ISSN 2007-9737

Distributed Geometric Multigrid Method: Analysis of a V Cycle Truncation Level Criteria

Matías Valdés, Sergio Nesmachnow

Universidad de la República,
Uruguay

{mvaldes,sergion}@fing.edu.uy

Abstract. This article presents the analysis of a V cycle truncation level criteria in a parallel implemention of a geometric multigrid method for solving partial differential equations, developed over a distributed memory system. The proposed system is implemented in C, using the Message Passing Interface library. A theoretical analysis of the proposed truncation level criteria is presented, and its evaluation is reported for the Poisson problem. The experimental analysis indicates that the proposed method achieves accurate speedup and computational efficiency, and shows a good scalability behavior to solve large problems by properly using more processing units.

Keywords. Multigrid, distributed memory, truncated V cycle, MPI.

1 Introduction

Multigrid methods are efficient numerical algorithms for solving partial differential equations by applying several discretization formulae, hierarchically organized [8, 16]. They are recognized within the most efficient techniques for solving partial differential Eqs. [16]. Furthermore, they can be expanded to employ higher level data structures to implement powerful multilevel methods, to address complex problems in various research domains by properly handling different matrix patterns, lattices, and other useful abstract mathematical structures for arranging the set of points that discretize a given problem domain.

Multigrid methods are also suitable for parallelism. A typical approach is to take advantage of the inherent parallel computations on the multiple grid components [10]. However, several important considerations must be taken into account to achieve a proper computational efficiency, including the sequential processing of elements in each grid level and the granularity of the parallel computations, which may be different for different levels.

One of the most useful applications of multigrid numerical methods is as solvers for elliptic partial differential equations. In fact, they are characterized as the fastest methods for elliptic problems [16]. One of the most notorious elliptic partial differential equation is Poisson equation. This equation usually appears as part of dynamical models for different physical phenomena. For example, it is related to the Navier-Stokes equations for incompressible flows, as it appears as a sub-problem in the discretization of these equations by using the MAC method, the projection method, and the fractional-step method [7]. Also, the incompressible Navier-Stokes equations may be reformulated into the Poisson pressure equation [9]. Poisson equation is also widely used as a test problem in numerical analysis and high performance computing [5].

In this line of work, this article proposes and analyzes the performance of a distributed memory implementation of a goemetric multigrid method, using truncated V cycles, and applied to a Poisson problem. The proposed approach consists in truncating each V cycle at the deepest possible level l that also guarantees that each processing unit is assigned a given constant number of vertices N. This truncation criteria was mentioned by Linden et al. [11], although it was not considered for V cycles.

We propose a theoretical analysis of the aforementioned truncation criteria, which characterizes the level reached in the truncated V-cycle, and its number of vertices, in terms of the problem size and number of processing units. We then present an experimental analysis of the computational performance of the implemented multigrid method on a distributed memory infrastructure.

The article is organized as follows. Section 2 describes the test problem and the main concepts about geometric multigrid methods. Section 3 presents a review of related works. Section 4 describes the proposed truncated V cycle criteria and the implementation details of the multigrid solver. The experimental evaluation is reported in Section 5. Finally, Section 6 presents the conclusions and formulates the main lines for future work.

2 Background and Theoretical Foundations

This section presents the test problem and the theoretical foundations for the proposed approach.

2.1 Test Problem: Poisson Equation

The Poisson problem was selected to test the performance of the implemented multigrid method. Poisson equation is an elliptic partial differential equation with several applications in science, which is commonly used for the evaluation of multigrid solvers (see the review of related works in Section 3).

The considered Poisson problem is defined on the unit square as domain, and Dirichlet boundary conditions are assumed, as expressed in Eq. 1. There, $\Delta u(x,y) = u_{xx}(x,y) + u_{yy}(x,y)$ is the Laplacian operator; $v(x,y)$ is a known *source* function; and $g(x,y)$ is also a known function that determines the boundary conditions. The Poisson equation is discretized by using centered finite differences, in a regular grid of $n+1$ vertices per dimension. The discretization generates a sparse linear system, with $(n-1)^2$ unknowns (one for each interior vertex) [2]. The resulting linear system is solved with multigrid to compute an estimation of $u(x,y)$ in the points of the grid:

$$\text{find } u : \Omega = [0,1]^2 \subseteq \mathbb{R}^2 \rightarrow \mathbb{R} \,/ \\ \begin{cases} -\Delta u(x,y) = v(x,y), & \forall\,(x,y) \in \text{int}\,(\Omega)\,, \\ u(x,y) = g(x,y), & \forall\,(x,y) \in \partial\Omega. \end{cases} \quad (1)$$

2.2 Multigrid Methods

Multigrid methods work under the idea of accelerating an iterative solver by using information provided by a global correction procedure, which operates on coarse grids to solve a fine grid problem which in turn is easier to solve. The recursive process, called the multigrid cycle, is applied until a direct solver can be applied without additional computation cost on the coarsest grid. V-cycle is one of the most popular type of multigrid cycles, characterized by computing the coarse grid correction down levels (the *restriction phase*), until finding a level where the direct method is applied.

Numerical computations on steps performed in coarse grids are quicker, as few vertices are considered, and the numerical error converges faster as the method goes down levels [8]. Then, an interpolation is applied (the *prolongation* phase) to determine values on upper level grids, until the finer grid is again reached. Multigrid methods are recursive in nature and a V-cycle can be extended to any number of levels.

In this article, geometric multigrid, implemented with cycles of type V, is used as a solver for the discretized Poisson problem. The problem domain is first partitioned into a uniform grid, in which an initial estimation is find by applying few iterations from an iterative *smoother* method. The residual of this initial estimation is then restricted into a a coarser grid, where it is corrected by applying few iterations of the smoother.

This process is repeated by taking successively coarser grids. When the coarsest grid is reached, estimations from the different grid levels are combined by interpolation to the finer grids, until the initial grid is reached. This process is illustrated in Fig. 1, where h is the width of the initial grid, n is the number of unknowns in each dimension of this grid, and R and I represent the restriction and interpolation operations, respectively.

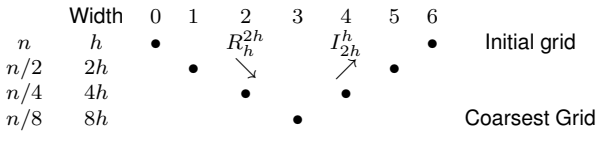

Fig. 1. A non-truncated V cycle for MG

Algorithm 1 Recursive multigrid V cycle: $V^h(A_h, r^h, u^0)$

Require: $h = 1/(m-1)$, $A_h \in \mathbb{R}^{m\times m}$, $r^h \in \mathbb{R}^{m\times 1}$, $\theta \in \mathbb{N}^+$

1: **if** actual grid is not the coarsest **then**
2: $x^h = \text{gs\_rb}(A_h, r^h, \theta)$ ▷ GS-RB from u^0
3: $r^{2h} = R_h^{2h}(r^h - A_h x^h)$ ▷ Restrict to coarser grid
4: $x^{2h} \leftarrow V^{2h}(A_{2h}, r^{2h}, u^0 = \vec{0})$ ▷ Recursion
5: $x^h = x^h + I_{2h}^h(x^{2h})$ ▷ Interpolate and correct
6: $x^h = \text{gs\_rb}(A_h, r^h, \theta)$ ▷ GS-RB with $u^0 = x^h$
7: **else**
8: $x^h \leftarrow \text{sor}(A_h, r^h)$ ▷ SOR until convergence
9: **end if**
10: **return** x^h

A recursive implementation for a multigrid V cycle is presented in Algorithm 1, following the idea by Briggs et al. [2]. The implementation uses Gauss-Seidel (GS) as smoother method. GS applies a Red-Black update order strategy and executes a fixed number of iterations θ. R_h^{2h} represents the restriction of the grid vertices to the next coarser grid; which is done by full-weighting of adjacent vertices. I_{2h}^h denotes the interpolation to the finer grid, which is performed by bi-linear interpolation of adjacent vertices. Finally, the Successive Over-Relaxation (SOR) method is applied as the coarse grid solver.

Gauss-Seidel Red-Black (GS-RB) is selected as smoother method, since it is a more parallelizable version of the traditional Gauss-Seidel solver. GS-RB is obtained by modifying the order in which grid vertices are updated. Vertices are first separated into two colors: red and black, intercalated. Then, they are updated with the usual GS expression, first applied to all red vertices, and then to the black ones [15]. This way, vertices of the same color may be updated concurrently. An important part of the smoother is the residual, as it is used as the independent term of the linear system solved by the multigrid method in each grid. In the case of GS-RB, the residual for black vertices is always null, as they are updated last. For the red vertices update, the residual coordinates associated to u^{k+1}, are given by Eq. 2 (for $i + j =$ even):

$$r_{i+j}^{k+1} = v_{i,j} + \frac{u_{i,j-1}^{k+1} + u_{i,j+1}^{k+1} - 4u_{i,j}^{k+1}}{h^2} + \frac{u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1}}{h^2}. \quad (2)$$

When solving a discretized Poisson linear system, GS-RB requires $O(n^4)$ operations in order to estimate a solution satisfying $\|r^k\| < \epsilon\|r^0\|$, $\epsilon > 0$. More efficient methods are Conjugate Gradient or Successive Over-Relaxation (SOR); both with $O(n^3)$ operations [4]. For the proposed implementation, SOR was chosen as coarse grid solver, as it may be obtained easily from the GS implementation. Specifically, SOR with Red-Black order (SOR-RB) is used. The linear system of interest is symmetric and positive definite, which implies that SOR-RB is convergent for any $w \in (0, 2)$ [4]. The value of w was chosen to maximize the convergence speed, as $w_{\text{opt}} = 2/(1 + \sin(\pi h))$. The coordinates of SOR residual, associated to u^{k+1}, are given by Eq. 3 (valid for red and black vertices):

$$r_{i,j}^{k+1} = v_{i,j} + \frac{u_{i,j-1}^{k+1} + u_{i,j+1}^{k+1} - 4u_{i,j}^{k+1}}{h^2} + \frac{u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1}}{h^2}. \quad (3)$$

Each multigrid iteration applies one V cycle, with initial condition taken as the previous V cycle final estimation: $u^{k+1} = V^h(A, b, u^k)$.

When developing a parallel implementation of a multigrid method in a distributed memory system, the number of vertices assigned to each processing unit decreases exponentially while descending in a V cycle. Thus, the cost of message passing may start to prevail over the

cost of computations, producing a degradation of the overall computational efficiency. A technique to overcome this problem is, instead of reaching the coarsest grid, using *truncated* V cycles [16], as illustrated in Fig. 2. When using a truncated V cycle, Algorithm 1 iterates until reaching the coarsest grid, which is determined by the truncation criteria.

The truncated V cycle strategy has a specific drawback: if the cycle is truncated too soon, the (truncated) coarse grid may have too many vertices, and solving the coarse grid linear system may become a bottleneck, affecting the overall computational efficiency of the solver. Thus, selecting an appropriate truncation level that takes into account the resulting trade-off is one of the most important challenges when implementing distributed memory multigrid methods [10].

3 Related Works

Sterk and Trobec [14] presented a parallel implementation of a multigrid method applied to solve the 3D Poisson equation, within a fluid flow simulation. The proposed algorithm was implemented in MPI and executed on a cluster of workstations. The authors performed several configuration experiments, including finding the grid size for switching from parallel to sequential execution. A comparison with a SOR method was reported. Both methods achieved similar sub-linear speedup values (i.e., below 0.8), but the results confirmed that the parallel multigrid method had a better scalability behavior than parallel SOR.

Gradl and U. Rüde [6] studied multigrid implemented over the Hierarchical Hybrid Grids framework for finite element problems and using the Metis mesh partitioning software. A specific algorithm design was proposed to lower the communication overheads, by reducing the number of messages exchanged between parallel processes.

The time per V cycle and the time to the overall solution were analyzed in weak scalability experiments. Results showed a good parallel scalability of the implemented multigrid method, but authors acknowledged that further improvement were needed for a full optimization of the communication patterns. The data structures used for calculation also allow applying adaptive mesh refinement methods, e.g., using hanging nodes or red-green refinement, to expand the applicability of the proposed solver.

Daley et al. [3] proposed two parallel mapping algorithms for addressing the scalability limitations of multigrid methods due to excessive communication costs in the coarser grids. Improved communication algorithms were conceived to map the mesh back and forth between an uniform grid and an adaptive mesh refinement procedure. The performance of the proposed parallel mapping algorithms was analyzed for a case study involving a multigrid Poisson solver using octrees block structures in FLASH, a multiphysics/multiscale method for flows simulations.

Experiments were performed in a Virtual Node with 4 MPI tasks per node and 512 MB of memory per MPI task, on a IBM BG/P platform. Results indicated that the proposed implementation allowed obtaining an increase on performance when increasing the number of computing resources, depending on the level used for switching from a refined mesh in the octree to a uniform grid (the higher the refinement level, the better the performance gain).

Müller and Scheichl [12] studied the scalability of numerical methods for solving elliptic partial differential equations for atmospheric fluid dynamics. Solvers based on Krylov subspaces and multigrid algorithms were categorized as the most efficient methods, after an experimental evaluation performed in the national supercomputer from the UK. Several algorithms were evaluated, already implemented in two well known libraries of routines for scalable parallel linear systems resolution: Distributed and Unified Numerics Environment (DUNE) and the Parallel High Performance Preconditioners (hypre).

In turn, two custom implementations were developed: a Conjugate Gradient solver using vertical line relaxation preconditioner and a tensor-product geometric multigrid algorithm. Results demonstrated that the overall computational efficiency of the tensor-product geometric multigrid method was better

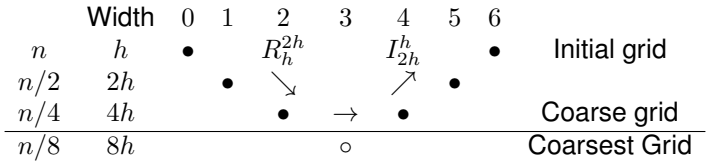

Fig. 2. A truncated V cycle for MG

that standard algebraic multigrid methods. Furthermore, the implemented multigrid solver was more robust than one-level methods when considering parameter variations.

The scalability of multigrid solvers available in hypre was also studied by Baker et al. [1]. Several multigrid algorithms were compared, including PFMG, SMG, SysPFMG, BoomerAMG, and AMS, to solve three benchmark problems: a 3D Laplace (i.e., Poisson with $v(x, y) = 0$) equation with Dirichlet boundary conditions, a system of two 3D Laplace equations with weak inter-variable coupling, and a simple 3D electromagnetic diffusion problem. The main results demonstrated the usefulness of considering assumed partition instead of global partitions, and both a distributed memory implementation using MPI and a hybrid distributed/shared memory MPI/OpenMP implementation were able to achieve accurate scalability values when solving large problem instances.

The truncation criterion proposed in this article is aimed to improve load balancing, by truncating each V cycle at the deepest possible level l that also guarantees that each processing unit processes a given constant number of vertices N. This criterion was suggested by Linden et al. [11], in their analysis of scalability aspects of parallel multigrid, but instead of using V cycles, they implemented a full multigrid method with W cycles in the intermediate grids. Experimental results indicated that the (truncated) coarsest grid calculations are determinant for scalability performance.

In a more recent article, Dexuan and Ridgway analyzed the convergence and efficiency of multigrid methods with truncated V-cycle, which they call U-cycle [17]. However, authors did not propose a specific criteria for selecting the coarsest grid, which was generally defined as "fine enough so that all processors are productively busy in doing the coarse-grid solver".

4 The Proposed Parallel Geometric Multigrid Solver

This section describes the proposed parallel multigrid method over distributed memory systems.

4.1 Truncated V Cycle: Criteria and Properties

The proposed method is based on choosing a truncation level which is deep enough to have few vertices per processing unit, but not as few as to have idle processing units. The formal definition is presented next.

Definition 1 (Truncation criteria)**.** *Consider an initial grid with $n = q \times 2^r$ vertices per dimension, where q is odd. In this case, each V cycle has a maximum of r levels. Given $N \geq 1$ and p processing units, truncate the V cycle at the deepest possible level $l \leq r$, which also satisfies: $n_l^2/p \geq N$, where $n_l = n/2^l$ is the number of vertices per dimension in the (truncated) coarse grid.*

Algorithm 2 presents a pseudocode for the proposed truncation criteria. Two conditions (line 3) control the iterative descending procedure. When using this criteria with an ideal load balancing, no processing unit is idle at the coarsest (truncated) grid. To compensate for some unbalance, the value of N should be chosen greater than one.

The level reached by the proposed criteria depends on the initial size n, the number of processing units p, and the value chosen for N. Theorem 1 characterizes this dependence.

ISSN 2007-9737

Algorithm 2 Proposed criteria for truncating a V cycle of geometric multigrid

Require: $n = q \times 2^r$, q odd, $r \geq 0$, $r \in \mathbb{N}$, $N \geq 1$, $N \in \mathbb{N}$

1: $l = 0$ ▷ Initial level of V cycle
2: $n_l = n$ ▷ Vertices per node at level l
3: **while** $l < r$ **and** $(n_l/2)^2 \geq Np$ **do**
4: $n_{l+1} = n_l/2$
5: $l = l + 1$
6: **end while**
7: **return** l, n_l

Theorem 1. *Let assume that the initial number of vertices per dimension is $n = q \times 2^r$, q odd, $r \in \mathbb{N}$, $r \geq 0$. Then, the level l of the truncated V cycle reached with the proposed criteria, is (for $p \in \mathbb{N}$, $p \geq 1$):*

$$l = \begin{cases} 0, & \text{if } \frac{n}{\sqrt{Np}} < 2 \\ \lfloor \log_2 \left(\frac{n}{\sqrt{Np}} \right) \rfloor \in [1, r-1], & \text{if } 2 \leq \frac{n}{\sqrt{Np}} < 2^r \\ r, & \text{if } \frac{n}{\sqrt{Np}} \geq 2^r. \end{cases}$$

Proof.

1. If $n/\sqrt{Np} < 2$, the second iteration condition (line 3 in Algorithm 2) is not satisfied at level $l = 0$, and the method does not descend any level.

2. To reach a given level $1 \leq l \leq r$, both iteration conditions must be satisfied at level $l - 1$. The first condition holds, as $l - 1 < r$. The second condition is:

$$n_{l-1}^2 = \left(\frac{n}{2^{l-1}} \right)^2 \geq 4Np \Leftrightarrow \frac{n}{2\sqrt{Np}} \geq 2^{l-1}$$

$$\Leftrightarrow \frac{n}{\sqrt{Np}} \geq 2^l.$$

   The reached level l is the last level, whenever the first or second stopping condition is satisfied at this new level. That is: if $l = r$, or:

$$n_l^2 = \left(n/2^l \right)^2 < 4Np \Leftrightarrow \frac{n}{2\sqrt{Np}} < 2^l.$$

3. Thus, level r is reached, if and only if:

$$\frac{n}{\sqrt{Np}} \geq 2^r.$$

4. In the rest of the cases, level l is reached, with $1 \leq l < r$. For those cases:

$$\frac{n}{2\sqrt{Np}} < 2^l \leq \frac{n}{\sqrt{Np}} \Leftrightarrow \log_2 \left(\frac{n}{2\sqrt{Np}} \right) < l$$

$$\leq \log_2 \left(\frac{n}{\sqrt{Np}} \right) = 1 + \log_2 \left(\frac{n}{2\sqrt{Np}} \right)$$

$$\Leftrightarrow l = \lfloor 1 + \log_2 \left(\frac{n}{2\sqrt{Np}} \right) \rfloor = \lfloor \log_2 \left(\frac{n}{\sqrt{Np}} \right) \rfloor.$$

□

Example 1. *Consider $n = 10240 = 5 \times 2^{11} = q \times 2^r$ and $p = 64 = 2^6$. Without truncating the V cycle, the deepest possible level is $l = 11$; resulting in a coarse grid with $n_c = 5$ vertices per dimension, and $n_c^2/p = 25/64 < 1$ vertices per processing unit (for an ideal load balance). Now consider truncating each cycle with the proposed criteria, using $N = 4$. In this case: $n/\sqrt{Np} = 5 \times 2^7 < 2^{11}$. Therefore, by Theorem 1, the value of l is given by Eq. 4:*

$$l = \lfloor \log_2 \left(\frac{n}{\sqrt{Np}} \right) \rfloor = \lfloor \log_2 \left(5 \times 2^7 \right) \rfloor =$$

$$\lfloor \log_2 (640) \rfloor = \lfloor 9.3219 \rfloor = 9. \quad (4)$$

The number of vertices per dimension at this level is $n_l = n/2^9 = 20$. Assuming an ideal load balance, the number of vertices per processing unit is $n_l^2/p = 6.25$. Fig. 3 shows the reached level l, coarse grid vertices per dimension n_l, and per processing unit n_l^2/p, for different values of p and N, as given by Theorem 1. Corollary 1 bounds the number of vertices in the coarse grid, independent from n and p.

Corollary 1. *For a truncated coarse grid reached at level $l \in \{1, ..., r-1\}$, the total number of vertices is bounded by $Np \leq n_c^2 < 4Np$, and the number of vertices per processing unit is bounded by $N \leq n_c^2/p < 4N$ (for ideal load balance).*

Proof. From the proof of Theorem 1:
$n/2\sqrt{Np} < 2^l \leq n/\sqrt{Np} \Leftrightarrow \sqrt{Np} \leq n_c = n/2^l < 2\sqrt{Np} \Leftrightarrow Np \leq n_c^2 < 4Np.$ □

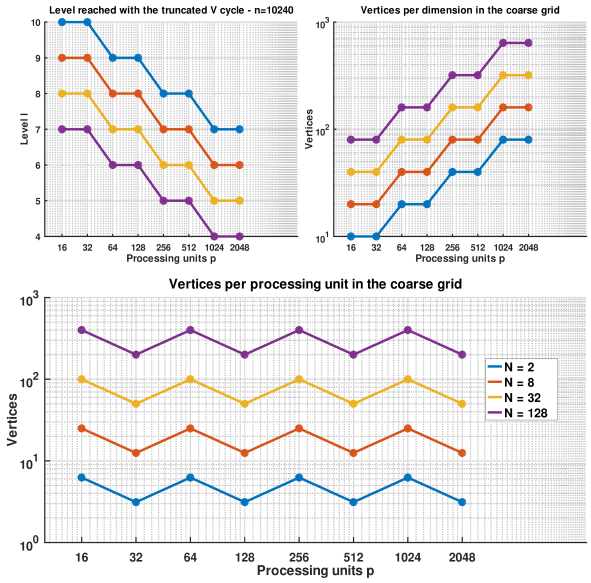

Fig. 3. Truncated V cycle for $n = 5 \times 2^{11}$, when using the proposed criteria: level reached (l), coarse grid vertices per dimension (n_l), and per processing unit n_l^2/p

4.2 Design and Implementation Details

This subsection presents the design considerations and implementation details of the proposed method.

4.2.1 Design Considerations

The discretization of the Poisson problem results in a square linear system with $(n-1)^2$ unknowns, $n = 1/h$. The solution of the system approximates the exact solution $u(x, y)$ in the selected grid vertices. Thus, to obtain a good spatial resolution, the value of n must be taken sufficiently large. This decision imposes two practical limitations: (1) the calculation time required to process an increasing number of vertices, and (2) the required memory to store the partial and final estimations of each vertex. A distributed memory parallel implementation is applied to overcome these practical limitations.

4.2.2 Domain Decomposition and Processing

The decomposition approach is based on dividing the domain grid into rectangular sub-domains, to be assigned to each of the available processing units. A Cartesian grid is created and the optimal number of processing units per dimension (N_x and N_y) is determined using the `MPI_Dims_create` function of MPI. The output information is then used to create a new MPI communicator with Cartesian topology, by using the `MPI_Cart_create` function.

The grid vertices are then distributed along the processing units. Let (q_x, q_y) the coordinates of a generic processing unit q in the Cartesian communicator, processing unit q receives the inner grid vertices with coordinates in the two intervals I_x^q and I_y^q, as defined in Eqs. 5 and 6, where the integer division operator is used:

$$I_x^q = \left[\frac{q_x(n-1)}{N_x} + 1, \frac{(q_x+1)(n-1)}{N_x}\right], \quad (5)$$

$$I_y^q = \left[\frac{q_y(n-1)}{N_y} + 1, \frac{(q_y+1)(n-1)}{N_y}\right]. \quad (6)$$

Once the data is partitioned, each processing unit applies the same MPI multigrid code to its own assigned vertices. The numerical resolution is performed in a coordinate way with other processing units, by exchanging *halo layers* of *ghost vertices*, as explained in the next subsection. Halo layers act as boundary values for each sub-domain, and contain values estimated by adjacent processing units. Fig. 4 illustrates an example of the domain decomposition.

4.2.3 Communications

Halo layers are exchanged by message passing between neighboring processing units, i.e., those adjacent to each other in the defined Cartesian communicator. The `MPI_Cart_shift` function is used to determine neighbors. Each processing unit updates its halo layers after receiving data from its neighbors, and then sends its updated vertices to act as halo layers for its neighbors. Message passing is done using the combined exchange mechanism implemented in the `MPI_Sendrecv`

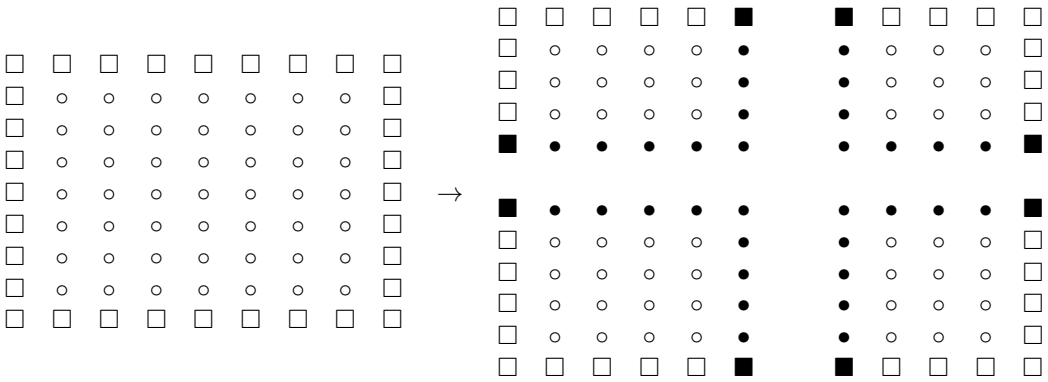

Fig. 4. Example of domain decomposition into four sub-domains, with an overlap region (halo layers) of width one. Inner vertices (white circles), boundary vertices (white squares), and ghost vertices (black circles/squares) are distinguished

function, which has the specific advantage of avoiding deadlocks between processes.

In order to improve the efficiency of the halo layers exchange, all the information is packed into MPI datatypes. Halo layers corresponding to a grid row are grouped into a contiguous datatype using the `MPI_Type_contiguous` function. In turn, for packing column halo layers, which are not contiguous, a vector datatype is created by using the `MPI_Type_vector` function. The exchange of halo layers is performed in the following stages: at every iteration of GS-RB and SOR-RB, after updating black or red vertices; after calculating the final residual vector of each GS-RB call; at the end of each restriction or interpolation operation.

Besides exchanging halo layers, a reduction operation (using `MPI_Reduce`) is applied: (1) to compute the residual, needed for the stopping criteria, after each multigrid cycle or SOR-RB iteration; (2) to compute the norm of the independent term at the beginning of the multigrid main iteration; and (3) to determine the overall execution time, as the maximum execution time of all processes.

4.2.4 MPI-IO Storage

During the multigrid execution, each processing unit stores in local memory the estimated values for its assigned vertices. In the developed implementation, the final estimation is stored concurrently into a unique binary file, by using MPI-IO. Each processing unit writes the solution of its sub-domain to the unique binary file using function `MPI_File_write_all`. The time spent in I/O operations is not included in the overall execution time when analyzing the performance of the proposed multigrid implementation, since it heavily depends on the capabilities (technology, transfer speed) of the hard disk. In turn, omitting the I/O times allows providing a baseline for comparison with other proposals.

4.2.5 Time Measurement

Execution time is measured as the difference between two calls of functions `time` (sequential) or `MPI_Wtime` (parallel). The parallel time is the maximum of all times of processing units, obtained with an `MPI_Reduce()` directive. The following initial steps are included in the overall execution time: determining the optimal number of processing units per grid dimension, creating a Cartesian communicator, assigning vertices to the processing unit, and determining neighboring processing units.

5 Experimental Evaluation

This section describes the experimental evaluation of the proposed truncation level criteria and discusses the main results.

5.1 Development and Execution Platform

The proposed multigrid method was implemented in the C programming language, using the MPICH implementation of the MPI standard (`www.mpich.org`). All floating point calculations use double precision.

The experimental evaluation was performed on HP ProLiant DL380 G9 servers with two Xeon Gold 6138 processors with 20 cores at 2.00 GHz each, and 128 GB of RAM, connected by Ethernet 10 Gbps, and running the Linux CentOS 7 OS, from National Supercomputing Center (Cluster-UY), Uruguay [13].

5.2 Test Problem, Convergence Criterion and Parameters

Tests were performed considering the Poisson problem defined in Eq. 1, whose exact solution given by Eq. 7:

$$u(x,y) = e^{-\left((x-\frac{1}{2})^2+(y-\frac{1}{2})^2\right)} + \frac{5}{100}\left(\sin(10\pi x)+\sin(4\pi y)\right). \quad (7)$$

The source function is taken as $v(x,y) = -\Delta u(x,y)$, and boundary values $g(x,y)$ are obtained by restricting the exact solution to the domain boundary.

Regarding the convergence criterion, SOR-RB stops execution when the residual of the corresponding linear system at step k satisfies $\|r_k\|_2/\|r_0\|_2 \leq 10^{-6}$. The same criteria is used for the multigrid main iteration, but with b instead of r_0. For multigrid the maximum number of iterations is set to 15 V cycles. For SOR-RB this is set to $5n_c$ iterations; where n_c is the number of vertices per dimension in the coarse grid. The truncation criteria uses $N = 5$.

The first multigrid iteration (first V cycle) uses $u^0 = \vec{0}$. For subsequent cycles, the initial estimation is taken as the final estimation of the previous cycle ("warm restart"). The GS-RB smoother does $\theta = 2$ iterations before descending to a coarser grid (pre-smooth), and after ascending to a finer grid (post-smooth).

5.3 Execution Time, Speedup and Efficiency

The *algorithmic speedup* metric is applied to analyzed the performance of the proposed method. Five independent executions were performed to reduce bias due to the utilization of a non-dedicated hardware platform. The chosen values for n are of the form $q \times 2^r$, with $q \in \{1,3,5\}$.

Table 1 reports the average execution times of the proposed method ($\overline{T_p}$), with their corresponding coefficient of variation ($\hat{V}$(%)), using an increasing number of computing units (p, up to 70) and cluster nodes (c), for different problem sizes (n from 8192 to 20480).

The execution time results reported in Table 1 Results demonstrate that the proposed implementation was able to steadily reduce the execution times of the multigrid method, as p increases. In particular, for $n = 20480$, the execution time reduced from 781 seconds for the sequential implementation to 14.0 seconds with $p = 70$. Despite using a non dedicated cluster, a robust behavior was observed in the execution times, with coefficients of variation less than 5% for most cases.

Table 2 reports the corresponding algorithmic speedup values (S_p) for the execution times reported in Table 1. A graphical analysis of speedup values is presented in Fig. 5.

The efficiency results in Table 2 indicate that the best speedup ($S_{70} = 55.6$) was obtained for the two largest problems. Constant speedup intervals are observed at different values of p; particularly for $[48,50]$ and $[56,60]$, possibly explained by the transition from 2 to 3 computing nodes, and the corresponding increase in communication latencies. In general, speedup is always increasing for the selected range of processing units, which may indicate that communications do not prevail over computations, which is the main objective of the proposed truncation level criteria.

Table 3 reports the computational efficiency of the proposed method, defined by the normalized value of the speedup when using p computing resources (Equation 8):

$$E_p = \frac{S_p}{p}. \quad (8)$$

Table 1. Average execution time ($\bar{T_p}$ seconds) and coefficient of variation ($\hat{v}$(%)) for different problem size (n) and number of computing resources (p, c)

$p\,(c)$	**problem size** (n)													
	8192		**10240**		**12288**		**14336**		**16384**		**18432**		**20480**	
	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)	$\overline{\mathbf{T_p}}$	$\hat{\mathbf{v}}$(%)
1 (1)	91.6	0.53	142	0.35	206	0.84	276	0.53	380	7.70	549	0.59	781	0.48
2 (1)	50.8	0.13	79.0	0.35	115	0.87	154	0.09	204	0.93	256	0.64	360	1.05
4 (1)	25.8	0.45	40.5	1.08	58.0	1.41	78.3	0.27	103	0.46	130	0.16	182	0.39
8 (1)	15.8	0.90	24.4	2.35	34.9	1.97	47.6	2.36	65.2	3.51	77.2	1.85	109	0.67
12 (1)	8.74	1.00	13.8	3.60	20.2	5.57	26.5	0.20	35.2	1.22	44.0	0.54	61.9	0.42
16 (1)	8.14	1.20	12.8	0.65	18.4	1.02	24.9	0.99	32.4	0.46	41.0	0.51	57.4	0.53
20 (1)	6.33	0.57	10.1	1.34	14.5	0.17	19.9	1.73	26.6	4.60	33.2	0.31	46.1	0.75
24 (2)	5.27	0.35	8.23	0.37	12.0	0.31	16.0	0.64	20.9	0.48	26.7	0.89	36.9	3.05
28 (2)	5.10	11.9	8.45	14.1	11.8	17.1	16.0	9.16	19.3	6.43	24.0	4.08	32.6	0.52
32 (1)	4.21	1.38	6.64	2.76	9.47	0.70	12.9	3.10	16.8	0.56	21.2	0.60	30.1	0.42
36 (1)	3.64	0.86	5.71	0.46	8.43	0.38	11.3	0.41	14.7	0.57	19.3	1.57	26.4	0.77
40 (2)	3.41	0.30	5.23	0.53	7.59	0.82	10.1	0.31	13.3	0.69	17.2	4.95	27.0	25.6
44 (2)	3.14	1.32	4.84	0.83	6.91	0.27	9.34	0.40	12.0	0.42	15.2	0.43	24.0	22.8
48 (2)	2.86	5.03	4.30	0.66	6.20	0.54	8.37	0.88	10.9	0.66	13.6	0.50	19.2	0.88
50 (3)	2.79	0.39	4.33	0.94	6.49	12.3	8.24	0.82	10.6	0.36	13.6	1.13	19.5	5.52
56 (2)	2.48	1.09	3.73	1.21	5.33	0.63	7.17	0.59	9.44	0.60	11.8	0.96	16.5	0.76
60 (3)	2.40	1.05	3.63	0.41	5.12	0.67	7.15	6.03	8.97	0.44	13.2	11.8	15.9	0.68
64 (2)	2.22	0.88	3.32	0.69	4.89	1.38	6.47	0.73	8.63	1.15	10.7	0.47	15.5	3.78
70 (2)	2.14	4.09	3.40	13.5	4.53	2.09	6.00	0.36	7.82	0.42	9.89	0.53	14.0	0.46

Table 2. Algorithmic speedup values for the execution times reported in Table 1

n	**computing resources** (p)																	
	2	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**	**36**	**40**	**44**	**48**	**50**	**56**	**60**	**64**	**70**
8192	1.8	3.6	5.8	10.5	11.2	14.5	17.4	18.0	21.8	25.2	26.9	29.2	32.0	32.8	37.0	38.1	41.3	42.7
10240	1.8	3.5	5.8	10.2	11.1	14.0	17.2	16.8	21.3	24.8	27.1	29.3	33.0	32.7	38.0	39.0	42.7	41.6
12288	1.8	3.6	5.9	10.2	11.2	14.2	17.2	17.4	21.8	24.5	27.2	29.9	33.3	31.8	38.7	40.3	42.2	45.6
14336	1.8	3.5	5.8	10.4	11.1	13.9	17.2	17.3	21.4	24.6	27.4	29.6	33.0	33.5	38.5	38.6	42.7	46.0
16384	1.9	3.7	5.8	10.8	11.7	14.3	18.2	19.6	22.6	25.8	28.5	31.6	34.9	35.7	40.2	42.3	44.0	48.6
18432	2.1	4.2	7.1	12.5	13.4	16.6	20.6	22.9	26.0	28.5	31.9	36.1	40.3	40.4	46.8	41.7	51.2	55.6
20480	2.2	4.3	7.2	12.6	13.6	16.9	21.2	23.9	25.9	29.5	28.9	32.5	40.6	40.0	47.4	49.1	50.4	55.6

The computational efficiency values ranged from 0.9 to 0.6. From $p = 16$ to $p = 70$, efficiency remained almost constant, in the order of 0.8 for the two largest problems, and 0.7 for the rest.

The best efficiency was 0.88, obtained for $n = 20480$, when using $p = 24$ in a two nodes environment.

A graphical analysis of efficiency values is presented in Fig. 6.

5.4 Scalability Analysis

A scalability analysis was performed to determine the capability of solving problems in rather similar execution times.

The setup for the analysis consisted in increasing the number of processing units p, together with the problem size n, whereas keeping constant the initial number of vertices per processing unit.

Table 3. Algorithmic efficiency for speedup values of Table 2 ($E_p = 100 \times S_p/p$)

n	computing resources (p)																	
	2	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**	**36**	**40**	**44**	**48**	**50**	**56**	**60**	**64**	**70**
8192	90.0	88.8	72.6	87.3	70.3	72.3	72.4	64.1	68.0	69.9	67.2	66.4	66.7	65.6	66.0	63.6	64.6	61.0
10240	89.5	87.5	72.5	85.3	69.3	69.9	71.7	59.8	66.7	68.8	67.7	66.5	68.6	65.4	67.9	65.0	66.7	59.4
12288	90.0	89.0	73.9	85.2	70.3	71.2	71.9	62.3	68.1	68.1	68.0	67.9	69.4	63.6	69.1	67.2	65.9	65.1
14336	89.5	88.2	72.5	86.8	69.4	69.6	71.8	61.7	67.0	68.2	68.6	67.2	68.8	67.0	68.8	64.3	66.7	65.7
16384	93.0	92.0	72.9	89.9	73.3	71.5	75.8	70.2	70.5	71.6	71.2	71.7	72.7	71.4	71.8	70.6	68.7	69.4
18432	107	106	88.9	104	83.8	82.8	85.6	81.7	81.1	79.1	79.8	82.0	83.9	80.8	83.5	69.5	80.0	79.4
20480	109	107	89.4	105	85.0	84.6	88.2	85.4	81.0	82.0	72.3	73.8	84.5	79.9	84.6	81.8	78.7	79.5

Table 4. Average execution time ($\bar{T}_p$ in seconds) and its coefficient of variation ($\hat{v}$(%)), for different number of computing resources (p, c), and problem size (n)

	problem size (n)							
	4096	**8192**	**10240**	**12288**	**16384**	**20480**	**24576**	**32768**
$p\,(c)$	1 (1)	4 (1)	6 (1)	9 (1)	16 (2)	25 (2)	36 (3)	64 (4)
$\overline{T_p}$	24.9	26.4	28.8	24.7	28.0	37.7	37.7	38.1
$\hat{v}$(%)	0.00	0.01	0.01	0.01	0.00	0.01	0.01	0.00

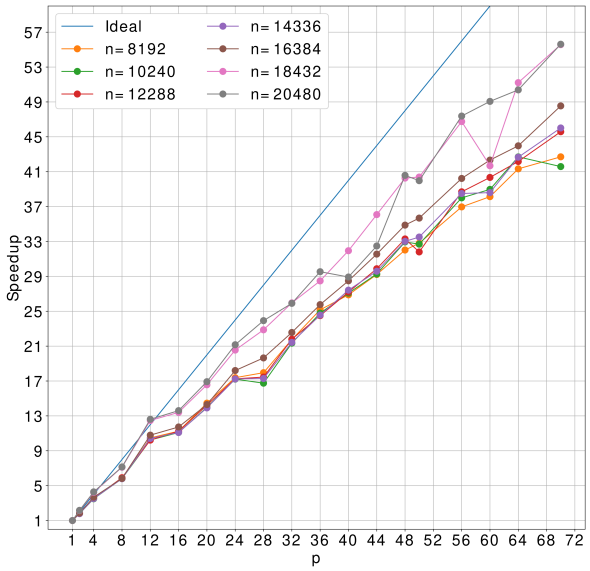

Fig. 5. Algorithmic speedup of the multigrid method with truncated V cycle ($N = 5$)

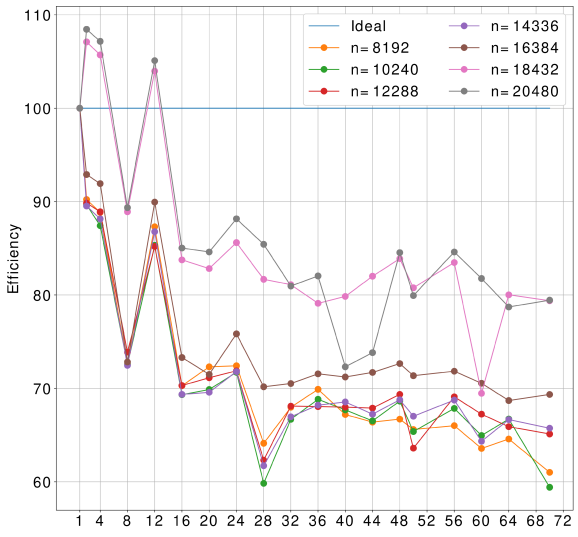

Fig. 6. Efficiency of the proposed multigrid algorithm with truncated V cycle ($N = 5$)

The value $n^2/p = 4096^2$ was chosen to have $p = 1$ for the smallest problem size, to fit in 1GB of RAM. Values of n are of the form $q \times 2^r$, $q \in \{1, 3\}$.

Table 4 reports the average execution times and coefficients of variation of the proposed multigrid implementation. Results demonstrate a good scalability behavior of the proposed implementation.

The execution time only increased 53% when the problem size n increased in a factor of 8

(from 4096 to 32768). Also, the execution time remained constant for n in 4096 to 12288 and 20480 to 32768.

6 Conclusions and Future Work

This article presented a truncating strategy for the V cycle of a parallel geometric multgird method. A theoretical analysis computed an explicit expression of the coarse grid level reached, and a bound for the number of vertices in the coarse grid, both in terms of the problem size and number of processing units.

The multigrid method with the proposed truncation criteria was implemented and evaluated in a distributed memory, non-dedicated cluster, for a Poisson problem. Accurate speedup and efficiency values were obtained for different problem sizes. Also, a weak scalability analysis revealed that problems of very different size may be solved in similar times by increasing the processing units.

The main lines for future work are related to analyze the performance with other coarse grid solvers, e.g., Conjugate Gradient or Chebyshev preconditioned with SSOR, which has fewer operations than SOR and CG. However, it uses two SOR steps per iteration, implying more communications. Thus, it may be interesting to analyze the computation-communication trade-off.

References

1. **Baker, A. H., Falgout, R. D., Kolev, T. V., Yang, U. M. (2012).** Scaling hypre's multigrid solvers to 100,000 cores. High-Performance Scientific Computing, Springer, pp. 261–279. DOI: 10.1007/978-1-4471-2437-5_13.
2. **Briggs, W. L., Henson, V. E., McCormick, S. F. (2000).** A multigrid tutorial. Society for Industrial and Applied Mathematics.
3. **Daley, C., Vanella, M., Dubey, A., Weide, K., Balaras, E. (2012).** Optimization of multigrid based elliptic solver for large scale simulations in the FLASH code. Concurrency and Computation: Practice and Experience, Vol. 24, No. 18, pp. 2346–2361. DOI: 10.1002/cpe.2821.
4. **Demmel, J. W. (1997).** Applied numerical linear algebra. Society for Industrial and Applied Mathematics.
5. **Dongarra, J., Heroux, M. A., Luszczek, P. (2015).** HPCG benchmark: a new metric for ranking high performance computing systems. Knoxville, Tennessee, pp. 42.
6. **Gradl, T., Rüde, U. (2008).** High performance multigrid on current large scale parallel computer. 9^{th} Workshop on Parallel Systems and Algorithm–workshop of the GI/ITG special interest groups PARS and PARVA, pp. 37–45.
7. **Guermond, J. L., Minev, P., Shen, J. (2006).** An overview of projection methods for incompressible flows. Computer Methods in Applied Mechanics and Engineering, Vol. 195, No. 44–47, pp. 6011–6045. DOI: 10.1016/j.cma.2005.10.010.
8. **Hackbusch, W., Trottenberg, U. (1982).** Multigrid methods: Proceedings of the conference held at köln-porz. Lecture Notes in Mathematics, Springer, Vol. 960 , pp. 23–27.
9. **Henshaw, W. D. (1994).** A fourth-order accurate method for the incompressible navier-stokes equations on overlapping grids. Journal of Computational Physics, Vol. 113, No. 1, pp. 13–25. DOI: 10.1006/jcph.1994.1114.
10. **Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U. (2006).** Parallel geometric multigrid. Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, Vol. 51, pp. 165–208. DOI: 10.1007/3-540-31619-1_5.
11. **Linden, J., Lonsdale, G., Ritzdorf, H., Schüller, A. (1994).** Scalability aspects of parallel multigrid. Future Generation Computer Systems, Vol. 10, No. 4, pp. 429–439. DOI: 10.1016/0167-739X(94)90007-8.
12. **Müller, E. H., Scheichl, R. (2014).** Massively parallel solvers for elliptic partial differential equations in numerical weather and climate

prediction. Quarterly Journal of the Royal Meteorological Society, Vol. 140, No. 685, pp. 2608–2624. DOI: 10.1002/qj.2327.

13. **Nesmachnow, S., Iturriaga, S. (2019).** Cluster-UY: Collaborative scientific high performance computing in Uruguay. International Conference on Supercomputing in Mexico, Springer, Vol. 1151, pp. 188–202. DOI: 10.1007/978-3-030-38043-4_16.
14. **Sterk, M., Trobec, R. (2003).** Parallel performances of a multigrid Poisson solver. Parallel and Distributed Computing, International Symposium on, pp. 238–238. DOI: 10.1109/ISPDC.2003.1267669.
15. **Strang, G. (2007).** Computational science and engineering. Wellesley-Cambridge Press.
16. **Trottenberg, U., Oosterlee, C., Schüller, A. (2001).** Multigrid. Academic Press, an Elsevier Science Imprint.
17. **Xie, D., Scott, L. (2009).** An analysis of parallel U-cycle multigrid method.

Article received on 02/05/2022; accepted on 03/10/2022. Corresponding author is Matías Valdés.