



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Computational Science and Engineering
(Int. Master's Program)

Master's Thesis

**Parallelization of a Multiscale CFD Solver within the
Peano Framework**

Daniel Butnaru





TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Computational Science and Engineering
(Int. Master's Program)

Master's Thesis

**Parallelisierung eines Multiscale CFD Lölers in der
Peano Framework**

**Parallelization of a Multiscale CFD Solver within the
Peano Framework**

1st examiner: Univ.-Prof. Dr. Hans-Joachim Bungartz
2nd examiner: Univ.-Prof. Dr. Thomas Huckle
Assistant advisor: Dr. Tobias Weinzierl
Thesis handed in on: 31.8.2009

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

31.8.2009

Datum

Daniel Butnaru

Acknowledgements

Several persons have contributed in one way or another to this thesis and I would like to explicitly thank them. First, I would like to thank Dr. Tobias Weinzierl for the constant support offered throughout the last couple of months. His guidance and suggestions were invaluable and allowed me to break the occasional barriers I encountered. Next, I would like to thank Dr. Miriam Mehl for creating a pleasant environment in which I could clearly focus on the challenges of the thesis. Eventhough they will not read this, I thank my family for understanding why lately I haven't called so often home. Finally, I want to thank my examiners Prof. Hans-Joachim Bungartz and Prof. Thomas Huckle for their efforts to examine this work.

Abstract

Modern computational fluid dynamics simulations are faced with an ever increasing need for computing power in fields such as computational steering or weather forecasts. A user performs a simulation where result data are of value only if they are delivered within a reasonable amount of time. In fields like turbulence research and analysis as well as multiphase flows, the amount of data generated by modern simulation codes has increased significantly. For all these important research areas, the computational fluid dynamic simulations are limited by the problem sizes which can be handled by single nodes due to their individual performance or memory available. These two reasons have made parallelization crucial for simulation codes.

The chair of Scientific Computing in Computer Science looks back to a long tradition in computational fluid dynamics. Recently, a new fluid dynamics solver has been established there. It is based on the general purpose partial differential equation (PDE) solver framework Peano which provides a set of state-of-the-art features for any PDE solver implemented upon it: dynamic h -adaptivity, low memory requirements, good cache behavior, multicore support. It offers support – utilities, algorithmic templates, out-of-the-box load balancing – for parallel PDE solvers due to a domain decomposition approach, too.

This thesis takes the existing fluid dynamics solver and makes it run in parallel within the Peano framework. For the regular Cartesian grid, a special case of adaptive Cartesian grids, a simple domain decomposition is implemented, data dependencies are studied and a communication scheme is set in place. While the decomposition and the communication scheme follow a standard text-book approach, the PDE-specific data dependencies were not analyzed before for the parallel case. Afterwards, the sequential fluid solver is adapted to the workflow of the parallel adaptive regular Cartesian grid. Here, the individual solver steps are analyzed, split up into individual substeps, and, realigned with the phases of the framework responsible for the domain decomposition. Afterwards, the overall workflow of the parallel simulation is set in place.

Finally, we place special emphasis on the modification of the solver for the pressure Poisson equation arising from the fluid equations: a Gauss-Seidel algorithm. This Gauss-Seidel is the basis of a multigrid solver for the CFD solver, and, due to the fact that most of the simulation's runtime is spent on this solver, an exact understanding of the parallel behavior of the solver is fundamental to make the overall application well-suited to tackle challenges not solvable before.

Several experiments show the parallel implementation offers parallel speedups for a balanced work distribution scheme. The balancing itself is up to the framework. The

combination of both features allows Peano's computational fluid dynamics solver to exploit nowadays massive parallel supercomputers. A section of this thesis furthermore shows how to tailor the parallel solver to the actual hardware such that the solver consequently does not only scale but also exploits the hardware efficiently. Hence, this thesis lays (part of) the foundations of a fluid dynamics solver that can tackle simulation tasks that are large enough to yield new scientific insight.

Zusammenfassung

Die Simulation von Strömungen erfordert heute immer mehr Rechenleistung für Anwendungen wie Computational Steering oder Wettervorhersagen. Simulationsergebnisse sind nur dann wertvoll, wenn sie in angemessener Zeit verfügbar sind. In Gebieten wie der Turbulenzsimulation oder für Multiphasen-Strömungen etwa nimmt die Menge an Simulationsdaten erheblich zu. In all diesen Forschungsgebieten sind Strömungssimulationen begrenzt durch die Größe der Szenarien, die von einzelnen Maschinen behandelt werden können. Das macht Parallelisierung entscheidend für Simulationsoftware.

Der Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen hat eine lange Tradition in der Strömungssimulation. Kürzlich wurde hier ein neuer Fluidlöser entwickelt. Er basiert auf dem Peano-Framework für partielle Differentialgleichungen und liefert folgende Eigenschaften die dem neuesten Stand der Technik entsprechen: dynamische h -Adaptivität, geringer Speicherverbrauch, Nutzung von Cacheoptimierungen, Support fuer Multicore-Architekturen. Die parallele Lösung partieller Differentialgleichungen gemäss eines Gebietszerlegungsansatzes unterstützt das Framework mit entsprechenden Hilfsroutinen, algorithmischen Grundstrukturen und vorgefertigte Lösungen zur Lastbalancierung.

Diese Arbeit parallelisiert einen existierende Strömungslöser im Rahmen dieses Peano-Frameworks. Für den Spezialfall regulärer kartesischer Gitter wurde eine einfache Gebietszerlegung implementiert, Datenabhängigkeiten wurden studiert und ein Kommunikationsschema entworfen. Während die Gebietszerlegung und das Kommunikationsschema bekannte Ansätze umsetzen, wurden die PDE-spezifischen Datenabhängigkeiten bisher für parallele Anwendungen nicht berücksichtigt. Der sequenzielle Strömungslöser wurde daraufhin an den Workflow des parallelen adaptiven kartesischen Gitters angepasst. Dabei wurden die einzelnen Schritte des Löser analysiert, in individuelle Teile zerlegt und an die Anforderungen des Frameworks angepasst. Schließlich wurde ein Gauß-Seidel-Algorithmus für die Lösung der aus den Fluidgleichungen resultierende Druckgleichung für die parallele Ausführung modifiziert. Ein Gauß-Seidel-Glätter bildet die Basis für Mehrgitterverfahren für den Strömungslöser und ist, weil zudem der Hauptteil der Rechenzeit in diesem Glätter verbraucht wird, von fundamentaler Bedeutung um größere Simulationsaufgaben angehen zu können.

Unterschiedliche Experimenten untersuchen den Speedup der die parallelen Implementierung für eine bereits balancierte Lastverteilung. Die Lastbalancierung selbst ist Aufgabe des Frameworks. Ein Unterkapitel der Arbeit zeigt, wie man den parallelen Löser an die verfügbare Hardware anpasst, so dass der Löser gleichzeitig skaliert und die Hardware effizient verwendet. In diesem Sinne legt diese Arbeit (zumindest zu einem guten Teil) die Grundlagen für einen Strömungslöser, der entsprechend große

Strömungsprobleme und daurch neue Fragestellungen angehen kann.

Contents

1	Introduction	13
2	The Peano Framework	15
2.1	Grid Structure	16
2.1.1	Regular Cartesian Grid	17
2.1.2	Adaptive Regular Cartesian Grid and k-Spacetrees	18
2.2	Traversal Strategies	20
2.2.1	Element-Wise Traversal and Information Transport	20
2.2.2	Traversal Events on the Regular Grid	21
2.2.3	Traversal Events on the Adaptive Grid	23
2.3	Parallel Computing in Peano	24
2.3.1	Master-Worker Paradigm	24
2.3.2	k-spacetree Decomposition	27
2.3.3	Parallel Level-Wise Depth-First Traversal	28
2.3.4	Partitions Induced by Space-Filling Curves	29
3	Computational Fluid Dynamics Solvers	31
3.1	Basic Equations	31
3.2	Phases of a CFD Solver	33
3.3	Degrees of Freedom Placement	33
3.4	Stencil Evaluation in a Cell-wise Traversal	34
3.5	Matrix-free Pressure Solver	35
4	Challenges	37
4.1	Regular Cartesian Grid	37
4.2	Adaptive Regular Cartesian Grid	39
4.3	Matrix-Free Poisson Equation Solver	40
5	Implementation	43
5.1	Regular Cartesian Grid	43
5.1.1	Domain Partitioning	43
5.1.2	Programing Model	44
5.1.3	Communication	45
5.1.4	Fluid Data Dependencies	46
5.1.5	Parallel Merge Phases	49

5.1.6	Global State Information	50
5.2	Adaptive Regular Cartesian Grid	50
5.2.1	Domain Partitioning	50
5.2.2	Programming Model	52
5.2.3	Communication Pattern and Phases	54
5.2.4	Fluid Data and Merge Operations	56
5.2.5	State Communication	60
5.3	Matrix Free Poisson Solver	62
6	Numerical Experiments	65
6.1	Message Size	65
6.2	Overlapping Communication with Computation	67
6.3	Strong Speedup	68
6.4	Weak Speedup	70
6.5	Gauss-Seidel vs. Blocked Gauss Seidel	74
7	Conclusions and Future Work	79
A	Parallel Adaptor	81
B	Border Traversal Algorithm for the Regular Cartesian Grid	87
C	Hardware	89

1 Introduction

Almost all simulation codes for partial differential equations are based on space discretization. To make a simulation solvable by a computer, only the solution in a finite number of points is considered. A discretized solution to continuous problems causes an inevitable loss of detail. However, new scientific insight benefits and relies on an increasing level of detail, i.e. on an increasing number of grid points. This facilitates a better representation of reality due to an increasing accuracy of the underlying models and numerical solutions. A higher level of detail means using finer and finer grids. This has two negative consequences on the efficiency of the simulation. First, the simulation produces more and more data while the practical maximum grid size is limited by the amount of memory available on the computational node. Second, with an increasing grid size and thus an increasing number of degrees of freedom also the execution time raises.

A common solution to tackle these limitations is parallel computing: make the application run concurrently on several computing nodes. This is achieved by identifying the computational intensive part of the application and dividing it among the available computing nodes. For computational fluid dynamics simulations (CFD) the method of domain decomposition is typically used, i.e. splitting the initial domain into overlapping or non-overlapping partitions. Each computing node is assigned such a partition on which it then performs the steps of the CFD solver.

In this thesis, we take an existing CFD-code [11] which runs in the Peano PDE framework [17] and enable it to run in parallel. We use domain partitioning with non-overlapping partitions. This approach requires the following considerations. First, how is the fluid solving process affected by the partitioning and, second, what needs to be done to obtain a consistent solution. Here, data dependencies between the fluid variables are to be considered. Third, the changes necessary to adapt the fluid solver to the specifics of the grid must be identified and addressed. Here, communication patterns and state synchronization are of primary concern.

The remainder is organized as follows. Chapter 2 presents the main features and concepts of the Peano framework. The available CFD solver is presented in Chapter 3. Here, the steps of the solver as well as the involved variables are explained. The challenges of running a parallel CFD simulation in Peano are presented in Chapter 4. The technical solutions to the previous challenges are detailed in Chapter 5, while

Chapter 6 shows performance results obtained with the parallel implementation. The thesis ends with a conclusion and an outlook.

2 The Peano Framework

This chapter introduces the Peano framework[17] and the main components it offers. The computational fluid dynamics solver considered for parallelization is running inside the Peano framework [12] and knowledge of the framework itself is necessary to understand the parallelization process. An overview of the framework's components is first given before presenting the two types of grids it supports for spatial discretization. The traversal strategies specific to each grid type are then presented and the chapter closes with the parallel computing support already available within the framework.

Developed at the chair for Scientific Computing at the Technische Universität München, Peano is a framework for grid-based solvers of partial differential equations (PDEs). It considers aspects which are attractive for PDE solvers such as time and spatial adaptivity, multiscale, arbitrary dimensions but also performance aspects such as multicore, hardware optimizations, fast algorithms, low memory footprint, and parallelization[12, 17].

Grid-based PDE solvers take the continuous computational domain and provide a discrete representation of the domain called a grid or mesh. For the solving process, this spatial discretization is used, and methods such as finite elements or finite differences break down the PDE into a discrete system of (linear) equations. There exists a zoo of (linear) system solvers which are then employed. Some use their own data structures to store and solve the equation system while others take advantage of the spatial discretization offered by the grid.

As a framework, Peano offers several important features needed to perform the solution process of a discretized PDE. First, a grid structure is provided on which information specific to the treated PDE is stored. Two types of grids are available: a **regular Cartesian grid** and an **adaptive Cartesian grid**. The adaptive grid offers criterion and scenario-based refinement or coarsening of the spatial discretization. Second, Peano provides access to the grid structure by offering traversals of the grid. An application would be aware of the traversal state and perform certain operations corresponding to a certain state during traversal. This knowledge of the grid structure and also of the state of the traversal is necessary for various processing steps such as pressure equation solving, visualization, and data processing.

Framework Overview

To access framework functionality or to integrate new applications in Peano, a modular approach is used. Each new application is a component in the framework and can be turned on, off, or used in combination with other components.

Figure 2.1 depicts a simplified structure of the current Peano framework. A break down into four major parts can be seen. Components referred to as the technical architecture **T-Arch** include XML configurations, runtime assertions, logging, and macros which are all used by the other components. The two grid implementations are the **trivialgrid** and **grid** components, while concrete applications such as the fluid solver **fluid** and the **poisson** equation solver use the two grid types as discretization support. The lowest level bundles together functionality needed by all PDEs like plotting, time integration, geometries for domain setup, and external solvers for linear systems of equations.

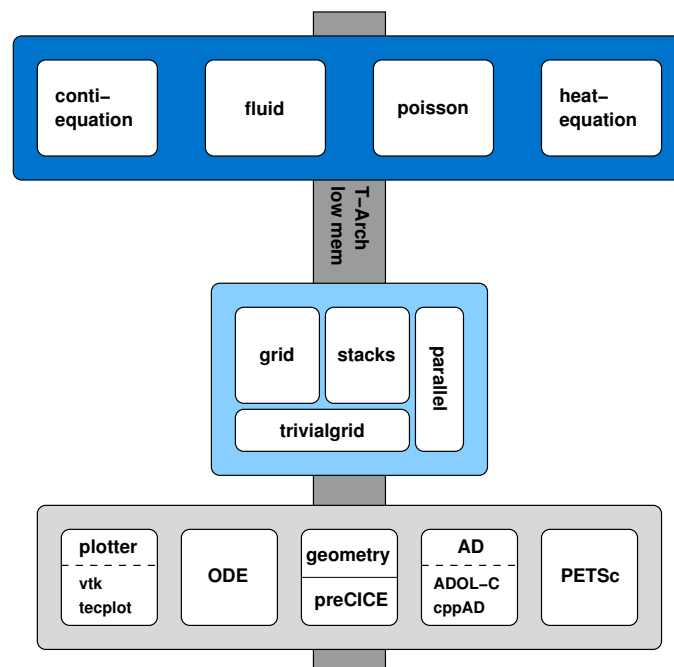


Figure 2.1: Components of the Peano Framework [17].

2.1 Grid Structure

A fundamental requirement for any PDE solver is a spatial discretization, i.e. a mapping from the real continuous domain to a discrete domain representation which can be handled by computers. This section presents the two types of spatial discretizations supported by Peano, and outlines their implementation and the interfaces offered to access them.

2.1.1 Regular Cartesian Grid

A **Cartesian** grid is a tessellation with congruent unit squares or unit cubes. Each cell in the grid can be addressed by index (i, j) in two dimensions or (i, j, k) in three dimensions, and each vertex has coordinates $(i \cdot dx, j \cdot dy)$ in 2D or $(i \cdot dx, j \cdot dy, k \cdot dz)$ in 3D for some real numbers dx , dy , and dz representing the grid spacing (mesh width). With respect to PDE solvers, this simple addressing is very well-suited for grid traversals, as spatial information can be derived from coordinates, i.e. there is no need to store spatial information within the cells or vertices.

Such **regular Cartesian grid** are used in Peano as a fast test bed for new applications where adaptivity is initially not important and the first step is to ensure the necessary data and the basic functionality is in place. As shown in Figure 2.2, the regular grid is constructed using a rectangular bounding box laid around the real geometry. The discretized geometry is implemented using cell markers. During the grid construction phase, based on the chosen mesh width, the vertices and cells are created and numbered. Several variations are supported regarding the mesh size. Depending on the PDE, different mesh sizes along different dimensions might lead to a better approximation. Mesh stretching is also implemented. Here we decrease the mesh width in one dimension when we approach the border of the domain.

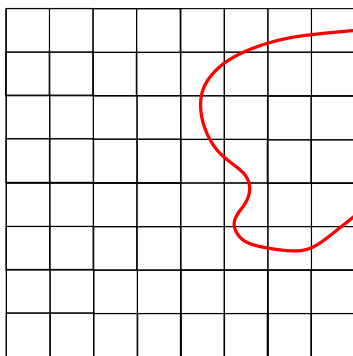


Figure 2.2: Discretization of a curved geometry using a regular Cartesian grid.

From the technical point of view, the regular cartesian grid follows a very straightforward implementation and does not take into account any performance aspects. For storage a one-dimensional array of cells and vertices is used which contains the degrees of freedom in lexicographical order. The memory overhead of grid-specific data in vertices and cells is minimal due to the strict structure of the grid, i.e. data can be accessed by array indices and no additional memory is used for spatial information.

2.1.2 Adaptive Regular Cartesian Grid and k-Spacetrees

The discretization of a domain affects the quality of the numerical solution due to the error introduced by discretizing the problem. To reduce the discretization error, a finer and finer grid can be used. However, this introduces even more geometrical elements (cells and vertices) which do not only require more memory but also increase the computing effort. If the discretization mesh size h of a regular Cartesian grid is reduced by a factor i , the number of unknowns N is increased depending on the dimension of the problem to $N \cdot d^i$. With an ever increasing number of degrees of freedom also the system of equations to be solved gets larger and requires more computing effort.

Depending on the PDE and a particular scenario, it is not necessary nor efficient to uniformly refine the entire mesh. This insight leads to the idea of using **adaptive grids**: only regions where the discretization error harms the numerical solution are refined. For fluid simulations, such regions are usually domain boundaries, obstacles, or fluid-structure interaction interfaces. While numerically desirable, adaptive grids bring with them the challenge of easily storing and accessing spatial data. For a regular grid, neighbor information is given by the simple construction. Cells and vertices are stored in arrays. With the help of multi-dimensional indices all elements can be easily addressed.

For an adaptive grid this is not that straightforward. There are a lot of methods to organize, store, and access spatial data in an efficient manner. One popular idea is to connect the adaptive structure of the grid to a tree representation and use the multitude of algorithms available for trees in order to efficiently store and access the grid. One example of such trees are **quadtrees** which are generated by bisecting cells in a recursive manner starting from a root cell. Figure 2.3d shows an example of such a quadtree while Figures 2.3a, 2.3b, and 2.3c show the grid levels represented by this quadtree. In this particular example, the decision to refine or not a cell is influenced by the presence of the curved geometry. This procedure is applied recursively until stopping criteria such as maximum tree level are fulfilled. Each decision to refine is reflected in the tree by adding four new children to the node representing the actual geometric element in the grid. One property of the quadtree is that it encodes several levels of refinement in a single tree and allows applications access to all these levels by simple traversals of the tree.

The quadtree is part of a the more general category of **k-spacetrees**. The k indicates the number of equidistant cuts applied per dimension. While quadtrees use bisection ($k = 2$), Peano uses $k = 3$, i.e each cube is cut into three pieces per dimension resulting in a total of 3^d subelements. It is important to highlight why a spacetree is well-suited for representing adaptive grids. The root of the spacetree corresponds to the largest geometric element that contains the whole computational domain. It is the starting point of the recursive grid construction. Each inner node in the tree matches a further refined grid cell, while the leaf nodes of the spacetree correspond to

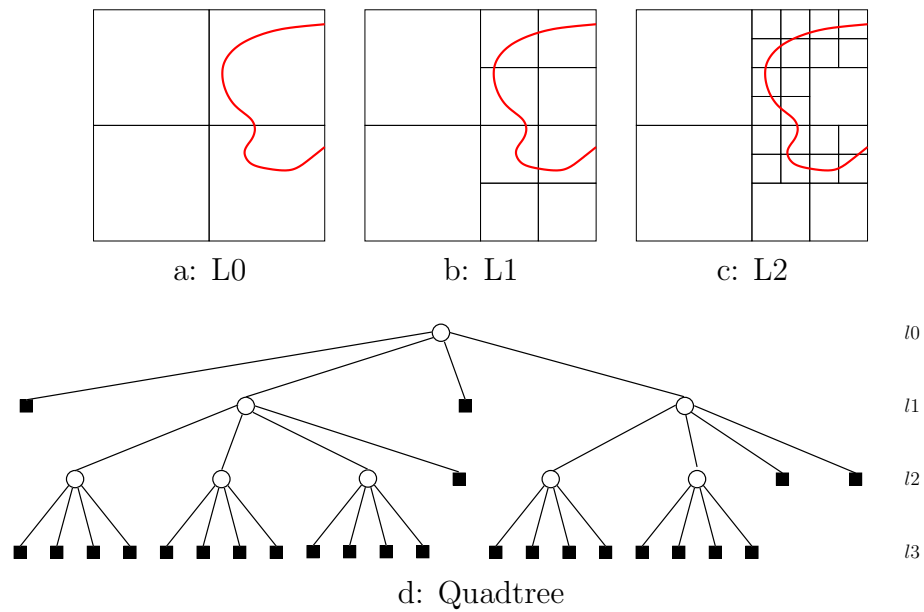


Figure 2.3: Example of an adaptively refined Cartesian Grid

cells in the actual grid, since they represent those cells of the grid that are not further refined. This means that each vertex that exists on level l_0 also exists on all sub-levels l_i down to the finest level and is also stored separately for each level. Similar for cells: geometric elements which are refined don't appear in the actual grid but are stored in the spacetime and will be traversed.

A k -spacetime explicitly preserves the hierarchy of adaptive levels, and the father-child relations cardinalities are fixed and invariant. The k -spacetime does not only encode the finest level of the grid but also all the intermediary levels up to the entire domain. This **multiscale** property facilitates many efficient and elegant algorithms. For example, geometric multigrid algorithms take advantage of this hierarchical encoding of the grid structure.

Hanging Nodes

One interesting question is what happens to the hanging nodes, i.e. those vertices lying on the border of cells with different refinement status. In Figure 2.4, an adaptive grid of a $k=2$ -spacetime is presented with the hanging nodes marked by squares. Such vertices are problematic for PDE solvers because operators are not defined on them. Instead, all neighboring nodes which are not hanging interpolate values to them in order to evaluate the operators[12].

As values assigned to hanging nodes are interpolated from values on the next higher level, Peano does not store such nodes but creates them whenever needed. After a traversal leaves a cell with hanging nodes they are discarded.

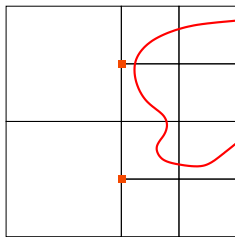


Figure 2.4: Adaptive grid with hanging nodes marked by squares.

2.2 Traversal Strategies

Every PDE needs access to the underlying grid in order to apply certain operators to cells or vertices. For that purpose, a grid traversal is to be provided. Traversing the grid depends on how the grid is structured and stored in memory. This section presents traversal strategies for the two types of grids in Peano and how an application can use the traversals.

2.2.1 Element-Wise Traversal and Information Transport

The grid consists of vertices and elements arranged in some data structure. Any application performing any operation on the grid would have to read each vertex or cell at least once. At least two traversal approaches are possible. A vertex-wise traversal would visit each vertex once and with each visit the cells sharing the vertex would also be available. The second approach, which is also the traversal method chosen in Peano, is **element-wise**. Such a traversal defines a total order on the geometric elements (cells) and runs through this data stream. Whenever the traversal reaches one element, access is provided to all the adjacent vertices. Each geometric element is visited only once, while vertices are visited multiple times corresponding to the number of elements that share them¹.

Both traversal types have applications for which one is better suited than the other. Visualization software prefers the vertex-wise traversal while finite element discretizations are more efficient on the second. The grid structure in Peano is implemented as a k -spacetree. Each node corresponds to a geometric element, hence it makes more sense to use an element-wise traversal as it can be implemented as a direct tree traversal. This opens the possibility off using standard tree algorithms like *depth-first* search on the adaptive grid. The cell-wise traversal implies also visiting each vertex at least once and can, when needed, easily accommodate the needs of visualization software.

¹ 2^d for non-hanging vertices

According to [17], the element-wise run on the grid is tightly connected to the speed the information propagates along the grid. In any given geometric element, the data sitting on the cell and the corresponding 2^d vertices are the only information sources available to an operator. No access to the neighboring cells is possible and the only (indirect) link to such cells are the vertices. Consider the traversal presented in Figure 2.5:

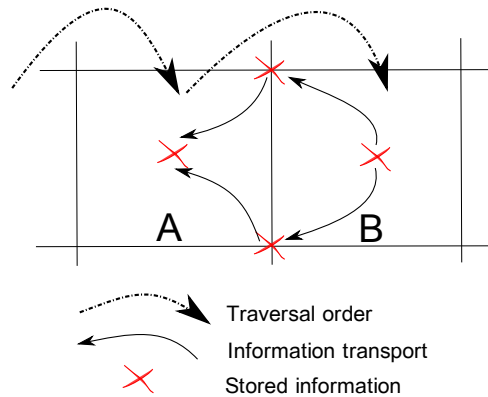


Figure 2.5: Information transport in a cell-based grid run

Cell A is visited before cell B. If information is required from B to update A, this information is not available: B has to write any information its neighbors need in the vertices. This information is then reconstructed by A from the vertices in the next traversal.

2.2.2 Traversal Events on the Regular Grid

Two predominant patterns for grid traversals are iterator and **event-based** traversals. The iterator pattern is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation. An application would receive from an iterator the next object without knowing what relations it has to previous or next objects. The advantage of this method is that the application explicitly controls the traversal.

Peano follows the **event-based** traversal pattern. During the traversal of the grid events such as *enter-cell* or *exit-cell* are fired. Unlike an iterator, the event-based traversal gives an application more information by firing events with different semantics. The same vertex can be returned twice but by different events. Unlike the iterator, the application is event-driven and cannot itself guide the traversal. The event-based traversal is well combined with the **adapter** concept [4]. This requires that any operator acting on the grid must abide to a given interface and act on the calls (events) to that interface. The grid acts as the adaptee by calling methods on the adapter. Various calls correspond to various locations and states in a grid traversal and also provide

certain local information to the adapter. With this information and with respect to the purpose of the adapter, changes to the application data residing on the grid can be performed. In Peano this concept is realized via static inheritance (templates) and allows the decoupling of the grid from the application and flexibility in choosing and switching among adapters (plugin mechanism).

Event	Description
<code>createDegreeOfFreedom()</code>	The method receives a vertex. The specific adapter initializes this vertex and returns it. Information about position and mesh size are also passed.
<code>createDegreeOfFreedom()</code>	The method receives a geometric element. The specific adapter initializes the element and returns it.
<code>isElementOutsideDomain()</code>	This method decides if the point identified by the passed position and surrounding environment is outside the computational domain or not.
<code>isElementInsideDomain()</code>	This method decides if the point identified by the passed position and surrounding environment is inside the computational domain or not.
<code>refine()</code>	This method decides if the point identified by the passed position should be refined.

Table 2.1: Grid creation and management events.

A grid traversal calls the methods in the adapter interface at the right time and passes the appropriate arguments. By their semantics, these events indicate a certain state in the traversal. Each PDE can then use this state information and perform specific actions. One class of events are grid creation and management events. They are called when the grid is initially constructed and allow adapters to intervene in this phase. They are listed together with their description in table 2.1.

One typical example of adapters using these events are the scenario adapters. Whenever a degree of freedom is created, they decide if the cell or vertex lies inside of the domain of the specific scenario and flag it appropriately. Furthermore, boundary conditions are set. The grid creation events are also heavily used by adapters who perform grid modifications (refinement or coarsening) where grid elements are created or destroyed.

The next class of events are the traversal events. On the regular Cartesian grid the cells are given a fixed number at grid creation. The numbering scheme respects a lexicographic order and this order also gives the traversal succession. As depicted in Figure 2.6 in 2D the typical run starts in the lower-left corner and proceeds left-to-right bottom-to-top to the top-right corner. Adapters running over the regular grid receive the events listed in Table 2.2.

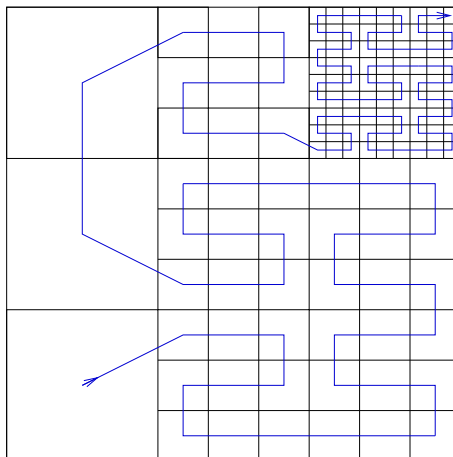


Figure 2.7: Example of a 2D Cartesian grid and its corresponding discrete Peano curve.

besides the ones for regular grids are fired. The complete list of events available on the adaptive grid are presented in Table 2.3. The Peano curve defines a depth-first level-wise traversal of the spacetree, i.e. before moving to a neighbor cell all the subcells of the current cell are traversed [17]. The Peano curve visits each cell exactly once. Such access to cell data is straightforward. For accessing vertex data a mechanism consisting of several stacks is used. Each vertex is pushed on an intermediary stack once and pop-ed only after all the surrounding geometric elements have been visited. Due to the nature of the space-filling curve traversal these intermediary stacks are guaranteed to be small and fit well in cache memory providing a good cache-efficiency to the traversal algorithm[17].

2.3 Parallel Computing in Peano

A better approximation of the solution of a discretized PDE is obtained by using a finer grid. This, however, causes an increase in the computational effort. Using an adaptive grid where the entire domain is not uniformly refined reduces the number of degrees of freedom and thus computational load. But as the problems get larger and the meshes finer, the need for parallelization arises. In Peano, this aspect is also considered and parallel computing is supported for the adaptive regular grid as a framework component. The next subsections will deal with parallel aspects in Peano.

2.3.1 Master-Worker Paradigm

There are three main factors which decisively influence the parallel efficiency of any algorithm. The sequential fraction is that part of the algorithm which cannot be

Event	Description
<code>beginTraversal()</code>	Start of an iteration.
<code>endTraversal()</code>	End of an iteration.
<code>enterElement()</code>	All necessary vertices for a cell are loaded.
<code>leaveElement()</code>	Counterpart to <code>enterElement()</code> .
<code>touchVertexFirstTime()</code>	The first time any cell next to this vertex is visited. The position, level and corresponding mesh width are passed.
<code>touchVertexLastTime()</code>	The last cell next to this vertex has been visited. At this step all geometric elements of this vertex have been visited.
<code>loadSubElement()</code>	Fired before going to the next finer level (top-bottom transition).
<code>storeSubElement()</code>	Fired before going to the next coarser level (bottom-top transition). This operation and its counterpart facilitate an interlevel information exchange.
<code>startStepsUp()</code>	Fired after the subelements have been processed and before the traversal will step up the tree.
<code>startStepDown()</code>	Fired right before the subelements are going to be entered.
<code>createPersistentVertex()</code>	Used to create new vertices when refining an existent vertex.
<code>destroyPersistentVertex()</code>	Used to destroy vertices following a coarsening.
<code>createTemporaryVertex()</code>	Used to create hanging nodes (touch first on hanging nodes).
<code>destroyTemporaryVertex()</code>	Used to destroy hanging nodes.

Table 2.3: Adaptive regular Cartesian grid management and traversal events.

parallelized and must be executed in serial by all nodes. The larger that fraction the less effective are the parallelization attempts. The parallel overhead is the amount of communication needed to maintain the progress of an algorithm consistent and correct. The third important factor is load balancing. A good algorithm must deal with all of these but the last factor can be that difficult that it dominates the design. Load balancing is also a problem for parallel PDE solvers and a few common characteristics can be identified in this direction:

- The workload associated with the tasks is highly variable in size and in occurrence. If their workload would be known, a static distribution on the available nodes could be used.
- The program structure of the computationally intensive portions of the application cannot be mapped into simple loops. This pushes the parallelization more and more to outer code blocks.
- The capabilities of the available nodes for the parallel computation vary across the parallel system, change over the course of the computation or are unpredictable.

The **master-worker** pattern is a good solution to the above mentioned problems. This pattern is a core aspect of the support for parallelization offered and also the guiding rule in running PDE simulations in parallel using Peano. This pattern is sketched in Figure 2.8. The solution consists of two logical elements: a master and one or more instances of workers. The master initiates the computation and sets up the problem. It then creates a bag of tasks. The master then initializes the workers and gives them a task to work on. It then either sleeps or works itself on a task. When the workers are done processing their jobs, the master collects the results and then either shuts down the computation or creates another bag of tasks and the steps are repeated.

In the classic pattern each worker enters a loop. At the top of the loop, the worker takes a task from the bag of tasks, does the indicated work, tests for completion, and then goes on to fetch another task. This continues until the termination condition is met, at which time the master wakes up, collects the results and finishes the computation. As long as the number of tasks greatly exceeds the number of workers and the cost of the individual tasks are not so variable that some workers take drastically longer than others, the master-worker algorithm has good scalability.

The textbook [10] variant described above is extended by introducing a recursive aspect of task distribution. Workers become themselves masters for other workers. Any worker also create its own bag of tasks by splitting the task it received from its master. By inspecting the pool of available nodes it then can hire workers to compute its sub-tasks and this way becomes a master itself. This has an advantage of a more simple work distribution pattern where no global accounting is performed and the responsibility for breaking the tasks into subtasks and distributing them is delegated to the workers.

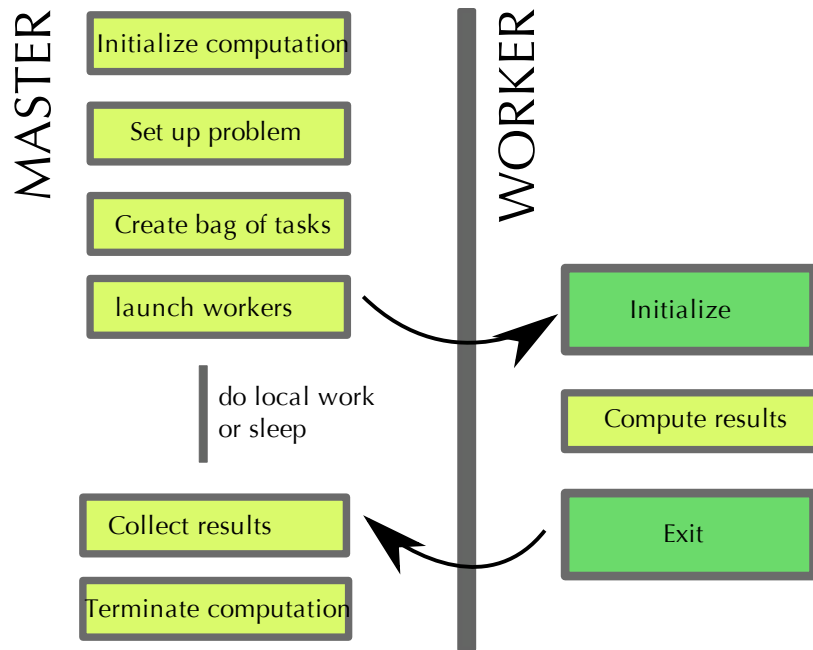


Figure 2.8: Master-worker design pattern.

In Peano the tasks each worker receives are **domain partitions**. Each worker is thus responsible for performing the PDE solving on a portion of the entire domain which it can refine or coarse in the process. During the refinement process a worker can decide to delegate itself a part of the local domain to idle workers. This allows for a simple recursive grid partitioning process.

One challenge of the master-worker pattern is to correctly detect when the computation is completed. If a bag of tasks is used then an empty bag is a termination condition. Another solution, which is used also in Peano, is to send a special termination message to mark termination. The workers react to this message by executing one last iteration, but do not send any data to the master at the end. When using a recursive work distribution, this execution termination message must also reach all the workers in the master-worker tree hierarchy.

2.3.2 k-spacetre Decomposition

The parallelization approach used for the adaptive regular grid in Peano is a domain decomposition approach. In such an approach, the global domain is split into several partitions and each computing node is assigned to a single partition. This way, each node only has to traverse its own local domain. Partition boundary operators and the exchange of the subdomain boundaries ensure that the parallel code preserves the sequential semantics of the algorithm. For all the nodes the start and end of the traversal act as synchronization points to ensure a consistent progress throughout the simulation.

The spacetree is the underlying representation of the domain, thus a logical domain decomposition is concretely implemented as a spacetree decomposition. A coloring scheme is used to indicate this decomposition. Each node has a number assigned, namely its rank. Each element in the spacetree gets assigned an attribute holding this number to indicate which node is responsible with the computations executed on it. The coloring of the spacetree considers aspects such as load balancing and available nodes. An example coloring is given in Figure 2.9. Three ranks are used to distribute the entire tree. Rank 0 starts by taking the top most node and distributes ranks one and three to other workers. As rank 0 is also responsible for administrative work it allocates itself the smaller branch.

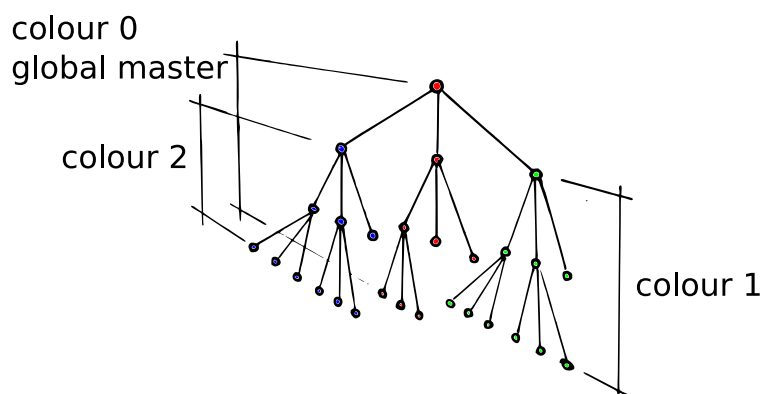


Figure 2.9: Master-worker Design Pattern

The coloring introduces a tree topology also on the involved computing nodes. This means that the adaptive grid can be distributed: each node is responsible for a subtree of the overall k -spacetree. It can happen that subparts of this local k -spacetree are colored with a different rank. In this case, a rank becomes master for another color. The recursive Master/Worker pattern described in the above subsection is applied here.

2.3.3 Parallel Level-Wise Depth-First Traversal

The domain decomposition implies a parallel traversal of the spacetree. Figure 2.10 illustrates the traversal strategy. The traversal is started by the global master. All other nodes reside in a node pool and must wait until a startup message is received. The traversal algorithm then loads k^d sub-elements in each recursion step. The elements are then checked to see if any of them are assigned to other ranks. If this is the case, the corresponding ranks receive a startup message and start iterate on their subtree. While waiting for the workers to finish, the global master can work on the elements assigned to itself.

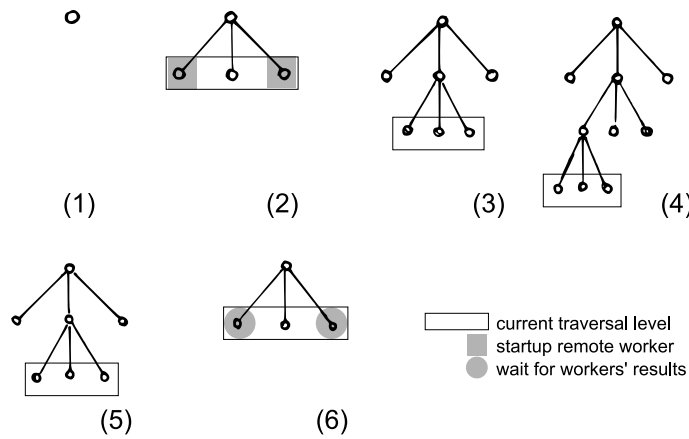


Figure 2.10: Parallel level-wise depth-first traversal. (1) The rank 0 node starts the traversal. (2) k^d sub-elements are loaded. If one sub-element is remote, the responsible remote node will be started. (3) Rank 0 then continues by descending in the local subelements. In (5) the traversal steps up the tree and in (6) the master node must wait for the workers to finish before collecting the results.

In the bottom-up phase of the depth-first traversal, the workers pass the results of their computation to the master. If the master is itself a worker to another rank, the results are recursively passed to the top.

2.3.4 Partitions Induced by Space-Filling Curves

When using domain decomposition, values of degrees of freedom at the borders of the subdomain must be exchanged with neighbor partitions. The longer the domain boundaries the more data have to be communicated. This implies an optimization problem with respect to partition size and shape. A good technique is to blend communication with computation. The computations effort is given by the volume of the partitions while the communication overhead by the surface. In this respect, good partitioning yields a big volume-surface ratio[17].

In Peano, the grid traversal runs through the individual spacetree levels following the Peano curve. It turns out that partitioning the Peano curve also creates quasi-optimal partitions of the computational domain[17]. This makes the Peano curve very attractive as the partitioning comes almost for free. Some resulting partitions of a two-dimensional domain are presented in Figure 2.11.

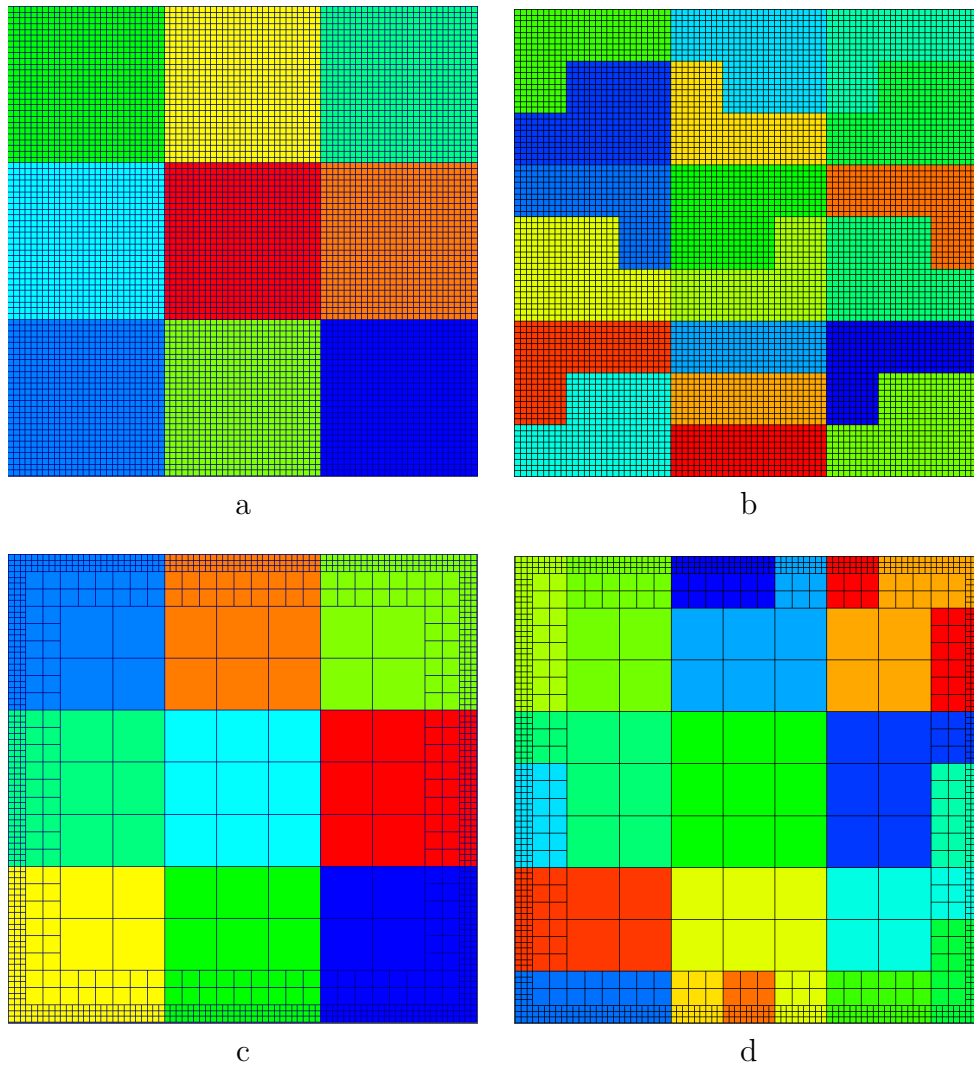


Figure 2.11: Domain partitioning along the Peano curve. (a) and (b): regular grid with 10 nodes respectively 20 nodes. (c) and (d): adaptive grid with 10 nodes respectively 20 nodes.

3 Computational Fluid Dynamics Solvers

This thesis takes an existing computational fluid dynamics (CFD) solver from [12] which already runs using the Peano framework and extends it to be able to use several computing nodes concurrently. Any parallelization attempt starts by looking first at the algorithm to be parallelized. Of interest here are the steps in algorithms and the data dependencies between these steps.

This chapter is a very brief overview of the fluid solver available in Peano and presents the main steps of a fluid dynamics simulation. It serves as a context into which the parallelization efforts are considered, explained, and evaluated in the following chapters.

3.1 Basic Equations

As mentioned, one key application which takes advantage of the Peano framework is an incompressible fluid solver. It is implemented in Peano as a component and uses the regular and adaptive Cartesian grids to perform fluid simulations. The flow of a fluid in a domain $\Omega \in \mathbb{R}^N$ ($N \in \{2, 3\}$) over the time $t \in [0, t_{end}]$ is characterized by the following variables:

- $\mathbf{u} : \Omega \times [0, t_{end}] \longrightarrow \mathbb{R}^N$ velocity field, and
- $p : \Omega \times [0, t_{end}] \longrightarrow \mathbb{R}^N$ pressure.

The Navier-Stokes equations describe incompressible fluids and have the expression [9]:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} - \frac{1}{\nu} \Delta \mathbf{u} + \nabla p = \mathbf{0}, \text{ and} \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{3.2}$$

The first vector equations are the momentum equations equivalent to a force balance, and the second scalar equation is the continuity equation which represents the incompressibility property. The goal is to find velocity \mathbf{u} and pressure p functions for all time and space positions:

$$\mathbf{u} = \mathbf{u}(t, x), \quad (3.3)$$

$$p = p(t, x). \quad (3.4)$$

The peano fluid solver **discretizes the space** with a finite element approach for the velocities. This results in the following set of semi-discrete Navier-Stokes equations:

$$A\dot{\mathbf{u}}_h + \underbrace{C(\mathbf{u}_h)\mathbf{u}_h}_{=:F} + \underbrace{D\mathbf{u}_h}_{=:gradP} + M^T p_h = \mathbf{0} \in \mathbb{R}^N, \quad (3.5)$$

$$M\mathbf{u}_h = \mathbf{0} \in \mathbb{R}^Z. \quad (3.6)$$

A, C, D, M are represented as global matrices applied to global vectors $\mathbf{u}_h \in \mathbb{R}^N$ and $p_h \in \mathbb{R}^Z$. Finite element theory is not necessary for understanding the parallelization efforts. Refer to [7] for more detailed information on this method. Using the (continuous) time-derivative of the semi-discrete continuity equations and inserting the semi-discrete momentum equations gives:

$$M\dot{\mathbf{u}}_h = \mathbf{0} \Rightarrow \mathbf{0} = M\dot{\mathbf{u}}_h = M \left(A^{-1} \left[-C(\mathbf{u}_h)\mathbf{u}_h - D\mathbf{u}_h - M^T p_h \right] \right) \quad (3.7)$$

$$\Leftrightarrow MA^{-1}M^T p_h = MA^{-1} \underbrace{\left[-C(\mathbf{u}_h)\mathbf{u}_h - D\mathbf{u}_h \right]}_{=: -F} \quad (\text{PPE}). \quad (3.8)$$

The second equation is called Pressure Poisson Equation (PPE) because the discrete operator on the left hand side is a Laplacian. The PPE replaces the continuity equation in the discretization scheme and may be used in order to compute a new pressure field out of given velocities. This new pressure is then responsible for making the velocity update discretely divergence-free. This equation is important as it represents the most computationally intensive part of the simulation in the case of an explicit time discretization. To solve it external libraries can be used which explicitly set up the system matrix before solving¹. Another solving option is the matrix-free solver introduced later in this chapter.

For the **discretization in time**, Peano uses finite differences for the time derivative of \mathbf{u}_h . The most straight-forward method is the explicit Euler method:

$$\frac{d\mathbf{u}_h}{dt}(t_0) \approx \frac{\mathbf{u}_h(t_0 + \tau) - \mathbf{u}_h(t_0)}{\tau}, \quad \tau \ll 1.$$

The time advancement is, then, a simple update of the velocities using the discrete momentum equations.

¹In Peano PETSc is used: <http://www.mcs.anl.gov/petsc/petsc-as/>

3.2 Phases of a CFD Solver

The Euler time stepping method is implemented as a time loop inside which new pressure and velocities are calculated. The main phases of the fluid solver are sketched in Algorithm 3.1:

Algorithm 3.1 Fluid solver algorithm as implemented in Peano.

```
prepareGrid()
while  $t < t_{end}$  do
  calculateF()
  assemblePPERHS()
  solveSystem()
  pressureReassembleAndGradientAndForces()
  plotOutput()
  updateVelocities()
end while
```

The algorithm starts by constructing the grid. This step is specific to the underlying grid used. Next, the F as specified in Equation 3.8 is accumulated for each vertex. The following traversal uses the accumulated F and computes the right-hand side of the pressure Poisson equation. With everything in Equation 3.8 available, the system of linear equations can be solved with a variety of solvers. Afterwards, the pressure solution is collected and the velocities are then calculated (Equation 3.7).

3.3 Degrees of Freedom Placement

For the implementation of Algorithm 3.1 several variables need to be stored. The velocity vector field \mathbf{u} , the pressure field p , and F data are variables needed during the simulation. The grid offers the possibility to store variables at the location of the vertices or in the cells which correspond to the center of the geometric element. Depending on the placement of variables, two grid types can be mentioned: collocated or staggered.

On a staggered grid, the scalar variables (pressure) are stored in the cell centers of the control volumes, whereas the velocity or momentum variables are located at the cell faces. This is different from a collocated grid arrangement, where all variables are stored in the same positions. A staggered storage is mainly used on structured grids for incompressible flow simulations. Using a staggered grid has some numerical advantages as it avoids odd-even decoupling between pressure and velocity [13]. Odd-even decoupling is a discretization error that can occur on collocated grids and which leads to checkerboard patterns in the solution.

Figure 3.1 shows where fluid specific variables are placed in Peano. The velocities are stored and located in the vertices as well as F and force data. The pressure lies

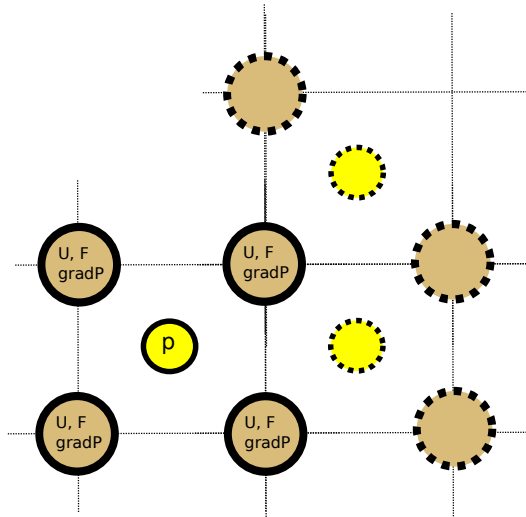


Figure 3.1: Location of fluid variables on the grid.

in the middle of the cells. Pressure values in the form of gradients are also stored in the vertices. These are needed in order to update the velocities by pulling the pressure from the middle of the cells to the vertices (where velocities lie). Beside numerical aspects, the positioning of the degrees of freedom on the grid also has implications on parallelization efforts. It determines what gets communicated and how the global values get reconstructed in order to maintain correctness of the computations.

3.4 Stencil Evaluation in a Cell-wise Traversal

The implementation of the finite element method for the fluid solver is adapted to the Peano framework. The degrees of freedom are located only in vertices and cells and access to them is only possible during a cell-wise grid traversal. This means the fluid solver must apply the operations described in Equations 3.7 and 3.8, having at any time during the traversal access to only one cell and its adjacent vertices. When considering values sitting in vertices, the nodal way of accumulating contributions is to apply a certain discretization dependent stencil to the neighboring nodes.

Figure 3.2 left illustrates this. In a strictly cell-wise traversal strategy the neighboring nodes are not available and the accumulation of the corresponding data on a node has to be done via neighboring cells. Figure 3.2 right shows the breakdown of the original stencil into partial stencils which will be applied consequently during the traversal. The same approach concerns stencils applied on cell values such as the pressure.

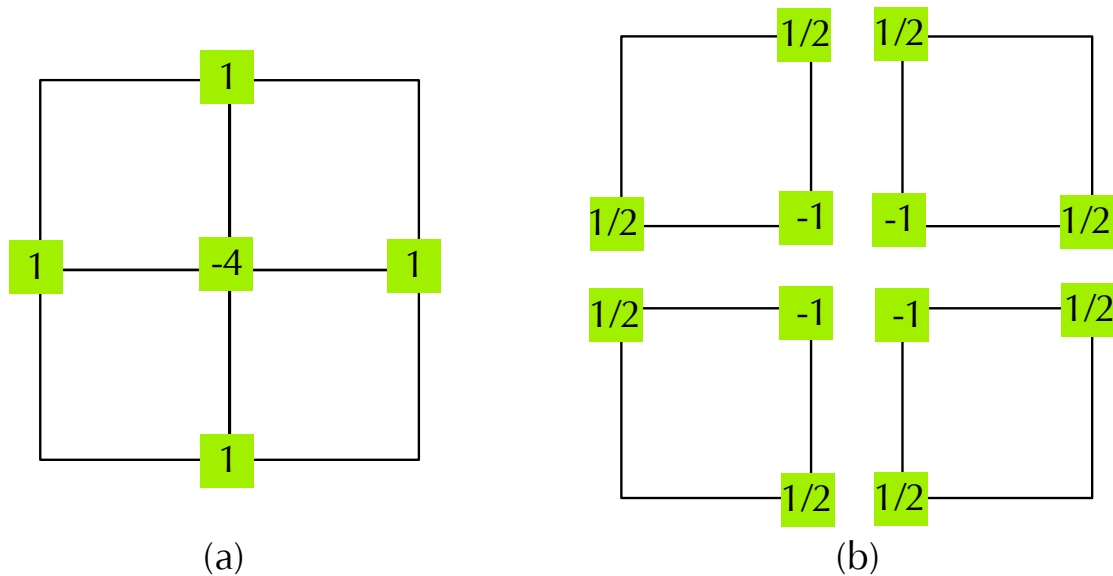


Figure 3.2: Operator evaluation for a nodal stencil. The usual nodal evaluation presented in (a) is not possible in Peano. Identical results are obtained by breaking down the stencil into cell operators (b).

3.5 Matrix-free Pressure Solver

One step of the fluid simulation is to solve the pressure Poisson equation. One way to do this in Peano is use external libraries such as PETSc. PETSc offers a multitude of linear solvers combined with a variety of preconditioners. In solving the *PPE*, PETSc explicitly sets up the global matrix $MA^{-1}M^T$. This has impact on the memory overhead of the entire simulation and also on parallelization. Another option is to use a matrix-free solver which does not assemble global matrices and can be implemented as a typical grid traversal. Peano currently offers an implementation of a Gauss-Seidel matrix-free solver.

Gauss-Seidel works by taking pressure in a cell and applying a stencil which updates the pressure in the considered cell. In this stencil, also pressure values from neighboring cells are involved. When assembling a global pressure matrix, these value are directly available but with a matrix-free approach they have to be obtained indirectly.

The matrix-free solver makes use of the vertices to store pressure gradients. These gradients are then used to reconstruct pressure values in the neighbouring cells. Figure 3.3 shows the data placement for the matrix-free solver. The solving process can than be summarized as follows:

- one iteration of the solver is implemented in one grid traversal
- upon entering a cell the skew five point stencil is applied

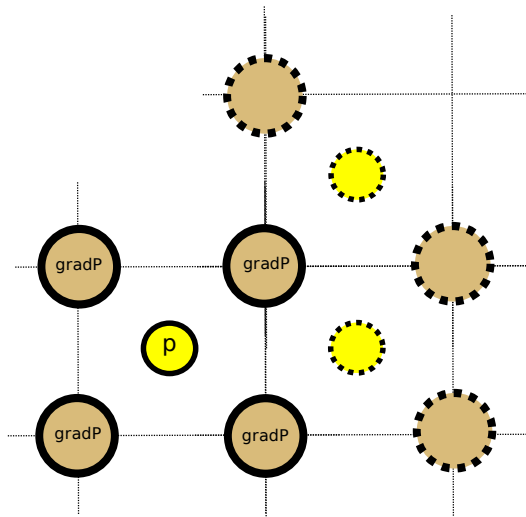


Figure 3.3: Location of matrix-free solvers's variables on the grid.

- the pressure values of neighbouring cells are reconstructed from the gradients sitting in the vertices
- after the pressure in the middle of the cell is updated, also the gradients are updated to reflect the change.
- the next cell will use updated neighbour gradients

4 Challenges

This thesis makes the CFD solver presented in the previous chapter run in parallel. For this, both a parallel implementation for the regular Cartesian grid but also an adaptation of the CFD solver to the already parallel adaptive Cartesian grid are necessary. Additionally, in preparation for a parallel multigrid solver for the pressure equation, a parallel version of a Gauss-Seidel solver was implemented to solve the pressure poisson equation.

This chapter highlights some of the challenges associated to parallel fluid dynamics in Peano using the two types of available grids: regular Cartesian and adaptive Cartesian. It starts with domain decomposition for the regular grid and presents some issues that must be solved to accommodate it. It then switches to the adaptive Cartesian grid and the issues caused by the shifted communication pattern in the master-worker concept. The last part of the chapter deals with specifics of the implementation of the parallel matrix-free solver.

4.1 Regular Cartesian Grid

The predominant approach to parallelize CFD solvers is to use domain decomposition. The computational domain is split into several overlapping or non-overlapping parts, and these are then distributed among the computing nodes. The problems divided this way are smaller and can then be solved faster. Furthermore, the application uses the combined available memory of several computing nodes.

The regular Cartesian grid of [17] lacks a parallel implementation. Consequently, the parallel design has to be done from scratch. The first task is to decide which partitioning strategy is to be used. This needs to consider aspects such as grid management, load balancing, simplicity to implement, and communication costs. As soon as the partitioning related aspects are in place, the next task is to consider how such a partitioning influences the fluid solver and how the parallel borders (between subdomains) are to be treated to maintain the solving process correctness. Here, the phases of the fluid solver presented in Chapter 3 must be analyzed to find out which variables (A , $gradP$, F , U , etc.) need to be exchanged with neighbors and which can be locally calculated.

Equation 4.1 recapitulates the semi-discrete Navier-Stokes with the definition of the two helper variables used by the fluid solver: F and $gradP$. They aggregate velocity and pressure values, respectively, and are used in the right-hand side of the pressure poisson equation (F) or in updating the velocities ($gradP$).

$$A\dot{\mathbf{u}}_h + \underbrace{C(\mathbf{u}_h)\mathbf{u}_h + D\mathbf{u}_h}_{=:F} + \underbrace{M^T p_h}_{=:gradP} = \mathbf{0} \in \mathbb{R}^N \quad (4.1)$$

Lets consider how F is accumulated during a grid traversal. Figure 4.1 (a) illustrates the F accumulation for the central vertex in the case of a sequential traversal of the grid. Due to the cell-wise traversal used in Peano, only cell accumulations are possible: when entering a cell, the velocity values sitting in the vertices of the cell are weighted and added to the F values sitting also in vertices. The traversal follows the space-filling curve and visits each cell at most once. At the end of the traversal, the F value for the middle vertex is fully accumulated.

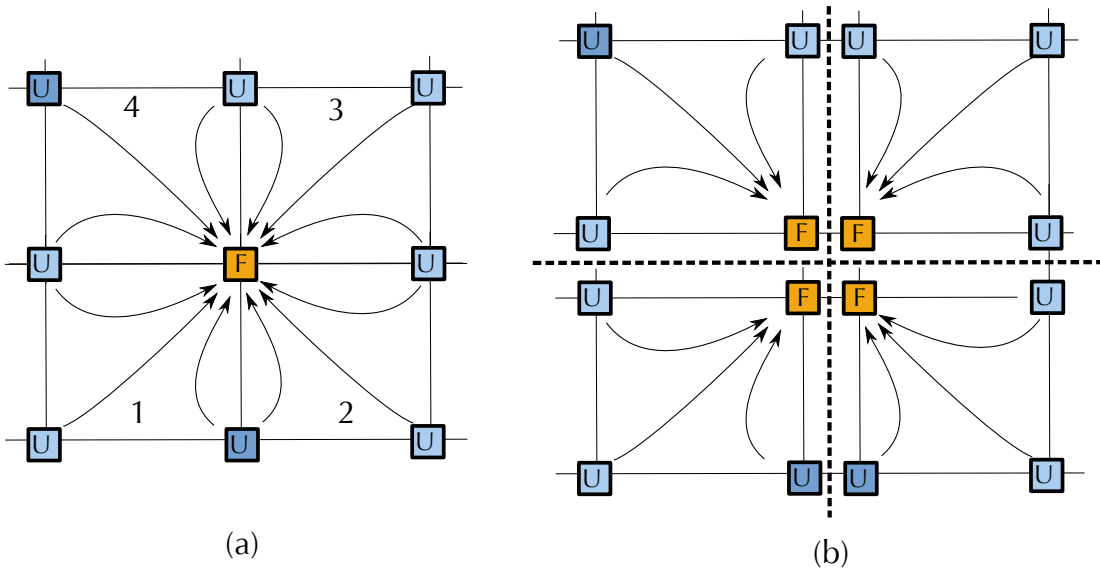


Figure 4.1: F accumulation on the regular Cartesian grid. In the sequential case (a) at the end of the traversal F is fully accumulated. For parallel border vertices (b) the same scheme only allows local accumulations.

Figure 4.1 (b) shows the same scenario for a non-overlapping domain decomposition. This is the simplest domain decomposition paradigm available, and we follow this paradigm as Peano's adaptive grid also does a non-overlapping decomposition. The vertex considered now lies on the border of four partitions. It can be easily seen that applying the same accumulation scheme as in the sequential case, will only lead to local accumulations of F values, i.e. the contributions from the remote nodes are missing. The challenge here is to investigate how to perform a global accumulation among

partitions such that, in the end, the accumulated values equal those of the sequential case. Important here is to consider the data dependencies between fluid variables and the update order needed by the fluid solver. Similar to F , the same considerations are needed for A , $gradP$, and U .

4.2 Adaptive Regular Cartesian Grid

As presented in Chapter 2, Peano already offers parallelization support for the adaptive Cartesian grid in the form of a non-overlapping domain decomposition implementing the master-worker paradigm. The main challenge here is to adapt the fluid solver to this pattern.

An important side effect of the master-worker pattern is the shifted communication pattern. Chapter 2 described how the master distributes domain partitions among workers. At the end of one grid traversal (iteration), the master collects the results and has the global information. The domain decomposition used in the CFD simulation requires nodes to exchange parallel border contributions. This exchange equals the regular grid case. However, such an exchange happens also between workers and their masters. Figure 4.2 illustrates this again for calculating F :

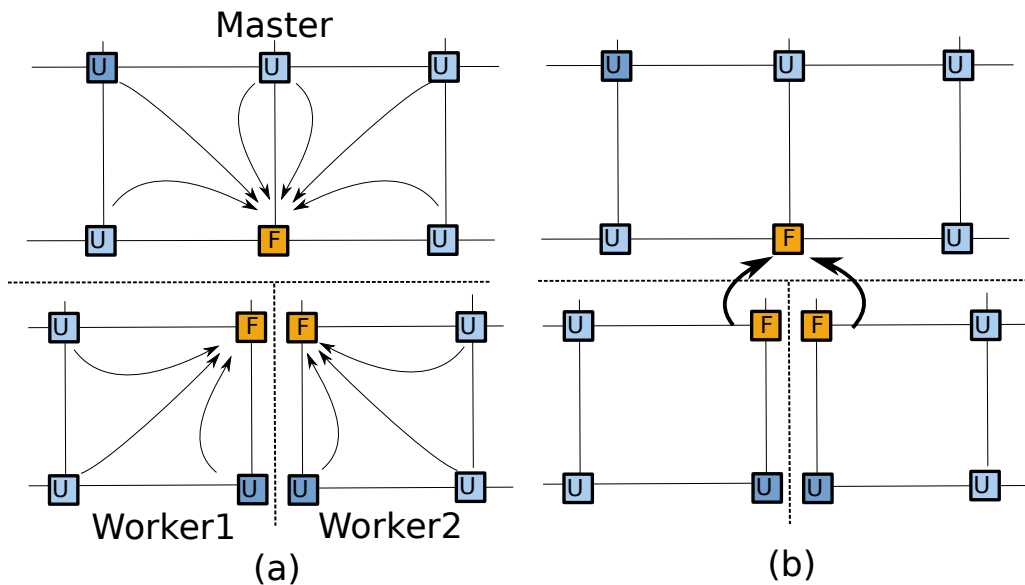


Figure 4.2: F accumulation on the adaptive regular grid. Each node first computes the local contributions (a). After the traversal workers must send border data (b). The master node has a full F accumulation while workers don't.

During one iteration the master and the workers accumulate their local contributions (Figure 4.2 (a)). The master-worker pattern requires workers to send border data

to the master at the end of one iteration (Figure 4.2 (b)). This means the master has accumulated the global F at the end of the iteration while workers have only local contributions. This is sufficient for a consistent simulation. However, the challenge here is to ensure workers also have consistent velocity and pressure data before the next iteration starts, i.e. also the timing of the data exchange must be considered.

A similar issue affects global variables used by the fluid simulation. For example, the time step size of the time integration scheme is part of the global state. This value depends among others on the maximum velocity in the entire domain. Individual workers do not have access to the entire domain and must get the new time step size from their master. This shows the need for some sort of state information exchange between master and workers and vice-versa.

4.3 Matrix-Free Poisson Equation Solver

The last task is the parallelization of the pressure poisson equation (PPE) solver. As above, similar aspects caused by the domain partitioning must be considered. The PPE solver uses conceptually a staggered grid but effectively stores the pressure information also in the vertices. While similar issues regarding data dependencies must also be considered here, one additional aspect is illustrated in Figure 4.3:

The solving process is again cell-wise. When entering a cell, the pressure gradients sitting on vertices are weighted and used to update the pressure sitting in the middle of the cell (Figure 4.3 (a)). Immediately after the pressure update, the gradients are updated to reflect the new pressure value (Figure 4.3 (b)). Because the pressure gradients are both used and changed during the same iteration, it is not possible to obtain the same contributions as with a sequential accumulation. The effect this has on the solving process will be measured in Chapter 6.

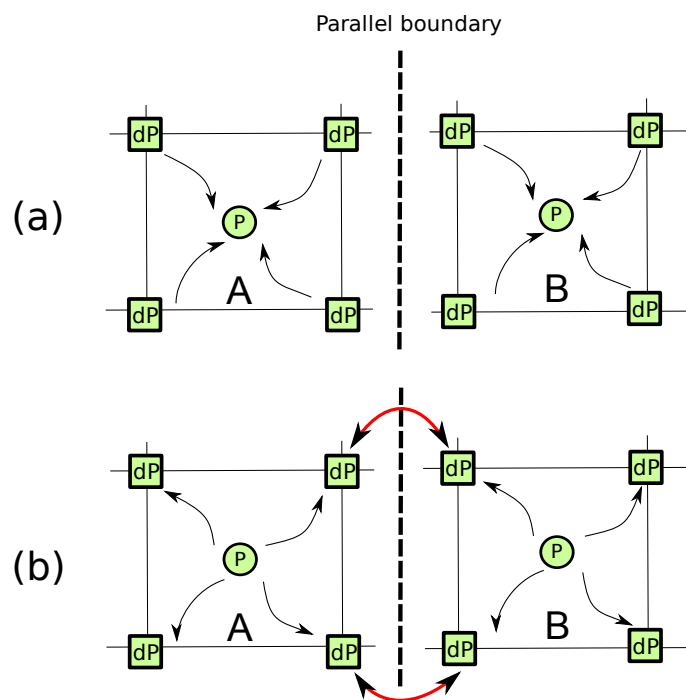


Figure 4.3: PPE solving. The pressure gradients sitting in the vertices are used (a). The new pressure is then used to update the pressure gradients (b).

5 Implementation

The solutions to the challenges presented in Chapter 4 enable the parallel run of fluid simulations in the Peano grid. This chapter follows the same structure of the previous one and it presents solutions and implementation aspects to the previously identified challenges. We look at the two types of grids and present for each grid type a particular parallel implementation for fluid simulations. The discussion for both grids follows the same structure: domain decomposition, programming model, exchanged data, and timing of communication phases.

5.1 Regular Cartesian Grid

The regular Cartesian grid is based on a simple implementation and it serves as a test bed for new ideas before they can be ported to the more efficient but also more complex adaptive grid. To avoid confusions, the following references to *grid* will concern the regular Cartesian grid unless otherwise mentioned. Also, a *node* refers to a computing unit. In hardware terms this is a processor or a core (for multicore processors). In a parallel environment each node is assigned an identification number called *rank*.

Several components of the Peano framework are involved in a fluid simulation. The *Geometry* component is responsible for providing scenario information such as which cells are inside of the domain and which are not (borders or obstacles). The *Trivialgrid* component uses this information to construct the grid in the initial phases of the simulation. It provides data structures for storing and accessing vertices and cells, traversal events, and grid management. The *ODE component* performs the time integration scheme by dealing with adaptive time-stepping as well as implicit and explicit time integration methods.

5.1.1 Domain Partitioning

To partition the computational domain a manual method is used. In the configuration file of the application a fixed computing node layout is given. The partitioning of the domain follows this given computational node configuration. For example a

5.1. Regular Cartesian Grid

2x2 node configuration would correspond to a 2x2 domain decomposition. Figure 5.1 illustrates this idea:

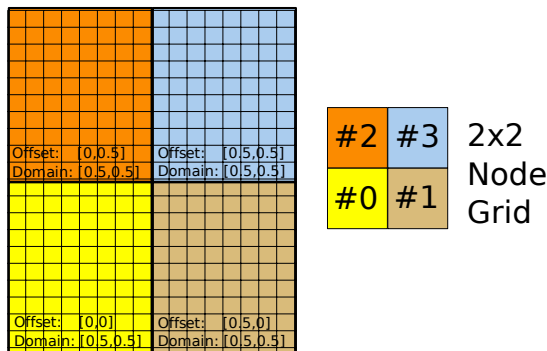


Figure 5.1: Regular Cartesian grid partitioning derived from a 2x2 node configuration. Each computational node calculates an offset and domain size based on the rank it has.

If the number of cells in each dimension cannot be equally distributed over the available nodes, the first ranks will have grid sizes with one additional cell per dimension. To implement the partitioning, each node creates and stores its part of the grid. The ranks available at the beginning of the simulation are arranged in the desired topology ($M \times N$) and ordered lexicographically. Based on its rank, each node computes its offset in the domain and the size of the local domain. The following routines encapsulate this functionality:

```
calculateGridDimensionOffsetAndSize()  
getNodeCoordinates()  
getNeighborCPURank()
```

5.1.2 Programing Model

The programming model chosen for the regular grid is single program multiple data (SPMD). This model describes a computational task in which the same program instructions are executed by all nodes but with different data. A grid traversal is such a task. During a particular step of the fluid simulation the traversal visits the grid cell by cell and performs on each the same operations. This type of parallelism is also a data parallelism in contrast to functional parallelism where functions (different operations) are run in parallel on the same or different data. Figure 5.2 shows the SPMD programming model applied to the fluid simulation.

Each node starts by obtaining its rank. With the rank the CPU coordinates are calculated. They are used to determine the global offset and the local domain size. The

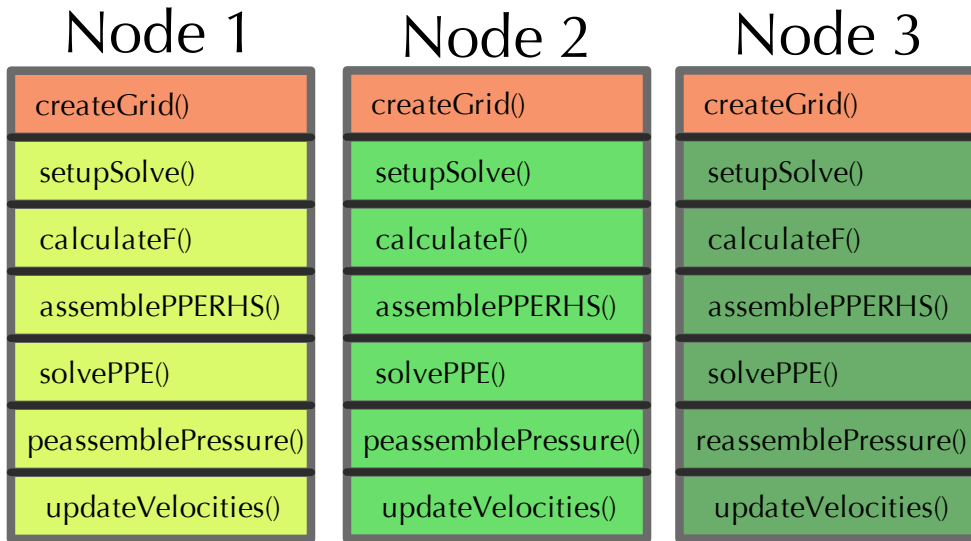


Figure 5.2: Single program multiple data model used for the fluid simulation on the regular Cartesian grid. Each node runs the same code. Each node creates only its own grid representing only a part of the global domain.

grid construction and storage is local. One-dimensional arrays are used to store cells and vertices. The vertices lying on a border of two partitions are duplicated. Each side has a copy of these as they play a crucial role in the border communication process.

5.1.3 Communication

Splitting the domain into several partitions also splits the information in that domain. To have the advantage of domain partitioning but at the same time maintain the consistency of the PDE on the entire domain, information transfer between partitions must take place. The Navier-Stokes fluid equations are continuous and a global correct solution over the entire domain cannot be obtained by solving on isolated sub-domains. Velocities and pressure information “travels” over the domain and at the borders of the sub-domains this transfer must be provided. The following routines of the regular grid implement the information transfer:

```
exchangeBoundaryData()
sendDataForCoordinateRange()
sendDataToSelfForCoordinateRange()
receiveDataForCoordinateRange()
mergeWithNeighbourData()
```

The idea here is to decouple the communication from the actual fluid solver. This has implementation advantages as new features can be added without considering parallel aspects. Also the readability of the code is improved. The communication func-

tionality is encapsulated in the grid and applications using the grid can make use of it. The `exchangeBoundaryData()` routine is such a general purpose boundary exchange function. When called, it runs over the domain boundaries and sends/receives vertices to/from the corresponding neighbors. The code of this routine is listed in Appendix B.

Deadlocks

The main challenge in performing communication is to ensure that the order of sent/received messages does not lead to any deadlock situations among the involved computational nodes. A deadlock can have several causes. It can occur in a simple two-way exchange when both partners start by going in a receive phase. A send-receive or send-send start phase would avoid the deadlock. When considering boundary exchanges a deadlock situation can also occur when the border on one processor has more elements than on the second processor. The second processor will continue execution as it will receive a correct number of vertices (albeit possibly non-corresponding) but the first processor will hang waiting for additional vertices.

The second type of deadlock can be avoided by a correct construction of the grid and becomes an issue with adaptive grids where the borders between partitions constantly change. The first case of dead lock can be eliminated by a conceptually correct succession of sends and receives. Such a scheme for a two-dimensional domain is presented in Figure 5.3. The communication is split in two phases. First, all the nodes traverse their boundaries and send vertex data to all other nodes which have a corresponding copy of the vertex. The semantics used here to send are *fire and forget*, i.e. as soon as the data to be sent is in a system buffer the send operation returns and the execution can continue. This does not guarantee that the data is also received. In the next phase all nodes go into a receive phase. This phase also will not block as everything that has to be received has been sent in the previous phase. The second phase also acts as a synchronization barrier for all the involved nodes. If any of the messages expected in the receive phase does not arrive, execution will not continue.

5.1.4 Fluid Data Dependencies

With the communication process in place, the next item to consider is what and when to communicate. Two reasons make this decision necessary. On the one hand, as few as possible communication leads to a better parallel efficiency. The amount of sent data can be reduced either by compressing messages or by locally computing some values. Peano supports compressing data structures in so called *PackedRecords* which reduces the amount of sent data[5, 6]. The packing/unpacking of vertex and cell data is provided by the framework by using a tool which automatically generates the needed data structures and their compression routines. Even though the messages have to be packed/unpacked at the sender/receiver the reduced communication overhead will pay

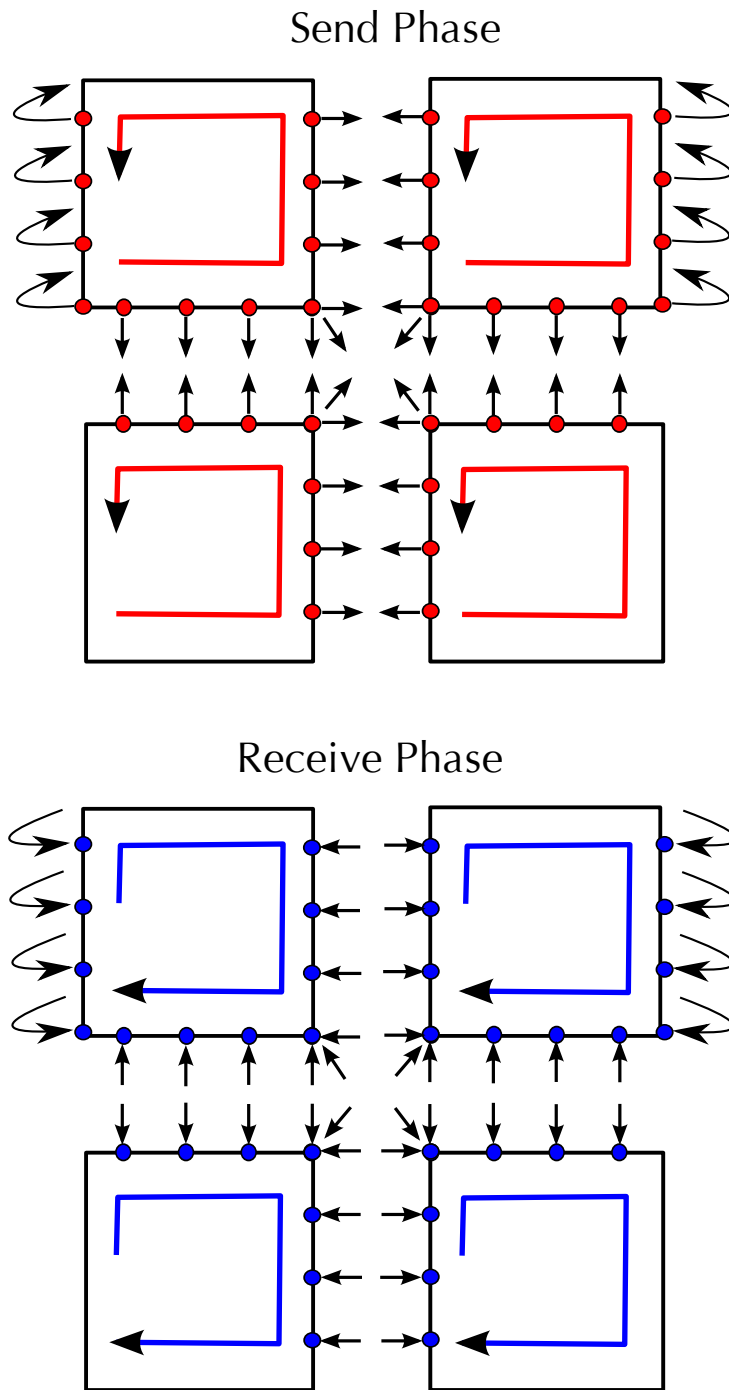


Figure 5.3: Exchange pattern of the vertices lying on the parallel boundaries. In the send phase each node traverses its boundary and sends shared vertices. In the receive phase the traversal is reversed and shared vertices are received. Exchange of periodic boundaries is also depicted. Periodic boundary condition considers the domain wrapped around one or more dimensions.

off.

On the other hand, to reduce communication overhead we avoid sending values which might as well be computed locally. This of course depends on the concrete application. The most important variables used in the fluid simulation were presented in the equations of Chapter 3. Specific to the fluid solver component the following data dependencies can be identified:

$$\begin{aligned}
 h &\longrightarrow A, \text{ done only once} \\
 U &\longrightarrow F \\
 F, A &\longrightarrow RHSPPE \\
 RHSPPE &\longrightarrow P \\
 P &\longrightarrow gradP \longrightarrow P, \text{ every PPE solver iteration} \\
 P &\longrightarrow gradP \\
 gradP &\longrightarrow U
 \end{aligned}$$

h is the meshsize. In a regular Cartesian grid the mesh size is constant along one dimension. A is the support of a vertex, i.e. the influence range of the vertex's stencil. U is the velocity vector located at each vertex position. As presented in Chapter 2, F is the diffusion/convection vector derived from the velocities. $RHSPPE$ is the right-hand side of the pressure poisson equation and is obtained from F and A . The $RHSPPE$ is then used in the pressure equation to obtain new pressure values. After the pressure solving process, pressure is known in the middle of the cell. To prepare for updating the velocities, this pressure is interpolated in the vertices: $gradP$ is the sum of the pressure contributions of each cell to a common shared vertex. With the pressure interpolated in the vertices, the velocities U can be updated and the solving process continues for the next time step.

From the above variables some must be exchanged while others are locally computed. As presented in the previous chapter, the exchange scheme needs to preserve the semantics of the sequential case, i.e from the same input values the same output values are to be obtained in a parallel as in sequential execution. Lets consider the A accumulation as an example. To compute A the mesh size is needed. For a regular grid the mesh size is identical for all the involved nodes. This means A can be calculated correctly for all vertices except the ones lying on a parallel border. Here, it is not known how many fluid cells are neighboring the vertex in the other partitions and so A must be exchanged to obtain the global accumulation. The entire A accumulation process consists of computing the local A and summing it up with all the A contributions received from other nodes sharing that vertex. This way the sequential semantics are preserved.

The U and F data also need to be merged on the boundary vertices. The decision made here is to exchange the F data. The velocities will be calculated locally using F and the pressure gradient. Inside the pressure poisson equation, data also needs to be communicated. What exactly depends on the type of solver used and the placement of

its degrees of freedom. Section 5.3 presents the case of the matrix-free staggered grid solver. $gradP$ is a sum of cell pressure contributions. Similar to the accumulation of A , vertices on the parallel boundary need the contributions to $gradP$ of several partitions.

To get a correct parallel execution the three fluid related values need to be exchanged: A , F , and $gradP$. In the calculation of these variables we only use data which are unchanged during partition traversals. F , for example, is computed from the velocities which, in the F traversal, are not modified. This guarantees that the summation of the individual contributions on each partition leads to the same result as in the sequential execution. As we shall see in Section 5.3, this is not the case for the pressure poisson equation solver.

In the code, information exchange is triggered by calling the corresponding send/receive routines on individual vertices. Each vertex needs memory space to store application data (A , F , $gradP$, etc.) and also grid data. As presented above, not all variables are necessary to be exchanged. To avoid sending unnecessary data, the send routines for vertices pack only data marked for parallelization and ignore the other. This helps to reduce communication overhead and improve parallel efficiency.

5.1.5 Parallel Merge Phases

To avoid code duplication, the generic `exchangeBoundaryData()` routine is used whenever something needs to be communicated. As we have seen, different values need to be merged at different points in the algorithm. To indicate which values should be merged a merge target is used. This is a global state variable which is used in the `exchangeBoundaryData()` routine. It indicates which variables are to be merged. This works as follows. The application sets the target for a specific variable, e.g. A and then calls the exchange function. The exchange function performs the boundary traversal and when preparing to merge two vertices it inspects the target to see what should be merged. The used merge targets are listed in Table 5.1.

Target	Description
MERGE_A	merge A data.
MERGE_F	merge F data.
MERGE_GRADP	merge pressure gradient data.
MERGE_MIDPOINTS	merge solver data.

Table 5.1: Merge targets are used to decide which variables are communicated in a specific communication step.

The parallel steps of the fluid simulation running on the regular adaptive grid are presented in Algorithm 5.1.5. The idea is to leave the steps of the serial fluid simulation

unchanged and only interleave communication phases. The entire simulation is unaware of the parallelization which is carried out by the grid.

5.1.6 Global State Information

The fluid simulation uses some global variables in order to compute various criteria. For example the maximum velocity in the entire flow field is needed in order to compute the size of the next time step. To obtain such global values each computing node sends its local value to the node with rank 0. This node aggregates all the received velocities and computes the correct next timestep size which it then broadcasts to the other nodes. A similar procedure must be done for all variables which characterize the global state of the fluid solver.

5.2 Adaptive Regular Cartesian Grid

Several lessons learned from parallelizing the solver on the regular Cartesian can be applied also to the adaptive grid. One crucial difference lies however in the work distribution and communication pattern required by the adaptive grid. The next subsections present a solution for running parallel fluid simulations using a hierarchical adaptive grid.

5.2.1 Domain Partitioning

The *grid* component in Peano offers the possibility to adapt the grid resolution or areas of interest specific to the PDE. The adaptive solver refines the grid at either fixed areas of the domain (borders, obstacles, etc.) or dynamically based on certain criteria that change during the simulation.

The domain decomposition is already built in the *grid* component. Figure 5.4 presents the decomposition of a square domain using 20 nodes. As presented in Chapter 1, the partitions correspond to a partitioning of the space-filling curve which gives the traversal order. The non-intersecting visit-only-once traversal gives also a quasi-optimal partitioning of the domain. The partitions are compact and have a good surface to volume ratio. This translates into a larger computational part compared to the communication phase and gives the possibility of overlapping computation and communication.

Algorithm 5.1 Algorithm of the fluid solver running in parallel using the regular Cartesian grid.

```
determineRank()
calculateGridDimensionOffsetAndSize(){ Based on rank compute do-
main responsibility.  }
3: constructGrid() { Create local grid.  }

    { Compute A for inner domain and exchange it.  }
6: countInnerCells()
setMergeTarget(MERGE_A){ Communicate and merge A once.  }
exchangeBoundaryData()
9:
    while  $t < t_{end}$  do
        { Compute F for inner domain and exchange it.  }
12: calculateF()
setMergeTarget(MERGE_F)
exchangeBoundaryData()
15:
    { Prepare and solve the pressure equation.  }
assemblePPERHS()
18: solveSystem() { Perform parallel solving.  }

    { Interpolate and exchange the pressure.  }
21: pressureReassembleAndGradientAndForces()
setMergeTarget(MERGE_GRADP)
exchangeBoundaryData()
24:
    plotOutput()
    updateVelocities()
27: end while
```

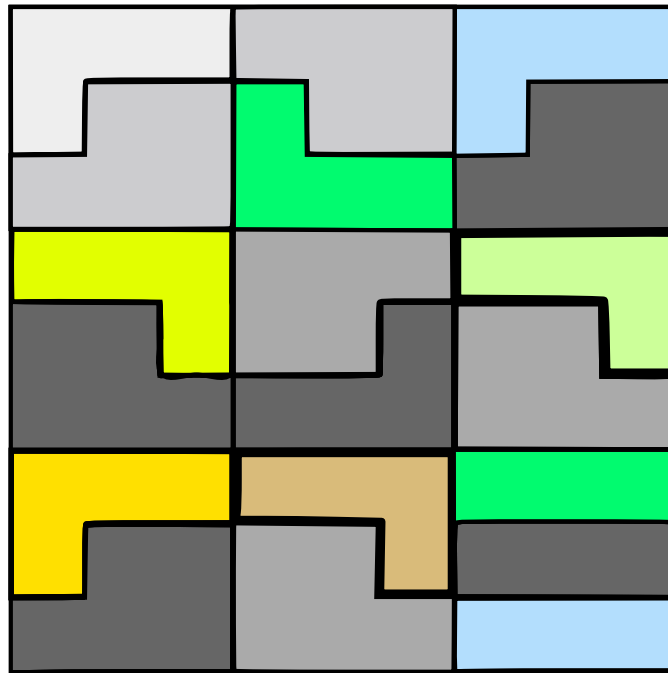


Figure 5.4: Partitioning of the adaptive grid.

5.2.2 Programming Model

The adaptive grid is not only hierarchical in terms of grid representation and storage, but also the topology of the master-worker ranks equals a tree. Figure 5.5 shows the allocation hierarchy of the grid partitioning from Figure 5.4. Rank 0 starts by owning the entire domain. After the first refinement step it has 9 cells to delegate to 9 other ranks. These then become workers for the master rank. After another refinement step each of the workers themselves has a maximum of 9 cells to delegate. Peano gives each of them another node from the remaining 10. The last idle node is allocated to that node which was faster in requesting it from the global pool of idle nodes. It is rank 10 in this example.

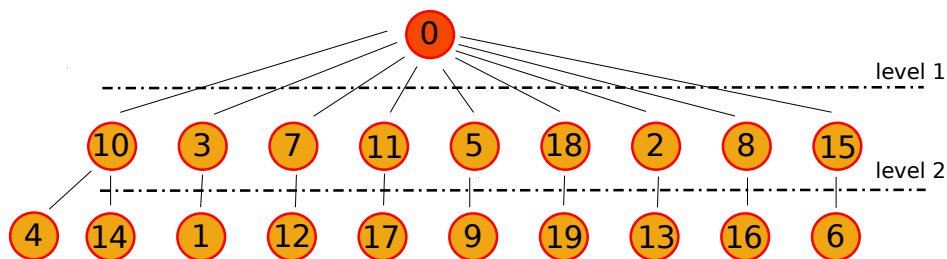


Figure 5.5: Node hierarchy for the grid distribution in Figure 5.4.

The adaptive grid follows both in the creation of the grid but also in subsequent

traversal the **master-worker** parallelization paradigm, while the regular grid uses the SPMD strategy where each computing node executes exactly the same lines of code and only the domains are different. The master-worker model requires a different behavior depending on the fact whether the node is a master, a worker, or both.

There are two types of master nodes to distinguish. The **global** master is the node with rank zero. It is responsible for the logic behind the parallel adaptive grid. Among its functions are: creating and managing the node pool, distributing the top levels of the spacetime, and setting the global state for all other nodes. A **non-global** master is a node which has distributed parts of the allocated spacetime to other workers but, at the same time, is also a worker for another node. This distinction is important for the functional decomposition used in the implementation.

In this thesis, the existing serial fluid simulation is adapted to the master-worker model due to the *master drives computation* approach. The master initiates the computation and provides the workers with necessary start-up data. Each worker uses the data to compute its local task. After finishing it goes in a wait phase until new tasks are available from the master. Figure 5.6 presents such a model with one master and two workers. The workers are displayed shifted to reflect the order of the execution start.

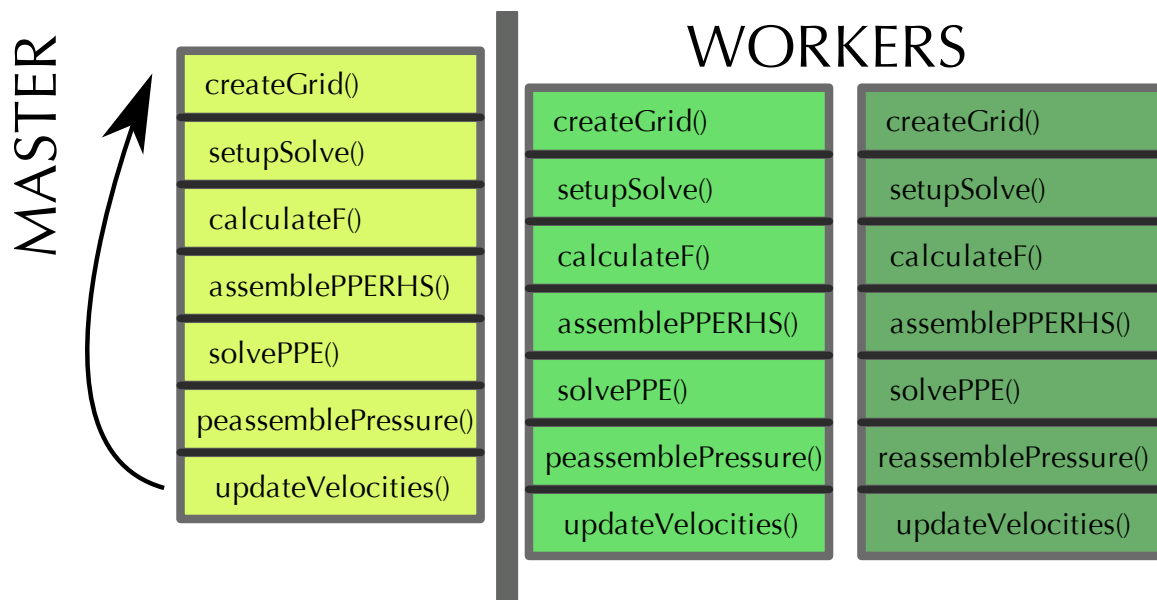


Figure 5.6: Master-Worker pattern for the fluid simulation. The master starts and drives the workers.

5.2.3 Communication Pattern and Phases

Unlike the regular Cartesian grid, the adaptive grid provides along with the partitioning also the exchange of the parallel boundary vertices. To make use of this facility several callback routines are available (Table 5.2). Inside these methods any application can perform the specific merge operations with regard to application data. The fluid solver uses these methods to merge grid, fluid, and solver data.

Callback	Description
<code>mergeNeighboursWithOwnProcessVertex()</code>	Called for merging a vertex which sits on the parallel boundary and is shared with other computing nodes on the same refinement level.
<code>mergeWithMasterProcessVertex()</code>	Called by a master in order to merge data received from a worker.
<code>mergeWithWorkerProcessVertex()</code>	Called by a worker in order to merge data received from its master.
<code>prepareSendToMaster()</code>	Allows additional operations before sending data to master.
<code>prepareSendToWorker()</code>	Allows additional operations before sending data to worker.
<code>prepareSendToNeighbour()</code>	Allows additional operations before sending neighbor.

Table 5.2: Callbacks for vertex merge operations.

The master-worker pattern makes the communication semantics differ dependent on the relation between computing nodes. Master-worker differs from worker-worker communication. The event callbacks during a traversal of the hierarchical adaptive grid were presented in Chapter 1 and are the locations where the communication is initiated and performed. Figure 5.7 presents the relation of these events to the communication scheme. A master process starts a traversal of the grid with the top element. It then descends on the next level by calling `startStepsDown()`. If any of the elements on the lower level were assigned during grid construction to other ranks, master-worker communication is needed. Before continuing with local work the master sends start-up information to the workers. When not active, each worker waits for job requests from the master. As soon as such a request comes and together with startup information, it proceeds by traversing its own part of the tree. The master-worker relation is recursive. A worker can also be a master for another rank and pass start-up information down the node hierarchy. The most important callbacks with their semantics are listed in Table 5.3 and presented in detail in Appendix A.

After finishing the traversal each worker sends data up to the master. This is a crucial aspect as it shows the update status after a complete tree traversal. Master

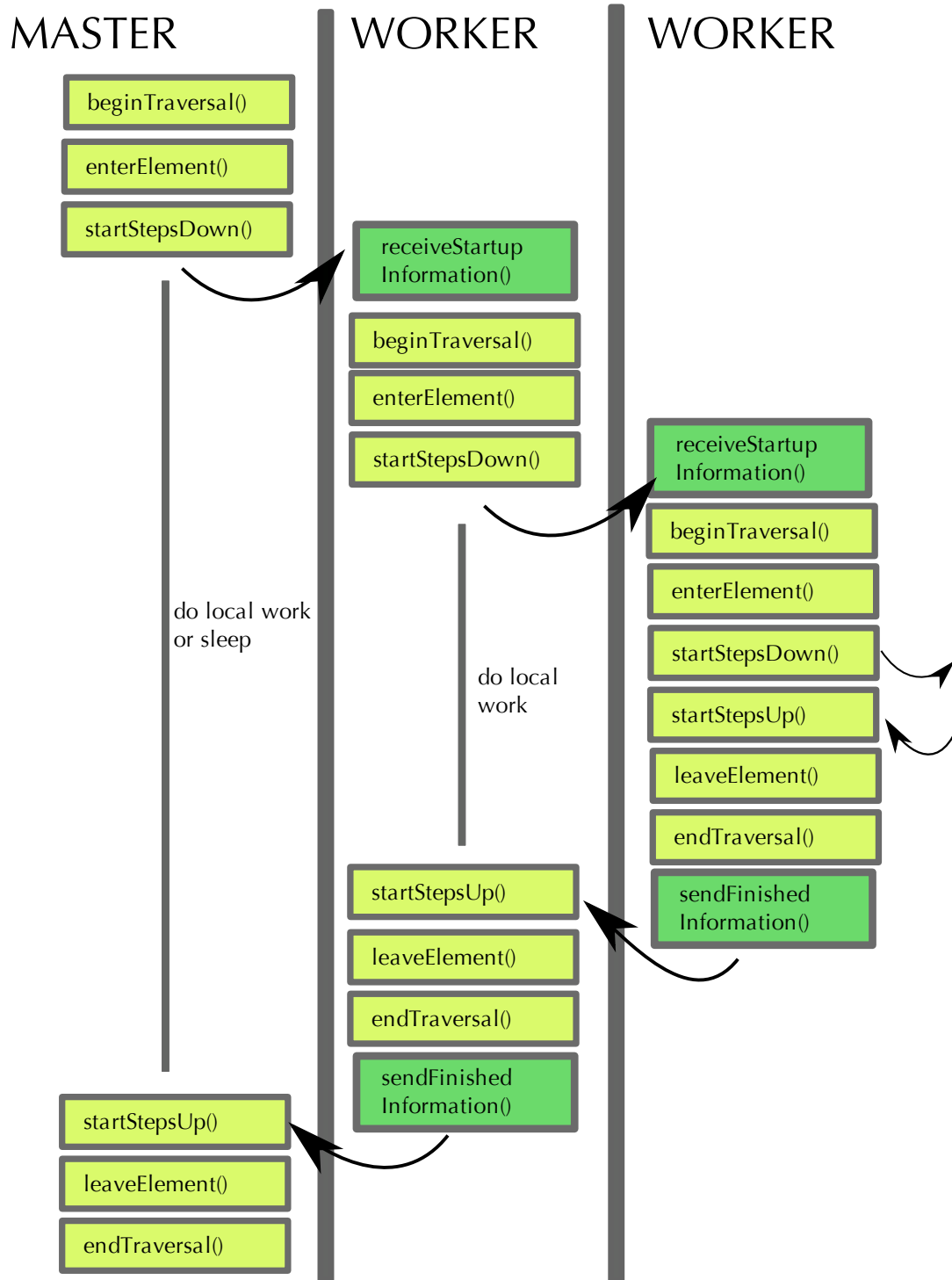


Figure 5.7: Master-worker communication steps and their relation to grid traversal events.

nodes have the updated information from their local part of the grid and the complete information of their workers. For example consider the maximum velocity in the flow field needed to compute the new time step. At the end of the `updateVelocities()` step the master has the global maximum velocity while the workers have only their local one. If the simulation continues like this, each worker will compute a separate wrong next time step. This shows the need for two distinct communication phases.

In **phase one**, the master receives data from all contracted workers and merges that data to obtain global values. In this phase the workers only send information at the traversal end. In phase one only the master is allowed to merge received data with its local data. The workers will get correct global information in **phase two** the communication process. In this phase the master will not merge and only workers merge with other neighboring nodes on the same level or with master start-up information. This communication pattern is illustrated in Figure 5.8.

Callback	Master	Worker
<code>startStepsDown()</code>	start workers and send start information	–
<code>startStepsUp()</code>	receive finish information from workers	–
<code>beginTraversal()</code>	–	receive start-up and state information
<code>endTraversal()</code>	–	prepare and send state information

Table 5.3: Event semantics for master and worker.

5.2.4 Fluid Data and Merge Operations

The fluid variables which need to be exchanged and merged have already been identified in the parallelization of the regular grid. The same operations now need to be performed for the adaptive grid. In merging fluid data, the implementation considers the two different communication phases presented in the previous sub-section.

The fluid simulation is implemented as a sequence of grid traversals with different adapters which perform operations on vertices and cells. Each adapter corresponds to a step of the fluid solver. For example, the F adapter uses the new velocities to recalculate the F values. Consider a master node which has delegated a part of its initial domain to a worker. Due to the nature of the master-worker pattern, at the end of the traversal, a master rank has the new F values on its local domain but also the correct accumulated F values on the border with its worker. This is the result of

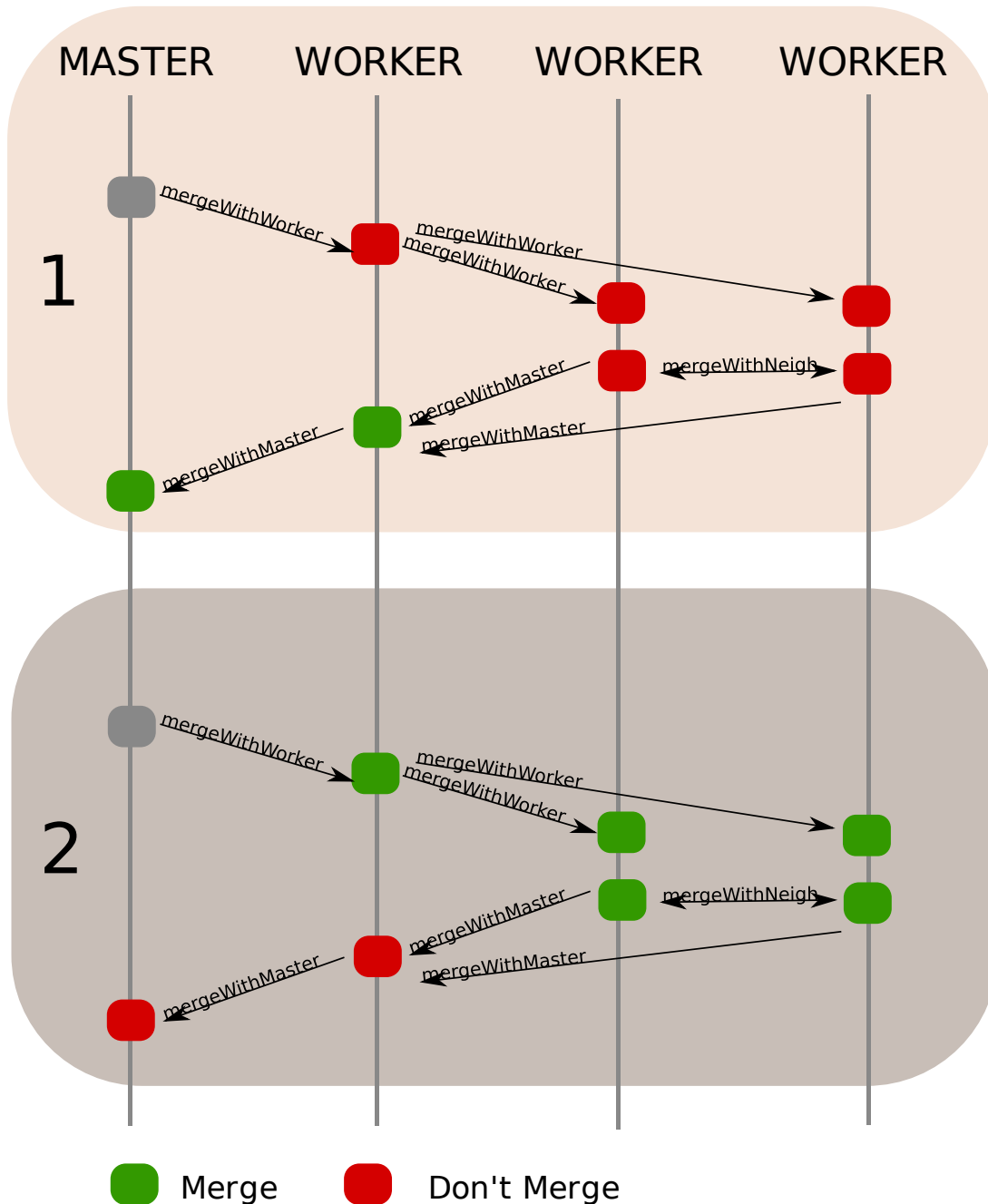


Figure 5.8: Communication Phases. The Master-Worker execution pattern imposes a shifted synchronization scheme between master and worker. To accommodate this, a complete communication exchange requires one iteration plus the beginning of the next iteration and can be structured in two phases. In phase one only master nodes can collect data from workers and merge it with local data. In this phase only *mergeWithMaster()* routines are allowed to merge. Phase two takes place at the beginning of a next iteration and here the workers can merge with both vertices from their respective master (*mergeWithWorker()*) and also from neighboring workers (*mergeWithNeighbors()*).

phase one communication. At this point the master could continue the solving process.

However, at the end of the traversal the worker does not have the fully accumulated F on the border with its master. It needs another iteration such that it also has consistent data. An additional grid iteration with this sole purpose would be expensive. A better solution is to reuse the immediately following iteration of the fluid solver to complete the communication. Here, a phase two type of communication takes place (workers merge).

To implement the communication phases, we use the same idea of merge targets. In this case however, the merge targets indicate not only which variable is to be merged but also which communication phase is to be applied. Table 5.4 lists the new merge targets. Phase one targets allow to merge on masters while during phase two targets merge data on workers.

Target	Description
MERGE_A_PHASE1	master merges A
MERGE_A_PHASE2	worker merges A with master and other workers
MERGE_F_PHASE1	master merges F
MERGE_F_PHASE2	worker merges F with master and other workers
MERGE_GRADP_PHASE1	master merges the pressure gradient
MERGE_GRADP_PHASE2	worker merges pressure gradient with master and other workers
MERGE_POISSON_PHASE1	master merges solver pressure gradients
MERGE_POISSON_PHASE2	worker merges solver gradients with master and other workers

Table 5.4: Merge targets for the adaptive grid.

The new simulation steps for the master with rank zero are presented in Algorithm 5.2. They are pretty similar to the usual SPMD simulation steps. As this code will only be executed by the master with rank zero, the merge targets are all in phase one. The workers themselves don't know which fluid solver step is next and rely on the global master to fix the state.

The fluid simulation as executed by the workers is presented in Algorithm 5.3. Here, the differentiation between phases is necessary: Consider the PRESSURE state. A computing node will only perform a merge in the `mergeWithMasterProcessVertex()` routine. This routine will only be called for nodes which are themselves workers but also masters for other workers. Simple workers will not receive this callback. This way, it is ensured that only masters collect data from their workers and perform a correct update on their level.

Algorithm 5.2 Fluid simulation algorithm for rank 0.

```

nextWorkerState(MOCK)
constructGrid()
3:
nextWorkerState(COUNT)
setMergeTarget(MERGE_A_PHASE1)
6: countInnerCells()

nextWorkerState(SETUP_SOLVER)
9: setupSolver()

while  $t < t_{end}$  do
12: nextWorkerState(F)
setMergeTarget(MERGE_F_PHASE1)
calculateF()
15:
nextWorkerState(PPERHS)
assemblePPERHS()
18:
while  $residual < tolerance$  do
nextWorkerState(SOLVE)
21: setMergeTarget(MERGE_POISSON_PHASE1)
solveSystem()
end while
24:
nextWorkerState(PRESSURE)
setMergeTarget(MERGE_GRADP_PHASE1)
27: pressureReassembleAndGradientAndForces()

nextWorkerState(PLOT)
30: plotOutput()

nextWorkerState(VELOCITIES)
33: updateVelocities()
end while

```

The next step in the fluid simulation following the `PRESSURE` step is `VELOCITIES`. Here, the phase two of the pressure gradient communication takes place. Any worker will receive first the callback `mergeWithWorkerProcessVertex()` where it can merge with master information. Also the callback `mergeNeighboursWithOwnProcessVertex()` will be called on all vertices on the parallel boundary which are not shared with a master.

5.2.5 State Communication

Algorithm 5.3 shows that the workers are running a different code branch than the node with rank 0. A worker receives a state object which informs it what adaptor to run next. This corresponds to the idea of the Master-Worker pattern where workers are unaware of global information and just process sub-domains and return the results to their masters.

For a worker to know only the step of the fluid simulation which will be performed next might not be sufficient for a correct execution. For example, one piece of information needed in the `VELOCITIES` phase is the next time step size. To calculate the correct time step size for the next fluid iteration is the responsibility of the global master. For the calculation it needs the maximum velocity in the entire domain. This will be communicated upward in the tree of nodes using **worker-to-master** state exchange.

With the global maximum velocities in the field, the global master computes the new step size and pushes this piece of information down the tree of nodes. The step size travels then recursively using the **master-to-worker** state exchange until it reaches the lowest workers. At this point all nodes have the correct next time step size. It can be that by the time the global state reaches the lowest workers some upper workers have already started traversing their k -spacetree partition. This does not cause any synchronization problems as the worker-to-master state exchange will bring all nodes in sync again. A master cannot start a new iteration until all workers have finished the first.

The step size calculation is an example for a general state information communication between computing nodes. Master nodes are responsible to provide correct global state information to workers. This simplifies the implementation of the workers and allows for a greater flexibility in the work distribution process. Table 5.5 lists other state information passed from master to worker ($M \rightarrow W$) and worker to master ($W \rightarrow M$) during the fluid simulation.

Algorithm 5.3 Job processing loop for worker nodes.

```

while waitForJob do
  if !stopMessage then
3:   if MOCK then
      mockIteration()
    end if
6:   if COUNT then
      setMergeTarget(MERGE_A_PHASE1)
      countCellsIteration()
9:   end if
      if SETUP_SOLVER then
        setMergeTarget(MERGE_A_PHASE2)
12:      setupSolverIteration()
      end if
      if SOLVE then
15:      setMergeTarget(MERGE_MIDPOINTS)
        solveIteration()
      end if
18:      if PRESSURE then
        setMergeTarget(MERGE_GRADP_PHASE1)
        reassemblePressureIteration()
21:      end if
        if F then
          setMergeTarget(MERGE_F_PHASE1)
24:          calculateFIteration()
        end if
        if PPERHS then
27:          setMergeTarget(MERGE_F_PHASE2)
            assembleRHSIteration()
        end if
30:      if VELOCITIES then
        setMergeTarget(MERGE_GRADP_PHASE2)
        updateVelocitiesIteration()
33:      end if
    end if
  end while

```

State	Meaning	Direction
phase	indicates next adaptor to run	M→W
tau	next time step size	M→W
odeStepNumber	step of the time integration scheme	M→W
x	accumulated pressure solution	W→M
dx	change in the pressure solution	W→M
uMaxPerDim	maximum velocity per dimension	W→M
uL2h	velocity L2 norm	W→M

Table 5.5: State information needed for a parallel fluid simulation.

5.3 Matrix Free Poisson Solver

In Algorithm 5.1.5 the solving of the pressure poisson equation is presented only briefly. This is actually the most computationally most intensive part of the fluid solving process and also involves parallelization. The currently used solver is matrix-free, i.e. no global system matrices are constructed [8, 18]. It makes use of the adapter pattern in Peano and during a traversal visits each cell and performs an update. The type of the solver is Gauss-Seidel using over-relaxation (SOR). The iterative solving process for a general system of linear equations with an unknown x can be summarized as follows;

$$x_i^{tmp} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k \right) \quad (5.1)$$

$$x_i^{new} = (1 - \omega) x_i^{old} + \omega x_i^{tmp} \quad (5.2)$$

A finite element stencil is applied on each cell. Depending on the stencil form this means several neighbor cells are involved in updating a single cell. The concrete stencil gives the a_{ij} coefficients. In any cell during the grid traversal an update will use values of the neighbor cells which are already updated and old values for not yet visited neighbor cells. This is expressed in Equation 5.1. The successive over relaxation method then proceeds by computing the final value of the pressure lying in that cell as a weighted sum of the previous value and the newly computed one (Equation 5.2).

The traversal type in Peano is cell-wise. When entering a cell, any adapter has access to the cell data sitting in the middle of the cell and the corresponding 2^D vertices. No direct access to the neighboring cells is possible. However, in SOR the stencil needs values from the neighboring cells. To overcome this limitation the vertices are used also to store indirect pressure information from neighboring cells. Figure 5.9 presents the steps of the pressure equation solving process. The vertices store indirect pressure information of all the surrounding cells in the form of pressure gradients. The solver stencil is then applied to the gradients and the new pressure value is placed in the cell.

The gradients are then updated to correspond to the new pressure contribution.

To parallelize the SOR solver it is enough to exchange the pressure gradients between neighboring partitions (Figure 5.10). Each node receives the gradient update from neighbor ranks and adds it to its own. Between any two pressure poisson iterations the vertices of the parallel boundaries have old values and the SOR deteriorates to a Jacobi solver. This will have an impact on the convergence rate as the number of partitions increases. The suitability of a parallel version of an SOR solver [15] is poor and can even lead to divergence if the omega factor is not adjusted with an increasing number of partitions.

The extended algorithm for solving the pressure equation also has a communication phase. The `MERGE_MIDPOINTS` target is used as a flag for the merge routine. Algorithm 5.3 presents the new solving loop.

Algorithm 5.4 Algorithm of the fluid solver running in parallel using the regular Cartesian grid.

```
while residual < tolerance do  
    solve()  
    setMergeTarget(MERGE_MIDPOINTS)  
    exchangeBoundaryData()  
end while
```

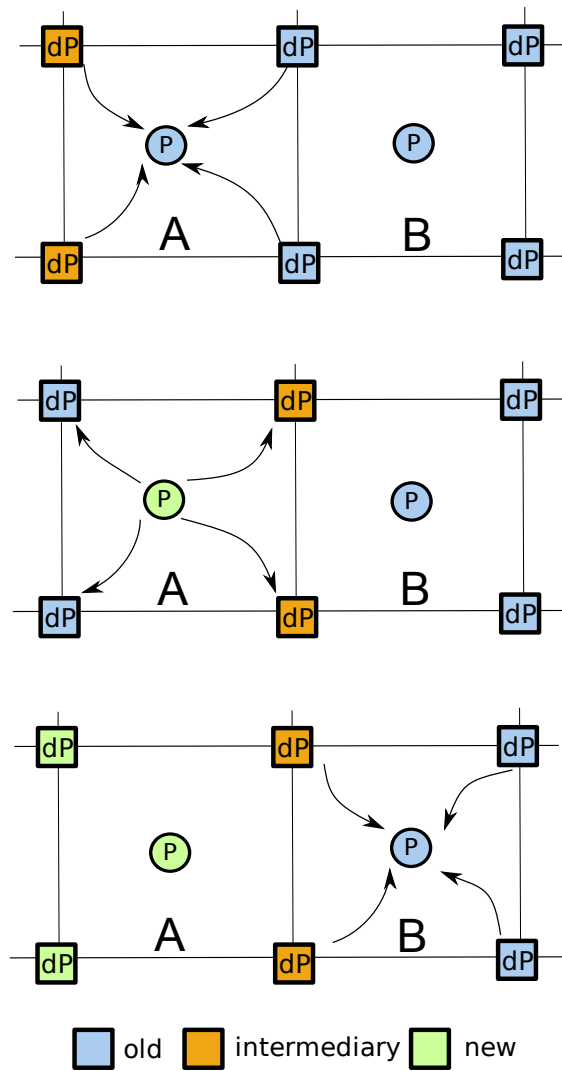


Figure 5.9: Solver Process. (1) Use vertex pressure gradient for new solution. (2) Update vertices with new solution. (3) Continue traversal.

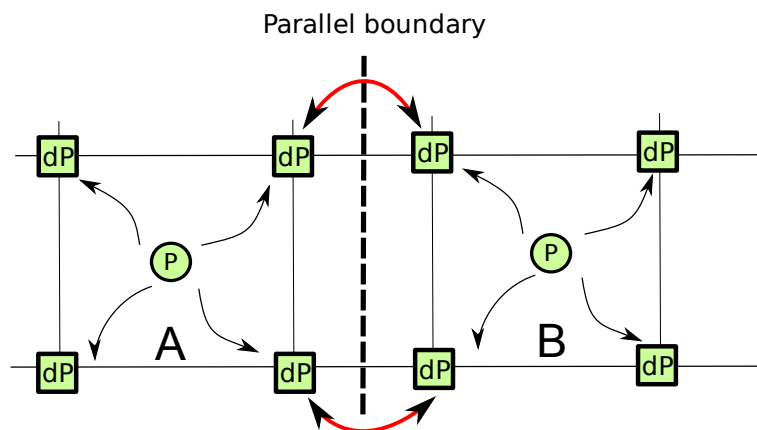


Figure 5.10: Communication in the parallel SOR solver.

6 Numerical Experiments

This chapter evaluates the performance of the parallel fluid solver using the parallel adaptive Cartesian grid. Eventhough the regular Cartesian grid is not focused on performance, it is still interesting to see its parallel performance. Some measuremets are also done for this grid type.

First, the influence of network properties and the impact of bundling messages is presented. Second, specific performance measurements for parallel implementations are introduced. These measurements are then used to inspect the parallel efficiency of the implementation from the previous chapter. The chapter closes with the blocked matrix-free pressure solver and its convergence speed with respect to the number of partitions.

All experiments were conducted on two parallel systems: the Infinicluster and the LRZ Linux cluster. Descriptions of the two systems can be found in Appendix C.

6.1 Message Size

The performance of parallel applications is influenced by both the hardware used and the quality of the software implementing the communication protocols. Peano uses the message passing protocol (MPI¹) for communication. It packs the records into messages which are then sent over the network to other computing nodes. Like any network transfer, runtime costs arise. In [10], a simple model for the total time of a message transfer is presented. It consists of a fixed cost plus a variable cost depending on the length of the message :

$$T_{message_transfer} = \alpha + \frac{N}{\beta}$$

The fixed cost α is called latency and is essentially the time it takes to send an empty message over the communication medium. Latency includes overhead due to software and network hardware plus the time it takes for the message to traverse the

¹<http://www.mcs.anl.gov/research/projects/mpi/>

communication medium. The bandwidth β is a measure of the capacity of the communication medium while N is the length of the message.

Latency and bandwidth can vary significantly between systems and it is a good practice to consider them when running parallel simulations. For example, in a system in which α is relatively large, it might be worthwhile to try to restructure a program that sends many small messages and instead aggregate the communication into a few large messages. In [3] several modern platform are evaluated with respect to network performance.

The purpose of the next experiment is to identify the largest number of messages that can be bundled together which has the biggest impact on reducing the execution time. It is a precondition to obtaining optimal measurements in the subsequent experiments.

Experiment

Peano offers the possibility to aggregate several messages and send/receive them in larger but fewer messages[17]. Such a feature reduces the fixed cost of latency. Figure 6.1 shows the effect of the network properties on the execution time of a parallel fluid simulation in Peano. The experiment consists of running the same simulation but varying the number of messages that are bundled together.

On the Infinicluster, message bundling can significantly influence runtime. Larger bundles lead to runtime reduction by almost a factor of two. For the considered simulation and hardware, an optimal message number is at least 32 records per send/receive call. Making the buffer too large has the opposite effect and slows down the computation. Presumably, the hardware fails to exchange very large messages in the background.

To show the influence of the architecture on the performance Figure 6.1 also presents the same experiment on the Linux cluster of the Leibniz Supercomputing Center ². Here, we see a different behavior than the Infinicluster environment. A message size of 32 is not optimal, while 64 and 128 buffer sizes offer best results. Again, we see that large buffer sizes (256) can worsen the performance.

The previous experiment is recommended before performing any parallel simulation in Peano. The bundle size depends on the attributes exchanged per vertex and on the number of messages placed in a bundle. As the attributes exchanged per vertex are PDE dependent, this measurement must be performed for each PDE and computing architecture.

²http://www.lrz-muenchen.de/services/compute/linux-cluster/linux_cluster_intro/index.html

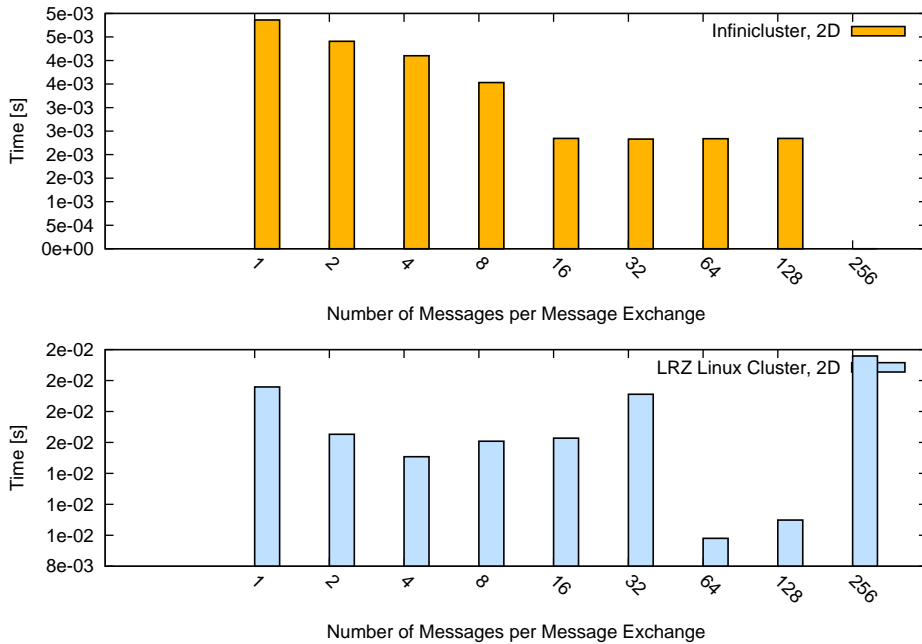


Figure 6.1: Influence of message size on the runtime per vertex for two different architectures.

6.2 Overlapping Communication with Computation

With increasing problem size, the communication will become dominant no matter how fast the communication medium might be. A good parallelization algorithm attempts to delay this by overlapping the communication with intensive computational parts of the algorithm. This ensures neighbor data will already be available during a traversal and no time is wasted inside receive operations. Peano does communication hiding by exchanging each vertex using a *fire-and-forget* semantics, i.e the actual data exchange runs in the background of the traversal [17].

Experiment

To measure how the extent of overlapping of communication and computations, the following experiment is set up. A simulation is run and the ratio between the time spent on waiting for incoming messages and the total execution time of one iteration is calculated. This experiment relies on determining the optimal buffer size from the previous measurement. Figure 6.2 presents the extent of communication hiding in Peano. The ratio is plotted for three buffer sizes. All three measurements follow a similar pattern, and it we see why, in the previous experiment, a buffer size of 32 is marginally better than 64 and 128: there is less wait communication overhead per iteration. Overall, the communication overhead is, on average, 4% of the iteration execution time. This indicates that communication is quite well hidden behind the

computation.

The same experiment was also repeated for the LRZ Linux cluster and the results are presented in figure Figure 6.3. A buffer size of 64 has a smaller communication overhead per iteration and this leads to a faster execution time. I cannot explain why the communication overhead pattern does not resemble the one on the Infiniclust. However, I assume that the hardware setup of the two systems could be responsible.

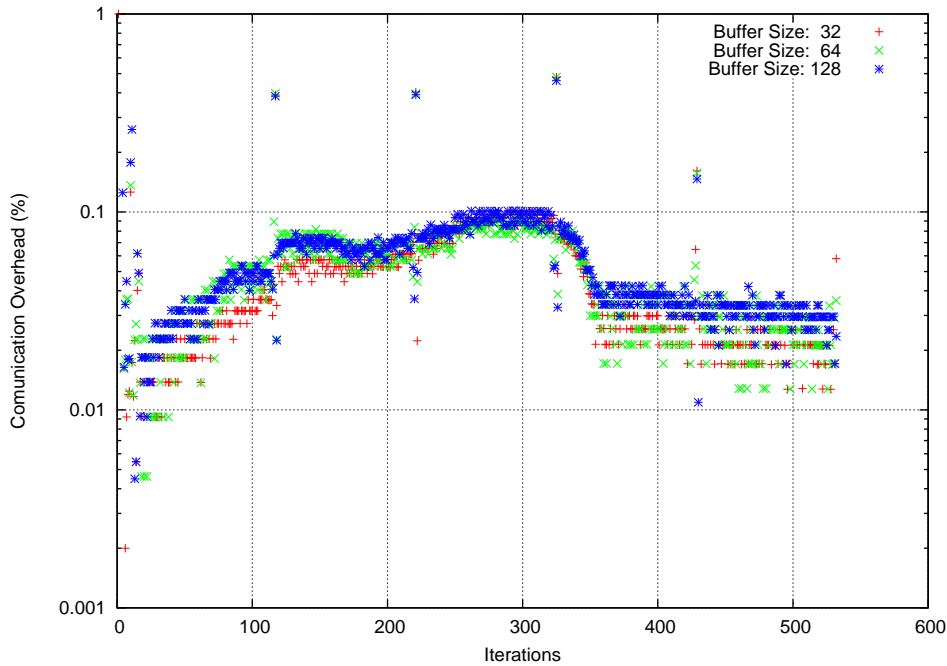


Figure 6.2: Communication overhead for a parallel fluid run on the Infiniclust.

6.3 Strong Speedup

After hardware properties have been studied, speedup measurements are performed. There are many heuristics to compare parallel implementations [1], but the most important measure of a parallel algorithm is the **speedup**. The first type of speedup treated here is the strong speedup which compares the serial execution time against the parallel time needed to solve a particular problem. If T_P denotes the execution time for an algorithm using P processors, and if T_1 gives the execution time for the same algorithm using only one processor, the strong speedup is defined as

$$S_P = \frac{T_1}{T_P}.$$

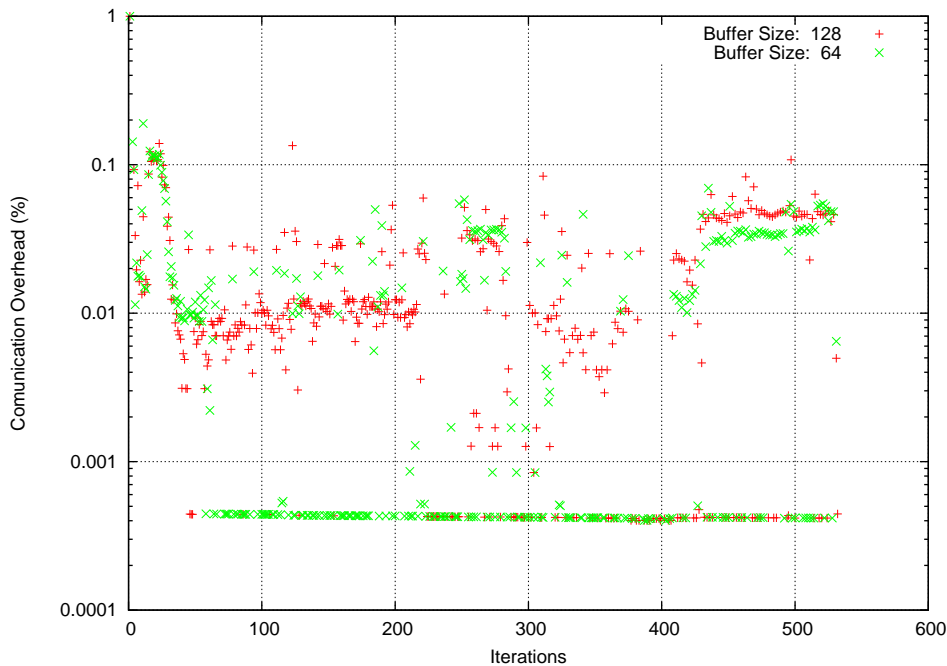


Figure 6.3: Communication overhead for a parallel fluid run on the LRZ Linux cluster.

A related measure is the efficiency E , which is the speedup normalized by the number of processors:

$$\begin{aligned} E_P &= \frac{T_1}{T_P} \\ &= \frac{T_1}{P \cdot T_P} \end{aligned}$$

For an optimal performance we want the speedup to be equal to P , the number of processors. This is called *perfect linear speedup*. Such speedup is hard to achieve due to the so called *serial terms* of any program. These are the part of the program that cannot be run concurrently. A *perfect linear speedup* would translate to an efficiency of 1, meaning the invested P processors are fully contributing to the reduction of the execution time.

Experiment

Figure 6.4 presents the speedup metric for the parallelization of the fluid simulation using the adaptive Cartesian grid. The steps of the experiment are presented in Table 6.1. The step pattern of the speedup curve is best explained when considering the distribution of the parallel spacetree. The tree node on level 0 corresponds to the entire domain. The first branching into 3^d elements means the initial domain is split into 9 (for $d = 2$) equal parts and if 9 computing nodes are available, one expects a linear

6.4. Weak Speedup

speedup. This can be observed in Figure 6.4.

Hardware Environment	Total of 10 nodes each with four AMD Opteron 850 processors at 2.4 GHz with 8 GB of main memory connected by an Infiniband network.
Fluid Scenario	Free flow through an empty square channel with $6.09 \cdot 10^4$ DOFs
Procedure	<ol style="list-style-type: none">1. Perform an initial measurement to determine T_1.2. Run the same problem with more nodes (step of 1)3. Calculate the strong speedup using T_1 as reference time

Table 6.1: Strong speedup experiment setup.

One characteristic of the master-worker model when used within the fluid simulation is that the node with the largest attributed domain partition determines the execution time. This is caused by the work distribution and the resulting wait pattern: A master has to wait for all workers to finish execution and the wait time is given by the slowest worker. In order to efficiently use all the nodes by avoiding idle node time, the load on each must be equal. The speedup for three and four nodes is a very good example for load imbalance. With three nodes, the nine elements can be equally split. With four nodes, work from one of the three nodes having each three elements will be delegated to the fourth available node but this will not speedup the computation as the other two nodes have unchanged (larger) workloads.

Speedup is one metric, but the involved cost/benefit ratio is also important. The efficiency as the ratio between speedup and invested nodes is a good metric for this. Figure 6.5 presents the efficiency for the speedup of the previous experiment. The connection between efficiency and load balancing can be very well observed: while three nodes due to the equal domain partitioning give an efficiency of almost 1, a run with four nodes yields an efficiency of only 0.75. The fourth node could be used for other purposes as it does not contribute to the overall efficiency of the simulation when compared to three node runs. This is however only the case in the current experiment scenario. When using adaptive grid the domain partitioning might be very different and the fourth node could speedup the computation.

6.4 Weak Speedup

The strong speedup performance measurements consider a fixed-sized problem. For fluid simulations the user typically wants to use meshes as fine as possible. Consequently, the effects of parallelization are interesting for an increasing problem size. Ideally, a parallel algorithm should scale with the problem size. For example, increasing the problem size by a factor of two and using now two instead of one processor to

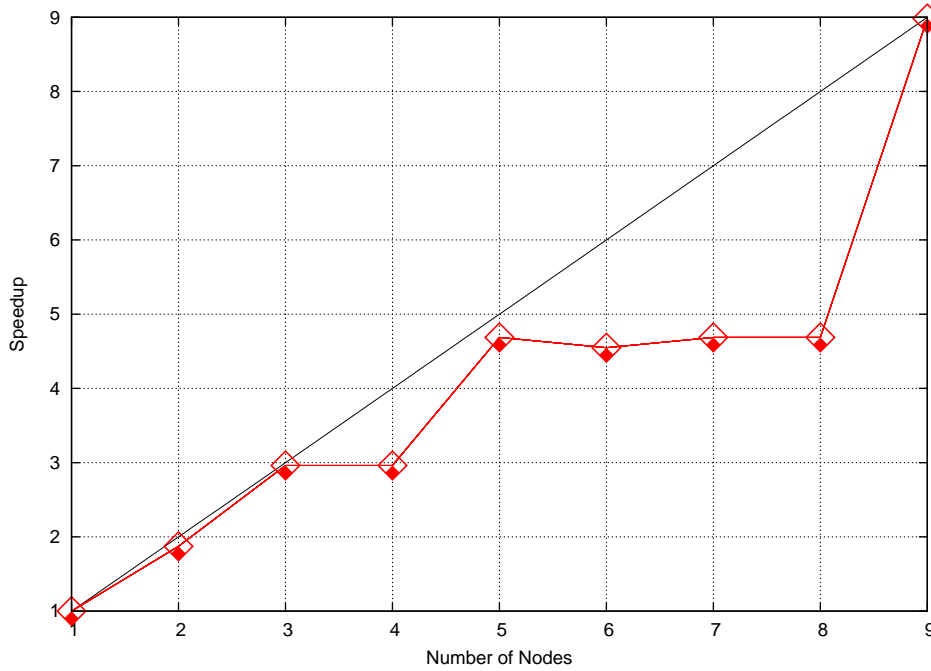


Figure 6.4: Strong speedup for a regular Cartesian grid on the Infiniclust

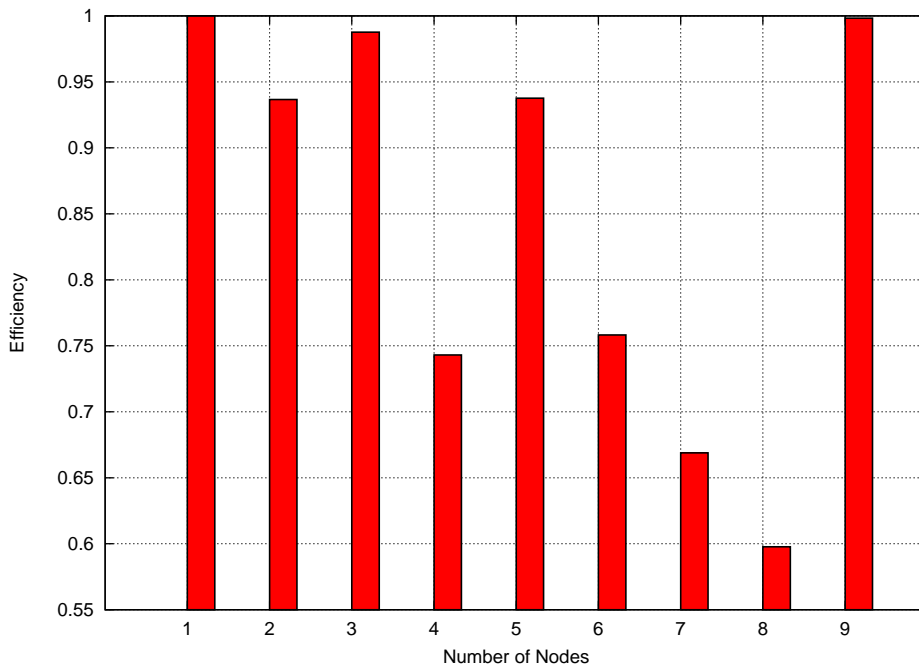


Figure 6.5: Strong efficiency for a regular Cartesian grid on the Infiniclust

solve the now larger problem should result in the same execution time as for running the initial problem size on a single processor.

This notion of speedup is called **weak speedup** and is a more realistic measurement of performance of scalable parallel algorithms on scalable problems. If T_p denotes the execution time of an algorithm using p processor on problem size S , then the weak speedup obtained when processing a problem of size $N \cdot S$ is given by the expression

$$\begin{aligned} S_w &= \frac{\frac{T_1}{S}}{\frac{T_p}{NS}} \\ &= \frac{T_1 \cdot N}{T_p}. \end{aligned}$$

The idea here is to assume that a problem N time larger would need $T_1 \cdot N$ time to execute on a single processor. In reality due to memory bounds and solvers which do not scale linearly, this may not be at all possible.

Experiment

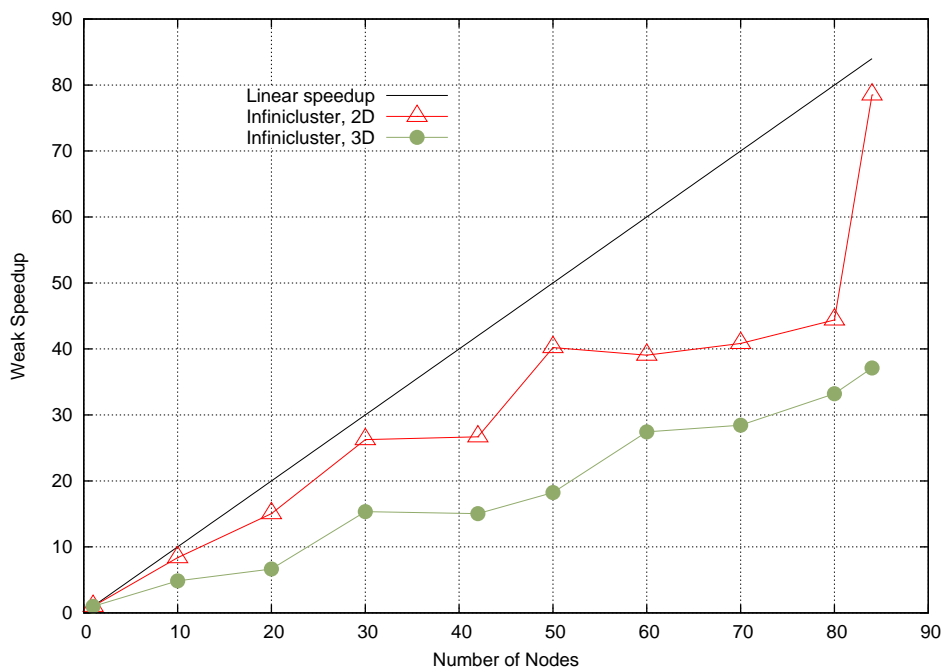
To measure the weak speedup of the parallel fluid simulation an experiment was done on the InfiniCluster by scaling at the same time the problem and the number of nodes. The experiment configuration and procedure are presented in Table 6.2. Figure 6.6 presents the weak speedup performance of the parallel fluid simulation code.

The step pattern resembles the steps in Figure 6.4 scaled up to a bigger number of nodes. The bigger the number of computing nodes and the larger the problem, the more additional nodes are needed to reach the next speedup level. The same observation regarding load balancing can be made here also. If the number of used computing nodes fits to the tree structure, a good speedup is obtained.

The results for a three-dimensional fluid simulation are also presented in Figure 6.6 and 6.7. We see the speedup curve does not show a similar profile to the two-dimensional case. This is due to the partial occupation of the computing nodes. To increase the problem size, we refine one additional level. In Chapter 2, we saw that each cell is refined by cutting each dimension in three. Thus, for 3D, one additional refinement level means a problem size 27 times larger. The maximum number of refinement steps that fitted on the InfiniCluster did not allow for a large enough problem to fully exploit the additional nodes. Refining once more, would create a problem size that does not fit into the memory available on the cluster.

Figure 6.8 and 6.9 show also speedup and efficiency results for the regular Cartesian grid. The implementation of the regular grid does not overlap communication with computation. Still, an efficiency of around 60% is obtained. The regular grid is a test

Hardware Environment	Total of 32 nodes each with four AMD Opteron 850 processors at 2.4 GHz with 8 GB of main memory connected by an Infiniband network
Fluid Scenario	Free flow through an empty square channel with initially $6.09 \cdot 10^4$ degrees of freedom. Scaled problem has $6.09 \cdot 10^5$.
Procedure	<ol style="list-style-type: none"> 1. Perform an initial measurement to determine T_1 on the largest problem size that fits in the memory of a single node. 2. Increase the problem size 9 times and use 10 nodes to compute the now larger problem size. Note that one node is responsible for the parallel coordination and performs no calculations. 3. Run the larger problem size using more nodes (step of 10). 4. Calculate the weak speedup using T_1 as reference time.

Table 6.2: Weak speedup experiment setup.**Figure 6.6:** Weak speedup of the adaptive regular Cartesian grid on the InfiniCluster for 2D and 3D simulations.

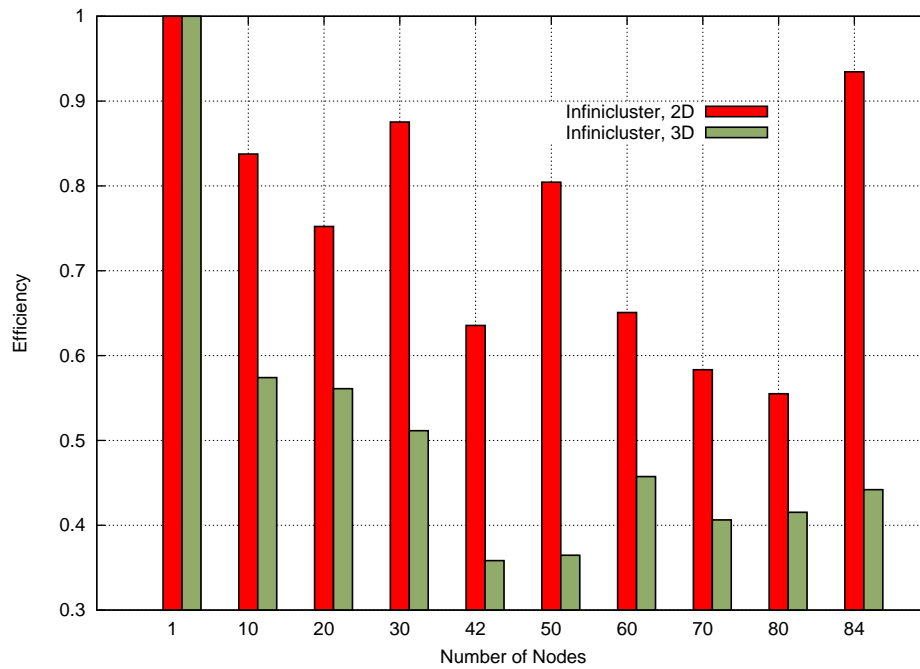


Figure 6.7: Weak efficiency of the adaptive regular Cartesian grid on the Infiniclust.

bed for new applications and extensions. For such purposes, the obtained efficiency should be sufficient.

6.5 Gauss-Seidel vs. Blocked Gauss Seidel

The pressure equation solver is the Gauss-Seidel iterative algorithm. We have seen that in order to run it in parallel it is sufficient to run a normal Gauss-Seidel on each partition with the sole exception that the values at the parallel boundaries will be treated slightly different. Boundaries are updated first with information from the local domain and, before starting the next iteration, also information from neighbor domains is added.

This means the information is propagated with a slower speed over the boundaries. The more partitions the more boundaries and the slower the information spreads. Considered as an algorithm, Gauss-Seidel deteriorates on the borders to a Jacobi iteration. In the worst case a node will be responsible for a domain consisting of a single vertex. The Gauss-Seidel becomes a pure Jacobi iteration over the entire domain. The convergence rate of Gauss-Seidel is approximately twice that of Jacobi, i.e. to reduce the discretization error by a factor of two Jacobi would need twice as many iterations as Gauss-Seidel. Practically, increasing the number of partitions would require more solver iterations to reach a fixed error bound and this way slow down the simulation.

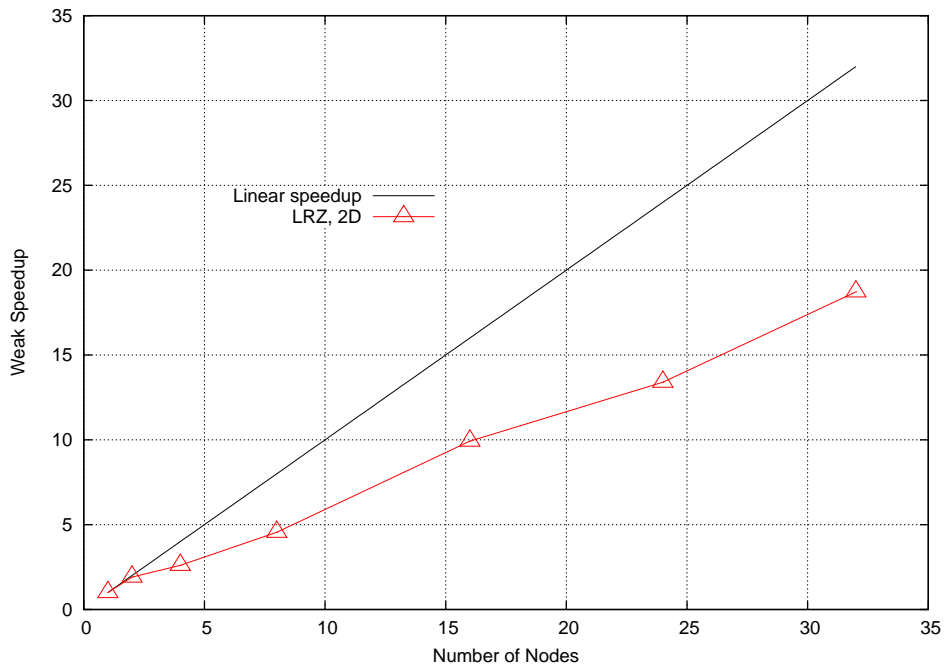


Figure 6.8: Weak speedup of a regular Cartesian grid on the LRZ Linux cluster for 2D simulations.

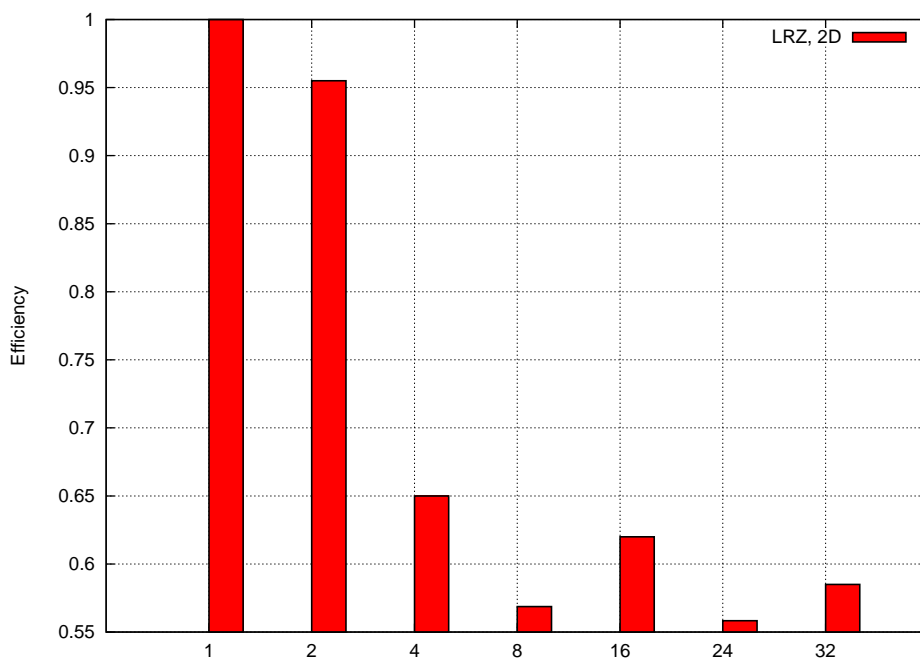


Figure 6.9: Weak efficiency of a regular Cartesian grid on the LRZ Linux cluster.

Experiment

To get an impression on how the number of partitions affects the overall convergence rate of the pressure equation solver and indirect the fluid solution, another experiment was run on the Infiniclust. Using a fluid scenario consisting of an channel with an obstacle in the center (Fig. 6.10) several partition cardinalities were noted with respect to convergence. Figure 6.11 presents the development of the error of the pressure equation along several thousands of fluid iterations for different partition numbers.

The curves of the error for 2 and 9 computing nodes are almost indistinguishable from the serial run. For 18 nodes however, a slightly slower convergence is observed. This is expected as the considered domain size in this experiment is quite small (81x81 cells, adaptive grid) and the ratio of vertices on the parallel boundary (Jacobi update) to inner vertices (Gauss-Seidel update) is larger as with 3 or 9 nodes.

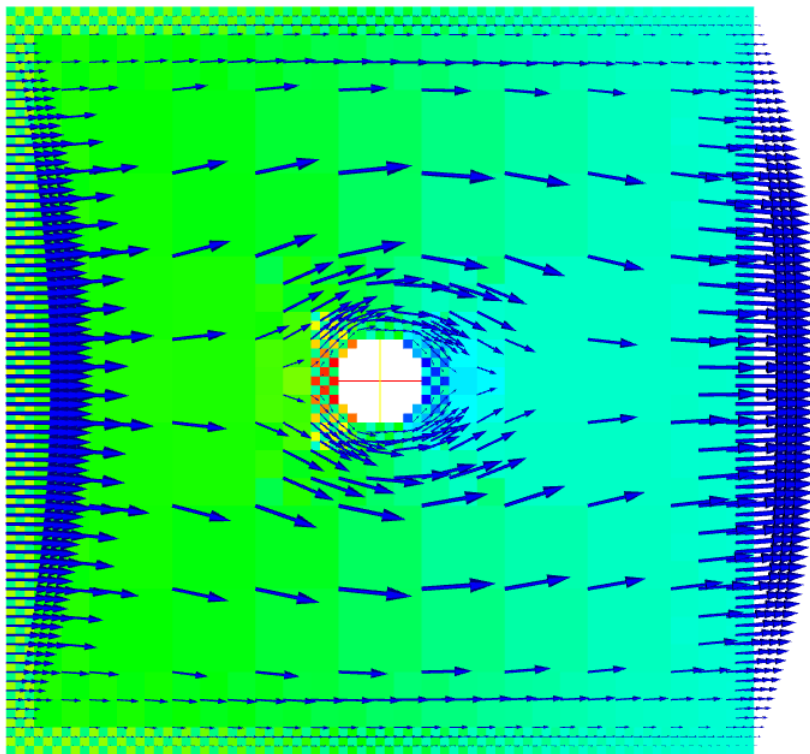


Figure 6.10: Channel with obstacle scenario ran in parallel with 10 nodes. Pressure and velocity vectors (blue) for the steady state are shown. The grid has a minimum mesh size of $1/81$ and maximum of $1/3$. An inflow with a velocity of 1 is applied from the left.

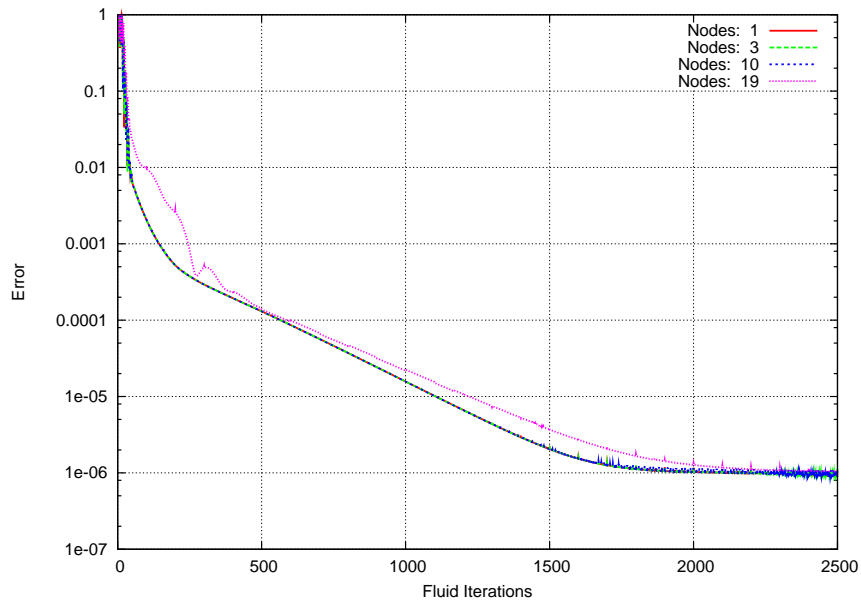


Figure 6.11: Convergence of the pressure equation error for a variable number of partitions. With a larger number of partitions a slightly slower convergence can be observed.

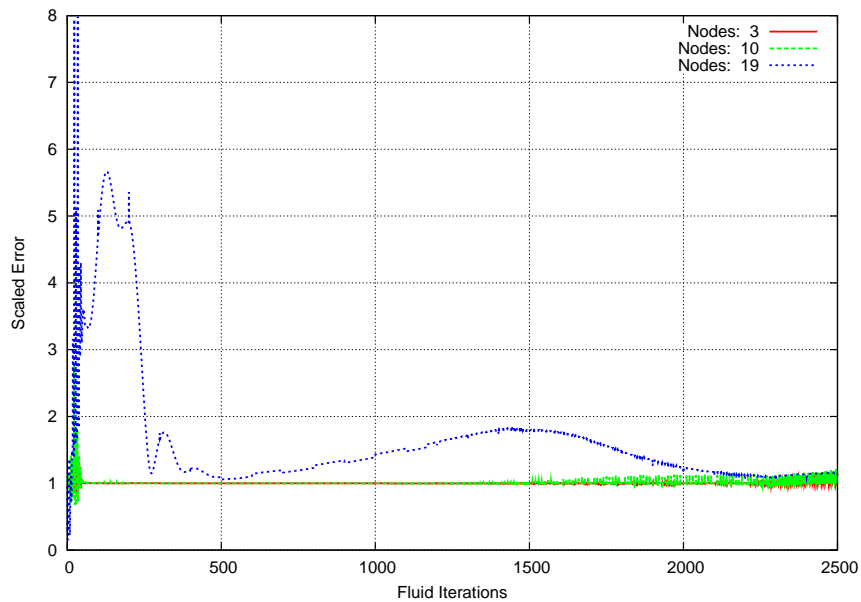


Figure 6.12: Scaled convergence of the pressure equation error for a variable number of partitions.

7 Conclusions and Future Work

This thesis has enabled a fluid dynamics solver to run in parallel within the Peano framework. Using domain partitioning with non-overlapping domains required an analysis of the operators evaluated by the fluid solver on the parallel boundaries to make sure consistency with the sequential algorithm is preserved. We proposed data exchange schemes for the various fluid variables which result in a correct parallel solution. Fluid simulations can now be performed without the help of external libraries such as PETSc by using the parallel version of the matrix-free poisson equation solver. The matrix-free solver shows a slower convergence rate with an increasing number of partitions. However, for its future function, as a smoother in a multigrid solver, this should not be an issue. Measurements have shown good results for the adaptive Regular Cartesian grid under conditions of good load balancing. Taken together, the extensions implemented by this thesis, allow considering larger fluid simulation scenarios and prepare the way for further developments in the Peano framework.

The parallel fluid solver has already been tested in [2] in the context of computational steering. [2] allows interactive steering of a fluid simulation by a graphical user interface that serves as a front-end for the flow solver of Peano. The parallel solver implemented in this thesis was combined with the parallelization of the visualization process. The resulting boost in performance allowed a smoother user experience and also access to larger simulations.

Moving geometries have been recently added to the Peano framework [16]. With this feature, objects of different shapes are placed and moved through the fluid flow. This is a first step in coupling the fluid solver with a structure solver with the future goal of performing simulations of fluid-structure interactions using arbitrary geometries. Performing such simulations also in parallel will be necessary in the future. The moving geometries do not work in parallel out of the box and enabling this is would be an interesting continuation of this thesis.

A Parallel Adaptor

Peano uses the adaptor pattern to perform a grid traversal. Each new operator can be implemented as a new adaptor and several callbacks will be fired to inform the operator where the traversal is at the moment. Based on this, and specific to the operator, an action may be performed or not. If an operator is only interested in working on elements, then the *enterElement()* callback is important while *touchVertexLastTime()* is not. For usage in a parallel traversal of the grid, each adaptor must be extended with some parallel constructs. The listing bellow presents the callbacks and their parallel additions. Of importance to the specific operator are the state information exchange constructs essential for configuring the workers before a traversal starts.

```
void loadSubElement(  
    level,  
    fatherCellVertices[FOUR_POWER_D],  
    fatherCellEventAttributes,  
    fatherCellFirstIndex,  
    fatherCellH,  
    fatherCellPosition,  
    childCellVertices[FOUR_POWER_D],  
    childCellEventAttributes,  
    childCellFirstIndex,  
    childCellH,  
    childCellPosition,  
    // switch the serial traversal automaton with the parallel one  
    parallelTraversalAutomatonProperties  
)  
{  
    // the following line is needed by the dynamic load balancing statistics  
    childCellEventAttributes.setDelta(fatherCellEventAttributes);  
}  
  
void storeSubElement(  
    level,  
    fatherCellVertices[FOUR_POWER_D],  
    fatherCellEventAttributes,  
    fatherCellFirstIndex,  
    fatherCellH,
```

```

    fatherCellPosition,
    childCellVertices[FOUR_POWER_D],
    childCellEventAttributes,
    childCellFirstIndex,
    childCellH,
    childCellPosition,
    // switch the serial traversal automaton with the parallel one
    parallelTraversalAutomatonProperties
){
    // the following line is needed by the dynamic load balancing statistics
    fatherCellEventAttributes.addWeight(childCellEventAttributes);
}

/**
 * This method is called before descending in the spacetree.
 * With regard to parallelization this is the place where state
 * data is sent from master to worker down the tree.
 */
void startStepsDown(
    vertices[FOUR_POWER_D],
    eventAttributes,
    firstIndex,
    h,
    position,
    subvertices[FOUR_POWER_D],
    subEventAttributes[THREE_POWER_D],
    subLevel,
    parallelTraversalAutomatonProperties& traversalAutomatonProperties
){
    // are there any workers responsible for the subelements of the current element?
    if (areThereRemoteNodes(subEventAttributes) ) {

        // if yes then inform them to start a new iteration
        startupRemoteNodes( vertices, eventAttributes, firstIndex, h,
            position, subvertices, subEventAttributes, subLevel,
            traversalAutomatonProperties, _maxJoinsAllowed);

        involvedNodes = getRemoteNodes(subEventAttributes);

        // use the master state to send workers any necessary data for the next step
        SimulationStateOfMaster state;

```

```

// fill in the state object with needed information
state.setState(_currentState);
state.setTau(someTau);
state.setSomeVariable(someVariable);

for (
    InvolvedNodes::const_iterator p = involvedNodes.begin();
    p != involvedNodes.end();
    p++) {

    // send state or fluid information to all remote nodes
    state.send(*p);
}
}

// now perform any local work
}

/**
 * This method is called before ascending in the spacetree.
 * With regard to parallelization this is the place where state
 * data is received from workers down the tree.
 */
void startStepsUp(
    vertices[FOUR_POWER_D],
    eventAttributes,
    firstIndex,
    h,
    position,
    subvertices[FOUR_POWER_D],
    subEventAttributes[THREE_POWER_D],
    subLevel,
    parallelTraversalAutomatonProperties
){
    eventAttributes.clearWeight();
    if (areThereRemoteNodes(subEventAttributes) ) {
        finishRemoteNodes(vertices,
            eventAttributes,
            firstIndex,
            h,
            position,
            subvertices,
            subEventAttributes,

```

```

    subLevel,
    traversalAutomatonProperties);

    InvolvedNodes involvedNodes = getRemoteNodes(subEventAttributes);

    for (
        InvolvedNodes::const_iterator p = involvedNodes.begin();
        p != involvedNodes.end();
        p++
    ) {

        // A worker state object is received from all workers
        SimulationStateOfWorker state;
        state.receive(*p);

        // collect and agregate worker informtion
        _globalResidual = _globalResidual + state.getResidual();
    }
}

forkIfAppropriate(
    vertices,
    eventAttributes,
    firstIndex,
    h,
    position,
    subvertices,
    subEventAttributes,
    subLevel,
    traversalAutomatonProperties,
    _forkFailed,
    _maxTopLevelElementsInConfiguration);

// continue with startStepsUp() semantics
}

/**
 * This method is called after a complete traversal of the spacetree.

```

```
* With regard to parallelization this is the place where a worker
* can send any local state or results to the master.
*/
endTraversal(){

    // only nodes that have a master can send information up the tree
    if ( !parallel::Node::getInstance().isMasterProcessAndNodePool() ) {

        // prepare the data to be sent
        SimulationStateOfWorker state;
        state.setLocalResidual(_residual);
        state.setUmaxPerDimmension(_umax);
        state.setSomeVariable(_someVariable);

        // send the data to the master
        state.send( parallel::NodePool::getInstance().getMasterNodeNumber() );
    }
}
```


B Border Traversal Algorithm for the Regular Cartesian Grid

```
// -1 is a flag for "index can run over entire dimension"
x[3] = {-1 , 0 , _numberOfGridPointsXDim-1};
y[3] = {-1 , 0 , _numberOfGridPointsYDim-1};
z[3] = {-1 , 0 , _numberOfGridPointsZDim-1};

// SEND PHASE

for (int i = 0; i<3;i++) {
  for (int j = 0; j<3;j++) {
    for (int k = 0; k<3;k++) {
      temp(0) = x[i];
      temp(1) = y[j];
      temp(2) = z[k];

      // don't run inside the domain. Only borders are traversed
      if (temp(0) == -1 && temp(1) == -1 && temp(2) == -1) { continue;}

      // calculate the start and end of the communication range
      for (int l = 0; l<DIMENSIONS;l++) {
        // go along this dimension
        if (temp(l) == -1) {
          start(l) = 0;
          end(l) = _numberOfGridPoints(l)-1;
        } else { // this dimension is constant
          start(l) = temp(l);
          end(l) = temp(l);
        }
      }
    }
  }

  // find the CPU we must communicate to
  rank = getNeighborCPURank(temp);

  if (rank != MPI_PROC_NULL && rank != myRank) {
```

```
        // trigger the communication
        sendDataForCoordinateRange(start,end,rank);
    }
}
}
```

```
// PERIODICITY PHASE
// RECEIVE PHASE
```

After the dimension and the indices for the traversal have been fixed for a coordinate range, the effective communication can take place.

```
sendDataForCoordinateRange(IntVector start, IntVector end, int rank)
{
    int vertexIndex;
    IntVector vertexCoordinates;

    for (int i = start(0); i<=end(0);i++) {
        for (int j = start(1); j<=end(1);j++) {
            for (int k = start(2); k<=end(2);k++) {
                vertexCoordinates(2) = k;
                vertexCoordinates(0) = i;
                vertexCoordinates(1) = j;

                // get the vertex corresponding to the current coordinates
                vertexIndex = getVertexIndex(vertexCoordinates);

                // send the vertex
                SendReceiveBufferPool.sendVertex(_data.getVertex(vertexIndex), rank);
            }
        }
    }
}
```

C Hardware

Infinicluster		
Processor		AMD Opteron 850
Location		Chair of Rechnertechnik und Rechnerorganisation Technische Universität München
Peak Performance	Perfor-	4.8 GFlops/s per core
Clock Rate		2.4 GHz
Cores		4 per node
Memory		8 GByte shared between 4 cores
Cluster Size		32
Net		4X Infiniband

LRZ Linux Cluster (Intel)		
Processor		Itanium2 Montecito Dual Core
Location		Leibniz Supercomputing Centre
Peak Performance	Perfor-	12.8 GFlops/s per socket (2 cores)
Clock Rate		1.6 GHz
Cores		2 per socket
Memory		12 GByte shared between 2 cores
Cluster Size		86
Network		Gigabit Ethernet interconnect

Bibliography

- [1] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.
- [2] Atanas Atanasov. Design and implementation for a computational steering framework for cfd simulations. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2009.
- [3] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An evaluation of current high-performance networks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.
- [4] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition*. Prentice Hall, September 2003.
- [5] Hans-Joachim Bungartz, Wolfgang Eckhardt, Miriam Mehl, and Tobias Weinzierl. Dastgen—a data structure generator for parallel c++ hpc software. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 213–222, Berlin, Heidelberg, 2008. Springer-Verlag.
- [6] Hans-Joachim Bungartz, Wolfgang Eckhardt, Tobias Weinzierl, and Christoph Zenger. A precompiler to reduce the memory footprint of multiscale pde solvers in c++. *Future Generation Computer Systems*, May 2009. (in press).
- [7] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems*. John Wiley and Sons Inc., 2003.
- [8] Michael Lieb. A full multigrid implementation on staggered adaptive cartesian grids for the pressure poisson equation in computational fluid dynamics. Master's thesis, Fakultät für Informatik, Technische Universität München, September 2008.
- [9] T. Neunhoeffler M. Griebel, T. Dornseifer. *Numerical Simulations in Fluid Dynamics*. SIAM, 1995.
- [10] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison Wesley, 2004.

- [11] Tobial Neckel. *The PDE Framework Peano: An Environment for Efficient Flow Simulations*. Springer, 2009.
- [12] Tobial Neckel. *The PDE Framework Peano: An Environment for Efficient Flow Simulations*. PhD thesis, Institut für Informatik, Technische Universität München, 2009.
- [13] S.V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980.
- [14] H. Sagan. *Space-filling curves*. Springer-Verlag, 1994.
- [15] L. R. Scott and Dexuan Xie. *Parallel Linear Stationary Iterative Methods*. 1995.
- [16] Kristof Unterweger. Cfd simulation of moving geometries using cartesian grids. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2009.
- [17] Tobial Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. PhD thesis, Institut für Informatik, Technische Universität München, 2009.
- [18] Tobias Weinzierl. Eine cache-optimale implementierung eines navier-stokes lösers unter besonderer berücksichtigung physikalischer erhaltungssätze. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2005.