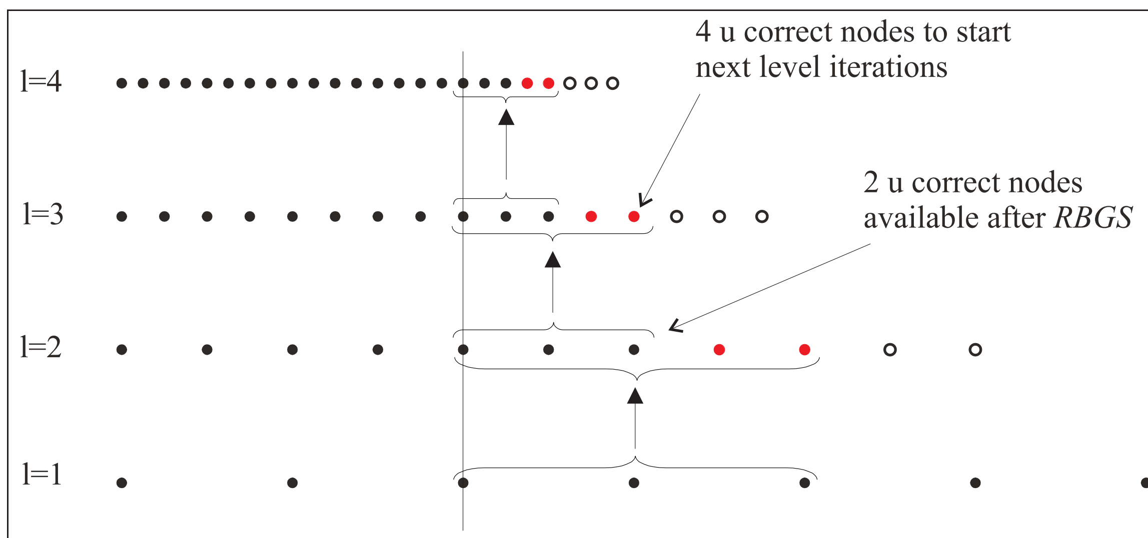

Figure 5.7: Evolution of the correct areas during the second DDV leg.

After this (for one-dimensional domain decompositions) each processor can evaluate f and u at its boundaries for all the levels. This two-stages construction of the correction equations is one of the key points of DDV.

One-dimensional and two-dimensional domain decompositions

Unfortunately, it is not possible to use the previous scheme using a single message for two-dimensional decompositions.

Consider a one-dimensional domain decomposition. The process of construction of the correction equations is as follows:

- First computation. Inner nodes data is evaluated.
- First communication. Halo data, evaluated by the neighbouring processors, is received.
- Second computation. Boundary nodes data is evaluated.

This process is represented in Fig. 5.8, at the left hand side, for an intermediate level. The nodes are filled in different colors, depending on their correctness before or after each communication or computation.

Consider now the case of a two-dimensional domain decomposition (which is in general a more efficient way to do the decomposition). It is presented at the right-hand-side of Fig. 5.8. The problem arises in the evaluation of f fields, so only them will be considered:

- First computation. Inner nodes data is evaluated, as in one-dimensional domain decompositions.
- First comunication. Nodes laying in the inner regions of other processors are received. But here, unlike one-dimensional domain decompositions, the halo regions are not only formed by inner nodes of neighbouring processors.

  As there are nodes that are simultaneously in the overlapping areas and in the boundaries, the neighbouring processors have not been able to evaluate them during the first computation.
- Second computation. Boundary nodes data is evaluated.

- Second comunication. Now, the data of the nodes in the intersection of the halos and borderlines can be received from neighbouring processors.
- Third computation. Only the intersection between both boundaries remains to be evaluated.

Thus, for two-dimensional domain decompositions, an additional communication and computation are needed to dash all the domain. The computation is not a problem, but the communication has a significant cost. Unfortunately, there seems to be no easy way to avoid this second communication episode.

The second communication is needed *because the halos and the boundaries have a non void intersection*. Such intersection areas do not exist in the one-dimensional decomposition (the only situation explicitely considered in [20]), so the problem does not arise in that case.

In summary, using the DDV strategy proposed by [20], that avoids pre-smoothing and does one post-smoothing iteration, ($\nu_1 = 0$, $\nu_2 = 1$), the each-to-neighbours communications can be reduced to one for the case of one-dimensional domain decompositions. Here, the technique has been extended to two-dimensional domain decompositions, where two communications are needed. To our knowledge, the case of three-dimensional domain decompositions has not been considered yet.

More than $\nu_2 = 1$ iterations are possible. The general condition that relates the number of *RBGS* iterations that can be done is [20]:

$$J \geq 4\nu_2 \tag{5.14}$$

However, for $\nu_2 > 1$, the size of the halo regions would probably be exceedingly large. This has two problems: the cost of the extra areas to be relaxed and the increase in the size of the coarsest level. In this work, these aspects have not been considered as the effort has concentrated in the new DDACM algorithm.

5.5.4 Benchmark

Different variants of a generic parallel MG algorithm have been benchmarked on the JFF cluster. Their main parameters are shown in Table 5.1.

Variant	**Cycle**	ν_1	ν_2	J	Δ	**Number of halo update operations per iteration**
1	V(2,2)	2	2	2	0	$6M - 1$
2	V(2,2)	2	2	2	1	$6(M-1)$
3	V(2,2)	2	2	4	1	$4(M-1)$
4	DDV(0,1)	0	1	4	1	2

Table 5.1: Parallel MG variants considered.

- Variant 1 is a straightforward parallelization of a sequential V cycle, using *RBGS* with a halo size of 2 and $\nu_1 = 1$ to have exactly the same results for any number of processors. It has been included in the benchmark to illustrate the importance of the direct solver for level 1 (the coarsest level)[6].
- Variant 2 is like variant 1 but it uses a direct solver for level 1[7].
- Variant 3 is numerically equal to variant 2 but it uses a larger halo region so two *RBGS* iterations can be done without updating the halos[8].

[6] The number of halo updates needed per iteration is $(2+2)M$ in the first leg plus $2(M-1)$ in the second leg

[7] The number of halo updates needed per iteration is $(2+2)(M-1)$ in the first leg plus $2(M-1)$ in the second leg

[8] The number of halo updates needed per iteration is $(2+1)(M-1)$ in the first leg plus $(M-1)$ in the second leg

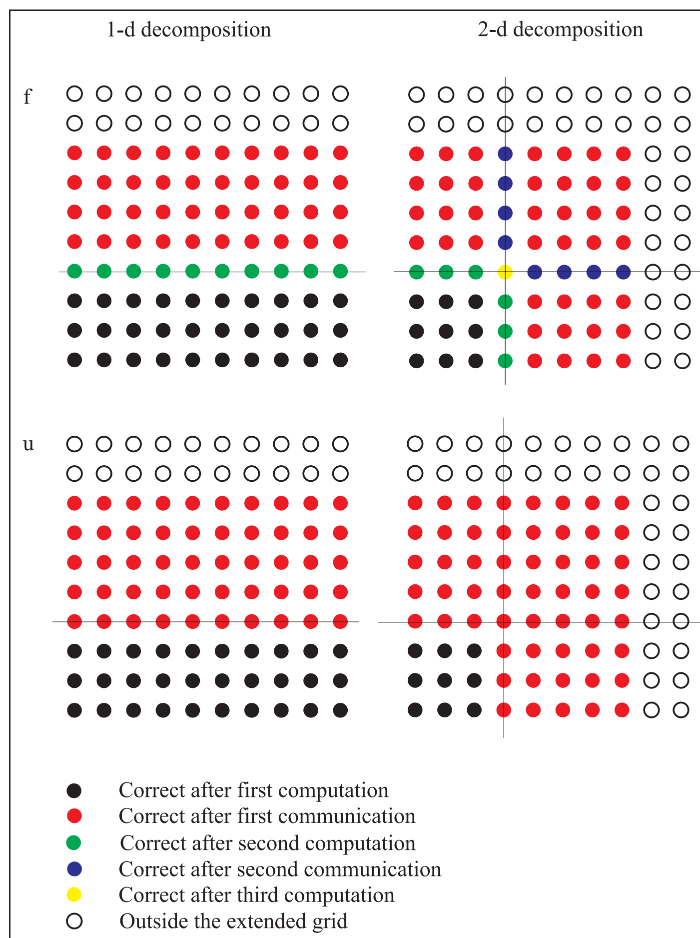

Figure 5.8: Evaluation of u and f before the first leg of a DDV algorithm in one-dimensional and two-dimensional domain decompositions.

- Variant 4 is the DDV(0,1) cycle, numerically equal to a V(0,1) cycle but with only two halo update operations, for any number of levels.

5.5.5 Numerical efficiency

For all the variants considered, the convergence process is independent of the number of processors. As a typical example, the convergence history of the problem model (section 2.7.1, see the second pararaph of next section) with $M = 8$, $N = 1536 \times 1536$ equations has been represented in Fig. 5.9.

For our problem model, if the criteria is the total number of iterations, the best algorithms are variants two and three, that converge to the specified accuracy in only four iterations for this example. Variant one, without direct solver for level 1, needs 17 iterations. Its behavior is even worse for other situations. The reason of its lower efficiency is that more than 2 *RBGS* iterations would be needed to converge level 1. The DDV(0,1) cycle, variant 4, needs 8 iterations. However, note that each V(2,2) cycle does four smoothing iterations per level and iteration while DDV(0,1) does only one.

In general terms, the restriction $\nu_1 = 0$ of DDV cycles limits their efficiency (measured in terms of the number of MG iterations needed to achieve convergence), compared with other V cycles such as V(2,2). It is important to note that the convergence ratios obtained with our implementation are consistent with the convergence ratios presented in [20]. The correctness of our implementations is additionally ensured by the fact that they are independent of the number of processors and by the

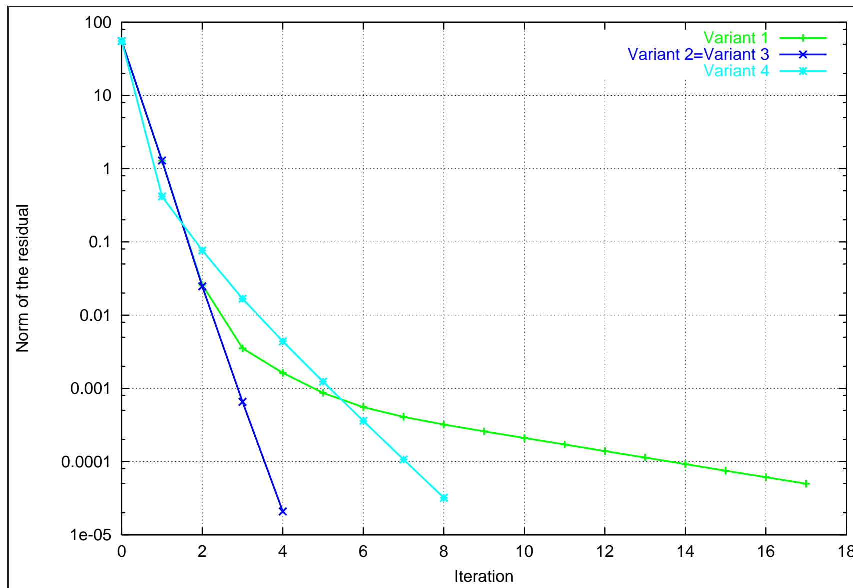

Figure 5.9: Convergence of the different parallel MG algorithms for a problem with $M = 8$, $N = 2.4 \times 10^6$ equations.

identical behavior of variants 2 and 3.

The question now is if the better computational behavior of DDV on loosely coupled systems can compensate for its worse convergence ratio. This aspect is considered in next section where the computing times with different numbers of processors are presented.

5.5.6 Computing times

The algebraic problem model (section 2.7.1), used for benchmarks in sections 3, 6 and 7, can not be directly solved by the geometric MG algorithms discussed here (which are designed to solve PDEs such as equation 5.1 and not linear systems of equations).

Thus, the PDE that generates a discrete equation for level M equivalent to the problem model (section 2.7.1) has been solved with variants 2, 3 and 4, for different problem sizes from $M = 4$ with $96 \times 96 \approx 9.2 \times 10^3$ equations to level $M = 8$ with $1536 \times 1536 \approx 2.4 \times 10^6$ equations. For shortness, results of variant 1 have not been presented as its efficiency is too low to be of interest.

A summary of the computing times[9] obtained on the JFF cluster is presented in Tables 5.2, 5.3 and 5.4.

M	N	N_{ite}	N_{hu}	$P = 1$	$P = 4$	$P = 9$	$P = 16$
4	96^2	5	90	1.18×10^{-1}	3.74×10^{-1}	7.20×10^{-1}	8.85×10^{-1}
5	192^2	4	96	4.04×10^{-1}	5.04×10^{-1}	8.01×10^{-1}	9.22×10^{-1}
6	384^2	4	120	1.75×10^{0}	9.55×10^{-1}	1.21×10^{0}	1.26×10^{0}
7	768^2	4	144	9.40×10^{0}	2.50×10^{0}	2.26×10^{0}	1.95×10^{0}
8	1536^2	4	168	2.82×10^{1}	1.07×10^{1}	5.10×10^{0}	3.78×10^{0}

Table 5.2: Execution times of parallel MG variant 2.

In tables 5.2-5.4, N_{ite} is the number of multigrid iterations needed to achieve convergence and N_{hu} is the total number of halo updates. Both are independent of the number of processors P. To ease the interpretation of the results, the computing times obtained for different problem sizes with

[9]The computing times presented are wall times in seconds. As there are variations in the time of each execution, the average of five executions has been presented.

M	N	N_{ite}	N_{hu}	$P=1$	$P=4$	$P=9$	$P=16$
4	96^2	5	60	1.15×10^{-1}	2.67×10^{-1}	5.35×10^{-1}	6.28×10^{-1}
5	192^2	4	64	4.02×10^{-1}	3.96×10^{-1}	5.61×10^{-1}	6.60×10^{-1}
6	384^2	4	80	1.72×10^{0}	7.68×10^{-1}	9.28×10^{-1}	9.19×10^{-1}
7	768^2	4	96	9.43×10^{0}	2.23×10^{0}	1.77×10^{0}	1.57×10^{0}
8	1536^2	4	112	2.81×10^{1}	1.03×10^{1}	4.46×10^{0}	3.16×10^{0}

Table 5.3: Execution times of parallel MG variant 3.

M	N	N_{ite}	N_{hu}	$P=1$	$P=4$	$P=9$	$P=16$
4	96^2	10	20	1.40×10^{-1}	2.05×10^{-1}	3.58×10^{-1}	5.64×10^{-1}
5	192^2	9	18	5.42×10^{-1}	3.28×10^{-1}	4.82×10^{-1}	5.79×10^{-1}
6	384^2	9	18	2.46×10^{0}	9.22×10^{-1}	8.26×10^{-1}	8.27×10^{-1}
7	768^2	8	16	1.12×10^{1}	2.90×10^{0}	2.02×10^{0}	1.38×10^{0}
8	1536^2	8	16	3.34×10^{1}	1.35×10^{1}	6.60×10^{0}	4.24×10^{0}

Table 5.4: Execution times of parallel MG variant 4.

9 and 16 processors have been represented in Figs. 5.10. The speedups for different mesh sizes, in Figs. 5.12 and 5.13.

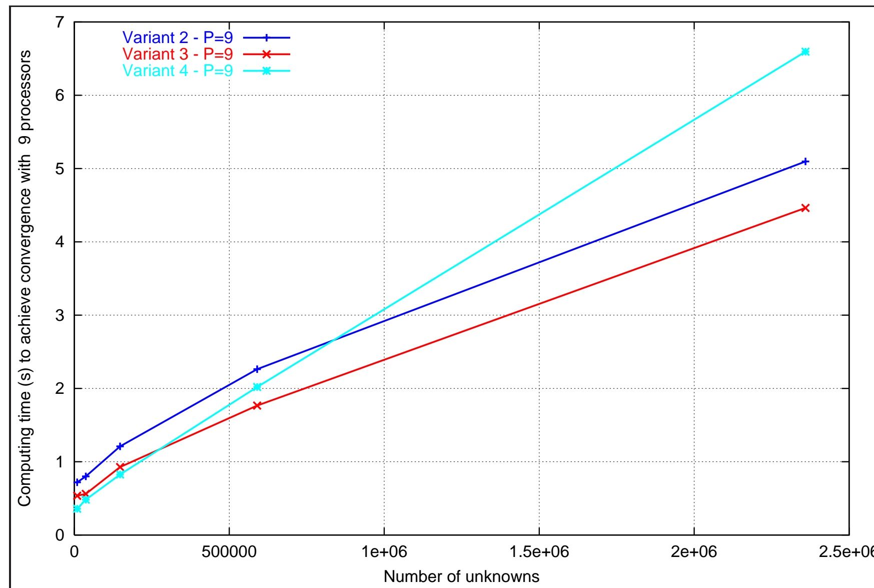

Figure 5.10: Computing time of the parallel MG variants with 9 processors versus number of unknowns.

The computing times for variant 4 are better for small problems. However, for large problems, the better convergence ration of variants 2 and 3 is the dominant factor and, in the conditions of the test, they are more efficient. The difference between variants 2 and 3 is due to the reduction of halo updates.

5.5.7 Breakdown of computing times

The fraction of total execution time spent in halo update operations and in the sequential solution of the coarsest level, for problems of different sizes, solved with $P = 16$ processors has been represented in Fig 5.14 and 5.15.

In variant 4, the computing time is never dominated by halo update operations, even for the

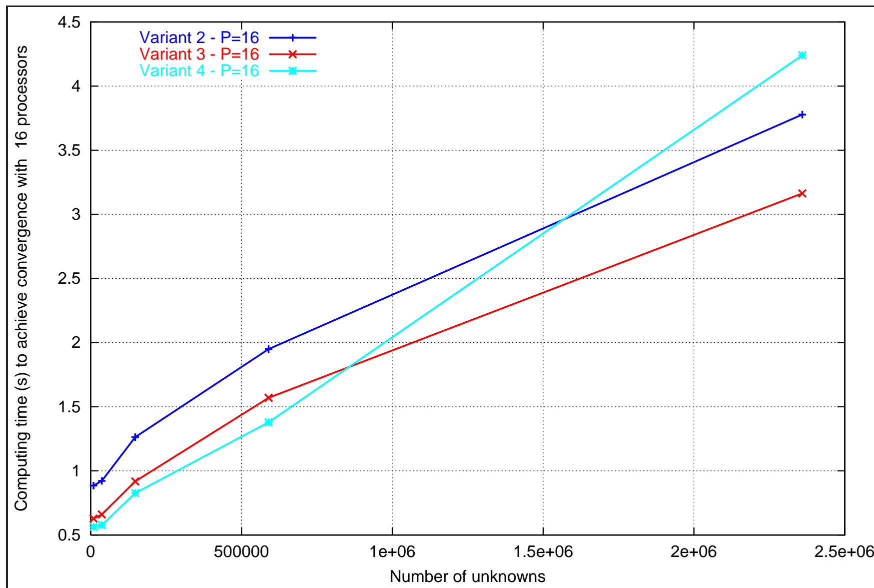

Figure 5.11: Computing time of the parallel MG variants with 16 processors versus number of unknowns.

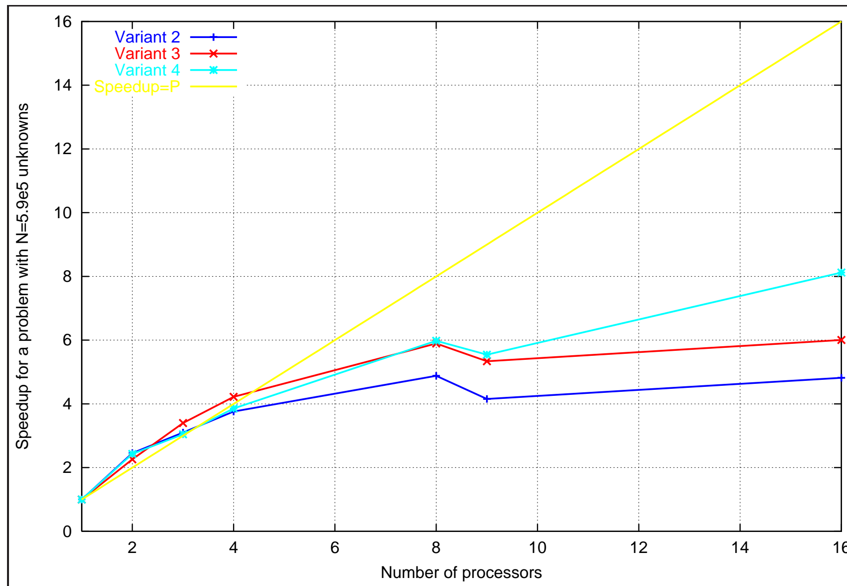

Figure 5.12: Speedup of the parallel MG variants for a problem with $N = 768^2 \approx 5.9 \times 10^5$ unknowns.

smaller problems considered. The situation is quite different for variants 2 and 3, where more than 50% of the total time is due to halo updates except for the largest problem considered.

5.6 Final remarks

Different strategies to increase the tolerance of parallel multigrid algorithms to high latency networks have been considered. The time to solve a model problem in the JFF cluster using four multigrid variants has been measured. The results illustrate their impact on the efficiency of the method:

- According to the results obtained for variant one, the use of a direct solver for the coarsest

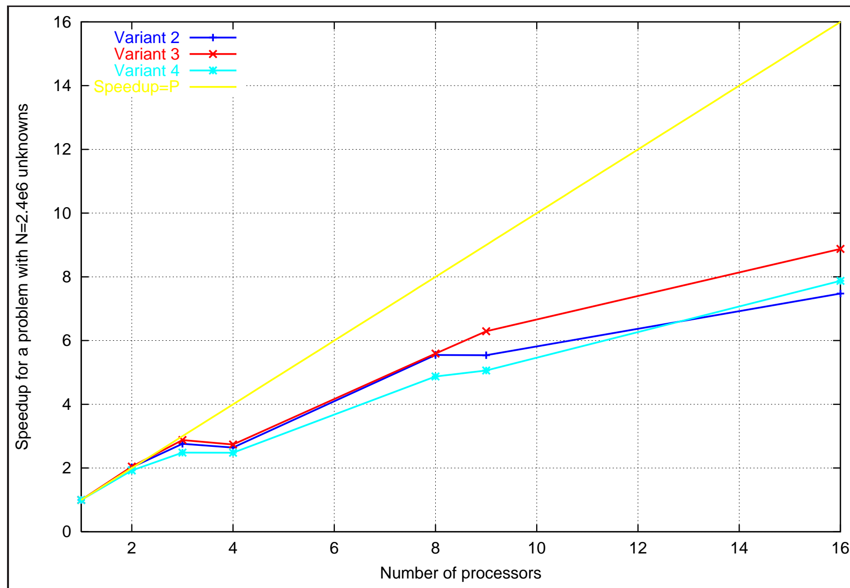

Figure 5.13: Speedup of the parallel MG variants for a problem with $N = 1536^2 \approx 2.4 \times 10^6$ unknowns.

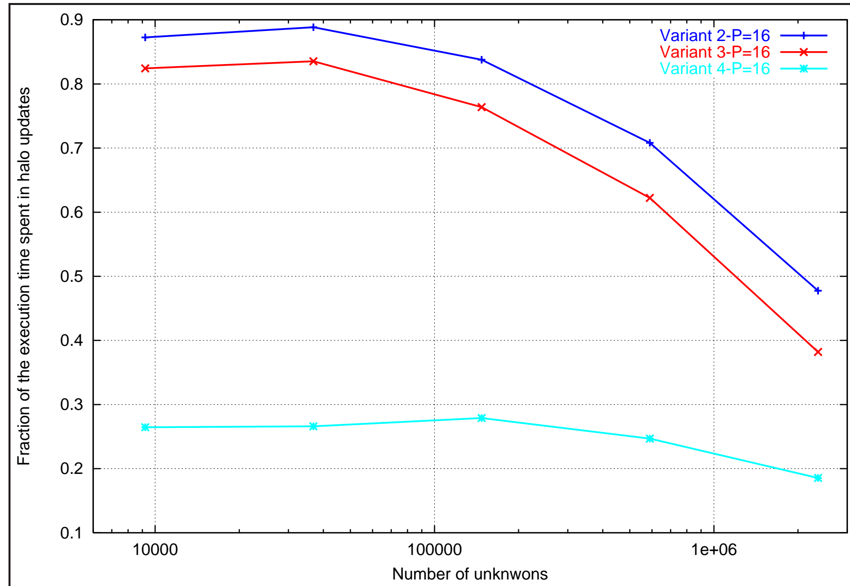

Figure 5.14: Fraction of total execution time spent in halo update operations by the parallel MG variants versus the problem size, with $P = 16$.

level is advisable. Its cost in terms of computing time is low, as it can be seen in Fig. 5.15, and its impact on the numerical efficiency is high, as can be seen in the example of Fig. 5.9.

- The use of overlapping areas that are relaxed to be able to do a group of smoothing iterations without updating the halos is also an interesting option, as can be seen comparing the difference of the computing times between variants 2 and 3 in Figs. 5.10 and 5.11, with halo regions of $J = 2$ and $J = 4$ nodes.

- The conclusion referent to the suppression of the pre-relaxation stage (DDV cycle) is not so clear. It allows an important reduction of the number of halo update operations, but this is at the cost of decreasing the convergence ratio. For small and medium size problems, the total

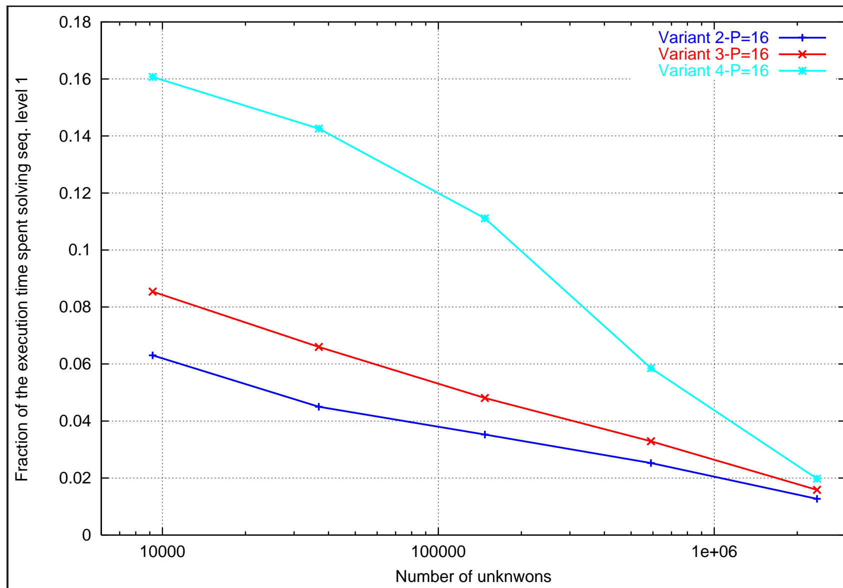

Figure 5.15: Fraction of total execution time spent by the parallel MG variants in the sequential solution of the coarsest level versus the problem size, with $P = 16$.

execution time of variant 3 is lower, but not for the larger meshes where the better convergence ratio of variant 4 is the dominant factor (Figs. 5.10 and 5.11).

However, most of the cost of each DDV iteration is due to computing time and not to halo update operations (Fig. 5.14). Thus, in a cluster with the same network but a faster processor (or a more efficient implementation of the algorithm) the conclusion could be the opposite.

As a general conclusion of this chapter, if care is taken to increase the tolerance to high latency networks, the parallel multigrid algorithms are an attractive option for implicit parallel CFD, even on loosely coupled systems such as Beowulf clusters with conventionally fast Ethernet networks. They provide reasonable speedups, and, most important, low absolute computing times.

In this thesis, the next goal is to implement the DDV concepts in an ACM context. The resulting algorithm has been called DDACM algorithm. The information obtained from the analysis of the DDV cycle has been useful for DDACM. The main aspects that will be reconsidered are the role of the direct solver and the suppression of the pre-relaxation stage.

5.7 Nomenclature

E^{num}	numerical efficiency
f	right-hand side
h	distance between nodes
$I_o^d, \hat{I}_o^d$	integrid transfer operators o origin level d destination level
j	number of halo rings where all the nodes are correct
J	halo size
L	generic differential operator
nb	neighbour
nr	residual norm
N	number of unknowns
N_{com}	number of communication operations
p	processor
P	number of processors
x, y	coordinates
r	residual
u	generic unknown
v	error

Greek symbols

Δ	use of a direct solver for level 1
ϵ	precision
γ	number of correct rings
κ	cycle index
ν	number of iterations
ν_{loc}	number of local iterations
ν_1, nu_2	number of pre and post-smoothing iterations
Ω	domain

Subindices

1,2,3	Cartesian components
k	level, number of levels
i, j	position in the discrete map

Superindices

$1 \cdots M$	levels **from coarse to fine**