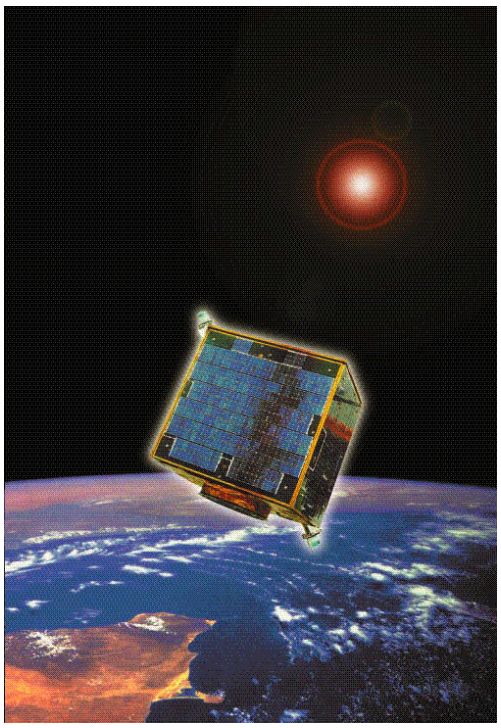
university of groningen / faculty of mathematics and natural sciences

MPI parallelization of the Poisson solver in COMFLO

J.C. Feitsma



Master Thesis in Applied Mathematics

August 2008

MPI parallelization of the Poisson solver in COMFLO

J.C. Feitsma

First supervisor(s): R. Luppes and A.E.P. Veldman
Second supervisor: A.J. van der Schaft

Institute of Mathematics and Computing Science
P.O. Box 407
9700 AK Groningen
The Netherlands

Contents

Chapter 1

Introduction

ComFlo `ComFlo` is a package of simulation software for free-surface flow in terrestrial and micro-gravity environments. It consists of multiple computer programs, developed and maintained since the 1980's by the *Computational Mechanics and Numerical Mathematics* Department of the University of Groningen. `ComFlo` models viscous incompressible flow in and around arbitrary geometries. At the free surface continuity of stresses is imposed; effects of capillarity are included. Also liquid-solid body interaction is included (in some versions).

Why parallelize? When the grid resolution increases, `ComFlo` results should consistently approach the real-world situation better and better. However, the required computer time normally also increases, often unproportionally to gained precision. Moreover, memory limitations will prevent users from using high resolutions.

With the emerging area of grid computing and the introduction of multi-core desktop processors, it is time for `ComFlo` to make its way to the play field of parallel programming. By making use of multiple processors during a simulation, we can achieve results on higher grid resolutions within shorter computation time than before. The process of writing code to divide work over multiple processors is called *parallelization*.

Outline of the thesis In chapter 2, the reader will be introduced to `ComFlo`. We will discuss briefly its history, several applications, and what variants are currently developed. In order to effectively parallelize a large code like `ComFlo`, its most time consuming components will be identified and analyzed. We will see that the pressure iteration procedure (`PRESIT`) is relatively quite costly, making it a main target component to be parallelized.

Chapter 3 treats several general parallelization concepts such as *speedup* and two application program interfaces to facilitate parallelization: MPI and OpenMP.

The next chapter introduces the parallel algorithm `PRESIT-P`, which will be our main weapon to achieve success.

Results and conclusions will be discussed in chapters 5 and 6.

Results The results will demonstrate that even the best possible parallelization effort is destined to fail on systems with distributed memory, as network bandwidth is limited on such systems and the code requires far too much communication time. On shared memory systems however, the code will yield fairly good speedup results, despite some technical anomalies.

Chapter 2

COMFLO

2.1 Liquid simulation

`ComFlo` is a series of Computational Fluid Dynamics (CFD) computer programs to simulate fluid motion. Its theoretical/computational model is based on the Navier-Stokes equations for 3D incompressible free-surface flow. This model includes capillary surface physics as well as coupled solid-liquid interaction dynamics. `ComFlo` consists of several special-purpose computer programs.
Typically, a `ComFlo` user specifies a static geometric 3D layout as well as several parameters, boundary- and initial conditions. This problem setting is translated into a numerical grid with dimensions (n_x, n_y, n_z). Then, `ComFlo` launches a time iteration, during which variables at each grid cell are updated (for instance the velocity vector $\vec{v}$, pressure p and density ρ). Users might be interested in the settled situation after a specific time interval, or in a detailed movie of flow behaviour over time at a certain subregion. After the simulation, the generated raw data is post-processed in order to visualize the results.

2.2 A brief history

The present code `ComFlo` is the successor of the model that was used in the early 1980's as a support to experiments on board SpaceLab (Veldman and Vogels [10]).
In 2005, experiments have been carried out with the satellite 'Sloshsat FLEVO' in an orbit around earth. This mini satellite has been built by the Dutch Aerospace Laboratory (NLR). The experimental data was used to validate numerical simulations performed by `ComFlo` (Veldman et al.[11], Luppes et al.[6, 7]).
Currently, `ComFlo` is also used for maritime, industrial and offshore free-surface flow applications (Fekken [3], Kleefsman et al.[4, 5]).

2.3 Software

Almost all `ComFlo` code is written in Fortran. Some older versions in use are still in F77, while newer versions are now developed in a modular way in F90 and F95.

Current development focuses on a two-phase method to better analyze wave impact in offshore environments [12]. In this thesis, we will work only on `SloshDP`, which is a special-purpose

code for validating the Sloshsat experiment (Veldman et al.[11]).

2.3.1 program structure

The main program loop of a typical `ComFlo` program consists of a time stepping loop. During each iteration, several functions are called in turn, to complete tasks like:

- determine if time step needs to be adapted
- update boundary conditions
- update cell labels (to distinguish full fluid cells from empty cells)
- update velocity vectors
- calculate pressure
- write a snapshot of the data to disk

The magnitude of the time step Δt mainly follows from the CFL stability limit. This means that in the $x-$direction $\Delta t\, U/\Delta x < L$ should hold for stable computations, with U the velocity component in $x-$direction and L the upper limit. Similar expressions should hold for the y and $z-$directions.
A lower bound for the CFL number is also used. During the simulations, the time step is doubled or halved to achieve $0.1 < \mathrm{CFL} < 0.3$. This may cause a deviation from the linear relation between time step refinement and grid refinement.

2.3.2 computational analysis

In this subsection, we will analyze briefly the individual calculation time of `ComFlo` components. With this information, we want to develop a strategy for parallelization. `SloshDP` is taken as reference code. It is one of the older F77 codes around and may be considered quite representative for all one-phase `ComFlo` variants.

Analysis is done with the profiling tool "gprof", which produces a list of percentages of the total time that a simulation has spent in each subroutine of the code. Checks were done on 4 grids: 30*20*20, 60*40*40, 90*60*60 and 120*80*80. Because of the geometry of the water tank inside Sloshsat, with these numbers of grid points the meshes are equidistantly spaced, with approximately equal mesh sizes in each direction: $\Delta x = \Delta y = \Delta z$.
On each grid, 1.0 seconds (real-time) of a typical flat-spin experiment was simulated. The number of time steps required for these grids were 30, 61, 167 and 247, respectively.

When the mesh is refined from 30*20*20 to 120*80*80 grid points, at least 4 times as much time steps are required for the same simulation period of 1.0 seconds. Because of the iterative SOR procedure to solve the Poisson equation for the pressure, the amount of work per time step increases with more than a factor of 4*4*4=64 in this case, as described below. Hence, the simulation time easily increases with more than 2 or 3 orders of magnitude when the grid is refined from 30*20*20 to 120*80*80 grid points.

subroutine	30*20*20	60*40*40	90*60*60	120*80*80
ZEESLAG	22.9 %	57.0 %	80.9 %	86.8 %
FLUIDFORCE	12.9 %	8.0 %	4.3 %	3.1 %
TILDE	12.9 %	6.7 %	3.3 %	2.4 %
VELBC	4.7 %	3.9 %	1.7 %	1.1 %
VFHN	5.6 %	2.7 %	1.1 %	0.7 %
total time	3.5 s	63 s	966 s	4435 s

Table 2.1: The most time-consuming subroutines and the total time for 1.0s simulation of a flat-spin manoeuvre on 4 different grids.

subroutine	description
ZEESLAG	one SOR iteration to solve the Poisson equation
FLUIDFORCE	computation of the fluid forces on the tank wall
TILDE	discretization of the momentum equations
VELBC	boundary conditions for the velocity components
VFHN	displacement of the free-surface

Table 2.2: Description of the most time-consuming subroutines.

In table 2.1 the most time-consuming subroutines are listed, together with the percentage of the total simulation time and the total simulation time itself. A short description of these subroutines is given in table 2.2. Note that the percentages in table 2.1 are not dependant on the number of executed time steps. The percentages only show the relative CPU consumption of the subroutines. It is clear that subroutine ZEESLAG, which takes care of one SOR iteration, becomes the most dominant subroutine with respect to CPU consumption when the mesh is sufficiently refined. Theoretically, the number of SOR iterations required per time step depends somewhere between linearly and quadratically on the number of grid points. Moreover, the number of operations per iteration increases cubically (in 3D simulations) in case of grid refinement in each coordinate direction. Hence, on the fine grids that are required for accurate simulations, the SOR iterations are the most time-consuming element of a simulation.

In the present project, parallelization of PRESIT will be subject of study. As there is recurrence in each SOR iteration, this parallelization is certainly not trivial, and a thorough study is required. The parallelization of the other subroutines, which in most cases consist of simple loops that can be parallelized trivially, shall be left for future parallelization projects.

Chapter 3

Parallelization

Traditionally, computers execute program instructions in a sequential fashion. Such kind of computers only have a single processor core. When programmers design an algorithm, this results in a sequence of steps, each step built on top of the result achieved by the previous step.

Parallel computing is a computing method that uses multiple cores within a single program; *parallelization* is the process of adapting a sequential program to be run on multiple cores. For many years, parallel computing was only applied by researchers on "exotic" supercomputers like Cray. During the past few years however, a significant shift towards commercial applications is seen. As processor manufacturers tend to develop multi-core processors rather than improve single core processor speeds, parallelization has made its way to the general public.

3.1 Advantages

The main advantage of parallelization is a possible speedup of program execution time. Suppose a certain parallel program requires $t_1 = 60$ minutes of computing time on a single-core processor. If we run it on two cores, ideally we would expect a runtime of $t_2 = 30$ minutes. In that case, the *speedup* would be 2 out of 2. Generally, the speedup on n cores is defined as follows:

$$s(n) = \frac{t_1}{t_n}$$

The extent to which ideal speedup can be achieved for a certain algorithm, depends on the parallelizable part of the algorithm. If we decompose $t_1 = t_{seq} + t_{par}$, then

$$t_n = t_{seq} + \frac{t_{par}}{n},$$

$$s_n = \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{n}},$$

$$\lim_{n\to\infty} s_n = \frac{t_{seq} + t_{par}}{t_{seq}} = \frac{1}{1-P}$$

with $P = \frac{t_{par}}{t_1}$ the parallelizable portion. The existence of this limit is known by *Amdahl's law* [1].

Another possible advantage of parallelization is that a program can make use of the aggregate memory of multiple separate computers at once.

3.2 Disadvantages

The main problems with parallel programming are based on interprocessor communication. Depending on the type of algorithm, each involved core may need to communicate with one or more other cores. This communication may slow down overall computation, especially when bandwidth is limited.

Programmers need to design the code very carefully to avoid deadlocks and race conditions. A *deadlock* is the situation that a scheduled data transmission never happens, because one of the cores that should send or receive data is not ready to do so, and never will be. Figure 3.2 illustrates a simple deadlock example on two cores, p_0 and p_1. In the left side situation, both cores are waiting for the other to go into listening mode, which will never happen. A corrected version is shown on the right side.

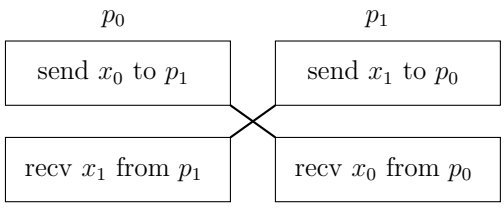


Figure 3.1: Deadlock example.

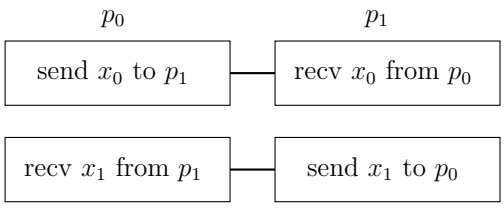


Figure 3.2: Correct use.

Race conditions may occur in a shared memory setting, i.e. when multiple cores have write access to the same variable. We can visualize this problem by a couple of horses racing to a finish line, where the outcome depends on whichever horse finishes first. As this problem will be of no concern to us, we will not go into further details.

3.3 Programming paradigms

When a parallel program starts, all cores execute the same code. During execution, each core is associated with an integer, so the cores can be distinguished from each other. Otherwise, they would all do the same thing, having no way of knowing about each other.

In this section, we will introduce two application program interfaces (API's) for parallel programming, as well as the parallel programming *master-slave* model.

3.3.1 master and slaves

In some cases, the task of dividing the work amongst all available cores is done by one special core. This core is called the *master*, as it controls the other cores, its *slaves*. Typically, the slaves await commands from the master, do their work and report back when done. The effectivity of this model depends on the way the work can be evenly distributed - if the master assigns more work to one slave than another, it may have to wait until the last slave has finished. The administrative tasks of distributing work and gathering results should be negligible compared to the actual computational work.

We will apply the master-slave model in section 4.2.2.

3.3.2 OpenMP

OpenMP (Open Multi-Processing) is an API supporting multi-platform shared-memory parallel programming in C/C++ and Fortran [8]. The programmer issues parallelization directives to the compiler, which works out the details. This allows for a moderately high-level of abstraction, as we trust in the compiler to take care of certain technical issues. Performance may differ, depending on the quality of the compiler.

The application of OpenMP to `ComFlo` is not within the scope of this thesis. There are plans to investigate this in the beginning of 2009.

3.3.3 MPI

MPI (Message Passing Interface, [9]) is an API specification for parallel programming on distributed memory systems. A MPI implementation in a given programming language offers a wide range of functions, from performing basic tasks like sending and receiving data, to more advanced functions.

Data typically needs to travel across a network from one core to any other. MPI can also be applied however on a shared memory system by subdividing the memory over all cores, effectively unsharing the memory. In this case, data transmission boils down to a mere copy of memory, which can be realized far more efficiently than any network transmission ever would. Hybrid constructions are also possible with MPI, for instance groups of cores on SMP machines collaborating intensively on a low level, while keeping in touch on a higher level across a network.

In this thesis we will focus only on using MPI to parallelize `ComFlo`. Below, we will show a MPI Fortran version `helloworld.f` of the famous program *Hello, world!*, to illustrate how MPI is typically used.

```
0       PROGRAM helloworld
        IMPLICIT NONE
        INCLUDE "mpif.h"

        INTEGER nrprocs     ! total number of cores
5       INTEGER noderank    ! rank of this core
        INTEGER err         ! error indicator

        CALL MPI_INIT(err) ! initialize
        CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nrprocs, err)
10      CALL MPI_COMM_RANK(MPI_COMM_WORLD, noderank, err)

        PRINT *, 'Hello World from node ', noderank, ' of ', nrprocs, '!'

        CALL MPI_FINALIZE(err)
15
        END ! end of program
```

Systems which have MPI installed, often provide compiler extensions which take care of the required header inclusion and library paths. On the `HPCIBM1` cluster at the University for Groningen, we can simply compile the program using the command `mpif77 helloworld.f`. The program can be started on for instance 3 nodes by invoking `mpirun -np 3 a.out`.

Chapter 4

PRESIT parallelized

In chapter 2.3.2 we have seen that the `PRESIT` component is by far the most computationally expensive part of a typical `ComFlo` simulation. A 90% portion of total simulation time is not uncommon. This is a first strong reason to investigate parallelization of `PRESIT`. Secondly, because this function iterates many times through the numerical grid, a grid decomposition strategy seems to be a natural way to get us started.

In this chapter, the new parallel algorithm called `PRESIT-P` will be introduced, based on the original un-parallelized code. Also, some technical implementation notes will be mentioned.

4.1 Prerequisites and features

Of course, the primary goal of `PRESIT-P` is to achieve a significant speedup on relatively large simulations. The extent to which this goal is achieved may be used to decide whether or not to spend more time in parallelizing other `ComFlo` components. Numerical tests will also show on which computer systems `PRESIT-P` performs the best.

Besides the main speedup objective, several secondary goals can be distinguished. Some were listed before our research even started, some were added during the code development whenever they came forth.

- reusable parallel program flow model
  Since more `ComFlo` components may be parallelized in the future, all nodes which are passive at a certain moment should be easily activated for whatever task is assigned to them. This requirement is met by employing a master-slave flow model, as introduced in section 3.3.1. For a detailed treatment, see section 4.2.2.

- minimal code change
  The process of integrating the new parallel component into an existing un-parallelized `ComFlo` code should require minimal effort. Users (or even developers) who want to benefit from the speedup `PRESIT-P` offers, should not have to be experts in parallel programming to actually use it in their own code.

- documentation
  Evidently, the code must be documented properly for future use. This is closely related

to the previous item. The question *How do I use PRESIT-P?* should have a clear easy answer. Part of this documentation will be found in this thesis of course, while the code itself is also thoroughly documented.

- consistent iteration behaviour
  During code development, a technical problem emerged. An intermediate version of PRESIT-P showed very good speedup results per iteration, yet convergence slowed down drastically, annihilating the speedup. The reason this problem occurred was the new order of iterating through the grid cells. By making sure the original numerical iteration process was reproduced, the problem was solved. More on this matter can be found in section 4.2.4 about *red/black ordering*.

4.2 Algorithm

First, let's take a look at the original PRESIT procedure, which will be the basis of our work. If we want to keep the amount of changes to the main ComFlo routine minimal, the parallel algorithm should resemble the original algorithm as much as possible.

4.2.1 un-parallelized algorithm

In the original PRESIT algorithm, the main PRESIT routine controls the SOR iteration process, calling SLAG as many times as required in order to converge. The routine SLAG iterates exactly once through the numerical grid, updating all pressure values. Let's assume the grid dimensions are $n_x \times n_y \times n_z$ and cells are labeled $i \in \{1, n_x\},\ j \in \{1, n_y\},\ k \in \{1, n_z\}$.

Within the pressure iteration, only *interior* cell values are updated. This boils down to the following:

```
do for all  i ∈ {2, n_x − 1},  j ∈ {2, n_y − 1},  k ∈ {2, n_z − 1}
in some order to be specified:
```

$$
\begin{aligned}
\mathrm{diff}(i,j,k) \;:=\; & \mathrm{div}(i,j,k) - p(i,j,k) \\
& - c_{xl}(i,j,k)\cdot p(i-1,j,k) - c_{xr}(i,j,k)\cdot p(i+1,j,k) \\
& - c_{yl}(i,j,k)\cdot p(i,j-1,k) - c_{yr}(i,j,k)\cdot p(i,j+1,k) \\
& - c_{zl}(i,j,k)\cdot p(i,j,k-1) - c_{zr}(i,j,k)\cdot p(i,j,k+1) \\
p(i,j,k) \;:=\; & p(i,j,k) + \omega \cdot \mathrm{diff}(i,j,k)
\end{aligned}
$$

It should be noted that only the six direct neighbours are involved, each with a certain given coefficient. The values of these coefficients as well as the value of the divergence are calculated by other ComFlo routines.

At the end of a SLAG iteration, two values are reported back to PRESIT. These variables, maxdiff and delta, are used to determine whether or not the iteration process needs to be aborted, when either convergence is achieved or the iteration process has failed. Moreover, the value of the SOR parameter ω may be altered based on the calculated residuals.

$$\text{maxdiff} := \max_{(i,j,k)\text{interior}} \left| \frac{\text{diff}(i,j,k)}{p(i,j,k)} \right|$$

$$\text{delta} := \left(\sum_{(i,j,k)\text{interior}} \text{diff}(i,j,k)^2 \right)^{\frac{1}{2}}$$

After many years of research by the Department of Numerical Mathematics, a strategy for choosing ω has been developed [2] which enables not only a robust but also a fast iteration process. These features have to be inherited by the parallel version `PRESIT-P`.

Figure 4.1 shows a schematical summary of `PRESIT`.

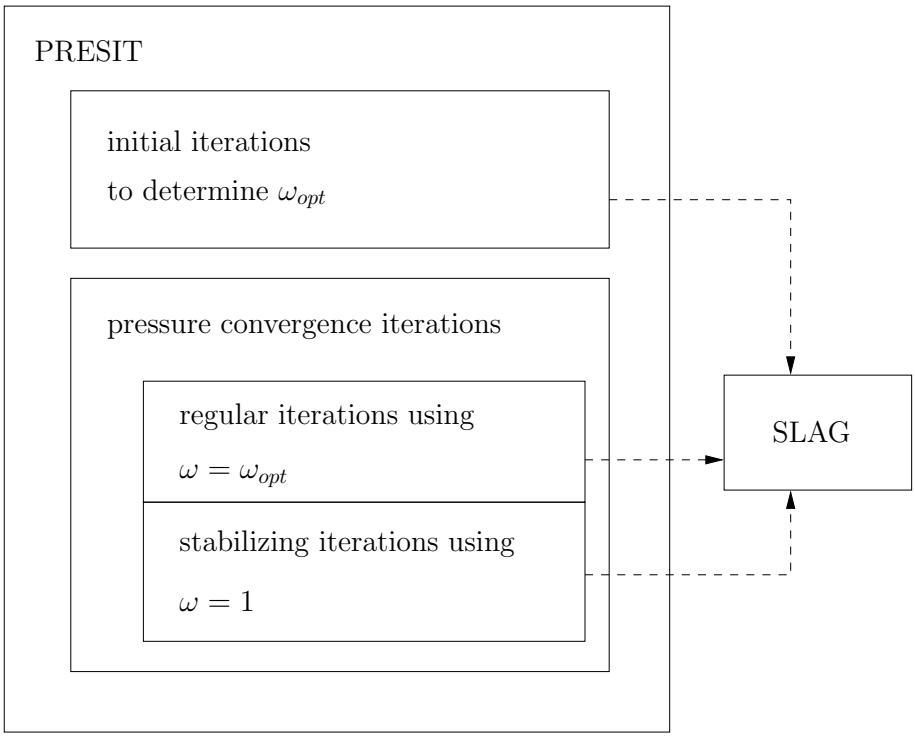


Figure 4.1: `PRESIT` schematics

In the following subsections, we will work towards the parallelized version of `SLAG`, called `PSLAG`. This routine will be the most important part of `PRESIT-P`.

4.2.2 master and slaves

Suppose there are M nodes available and they are numbered $m \in \{0, \ldots, M-1\}$. We apply the previously introduced *master-slaves* model in `PRESIT-P` (see chapter 3.3.2). The master is the one and only node executing the main `ComFlo` code, usually with rank 0. In the meantime, all $M-1$ slaves are put in a dormant state, waiting for an activation signal from the master

node in a so-called *slave-loop*. The benefit of this paradigm is that it allows the slaves to be used in other components as well, should they be parallelized in the future.

At the beginning of `PRESIT-P`, the grid cells are equally partitioned in strips by the master over all nodes. Since the administrative tasks of the master node are negligible compared to the grid iteration, the master also assigns a strip to itself.

Let's assume for the time being that the grid is partitioned in the z dimension. The strip S_m consists of interior cells assigned to node m:

$$S_m = \{(i,j,k) \ : \ i \in \{i_{lo}, i_{up}\},\ j \in \{j_{lo}, j_{up}\},\ k \in \{k_{lo}, k_{up}\}\}$$

with

$$\begin{aligned} i_{lo} &= 2 \\ i_{up} &= n_x - 1 \\ j_{lo} &= 2 \\ j_{up} &= n_y - 1 \\ k_{lo} &= \lfloor (n_z - 2) \cdot m/M \rfloor + 2 \\ k_{up} &= \lfloor (n_z - 2) \cdot (m+1)/M \rfloor + 1 \end{aligned}$$

4.2.3 interaction

We now introduce the *boundary planes* $B_{m,1}, B_{m,2}, B_{m,3}, B_{m,4}$ for node m. For each plane, we take $i \in \{i_{lo}, i_{up}\}$ and $j \in \{j_{lo}, j_{up}\}$.

$$\begin{aligned} B_{m,1} &= \{(i,j,k_{lo}-1)\} \\ B_{m,2} &= \{(i,j,k_{lo})\} \\ B_{m,3} &= \{(i,j,k_{up})\} \\ B_{m,4} &= \{(i,j,k_{up}+1)\} \end{aligned}$$

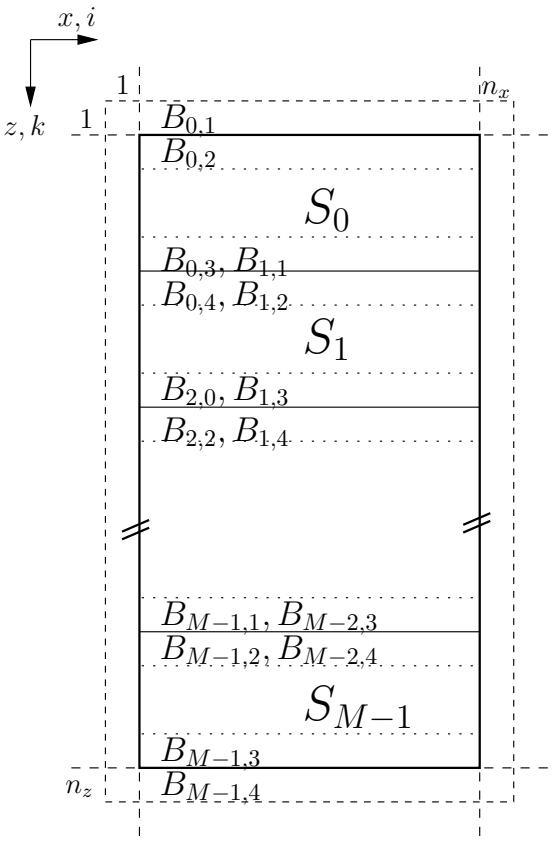


Figure 4.2: strips and boundary planes, y dimension omitted

Within `PSLAG` each slave at some point needs to send the data from its planes B_2 and B_3 to the corresponding neighbours, which will store the planes at respectively B_4 or B_1. Throughout the next two sections we will explain this in more detail.

At the end of `PSLAG`, the master will be responsible for gathering `maxdiff` and `delta`, so `PRESIT-P` can use these values as if they were produced by the original un-parallelized code. This is accomplished through the standard MPI reduction routine `MPI_REDUCE`.

4.2.4 red/black ordering

How should the nodes iterate through their numerical domain? At first sight, one might think that a linear grid traversal would be the easiest method, for instance with k iterating in the outermost loop, and i, j in the inner loops. Even though the original un-parallelized code uses a red/black ordering, we initially performed some tests with the linear ordering, which indicated that this method yields good speedup results per iteration. A parallel algorithm called PSOR [13] has been devised and used for this grid ordering.

Unfortunately, the linear ordering cannot be applied. Despite the fact that speedup within one single iteration showed to be good (if not optimal), the calculated values of `maxdiff` and `delta` disrupted the strategy for choosing ω within the `PRESIT` routine. This lead to an unstable iteration process within `PRESIT`: the number of required iterations increased with the number of nodes M.
We concluded that the red/black ordering in the original `SLAG` could not be circumvented (see also the fourth prerequisite in the previous section). Theoretically, the reproduced numerical process should yield exact equal values of `maxdiff` and `delta` as the original code would have delivered. In practice, numerical errors will slightly distort the values, but this will have a marginal influence in the number of `PRESIT` iterations, as we will see in the chapter on results.

Let us examine more closely the grid iteration order. All red values will be updated first, using only black neighbour values besides themselves. No red values are directly depending on each other during one iteration. Thus, they can all be updated in parallel.
Once all red values have been updated, an interaction step seems necessary for the black values at B_2 and B_3 since they need the updated red values. This presents us with a substantial problem, as interaction costs are normally quite large with respect to the calculation costs. We might split the amount of data to be sent in two by applying a stride, but this still requires more interaction than we would like to see.

4.2.5 correction phase minimizes communication

Dropping the interaction step between calculating the red and black values will result in contamination of some of the black values, namely those in all four boundary planes. After the boundary planes are exchanged, a correction phase is required to set the values straight.

Consider a contaminated black value $p(i, j, k_{lo})$ at B_2. It is calculated using an old red value $p^o(i, j, k_{lo} - 1)$. During the interaction phase, this old value is overwritten by $p^n(i, j, k_{lo} - 1)$, which is exactly the value that should have been used during the black update phase. As we can see from the update recipe from section 4.2.1, the difference between the values needs to be multiplied by ω as well as the corresponding coefficient, c_{zl} in this case. The correction term ε for $p(i, j, k_{lo})$ then becomes

$$\varepsilon = -\omega * c_{zl}(i, j, k_{lo}) * (p^n(i, j, k_{lo} - 1) - p^o(i, j, k_{lo} - 1)) .$$

A similar argument holds for the other three planes. The correction terms ε in the black values after the interaction phase are given by

$$
\begin{aligned}
B_{m,1} : \varepsilon(i,j,k_{lo}-1) &= -\omega * c_{zr}(i,j,k_{lo}-1) * (p^*(i,j,k_{lo}) - p^o(i,j,k_{lo})) \\
B_{m,2} : \varepsilon(i,j,k_{lo}) &= -\omega * c_{zl}(i,j,k_{lo}) * (p^*(i,j,k_{lo}-1) - p^o(i,j,k_{lo}-1)) \\
B_{m,3} : \varepsilon(i,j,k_{up}) &= -\omega * c_{zr}(i,j,k_{up}) * (p^*(i,j,k_{up}+1) - p^o(i,j,k_{up}+1)) \\
B_{m,4} : \varepsilon(i,j,k_{up}+1) &= -\omega * c_{zl}(i,j,k_{up}+1) * (p^*(i,j,k_{up}) - p^o(i,j,k_{up}))
\end{aligned}
$$

The correction phase adds these terms to the corresponding black values, so each node ends up with the latest correct values without having to perform extra interaction.

4.2.6 `PSLAG`

Previous subsections can now be combined into the new `PSLAG` routine. Before the first call to `PSLAG`, the strips will be assigned to each slave by the master node and all data (coefficients, divergence) will be distributed. This work is done in `PPRESIT_INIT`.

The new `PSLAG` routine consists of the following phases.

- *initialization*
  Slaves receive a signal from the master to help with the pressure iteration. Several variables are initialized, ω is distributed. Each node stores the four boundary planes B_1, B_2, B_3 and B_4 for later use during the correction phase.

- *updating of red values*
  Each node updates the red values in its grid strip.

- *partial update of black values*
  Update all assigned black values, including those at the boundary planes which will need correction.

- *interaction*
  Send $B_{m,2}$ to node $m-1$ while receiving $B_{m,4}$ from node $m+1$. Send $B_{m,3}$ to node $m+1$ while receiving $B_{m,1}$ from node $m-1$.

- *correction of black values*
  Correct the black values in all four boundary planes by using the just received data and the values that were stored at the initialization phase.

- *finalization*
  The slaves report their partial values of `maxdiff` and `delta` to the master, which delivers them to the `PRESIT` control routine.

4.3 Implementation

All `PRESIT-P`-code is available by means of a special module `cfmpi_mod.f`.

4.3.1 data memory-alignment

In the previous section, we have assumed that grid partitioning is done along the z dimension. The main advantage of this choice is the convenient memory alignment of values to be transmitted. When transmitting a block of data, we need to specify a contiguous array of data, namely the (memory address of) the first element and the number of elements to send/receive. Partitioning in the z dimension does not require an array reshape operation, as the boundary planes are already stored contiguously in memory.

However, if $n_x > n_z$ we would rather partition in the x dimension as this minimizes the amount of data to be transmitted. It would seem that this approach requires memory reshape operations before and after the transmission of a boundary plane. Fortunately, by transposing the entire system (pressure values, coefficients, divergence) before the first `PSLAG` call, we can leave the code in `PSLAG` intact and thus benefit from optimal memory alignment. The performance penalty of this transposition in `PPRESIT_INIT` will prove to be negligible.

Throughout the code, we will use special variable names `P2`, `DIV2`, etcetera for the transposed system.

4.3.2 MPI specifics

During the interaction phase within `PSLAG`, the following code is executed. (For details on the arguments to `MPI_SENDRECV`, please refer to the MPI manual.)

```
! the number of elements in a full XY-plane
elcount = (iup - ilo + 3) * (jup - jlo + 3)

! send B3, receive B1
CALL MPI_SENDRECV(P2(ilo-1,jlo-1,kup   ), elcount,
                  MPI_DOUBLE_PRECISION, mpiNextNode, 707,
                  P2(ilo-1,jlo-1,klo-1), elcount,
                  MPI_DOUBLE_PRECISION, mpiPrevNode, 707,
                  mpiActiveComm, mpiStatus, mpiErr)

! send B2, receive B4
CALL MPI_SENDRECV(P2(ilo-1,jlo-1,klo   ), elcount,
                  MPI_DOUBLE_PRECISION, mpiPrevNode, 707,
                  P2(ilo-1,jlo-1,kup+1), elcount,
                  MPI_DOUBLE_PRECISION, mpiNextNode, 707,
                  mpiActiveComm, mpiStatus, mpiErr)
```

A call to `MPI_SENDRECV` is equivalent to a simultaneous blocking combined send and receive operation. The efficiency of this call is dependent on the underlying MPI implementation. Perhaps it is worth the effort to explicitly decompose this block of code. On the other hand, we should not resort to this kind of tweaks, as it degrades code clarity and may very well prove to have no positive effect on performance.

Other possible improvements in the interaction phase would be to use non-blocking routines, and/or to send packed data. These improvements are not explored in this thesis.

4.3.3 memory limitations

In the old-fashioned single-node setting, all cell variables are maintained in several large static 3D arrays. This does not translate well into the parallel setting, as every node will allocate far more memory than required. Therefore, the memory blocks that are used in `PRESIT-P` will be allocated dynamically whenever required via F90 modules `CFDYNMEM_MOD` and `CFSTATMEM_MOD`.

4.4 Embedding the code

In this section, we will touch briefly on how the `PRESIT-P` component has been incorporated in the original `sloshdp` application. These adaptations can be viewed as a guideline to build `PRESIT-P` into other `ComFlo` programs.

4.4.1 main procedure

The first step is to initialize MPI in the main routine. This is accomplished by loading the required modules, and placing a piece of code just below the variable devlarations. An example follows.

```
PROGRAM COMFLO

! load required modules
USE CFMPI_MOD
USE CFDYNMEM_MOD
USE CFSTATMEM_MOD

! other includes, local variable declarations
! ...


! start of main procedure

! all nodes: initialize MPI (see cfmpi_mod.f)
CALL CFMPI_INIT

! the number of nodes to use in ppresit
mpiNumActiveNodes = mpiNumTotalNodes

! activate nodes
CALL CFMPI_SET_ACTIVE

! put slaves into passive/listen-mode
IF (mpiNodeRank > 0) THEN
   CALL CFMPI_SLAVELOOP
   ! slaves only exit the loop when the master tells them
   ! no more work will come
   CALL CFMPI_FINALIZE
   ! slaves should abort at this point
   STOP
```

```
30 END IF

   ! remainder of main procedure
   ! ...
```

The module `CFMPI_MOD` is the extension for the standard MPI module we have built.
It is very important that the slaves are kept clear from the main procedure after finishing the slave-loop. If they accidentally execute the main procedure, the following things may happen:

- superfluous work: the slaves will perform the same operations as the master node
- multiple masters: all nodes may assume the master role at some point in the code
- I/O problems: multiple nodes may write to the same file at the same moment, leading to I/O errors

4.4.2 `PRESIT` procedure

Changes to the `PRESIT` routine are quite straightforward.

- Initialize `PRESIT-P` by calling `PPRESIT_INIT` at the beginning. In this routine, the master will wake up the slaves and distribute the grid strips.
- Replace each call to `SLAG` with a call to `PSLAG`.
- Finalize `PRESIT-P` by calling `PPRESIT_FINISH` at the end. This routine will make each slave send the new pressure values from its strip to the master.

A schematic view is given in figure 4.3.

4.4.3 global data

As mentioned in the subsection on memory limitations, `PRESIT-P` requires two memory modules. This may lead to some minor modifications in the main `ComFlo` code, especially if global variables are managed via `COMMON`-blocks.

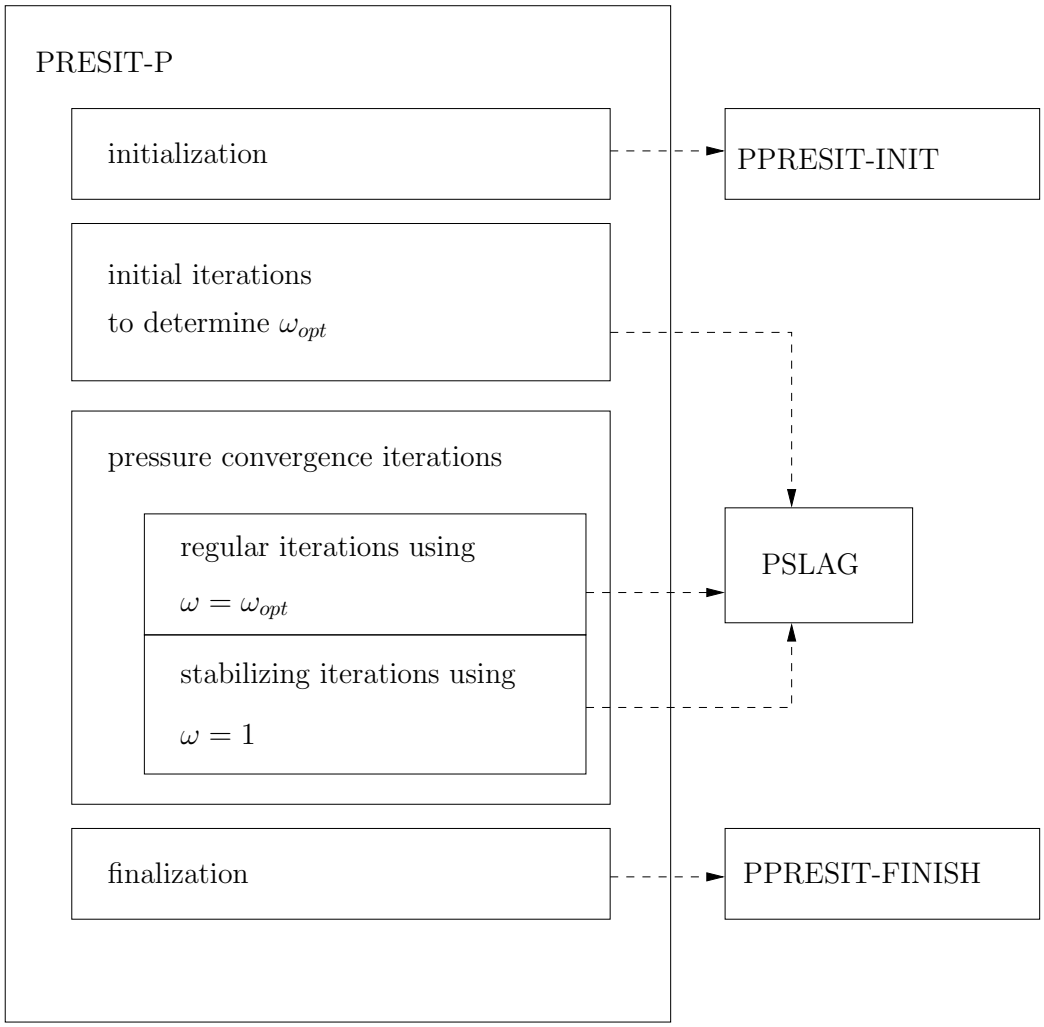


Figure 4.3: `PRESIT-P` schematics

Chapter 5

Results

In this chapter, we will present speedup measurements of `PRESIT-P` on two different machines.

- `HPCIBM1`
  The *Opteron Cluster* (also known as `HPCIBM1`) consists of 200 nodes, each having a dual-core AMD Opteron processor. Most nodes have 1GB memory, some special nodes are equipped with 4GB. Point-to-point bandwidth is estimated at 22ms/MB.
- `SI01`
  The other machine is a large SMP (Shared Memory Processor) machine called `SI01`. It consists of 4 quad-core processors and is equipped with 128GB of shared memory.

All timing- and speedup measurements are taken from the first 10 `PRESIT` iterations of `SloshDP`, after which the program simply aborts. The original `SloshDP`-code is used as basis to calculate the various speedup variants.

5.1 Notation

The following symbols are used throughout the tables, figures and accompanying text. Time is always measured in seconds.

- n_p: number of processors in MPI-mode $np =$`orig` designates the original code.
- m_1: the total number of `PRESIT-P` iterations, normally fixed at $m_1 = 10$.
- m_2: total number of `PSLAG` iterations during the m_1 `PRESIT-P` iterations.
- t_{tot}: total elapsed time of the code (after m_1 `PRESIT` iterations).
- t_{par}: *parallel time*, the part of t_{tot} spent in `PRESIT-P`.
- %par $= t_{par}/t_{tot}$.
- t_{seq}: *sequential time*, i.e. the portion that cannot be reduced by parallelization, thus

  $$t_{seq} = t_{tot} - t_{par}.$$

- s_1: standard speedup based on t_{tot}:

$$s_1 = \frac{t_{tot}(\text{orig})}{t_{tot}(n_p)}.$$

- s_2: speedup of the parallel-only portion of the code, i.e.

$$s_2 = \frac{t_{par}(\text{orig})}{t_{par}(n_p)}.$$

- `redblack`, t_{rb}: how much time was spent on updating the red and black values.
- s_3: speedup of the red-black values update portion.
- `comm`, t_{comm}: total time used by the communication phase.
- `other`, t_o: total time in `PRESIT-P` initialization and finalization, `PSLAG` correction phase, `PSLAG` initialization phase. When no numerical accuracy loss occurs, we should see

$$t_{tot} = t_{seq} + t_{rb} + t_{comm} + t_o$$

- `diff`: discrepancy (%) in time measurements, possible due to rounding errors:

$$\text{diff} = 100\ \frac{t_{tot} - t_{seq} - t_{rb} - t_{comm} - t_o}{t_{tot}}$$

5.2 HPCIBM1

5.2.1 low resolution

We start with a relatively small grid: $50 \times 50 \times 100$.

np	ttot	tseq	m2	s1	s2	s3	redblack	comm	other	diff
orig	107.8	14.9								
1	94.3	12.4	6124	1.14	1.14	1.00	79.8	0.1	1.9	0.04
2	80.3	11.6	6124	1.34	1.35	1.52	52.6	7.4	5.3	4.22
3	69.2	11.6	5854	1.56	1.61	2.33	34.2	8.6	9.0	8.27
4	78.0	11.7	6109	1.38	1.40	1.91	41.7	10.1	9.4	6.61
5	69.7	11.6	5886	1.55	1.60	2.85	28.0	10.9	11.7	10.66
6	71.6	11.7	5997	1.51	1.55	3.52	22.7	14.6	11.8	15.08
7	73.6	11.7	6027	1.46	1.50	3.41	23.4	13.2	13.9	15.48
8	70.3	11.9	6117	1.53	1.59	3.91	20.4	15.0	13.4	13.72

Table 5.1: `HPCIBM1`, $50 \times 50 \times 100$

We see a slight yet surprising 14% improvement of the single-processor MPI code over the original code. This might be due to (the lack of) compiler optimizations, despite all code is compiled with option `-O3`. Theoretically, both codes should agree closely as MPI overhead is expected to be negligible. On the other hand, it is difficult to tell what exactly happens during compilation.

The column s_3 shows suboptimal speedup, since we would expect $s_3(n_p) = n_p$. This is because the grid planes are partitioned over all processors and s_3 reflects the computational work at those planes only, not being affected by communication costs or whatsoever. In general, the number of planes is not a multiple of n_p, yielding a slight fractional performance loss of about $\frac{n_p}{n_z}$, since some processors have been assigned one plane more than some others. This effect should vanish when the number of planes increases, but on the other hand, the values in table 5.1 are far too bad. For instance, at $n_p = 8$ some nodes have been assigned 13 planes and some only 12, thus s_3 is bounded by $\frac{100}{13} \approx 7.6$.

Observe the erratic iteration counts in m_2, as announced in section 4.2.4. During the pressure iteration, data travels in a slightly different manner through the numerical grid than in the original code. A certain `PRESIT-P` call might take a few iterations more or less to converge, depending on the grid partitioning. This explains why the total iteration counts seem to be randomly distorted.

Timing measurements are done using `MPI_WTIME`. This function has only a resolution of about one millisecond, so for small grids such as this one, rounding problems might distort some of the numbers. More precisely, when for instance the `PSLAG` correction phase finishes within 0.5ms, that timing result might be truncated. Thus, timing results after 6000 `PSLAG` iterations are off by 3 seconds in the worst case, which resembles the observed order of discrepancy in `diff` quite well.

Communication overhead increases with n_p, indicating that either the network or the MPI implementation performs worse than expected. We will look more closely into this matter in section 6.1.

The difference between s_1 and s_2 is typically only a few percent. This is explained by the fact that almost all time is spent in `PRESIT-P`, and this portion will grow even more when grid resolution increases. From now on, the s_2 column will be omitted as it does not add any significant information.

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	86.6	13.7						
1	86.5	12.0	5994	1.00	1.00	69.9	0.0	4.4
2	69.2	12.2	5922	1.25	1.83	38.2	7.3	6.7
3	71.1	11.9	5994	1.22	2.06	33.9	9.0	10.1
4	87.9	12.0	6022	0.99	1.43	49.0	10.1	10.1
5	69.0	12.1	5955	1.25	2.64	26.5	11.1	12.1
6	67.3	12.2	5952	1.29	3.58	19.5	13.7	12.2
7	69.7	12.4	5962	1.24	3.70	18.9	14.4	13.2
8	71.8	12.3	5952	1.21	3.28	21.3	15.8	12.1

Table 5.2: `HPCIBM1`, $100 \times 50 \times 50$

Stretching the grid in the other direction (table 5.2) will activate the transposition mode of `PRESIT-P`, as the code within `PSLAG` requires z to be the longest direction. This should show an increased portion of time spent in the category `other`, as it contains `PRESIT-P` initialization and finalization.

As with $50 \times 50 \times 100$, the values of s_3 are far from optimal and the communication phase is too expensive. The values in `other` are quite comparable to the ones associated with the untransposed grid, and it indeed seems that the influence of `PRESIT-P` initialization and finalization is marginal.

Nonetheless, both small grids exhibit a disappointing performance of `PRESIT-P`. Let us now move on to higher resolutions.

5.2.2 high resolution

The memory usage of the (master) `SloshDP` code is about 440 bytes per grid cell, which means we cannot increase the resolution much further than $200 \times 100 \times 100$ since a `HPCIBM1` node has 1GB memory.

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	1.629e+03	9.28e+01						
1	1.952e+03	9.48e+01	17743	0.83	1.00	1.81e+03	2.00e-01	4.95e+01
2	1.575e+03	8.87e+01	17727	1.03	1.34	1.35e+03	6.88e+01	5.00e+01
3	8.954e+02	8.80e+01	17738	1.82	2.91	6.21e+02	1.00e+02	6.66e+01
4	1.183e+03	8.81e+01	17768	1.38	2.04	8.85e+02	1.24e+02	6.40e+01
5	9.121e+02	8.86e+01	17845	1.79	3.15	5.74e+02	1.52e+02	7.35e+01
6	7.378e+02	8.78e+01	17636	2.21	4.74	3.81e+02	1.66e+02	7.16e+01
7	6.397e+02	8.98e+01	17828	2.55	6.69	2.70e+02	1.68e+02	7.59e+01
8	8.398e+02	8.93e+01	17799	1.94	4.15	4.35e+02	2.11e+02	7.46e+01

Table 5.3: `HPCIBM1`, $200 \times 100 \times 100$

At this grid resolution, s_3 performs a bit better than before. Unfortunately, the communication phase has a far too large influence, annihilating the speedup s_1.

Again, we also look at the transposed version, in this case $100 \times 100 \times 200$ (table 5.4).

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	1.806e+03	9.51e+01						
1	1.550e+03	1.00e+02	18775	1.17	1.00	1.40e+03	2.00e-01	3.08e+01
2	2.119e+03	8.97e+01	18775	0.85	0.74	1.91e+03	6.93e+01	3.79e+01
3	9.959e+02	8.78e+01	18840	1.81	1.91	7.34e+02	1.04e+02	5.16e+01
4	1.173e+03	8.78e+01	18796	1.54	1.59	8.84e+02	1.30e+02	5.47e+01
5	1.208e+03	8.87e+01	18827	1.50	1.65	8.50e+02	1.61e+02	6.87e+01
6	9.349e+02	8.81e+01	18838	1.93	2.48	5.66e+02	1.71e+02	7.14e+01
7	8.471e+02	8.83e+01	18744	2.13	3.27	4.29e+02	2.10e+02	7.84e+01
8	8.223e+02	8.79e+01	18904	2.20	3.58	3.92e+02	2.28e+02	7.83e+01

Table 5.4: `HPCIBM1`, $100 \times 100 \times 200$

Surprisingly, these results are significantly worse (mainly due to s_3) than at $200 \times 100 \times 100$, despite the fact that no grid transposition is applied within `PRESIT-P`. We might look into this curious matter further, if it weren't overshadowed by the fact that the communication phase again seems to be too expensive. A more detailed analysis will follow in section 6.1.

5.3 SI01

5.3.1 low resolution

On SMP machines like SI01, communication costs should have far less impact than on HPCIBM1.

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	56.4	7.1						
1	76.6	6.6	5911	0.74	1.00	68.3	0.0	1.5
2	41.3	7.0	5898	1.37	2.18	31.4	0.7	1.9
3	25.7	6.5	5915	2.20	4.14	16.5	0.8	1.5
4	18.5	6.9	5969	3.05	7.76	8.8	0.9	1.6
5	14.3	6.3	6013	3.95	13.39	5.1	1.0	1.4
6	12.0	6.2	5942	4.69	22.03	3.1	0.9	1.2
7	13.8	6.3	5970	4.10	15.88	4.3	1.0	1.6
8	12.8	6.7	5970	4.40	24.39	2.8	1.1	1.8

Table 5.5: SI01, $100 \times 50 \times 50$

Somehow, the single-node MPI code performs dramatically worse than the original code. If this is caused by a structural problem in the code and if that problem would be solved, we should see s_1 improves, not only for $n_p = 1$, but for all n_p.

The red-black update shows a remarkable case of *super-linear speedup*. As explained before, we would theoretically expect $s_3(n_p) = n_p$, regardless of underlying machine architecture intrinsics such as network bandwidth and cache sizes. The super-linear speedup possibly reflects a "lucky cache strategy", so we should not get too excited about this.

As foreseen, t_{comm} is very small compared to t_{tot}, so the communication overhead is practically gone on this machine. At this point, we should place a remark regarding the SI01 architecture. The machine has its processors clustered in four groups of four cores each. When we start a PRESIT-P timing measurement, the system activates a certain number of cores on its own depending on the availability at that specific moment. In terms of communication, it might turn out that some cores are more close to each-other than others. On the other hand, this effect is unnoticeable since communication costs are small anyhow.

Table 5.6 shows really good speedup figures.

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	69.2	8.2						
1	35.6	6.2	5986	1.94	1.00	28.4	0.0	0.8
2	31.5	6.2	5986	2.20	1.21	23.5	0.6	0.9
3	16.9	6.2	6060	4.10	3.26	8.7	0.7	1.0
4	14.5	6.2	6103	4.78	4.58	6.2	0.9	0.9
5	13.4	6.2	6086	5.18	5.80	4.9	0.8	1.1
6	11.9	6.2	5989	5.83	8.11	3.5	0.8	1.0
7	11.5	6.2	6004	6.03	9.47	3.0	0.9	1.1
8	11.3	6.2	5954	6.12	10.14	2.8	0.9	1.1

Table 5.6: SI01, $50 \times 50 \times 100$

Again, we observe super-linear speedup at s_3, with the sole exception at $n_p = 2$. Compared to the original code, the single node MPI code performs almost twice as good, but not much better when 2 instead of one node is involved. This can be explained by the fluid configuration of our test case, see chapter 6.2.

Generally, `PRESIT-P` seems to yield very good speedup if we do not use too many cores.

5.3.2 high resolution

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	9.837e+02	4.58e+01						
1	1.902e+03	5.37e+01	17626	0.52	1.00	1.82e+03	1.00e-01	2.26e+01
2	9.870e+02	5.05e+01	17671	1.00	2.00	9.11e+02	5.40e+00	1.87e+01
3	6.871e+02	5.11e+01	17693	1.43	3.00	6.08e+02	7.40e+00	1.89e+01
4	7.584e+02	5.01e+01	17618	1.30	2.70	6.77e+02	1.01e+01	2.05e+01
5	5.719e+02	5.00e+01	17713	1.72	3.72	4.90e+02	1.01e+01	2.04e+01
6	4.810e+02	5.01e+01	17567	2.05	4.59	3.98e+02	1.05e+01	2.03e+01
7	5.077e+02	4.94e+01	17727	1.94	4.34	4.21e+02	1.28e+01	2.21e+01
8	4.744e+02	4.96e+01	17522	2.07	4.77	3.82e+02	1.45e+01	2.56e+01

Table 5.7: `SI01`, $200 \times 100 \times 100$

As with $100 \times 50 \times 50$, the single-node MPI code performs far worse than the original code. The numbers almost suggest that the compiler silently puts two cores to work on the original code - compare $t_{tot}(\text{orig}) = 983.7$ to $t_{tot}(2) = 987.0$. Unfortunately, we can't draw any real conclusions from the code yet. If we can either stop the original code from cheating, or apply the same cheating to `PRESIT-P`, the speedup s_1 probably becomes twice as good.

The column of s_3 shows fair speedup until $n_p = 3$. Beyond that point, speedup stagnates without any clear reason.

The previous tables of `SI01` benchmarks have shown some numbers that cannot be explained at the moment. We might even flag them as *contaminated*, assuming there exists some compiler option to relieve our suspicions. More contaminated speedup values can be seen in table 5.8.

np	ttot	tseq	m2	s1	s3	redblack	comm	other
orig	1.320e+03	5.22e+01						
1	9.294e+02	4.90e+01	18836	1.42	1.00	8.71e+02	1.00e-01	9.00e+00
2	8.837e+02	4.88e+01	18836	1.49	1.06	8.19e+02	5.00e+00	9.80e+00
3	6.143e+02	4.89e+01	18847	2.15	1.61	5.42e+02	8.50e+00	1.35e+01
4	5.214e+02	4.89e+01	18792	2.53	1.94	4.49e+02	9.40e+00	1.28e+01
5	5.811e+02	4.88e+01	18852	2.27	1.74	5.01e+02	1.14e+01	1.82e+01
6	4.234e+02	5.03e+01	18865	3.12	2.54	3.42e+02	1.04e+01	1.74e+01
7	3.853e+02	4.88e+01	18844	3.43	2.86	3.05e+02	1.15e+01	1.59e+01
8	3.510e+02	4.88e+01	18813	3.76	3.24	2.69e+02	1.20e+01	1.74e+01

Table 5.8: `SI01`, $100 \times 100 \times 200$

The single-core MPI code performs 42% better than its original counterpart. On the other

hand, adding one extra core has marginal influence, again explained by the fluid configuration (chapter 6.2).

With the contaminated value $t_{rb}(1) = 871$, the entire column s_3 gives a misleading picture. Suppose we would have $t_{rb}(1) = 1.5 \cdot 10^3$, then speedup would be far more agreeable, for instance $s_3(2) = 1.8, s_3(3) = 2.8$ and $s_3(8) = 5.6$.

Chapter 6

Discussion and conclusions

In this thesis, we first established that the `PRESIT` component of `ComFlo` is the most time consuming component. This procedure, which solves the discrete pressure Poisson equation, has been parallelized using MPI. The new parallel procedure was named `PRESIT-P` and it has been tested on two machines. Although we tried to minimize inter-processor communication costs, results on the `HPCIBM1` cluster were far from good. Section 6.1 will explain this in more detail. On the other hand, results on the shared memory processing machine `SI01` have shown good speedup measurements.

6.1 Bandwidth bottleneck

The results on `HPCIBM1` have raised numerous questions. Why does the single-processor MPI code sometimes produces its results much faster or slower than the original code? To which extent does the compiler optimize code? How is it possible we don't see near-perfect speedup at s_3?

All these questions would certainly be worth the effort of further investigations, if we could expect better results. However, it is a simple fact that the bandwidth is the main reason `PRESIT-P` does not perform good on `HPCIBM1`. Let's do a little heuristic analysis to firmly support this statement.

In an ideal setting, the time spent in the pressure value update should parallelize perfectly. Let's approximate the required total time by $\tilde{t}$ in an optimistically way, not taking for example the correction phase into account. We use notation as introduced in section 5.1.1.
If we assume α is machine-dependent parameter indicating the amount of time required to update one grid cell, we may estimate

$$\tilde{t}_{rb}(n_p) = m_2 \cdot \alpha \cdot \frac{n_x \cdot n_y \cdot n_z}{n_p} \approx t_{rb}(1).$$

As mentioned before (see chapter 4.3.2), in the best case the communication phase takes

$$\tilde{t}_{comm}(n_p) = 2 \cdot m_2 \cdot \beta \cdot n_x \cdot n_y,$$

assuming z is the longest dimension, with β the time to transfer one pressure value to a neighbor processor. `HPCIBM1` has a bandwidth of about 22ms/MB and each pressure value is 8 bytes (double precision), thus $\beta \approx 1.8 \cdot 10^{-7}$.

If we use the equalities above to estimate the optimal general speedup s_1^*, we get

$$s_1^*(n_p) = \frac{t_{tot}(\text{orig})}{t_{seq}(n_p) + \tilde{t}_{rb}(n_p) + \tilde{t}_{comm}(n_p) + t_o(n_p)}.$$

In figure 6.1, these optimal speedup estimates are shown next to the `HPCIBM1` measurement values from section 5.2.

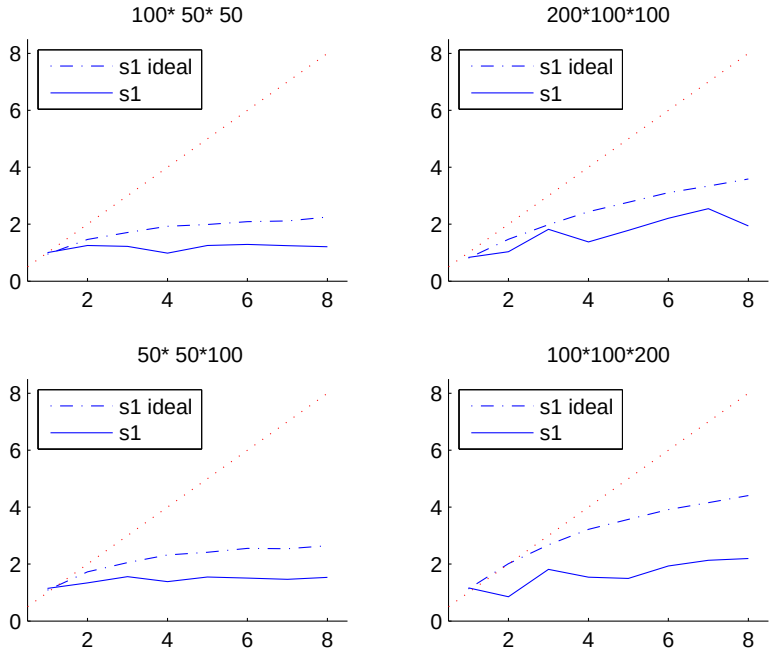


Figure 6.1: optimal `HPCIBM1` speedup estimates

Although there is some room for improvement, it remains to be seen to which extent it can be achieved. Therefore we conclude that the bandwidth limitations render `PRESIT-P` practically useless on `HPCIBM1`. On `SI01` the "bandwidth" is high enough to prevent the problems mentioned above.

6.2 Fluid configuration

As the PRESIT routine only affects full and mixed fluid cells, and we abort the program after 10 PRESIT calls, the speedup measurements depend highly on the used fluid configuration. Our measurements were conducted on a setup where the fluid is located in the lower z area, with a fill ratio of about 38%. This explains the bad speedup when we partition in the z dimension: if we use two instead of one core, one of both only has air cells to update. Users should be aware of this phenomenon when choosing a grid resolution. Perhaps the decision in which dimension to subdivide the grid can be based upon the fluid configuration.

6.3 Shared memory

Results on `SI01` are generally quite good. This has two main reasons:

- high bandwidth
  Opposite to `HPCIBM1`, MPI does not need to send its data across a network. Since all memory is shared by all cores, the transmission of a boundary plane effectively is a copy of memory from the allocated space of one core to another, which can be achieved very fast. On the other hand cache effects and bus limitations may hamper speedup, as illustrated by figure 6.2.

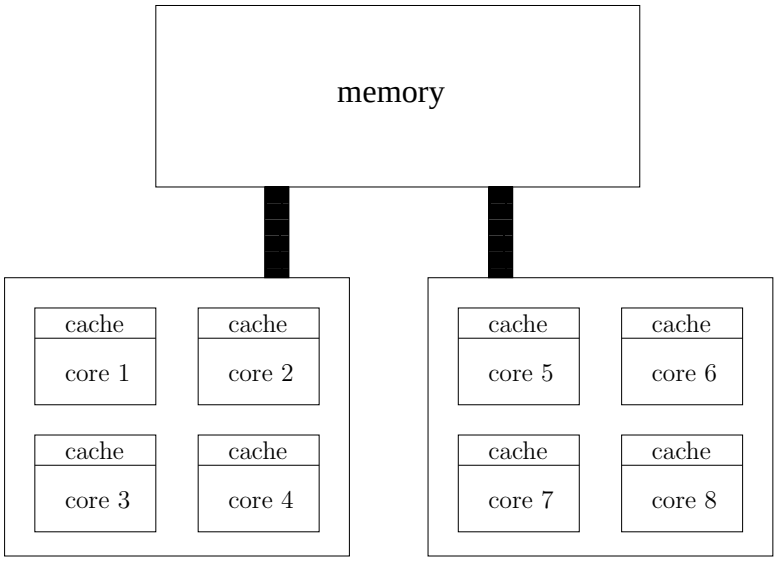


Figure 6.2: cache and bus effects

- high resolutions
  As mentioned above, all memory is shared by all cores. This gives us the possibility to look at larger grids than was previously possible. A nice property of `PRESIT-P` is that speedup should become better as grids grow, so this is definitely a feature to put to good use.

6.4 Concluding remarks

A distributed memory system such as `HPCIBM1` is not fit for the developed parallel MPI code `PRESIT-P`. The main reason for this is that the bandwidth is too low, i.e. the transferral of data from the boundary planes takes too much time compared to the actual computation work within the Poisson solver. However, when we run the program on a SMP system such as `SI01`, much better performance can be expected as bandwidth is not an issue anymore. As long as no SMP specific code is developed, for instance using OpenMP, our `PRESIT-P` will prove worth-wile on SMP machines.

6.5 Suggestions for future work

6.5.1 compiler technicalities

Judging from several strange values in section 5.3, it may very well be possible that the compiler is performing certain actions, resulting in distorted speedup measurements. Perhaps it would help if someone with more experience in this area would take a look at this.

6.5.2 possible MPI improvements

As mentioned in section 4.3.2, a possible improvement might be to use specific MPI features such as non-blocking interaction routines. Another possibility is to somehow pack the data from the interaction planes, although it seems unlikely that this is worth the effort.

6.5.3 grid partitioning choice

Currently, the grid is partitioned in the longest dimension. Since fluid configuration plays a large role in the expected speedup, we might use this to choose a better grid partitioning strategy. This work could be done on PPRESIT_INIT.

Bibliography

[1] Amdahl, G. *The validity of the single processor approach to achieving large-scale computing capabilities.* Proceedings of AFIPS Spring Joint Computer Conference, pp. 48385, Atlantic City, N.J., April 1967. AFIPS Press.

[2] Botta, E.F.F. and Ellenbroek, M.H.M. A modified sor method for the poisson equation in unsteady free-surface flow calculations. *J. Comput. Physics*, 60:119–134, 1985.

[3] Fekken, G. *Numerical Simulation of Free-Surface Flow with Moving Rigid Bodies.* PhD thesis, University of Groningen, The Netherlands, 2004.

[4] Kleefsman, K.M.T. *Water impact loading on offshore structures - a numerical study.* PhD thesis, University of Groningen, The Netherlands, 2005.

[5] Kleefsman, K.M.T., Fekken, G., Veldman, A.E.P., Iwanowski, B. and Buchner, B. A volume-of-fluid based simulation method for wave impact problems. *J. Comput. Physics*, 206:363–393, 2005.

[6] Luppes, R., Helder, J.A. and Veldman, A.E.P. Liquid sloshing in microgravity. In *56th International Astronautical Congress.* International Astronautical Federation, 2005. IAC-05-A2.2.07.

[7] Luppes, R., Helder, J.A. and Veldman, A.E.P. The numerical simulation of liquid sloshing in microgravity. In *Europ. Conf. Comput. Fluid Dyn.: ECCOMAS CFD 06.* ECCOMAS, 2006. paper 490, ISBN 909020970-0.

[8] OpenMP Website, URL: http://openmp.org

[9] MPI specification (website), URL: http://www.mpi-forum.org/docs

[10] Veldman, A.E.P and Vogels, M.E.S. Axisymmetric liquid sloshing under low gravity conditions. *Acta Astronautica*, 11:641–649, 1984.

[11] Veldman, A.E.P, Gerrits, J., Luppes, R., Helder, J.A. and Vreeburg, J.P.B. The numerical simulation of liquid sloshing on board spacecraft. *J. Comput. Phys.*, 224:82–99, 2007.

[12] Wemmenhove, R. Numerical simulation of two-phase flow in offshore environments. PhD Thesis, University of Groningen, The Netherlands, 2008. URL: dissertations.ub.rug.nl/faculties/science/2008/r.wemmenhove.

[13] Xie, D. and Adams, L. SIAM Journal on Scientific Computing, Volume 20 , Issue 6 (November 1999)