Parallelization of Air Quality Models on Structured Grids Using MPI

By

Philipp Miehe

A PROJECT REPORT

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY

August 2001

This project report, "Parallelization of Air Quality Models on Structured Grids Using MPI," is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

Committee members:

Dr. Adrian Sandu (Advisor)

Dr. Jean Mayo

Dr. Steve Seidel

Dr. David Hand (CEE)

DEPARTMENT Computer Science

Signatures:

Project Advisor ______________________________
(Adrian Sandu)

Department Chair ______________________________
(Linda Ott)

Date ______________________________

Abstract

The technological development during the past two centuries has had a significant impact on our environment. One of the greatest concerns is air pollution. Air pollutants affect human health, the biosphere and damage property. Because of these effects air pollution is now monitored and regulated worldwide, for example by the Clean Air Act in the United States. In order to understand air pollution and design effective control strategies, comprehensive computer models are necessary. Air quality models simulate the chemical reactions and regional movement of pollutants in our atmosphere. The complexity of the computations and the resulting high time-to-solution call for the application of parallelization strategies. This work reviews some approaches taken by other researchers to efficiently parallelize air quality models. Further a MPI-based communication library is implemented to parallelize air quality models using uniform structured grids. The routines serve for distribution, gathering and reshuffling of typical arrays used in these models. Portability of the communication library is proved by testing it on two different architectures, a Beowulf cluster using distributed memory and a Sun Enterprise using shared memory. Our domain decomposition approaches accommodate the most general transportation schemes. Different data decomposition strategies are implemented and compared with respect to performance. Finally the library is used to parallelize STEM-III, a state-of-the-science air quality model developed at the University of Iowa. The parallelization of STEM-III on the Beowulf cluster results in a time decrease from 1 hour and 15 minutes (serial code) to less than 2 minutes. The load imbalance of the different domain decompositon approaches is discussed in this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides the motivation and introduction to our work. It describes the importance of air quality models in our world today and illustrates the various environmental processes simulated by air quality models.

1.1 The Importance of Air Quality Modeling

From the start of industrialization in the 19th century until today mankind has achieved a high level of technology. This development during the past two centuries has had a great impact on our environment. The driving force for technological development was the invention of the steam engine. While the name does not suggest negative effects since steam is used to refer to vaporized water, the exhaust of the steam engine is not only water but all the chemical species involved when burning coal or later fuel. Serious pollution of our environment started with this technologically important invention.

Life strives on the air that surrounds us. Scientists studied the impact of air pollution on our world during the past years. In order to preserve the world present for our children, governments throughout the world are now regulating the admissible levels of pollutants in the atmosphere.

The Clean Air Act [15] in the United States describes the various chemical species in the air as well as pollutants, that are released in the atmosphere by today's technology. Air pollution creates negative effects on the health of humans and can damage the environment and property. Air quality models (AQMs) are used to simulate the reactions and effects of certain chemical species in our atmosphere.

Ozone (O_3) is the main component of smog. While ozone in the stratosphere prevents most of the deadly ultraviolet radiation from entering our atmosphere, ground-level ozone can lead to breathing problems for people and damage plants, rubber and fabrics. As the primary component of smog, ground-level ozone is produced by many different smog-forming air pollutants such as Volatile

Organic Compounds (VOCs) and Nitrogen oxides (NO_x). VOCs can cause cancer in addition to the effects ozone already has on human health. Nitrogen (NO_2) and Sulfur dioxide (SO_2) can damage trees and lakes as well as buildings and monuments since these species are contained in acid rain. Carbon monoxide (CO) binds hemoglobin, the protein in our blood that transports oxygen. It therefore can be seriously damaging to our body; cells that do not receive oxygen will die. Lead (Pb), as another pollutant, can damage the nervous system of humans. Particulate matter such as dust, smoke and soot also need to be monitored as part of air pollution.

Due to their negative effects on human health and the environment the chemicals mentioned above should be prevented from entering the air. The process of eliminating pollution from our manufacturing industry becomes difficult because of the exceptionally high costs. Since most damaging chemicals form by burning coal, natural gas and gasoline – processes that are considered the "engine" of today's technology and therefore widely spread throughout the world – prevention of air pollution eventually will be the final result of the slow process of reduction steps taken by officials.

This leads to the need to simulate the state of the atmosphere, done by AQMs, to predict the level of danger for humans and nature and calculate the amount of regional pollution reduction that is necessary. Further needs for AQMs are due to the impact of chlorofluoro carbons (CFCs) that destroy the important ozone layer in the stratosphere, and to global warming processes due to anthropogenic emissions of greenhouse gases like CO_2. Therefore air pollution can affect the desertification, and therewith decrease the potential of life on our planet.

1.2 The Structure of Air Quality Models

AQMs include various submodels that cooperate to tackle the complex simulation problem. Monitoring the reactions of chemical species over time is just part of the task. Physical factors such as sun light and wind affect the reaction rates and species concentrations and have to be modeled as well in order to simulate the air quality. Atmospheric chemical transformations, cloud forming, radiative transfer simulations, transport considerations, emissions and depositions have to be taken into account in order to obtain reliable results that are necessary to provide accurate predictions. The overall performance and accuracy depend on the quality of each module. Used by many environmental researchers as investigation tools, AQMs can also be useful for regulating purposes as well as simulating different policies and pollution reduction scenarios.

Typically a regional air quality model simulates about 100 chemical gas-phase species and uses a resolution of 100×100 grid points in horizontal direction as well as 20 vertical layers. A global model usually incorporates a vertical resolution of $1^o \times 1^o$ longitude by latitude and 20 vertical layers, the size of the grid cells is larger than for regional models. Due to the complex computations done in the atmospheric chemistry model of an AQM, there is a need for faster machines as the number of

	Horizontal Resolution	Vertical Resolution	Chemical Species	ST/RT[1] Ratio	Computational Power
Regional Model	100×100	20 layers	100 gas phase species	1:1	1 GigaFlop/s
Regional Model	100×100	20 layers	100 gas phase species	100:1	100 GigaFlop/s
Global Model	$2^o \times 2^o$	10 layers	70 gas phase species	100:1	20 - 30 GigaFlop/s
Global Model	$1^o \times 1^o$	10 layers	70 gas phase species	100:1	100 - 200 GigaFlop/s
Global Model	$0.5^o \times 0.5^o$	20 layers	70 gas phase species	100:1	0.8 - 2 TeraFlop/s
Global Model	$0.5^o \times 0.5^o$	20 layers	70 gas phase species + aerosols	100:1	$\sim$ 30 TeraFlop/s

Table 1.1: Computational power required for global and regional models, [1, 5]

simulated species increases. Predictions have to be done faster and include more information in order to be useful. Today's most powerful workstations have computational power of several GigaFlop/s. Model simulations utilizing above described grid sizes and performing a desirable simulation time to real time ratio of 100:1 require about 100 GigaFlop/s as seen in table 1.1. When including aerosol simulations, computational requirements increase to a couple of TeraFlop/s. As this cannot be handled by single workstations anymore the application of parallel power becomes evident. Today's fastest parallel systems incorporate around 10 TeraFlop/s (the best ranked ASCI White SP in the TOP500 Supercomputer list[2] was the first to break the double digit TeraFlop speed barrier with 12.3 TeraFlop/s). Clearly the power of parallel computing can provide faster time-to-result for many simulation scenarios, thereby presenting a possible solution to our time problem.

An AQM works on a discrete grid covering the atmospheric region to be simulated. A regular grid can be logically mapped onto a cube, mapping the earths surface on the bottom side while its volume represents the atmospheric region of interest. The full cube is partitioned into many small subcubes by extending the grid on the bottom surface in vertical direction and dividing the full cube into horizontal layers. Each resulting subcube represents an air parcel that is effected by pollution from industry and technology on the ground as well as by the polluted air from the surrounding cube cells, e.g. blown in by wind. The transportation of chemicals by wind as well as by diffusion

[1] ST/RT - Simulation time to real time

[2] http://www.netlib.org/benchmark/top500/top500.list.html

is simulated in time steps; the chemical reactions can be calculated independently in each subcube.

1.3 The Structure of This Project

The need for faster machines is created by the complexity of the problems to solve. Air quality modeling is one of the grand challenge problems of parallel computing. The application of parallel power to AQMs reduces the time-to-result, thereby provoking the appliance of larger problems including longer simulation times. This report discusses the implementation of an MPI-based communication library to parallelize air quality models using uniform structured grids in FORTRAN. It was tested on a Sun Enterprise and a Beowulf cluster using Sun MPI and MPICH, respectively. Based on the library a parallel version of the state-of-the-science AQM STEM-III, developed at the University of Iowa, was implemented. This document describes the functionality of the communication library as well as the strategies used for parallelizing STEM-III. Further this work includes timing results of the parallel version on the Beowulf cluster.

The paper is organized as follows: The following chapter, chapter 2, presents previous work on AQMs done by scientists all over the world. First the problem of air quality modeling is described followed by the presentation of parallelization work and load balancing experiments done by scientists around the globe. Chapter 3 presents parallelization strategies for AQMs in general. Then it describes STEM-III and the approach that was used to implement parallel STEM-III. Environment descriptions and dependency analysis are included in chapter 4. The communication library applicable to AQMs using structured grids is described in detail in chapter 5 and comparison results of the library routines on different architectures are presented. Measurements of the parallelization of STEM-III and important results are illustrated in chapter 6. Finally chapter 7 draws some conclusions and pinpoints to parts of our work where future research can be undertaken. In the appendix the user's manual for the communication library is included.

Chapter 2

Previous Work

After introducing air quality models in the last chapter, here approaches taken by other researchers to optimize the speed of simulations by developing suitable numerical methods as well as applying parallelization strategies to air quality models are presented.

2.1 Efficient Air Quality Modeling

Carmichael et. al. [2] describe the chemical and computational framework of air quality modeling. The authors mention the complexity of the processes incorporated into AQMs. Most phenomenon are not completely understood and have to be simplified and parameterized in order to be integrated in models. The theoretical basis for modeling is the mass balance equation:

$$\frac{\partial C_i}{\partial t} + \frac{\partial(U_j \cdot C_i)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[K_{jj} \cdot \frac{\partial C_i}{\partial x_j}\right] + R_i + E_i + G_i \tag{2.1}$$

Here i represents each species, C_i stands for gas phase concentrations, U_j for velocity components and x_j for spatially coordinates (three dimensional). K_{jj} are the diffusitives, R_i the rate of chemical reaction, G_i the rate of mass transfer and E_i the rate of emissions.

The paper describes mechanisms and algorithms used to model several natural processes. It focuses on improving algorithms for the chemistry module in order to reduce the total CPU time by introducing different methods for solving ordinary differential equations (ODEs). Initially this involves improving quasi-steady state approximations (QSSA) but Carmichael presents different approaches using implicit integrators and linear algebra in order to solve atmospheric chemistry problems more efficiently.

The authors explain advantages and difficulties of parallel implementations while directing attention to the differences of parallel computers (e.g. vector machines and message passing systems). Minimizing communication is the goal for the parallel version in order to achieve high performance. For each time step, the horizontal $(T_\mathrm{x}, T_\mathrm{y})$ and vertical transportation (T_z) as well as the chemical reactions (C) have to be calculated. The authors describe a scheme that saves communication time. Normally each time step is ordered as $T_x \cdot T_y \cdot T_z \cdot C$. Since a reshuffling of the rows and columns is necessary after the horizontal and before the vertical transport, because a whole row/column of data is required for the horizontal/vertical calculations respectively, we need two communication procedures for each time step. This can be reduced to one per time step when applying the authors scheme. Consider two time steps and their calculations in the following order: $T_x^1 \cdot T_y^1 \cdot T_z^1 \cdot C^1 \cdot C^2 \cdot T_x^2 \cdot T_y^2 \cdot T_x^2$. Since reshuffling takes place once in between horizontal and vertical transportation, we need only two reshufflings for the two merged time steps.

In [3], Carmichael et. al. discuss further computational challenges introduced by air quality models. The paper focuses on aerosol and gas phase processes describing the simulation of atmospheric aerosols using thermodynamic and kinetic approaches. Aerosol in the stratosphere can destroy the ozone layer. As mentioned by the authors, inorganic aerosols have been studied for some years but organic aerosols might have a greater effect than known so far. This adds another complex computation to the already cost-expensive process of air quality modeling. The major part of the paper discusses a variety of methods to effectively calculate and simulate the aerosol reaction processes in the air.

2.2 Parallelization of Air Quality Models

Elbern [7, 8] put major effort into load balancing of AQMs. Specifically he worked on the EURAD (European Air Pollution Dispersion Model). In his work he tries to examine the contributions of dynamics, physics and chemistry modules to load imbalances and therefore introduces a family of load balancing schemes for a wide range of platforms. By categorizing existent load balancing strategies into combined global and local decision and migration bases, Elbern obtained a variety of concepts to chose from. A local decision base corresponds to comparing the work-load of the selected processor with neighbor nodes only, whereas in a global decision base, the load at all nodes are considered. The migration base is called local if the computational work units are handed over to neighbors only; however in a global migration base the load balancing can take place between any two processors. The author chooses four grid partitioning strategies to balance the computational load. Equal area partition (EAP) is a static partitioning scheme where each processor maintains the same amount of non-neighboring grid cells. Static load balancing (SLB) arranges the grid rows and columns depending on the prior known distribution of computational work once before the simulation. One-dimensional dynamic load balancing (DL1) globally changes the row and column positions after each time step depending on the computational load on each processor. Hereby the decision is made based on slow temporal variances of the load as well as quickly varying loads by recognition of regular imbalance patterns (e.g. during dusk and dawn solar radiation changes and

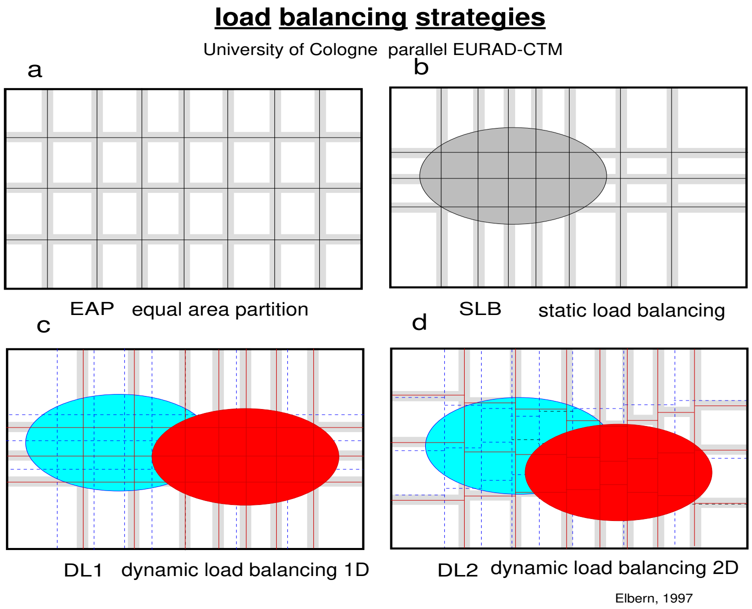

Figure 2.1: Grid partitioning strategies, found in [7, 8]

enforces small chemical time steps). The final approach, a two-dimensional dynamic load balancing scheme (DL2), works similar to the one-dimensional algorithm, but each processor in a column can adjust its north and south bounds separately. Therefore a processor can have more than two horizontal neighbors. A graphical description of the four schemes can be found in Figure 2.1.

The performance was measured for each load balancing method using 2 to 128 processors on a message passing machine (Intel Paragon). The conducted experiments emphasized the high load imbalances. The full adjustment scheme (two-dimensional load balancing) produced the best results in balancing the work. Comparisons of the communication costs using the static and the two-dimensional dynamic load balancing scheme show only a modest additional overhead due to message passing for the global decision base of work-load. In conclusion, the author states that full processor use without any idle time cannot be achieved due to the high number of different complex tasks that have to be carried out at each time step. But the introduced highly dynamic load balancing algorithm DL2 resulted in an improvement in efficiency from 38 percent to 62 percent.

Dabdub et. al. [6] present a portable parallel implementation of an AQM for distributed memory MIMD machines. The authors developed a performance model to be able to predict execution times of a program depending on the number of processors using machine and application-dependent parameters. For their experiments they use the California Institute of Technology (CIT) photochemical model. By reporting times for 12 different machines the authors found that 87 percent of the total time is consumed by chemical and vertical transport computations. Reading input and writing output data files requires 7.5 percent of the computation time, while 5 percent of the total time are spent on horizontal transport computation.

Dabdub et. al. used a master-worker strategy to parallelize the CIT AQM. Here the master does the I/O handling and distributes the work to other processors. During computation time of the processors the master can already read the input for the next time step. In order to provide portability the authors implemented an interface that fully utilizes the libraries on the given systems – MPI, PVM and p4 are not used directly because these libraries were not available on all the platforms used. This interface made it possible to port the AQM to different parallel machines (Intel Delta, Intel Paragon, IBM SP2, Cray T3D) while using different parallel libraries.

The drawback in the performance analysis is that the idle time due to load imbalances is ignored. Only computation and communication times are used in the formulas to describe the model resulting in the following formula of total time taken by each worker:

$$T = 92.5\% \cdot \frac{T_{seq}}{N} + L \cdot aN + \frac{bN + c + d/N}{BW} \tag{2.2}$$

The parameters are defined as the number of workers N, the latency L for sending a message and the interconnection bandwidth BW. The size of the arrays included in the CIT AQM data correspond to the parameters a, b, c and d; these parameters can be obtained by inspecting the parallel program. The product aN represents the number of messages sent by a processor increasing linearly with the number of processors; b represents data replicated at each processor; c represents data evenly divided among the workers and d represents data evenly divided that has to be exchanged by workers in parallel. These parameters can be determined a priori (prior to any execution of the code) and the authors show that the observed behavior agrees with their model. As mentioned by the authors, the introduction of load balancing complicates the implementation and causes message overhead, thereby unbalancing the communication phase, which will result in a more complex performance model. The load balancing issue is left for future work.

In [16] the Danish Eulerian Model (DEM) is parallelized by W. Owczarz and Z. Zlatev on an IBM SMP machine. Due to the special structure of the machine not only general optimizations, but also machine dependent features were integrated into the code by the authors. The used IBM SMP had 16 processors where 8 processors are on one node creating a shared memory level for those 8 processors, but the memory between the two nodes used is distributed. First the authors described the optimizing of the sequential code using the cache memory of the processors more effectively. This can be done for any machine where cache memory is available using blocking methods, by dividing the data into many small chunks that fit into the cache. Further optimization examples given by the authors include the reversing of loops (`DO` and `FOR`-loops) in parts of the program. As shown by the authors, the above examples are not optimizations for every system. For example vector machines perform poorly when blocking techniques are used because they were built to handle large arrays faster. The optimization proposed in the paper already lead to a speedup of 3.7 of the serial code of the DEM. Further, the authors used openMP to run the code in parallel on the shared memory part of the IBM SMP (one node only). For the parallel run on two nodes (16 processors) MPI was used in the implementation. Communication overhead for the distributed memory part is higher, therefore the implementation had to be done very carefully in order to effectively utilize the shared memory effects on each node. The authors proved their code to be scalable to some extend, when moving from a 96×96 grid to a 288×288 grid the expected outcome matches the experimental results. Overall the authors acquired impressive performance on the parallelization of the serially optimized code: speedup of 6.54 for one node (8 processors) resulting in 82 percent efficiency and speedup of 12.32 representing 72 percent efficiency for two nodes (16 processors). Load balancing is not discussed by the authors in this paper. In conclusions the writers mention that speed will always be an issue even while machines are getting faster all the time, because comprehensive simulation processes containing many scenarios still cannot be executed in timely manner.

2.3 Preliminary Results

Preliminary experiments have been conducted by Sandu [18] using a Beowulf cluster with 64 nodes. The author describes the parallelization strategy of the code used for the experiments. Input and output files are stored in the main storage accessible to the central box of the Beowulf cluster. Each node has access to local storage only, thereby demanding a master-worker parallelization strategy where the master process performs input and output and distributes the data to the worker processors.

Data dependencies in the code result from the implicit time-stepping transport scheme. For transport in any direction the whole row or column of grid points in that direction are needed at one processor. As the author explains, first the problem is reduced to a two-dimensional problem by not splitting vertical columns (z-direction), resulting in possible splittings in x and y-direction only.

Figure 2.2 on the next page shows the partitioning done by the master process as well as the communication required during time steps. The shuffling (X to Y and Y to X) is an all-to-all exchange communication including all processes but the master.

In the end the author shows his results, presenting a computational time decrease from 1 hour and 15 minutes to 2 minutes by application of 62 processors. Figure 2.3 on the following page shows the speedup for some numbers of processors. The graph for all numbers of processors possible would show the load imbalance of this scheme, in [18] a speedup graph evolving in stairs can be found.

In the diagram the bisecting line of the angle (green line) represents the ideal speedup, that can be reached when no additional overhead is introduced by the parallelization. The blue dotted line for theoretic speedup is calculated by including the parallelization overhead and considering unparallelizable parts, e.g. input/output routines, that decrease the efficiency when more than one processor is used. Finally the red line represents the real measurements. The difference between

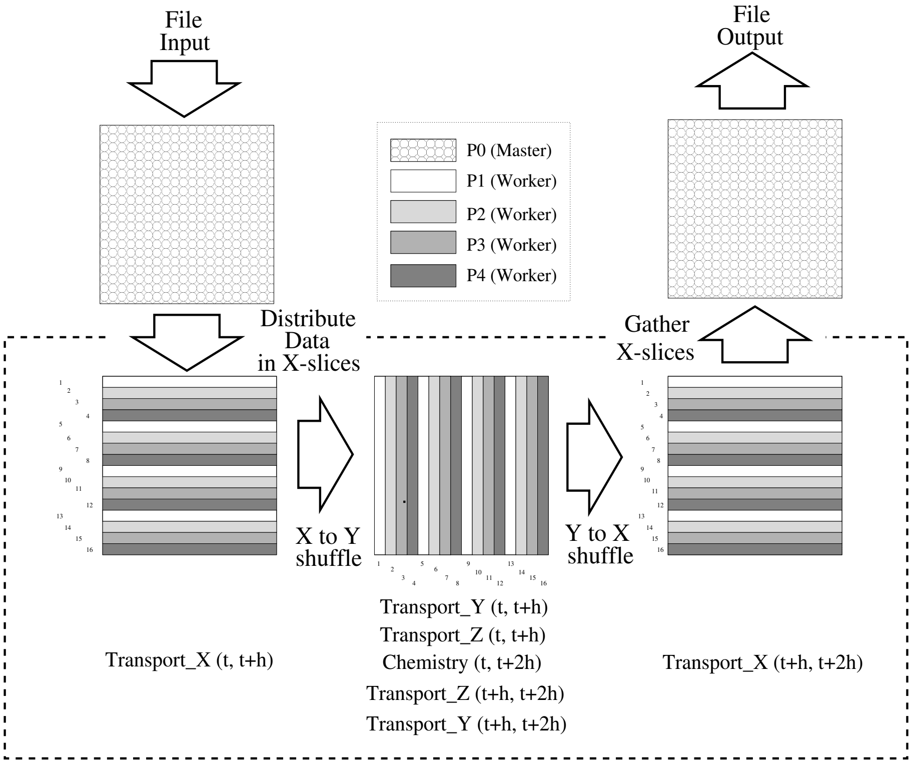

Figure 2.2: Data partitioning scheme for parallel STEM-III using five processors on a 16×16 grid from [18]

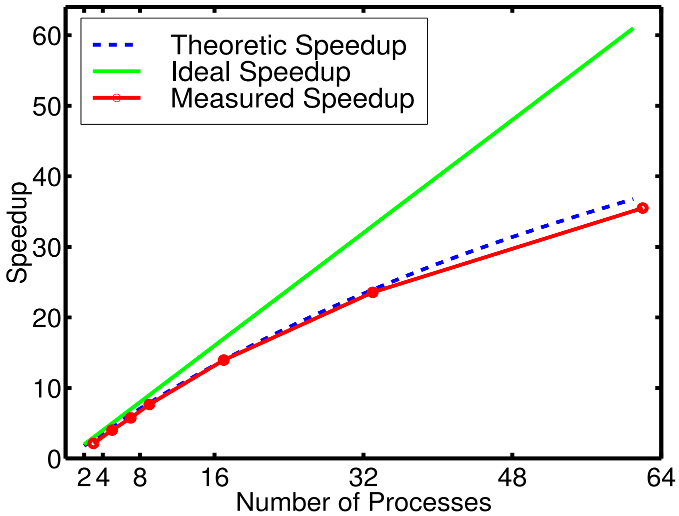

Figure 2.3: Speedup obtained using the sliced data partitioning scheme from [18]

the red and blue lines describes the load imbalance in the simulation that is not considered in the prediction calculations.

Due to the sliced partitioning and the implicit time stepping scheme for transportation all data need to be communicated between the processors. This results in very high communication cost. Clearly the disadvantages of this scheme are the total data movement at each time step in order to do the X and Y-transport. Therefore load balancing is almost impossible to implement as Sandu mentions. The parallelization depending on the number of slices given also reduces the scalability of the program (i.e. using a number of processors exceeding the grid size limit will result in loss of efficiency and no or only small time reduction).

In order to enable load balancing and to take advantage of the number of processors used non-sliced partitioning for chemistry needs to be researched. Other future work mentioned by the author is the achievement of finer granularity by redistribution of data for chemistry computations.

Chapter 3

Parallelization of AQMs

After summarizing parallelizations developed by researchers all over the world In the following the strategies investigated during the project are presented. This chapter first introduces general parallelization issues and then describes STEM-III, the AQM to be parallelized and illustrates the domain decomposition schemes applied.

3.1 Air Quality Models on Structured Grids

Most of todays AQMs, as well as meteorological simulation codes, are based on structured grids. At Pennsylvania State University the National Center for Atmospheric Research developed the mesoscale model in the early 70's, representing the standard of weather forecast models up to today's fifth generation (MM5). Now a new standard will be introduced, the Weather Model for Forecasting (WMF) in America and the European Community Weather Model for Forecasting (ECWMF) in Europe, replacing MM5. Investigations of structured grids are presented in the following sections. Recently the use of non-structured grids in AQMs has been investigated [20]. Such grids use nested structures based on tetrahedra or rectangular cell shapes, where selected areas of the grid can have higher resolutions then the rest of the grid. Non-structured grids will not be considered in this work.

3.1.1 The Grid

A structured grid describes a cube (figure 3.1) representing a part of the atmosphere, from the earth's surface up to a certain height. Depending on the resolution of the grid the cube is partitioned into many subcubes that enclose small volumes of the full cube.

The data structures for simulation on structured grids require different arrays to hold the simulation data. For each chemical species there is a concentration entry for each subcube in the concentration array, that represents the major information computed by the AQM. All other arrays

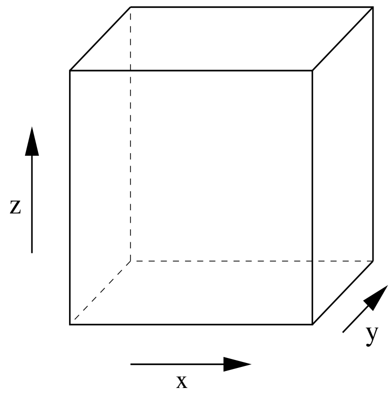

Figure 3.1: Cube model illustrating the 3 dimensions

represent important data necessary to recompute and update the concentration array using scientific methods.

The size and resolution of the grid is the main factor that determines the computation time. A high resolution will result in larger arrays that have to be read in, updated at each time step as well as written to in the end. Considering a constant resolution the enlargement of the grid would certainly lead to larger arrays too, therefore contributing to time increases during computation.

3.1.2 Data Types

In order to simulate the chemical reactions and natural behavior of the species monitored, the data needs to be arranged in efficient data structures found in table 3.1, that can be used for the calculations. The species concentrations, as the major data of interest here, are stored in a four dimensional array (4D) containing the concentration of each species in each grid cell. The grid cell layout is associated with the first three dimensions x, y and z and for each species the concentration needs to be recorded, that gives the species dimension (Ns). This is the structure where all the computations will take place on and that will produce the result to be written to an output file after the simulation is concluded.

description	dimension
4D array	(Nx,Ny,Nz,Ns)
2D array	(Nx,Ny)
3D array	(Nx,Ny,Nz)
2DN array	(Nx,Ny,Ns)
BDx array	(Nx,Nz,2,Ns)
BDy array	(Ny,Nz,2,Ns)

Table 3.1: The arrays representing a structured grid

Further data structures are needed for wind components in three directions, horizontal and vertical diffusion, emission rates as well as boundary information on the sides of the grid. These data are stored in two, three and four dimensional arrays respectively. Two dimensional arrays (2D), for instance the top layer height, consist of dimensions x and y of the grid. Three dimensional arrays can have different dimensions. The wind field representation consists of arrays in the three grid dimensions x, y and z (3D). Top and bottom boundary arrays are represented by arrays in x and y dimension of the grid as well as a species dimension (2DN). Boundary arrays (BD) are four dimensional structures, consisting of 2 three dimensional arrays in x and z (BDx) or y and z (BDy) direction of the grid and species dimension. The east and west as well as north and south boundary form the fourth dimension (only 2 entries).

All these arrays are set up in the input step of the simulation using the preprocessors. Only the 4D array containing the species concentration is used for the output process, it then serves as input to the visualization tools for building maps displaying the human health influencing species using colorful concentration representations.

3.2 The Parallelization Problem

As explained in the introduction, air quality models are parallelized in order to lead to faster time-to-result. Although the simulation process is complex the parallelization has to be as simple as possible using modules and functions already given by the serial version. The application of many processors requires a partitioning of the model leading to smaller local problems to be tackled at each processor. The decrease in size can lead to cache effects that, depending on the parallel overhead of the implementation, may introduce close to perfect speedup where speedup is the serial time over the parallel time resulting in perfection when the number of applied processors is obtained.

$$Speedup = \frac{t_{ser}}{t_{par}(N)} \tag{3.1}$$

$$Efficiency = \frac{Speedup}{N} \tag{3.2}$$

t_{ser} - serial time

t_{par} - parallel time

N - number of workers

Detailed definitions for parallel vocabulary like speedup and efficiency can be found in [12]. In order to parallelize a problem the following factors have to be considered:

- the possibilities of partitioning the problem into independent tasks,
- more efficient problem solving strategies and
- the targeted parallel architecture,
- the parallel library (MPI, PVM or others) to be used,
- other factors related to the computing environment (e.g. compilers).

These factors can not be viewed independently, the architecture to be used may influence the computing environment constraints as well as the parallel library possibilities. After careful consideration of these issues the efficiency can be estimated and the applicability of parallel power to the problem will become known.

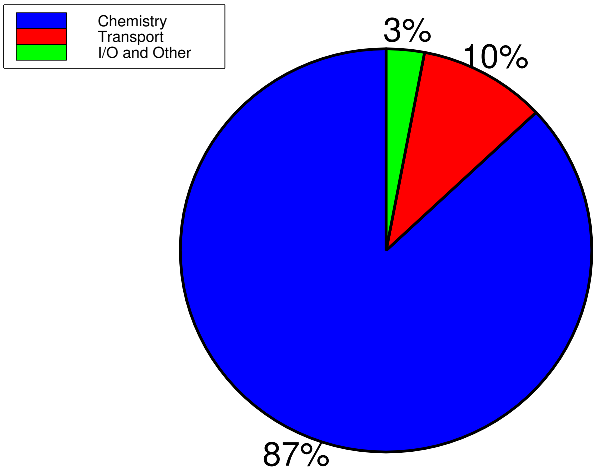

Figure 3.2: Typical CPU load for AQMs

3.2.1 Analysis of Data Dependencies

A time step consists of the following computations: $T_x^1 \cdot T_y^1 \cdot T_z^1 \cdot C^1 \cdot C^2 \cdot T^2 \cdot T_y^2 \cdot T_x^2$. As mentioned in chapter 2 this construction actually represents two merged time steps. The chemical reaction computations represent an embarrassingly parallel problem, however wind affects the chemical concentrations in each grid cell, therefore communication is required at each time step in between transportation computations. Since the chemical module takes most of the computational time, as seen in Figure 3.2, most effort is put into parallelizing this part.

The coupling of the transportation methods puts most restrictions on the parallelization. Using an implicit transport scheme requires a whole row or column of the grid structure along the transportation direction to be present at each processor participating in the computation. Therefore the three possible transportation directions create strong data dependencies. As an alternative, the explicit transport scheme requires information of the neighboring grid cells only, thereby resulting in less communication between the processors, however it uses more specialized computation techniques on the transportation. As can be seen in figure 3.3 on the next page, the transportation calculations for each concentration in step n+1 require knowledge of previous concentrations from from k neighboring concentration fields right and left of the field to be computed, where the typical value for k is 2. During implicit transportation all information from the last step is needed. This dependency limits the strategies applied to domain decomposition.

An important aspect of the performance obtained after parallelization of an AQM is the level of load imbalance during the execution. Several load balancing schemes have been investigated by different authors as seen in chapter 2. An explicit transport scheme provides the basis to apply dynamic

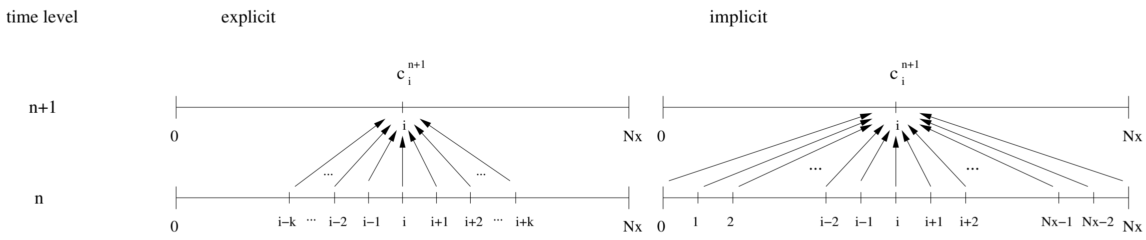

Figure 3.3: Stencil of explicit and implicit transportation scheme on X-transport

load balancing approaches and measure their performance, however we concentrated on the more general implicit transport scheme and investigated various inherent load balancing configurations dependent on the domain decomposition strategies.

3.2.2 Requirements and Constraints

The parallel version of an AQM code has to behave similarly to the serial version. Not only should it use the same input files and produce the same output files, but it is required to integrate the same numerical methods as the serial code. Science modules that have been developed over years of work and are continuously subject to modifications, due to new insights into the chemical and physical processes, need to be replaceable in the parallel version. Therefore the parallelization should provide module independence to a certain level.

The development of the code needs to be possible in serial mode while the production runs for simulation results will be done in parallel. This is another reason for the modularity of the parallelization, no changes in the serial modules should be allowed.

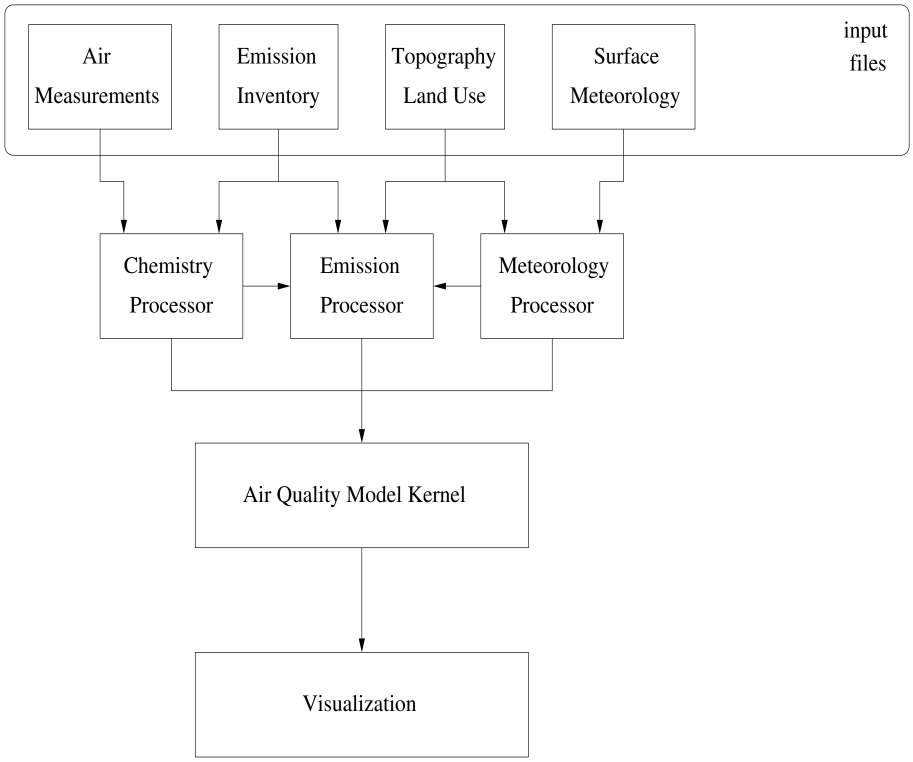

Figure 3.4: STEM-III architecture, as described in [4]

3.3 STEM-III

The Center for Global and Regional Environmental Research at the University of Iowa has developed the air quality model STEM. It has been updated and improved during the past two decades resulting in more advanced versions using the newest scientific methods to accurately simulate regional pollution. The current version, STEM-III, consists of a database, preprocessors, the AQM kernel and postprocessors [4]. The database is given as an input file (either ascii or ioapi) and holds information about air measurements, emission data, topographic information about the region's surface as well as meteorological data. Preprocessors are used for chemistry, emission and meteorology data processing and the combined result serves as input to the AQM kernel. The postprocessors are used to visualize the results of the created output file (ascii or ioapi) and are not part of the parallelization task. The topology of the model is summarized in figure 3.4.

The chemistry processor extracts the chemical property data from the input file such as molecular weight, Henry's constant, equilibrium constants, molecular diffusity and many more details for each species. Based on these data chemical reaction rates are calculated and corresponding Jacobian matrices are built that are processed later by the AQM kernel. Building a gridded emission field from the input data is the task of the emission processor. Longitude and latitude information is transformed to the corresponding projection data. Also natural emissions of NO_x, hydrocarbons and aerosols are calculated. The temperatures and wind fields necessary in these computations are included by the meteorological processor, which also uses the surface topography input data as well as meteorological information in order to calculate the necessary factors.

The basic computations in the AQM kernel can be divided into the following four steps: Transportation along the x-axis, transportation along the y-axis, transportation along the z-axis and chemistry reaction calculations. Each of these four main components uses the given input data to update the concentration array of the chemical species. As we can see, the only changes of data during computation are concentration updates due to the various influencing effects. All the other data used for the calculation is fixed after the input files are read.

The four main computation steps occur during each time step. A simulation uses several time steps representing real time intervals that dependent on the simulation to be carried out.

STEM-III mainly consists of a double loop. After setting up the grid and reading part of the input the outer loop starts, representing a six hour interval. Then more input, necessary for the computations in the current step of the outer loop, is read. The grid is initialized and the inner loop, representing a 20 minute interval, starts, computing the above described four steps: X-transport, Y-transport, Z-transport, chemistry reaction calculations and again Z-transport, Y-transport and X-transport as has been introduced and explained by Carmichael et. al. in [2]. Finally both, the inner and the outer loops end concluding the simulations after writing the computed data to output files.

To parallelize STEM-III it is only necessary to visualize the general structure of the simulation process. The calling of different modules and functions as described above provide the global view to apply a parallelization strategy. All the modules integrated in the process, performing the calculations and their complex structures and interconnections, have no influence on the parallelization approach, because the same routines will be used on each participating processor with access to only the local view of the system.

A more detailed description and explanation of STEM-III and the used mathematical and scientific methods can be found in [10].

3.4 Parallelization of STEM-III

STEM-III uses uniform structured grids. By keeping our parallel implementation general we were able to implement a communication library not only to parallelize STEM-III but providing a solid foundation to parallelize any AQM using structured grids.

3.4.1 Parallelization Strategy

A master-worker parallelization was implemented, where the master handles the reading and writing of the input and output files as well as the distribution of the global data into the desired partition on the workers in the beginning, and gathering of the computed data in the end. The master will not take part in any of the simulation computations. This structuring is consistent with the architecture of a Beowulf cluster where a central box has access to the main storage and nodes connected by fast Ethernet have access to local storage only.

Advantages of this approach mainly lay in future extensions of the implementation, because the idle processor time at the master can be used to manage and organize dynamic load balancing overhead. Clearly the overlapping of input-file reading and simulation computations are another advantage, but the data set used for testing our simulation didn't include time-step dependent input that could have been overlapped with work, therefore we did not investigate this issue in depth.

We also see a few disadvantages of the master-worker strategy. The master stays idle for a long time, the full processing power of the system is not used to its limit. The use of collective communication routines is somewhat irregular, because existing implementations are usually based on equal work sharing constructs. Consider a distribution using a scatter function. In our case the master does not receive anything, naturally a scatter is implemented such that every participating process receives an equal part of the global data. Another disadvantage can be found when viewing the hierarchy structure. In order to group the processors for the communication routines a master-worker strategy has two different categories: the communication routines where the master takes part in and the routines where the workers communicate between themselves.

While most disadvantages deal with implementation difficulties the advantageous overlapping of input reading with computations outweighs the higher time investment in code development, therefore the master-worker strategy was favored over an equal work partitioning.

3.4.2 Domain Decomposition

Very important to the effectivity of a parallelization strategy is the partitioning scheme used. A partition affects load balances, distribution time as well as book keeping effort necessary. We assume a homogeneous parallel system, each machine integrated in the computation has the same processor speed and memory capacities.

XY-partitioning

The XY-partitioning layout can be seen in figure 3.5 on the next page. All points in the cube having the same y coordinate create an X-slice of the cube. Figure 3.5 (a) shows five X-slices combined to a cube of dimension (5,Ny,Nz). A Y-slice is a slice of the cube in YZ-direction consisting of one

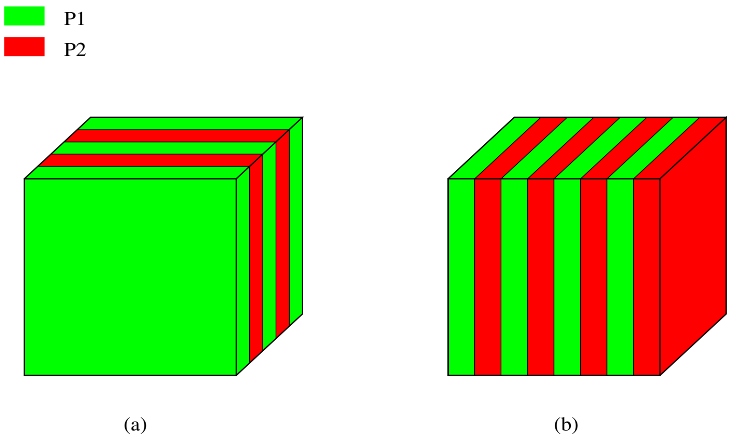

Figure 3.5: Regional model with partitioning in (a) X-slices and (b) Y-slices

grid point in X-dimension. Figure 3.5 (b) shows eight Y-slices combined to a cube of dimension (Nx,8,Nz).

Using the implicit transport scheme the full length of a row or column in any transport direction is needed in order to do the transport calculations. X-slice format is suitable for X and Z-transport while Y-slice format provides the layout for Y and Z-transport. Repartitioning of the cube, consisting of data shuffling from X-slices to Y-slices and from Y-slices to X-slices, is necessary during each time step in order to do the different transport computations. A disadvantage is the parallelization dependency on the number of X- and Y-slices. Not more processors than the number of X or Y-slices given in the grid can be used. Due to the order of computations the chemistry calculations are done when the system is partitioned into Y-slices, therefore there exists a higher parallelization dependency on the number of Y-slices.

The distribution of the X and Y-slices takes place in a circular fashion, assigning the first slice to the first processor and the next slices to the next processors until each processor has a slice and then starting over with the assignment at the first processor.

HV-partitioning

HV-partitioning stands for horizontal-vertical partitioning. Its layout can be seen in figure 3.6 on the following page. Figure 3.6 (a) identifies H-slices in comparison to V-columns in figure 3.6 (b). All points in the cube having the same z index pertain to the same H-slice of the cube. A V-column consists of one grid point in X and Y-dimension and the full column in Z-dimension. The cube shown has dimensions $Nx \times Ny \times Nz = 16 \times 10 \times 8$. Again only a combination of both partitions can be used to do the parallelization of the AQM. A shuffling from H-slices to V-columns and back has to take place during each time step. Here the parallelization is dependent on the number of grid

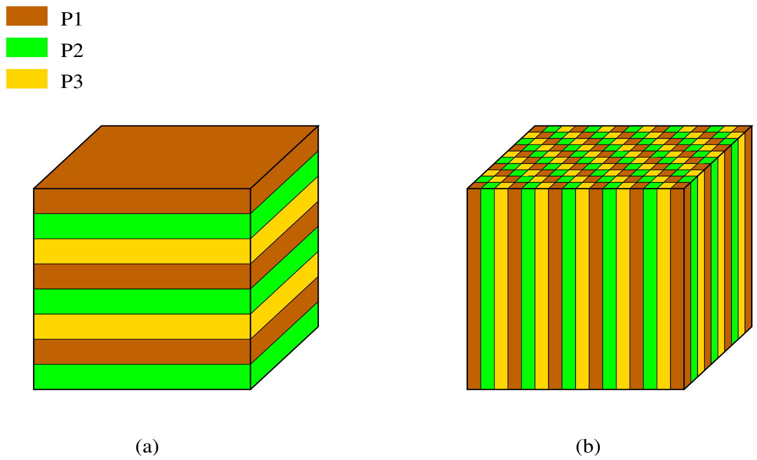

Figure 3.6: Regional model illustrating HV-partitioning on 3 processors

points in Z-dimension - only Nz processors can be active during H-partitioning. But on the H-slice only the X and Y-transport can be done, therefore the time taking chemistry calculations will be done on V-columns and there are almost no limitations here because we have (Nx·Ny) V-columns that can be distributed to the processors.

Distribution proceeds by going through H-slices in round robin fashion and assigning V-columns first going through the x-dimension and then through the y-dimension.

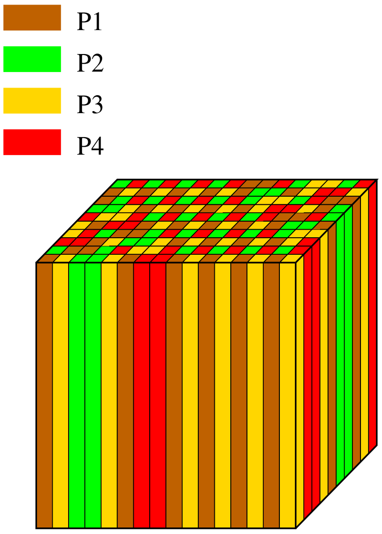

Figure 3.7: Diagonal HV-partitioning on 4 processors

The HV-scheme has several advantages over the XY-scheme. While the dependency on the grid size is softened for HV-partitioning due to the many V-columns available, the inherent load balancing will be more effective as can be seen in chapter 6. The HV-mapping of 3 processors results in diagonal slices of the $16 \times 10 \times Nz$ cube seen in figure 3.6 (b). But applying 7 processors to the same cube gives us a somewhat more arbitrary distribution. Still the used distribution as presented in figure 3.6 can be described as regular. Therefore we also implemented an irregular distribution by assigning the V-columns to the processors by going through the diagonals, for four processors this results in the structure seen in figure 3.7.

All the arrays representing the grid as explained in section 3.1.2 need to be distributed in the beginning of the parallelization in order to do the time step computations. Also a few constants and various (mostly one-dimensional) input variables have to be broadcasted to each process, but the main communication time is consumed by distributing the large arrays described above.

The two presented partitioning schemes (XY and HV-partitioning) were investigated as part of our project. Other partitioning schemes, such as partition into different species, will result in heavy load balancing issues while our partitions include inherent load balancing as long as there are enough slices that can be given to the participating processors.

Although not implemented here another partition possibility needs to be mentioned. When using explicit transportation algorithms we can distribute subcubes of the grid to the working processors resulting in block matrix distribution. Here different load balancing strategies can be easily applied. Furthermore this partition will be more scalable in terms of the number of processors that can be applied. However, we kept our approach more general, i.e. both transportation variants (implicit and explicit) can be applied as is guaranteed by both domain decompositions (XY and HV).

3.4.3 Synchronization

The parallelization of STEM-III is synchronized at many points during computation. The first point of synchronization is the distribution of data after reading the input files. Synchronization is achieved by calling of the communication routines because no communication routine can return before the data movement has finished. Therefore we do not include explicit synchronization points into the computation by `MPI_BARRIER`-calls, because these could increase the idle time of processors. Consider the gathering process at the master. Including a barrier before the gathering makes sure that each process has finished its computation. But the gathering can only take place in serial fashion. Here we can decrease the idle time by removing the barrier. Now a process finished with the computation can send its data to the master right away without having to wait for other processes to finish, using part of the former idle time for transmission and decreasing the gathering time when the last process finishes its computation. Thereby the load imbalance of the system is used to reduce the communication overhead. During reshuffling of data further synchronization is introduced because each process is in communication with all other processes for our domain decomposition schemes, therefore every process has to wait until the last process reaches the reshuffling routine.

Chapter 4

Environment and Dependencies

This chapter explains the computational environments used to implement the parallel strategies of chapter 3 Further dependencies and requirements for portation of our implementation are pointed out.

4.1 Computational Environment

For the parallelization of STEM-III we use the Beowulf cluster of the computer science department at Michigan Technological University.

A Beowulf cluster is a parallel machine consisting of a number of PCs running a stripped down operating system (usually Linux) to increase the performance. The cost-effective PCs are connected with a high-speed network and a switch yielding a fully connected system.

The Beowulf cluster at the CS department at MTU has 64 nodes, each consisting of a 200 MHz dual-processor Pentium Pro with 128 MB memory. The connection is acquired using Fast Ethernet (100Mb/sec) and a 72-port Foundry switch. Each node runs Red Hat Linux, kernel version 2.2.12 as its operating system. [1]

In order to explain results presented in chapter 6 we performed latency and bandwidth measurements on the Beowulf cluster using the program `mpptest`, one of the MPICH examples. Figures 4.1 and 4.2 on the following page present the obtained results, which are summarized in table 4.1. Our measurements are not as detailed for different message sizes as can be found in [21] for the Sun Enterprise, but give an overall estimate.

We have been using the Sun Enterprise 4500 for development and testing of the communication library. The Sun Enterprise 4500 is a shared memory machine. At the department of computer science at Michigan Tech University the Sun has twelve 400 MHz processors, each using uniform access to the 8 GB memory and supporting cache coherence. Equipped with 16 KB instruction and

[1] More information can be found at http://www.cs.mtu.edu/beowulf/

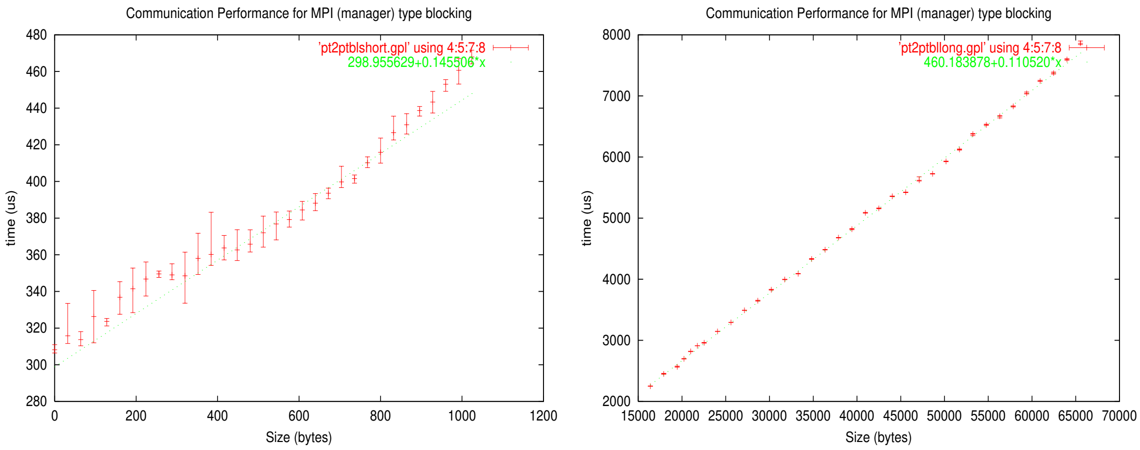

Figure 4.1: Latency and bandwidth measurements using blocking communication

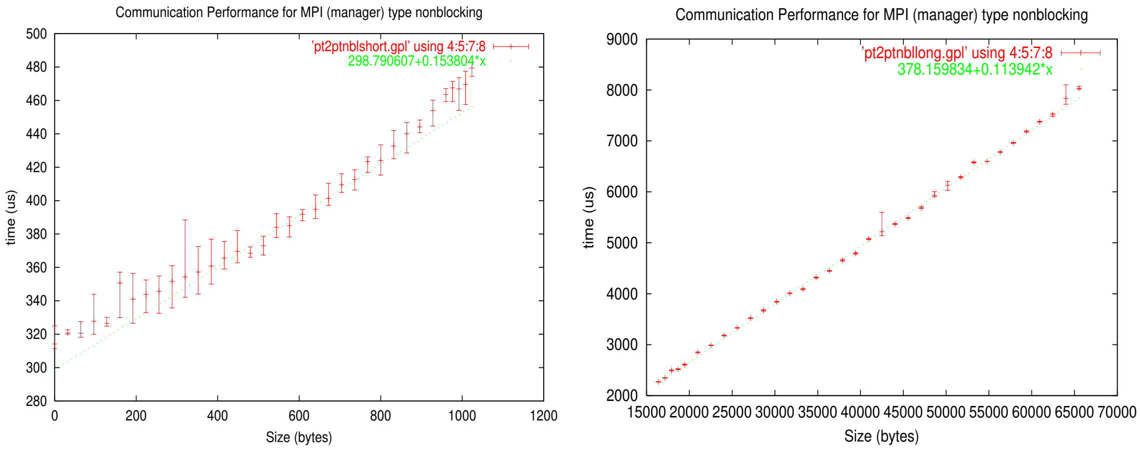

Figure 4.2: Latency and bandwidth measurements using non-blocking communication

16 KB data on chip primary cache for each processor it is a high performance machine providing low latency and high bandwidth system interconnects. As operating system the machine runs one version of the Sun OS 5.7 for all nodes.

VanVoorst and Seidel [21] measured the latency and bandwidth on the Sun Enterprise using Sun MPI, LAM and MPICH. Table 4.2 on the next page describes the different measurements.

Users of the Sun have to be aware that the operating system handles the management of the threads to be run, e.g. assigns the threads to CPUs during computation. Therefore the operating system takes care of some load balancing issues at all times, the overhead due to management of this system is unknown.

We used the Message Passing Interface (MPI) for implementation of the communication library. MPI [11, 17, 19] has been developed by a committee of vendors, implementers and users to act

Type	Message size (in bytes)	Latency (μsec)	Bandwidth (MB/sec)
Blocking	0 - 1K	298.96	6.9
	16K - 64K	460.18	9.0
Non-blocking	0 - 1K	298.79	6.5
	16K - 64K	378.16	8.8

Table 4.1: Bandwidth and latency of blocking and non-blocking communication on the Beowulf cluster

Message size (bytes)	Latency (μsec)	Bandwidth (MB/sec)
0 - 256	6	41.53
256 - 512	10	236.7
512 - 1K	9	158.2
1K - 16K	11	182.0
16K - 32K	29*	197.9*
32K - 1M	35*	208.6*
1M - 2M	277*	219.7*
2M - 4M	594*	225.1*

Table 4.2: Sun MPI cost model (* imprecise due to large variance) from [21]

as a widely used standard library specification for writing message-passing code. It was designed for high performance on parallel machines as well as networks of workstations. There are many implementations of the MPI specification for C or FORTRAN code widely available, some of them are vendor-supplied, others are free. We have been using MPICH 1.2.1 on the Beowulf cluster and Sun MPI on the Sun. MPICH does not offer implementations of all the MPI-2 functions yet. Since our communication library package was developed using Sun MPI where many MPI-2 functions are implemented we had to rewrite part of the package replacing some of these functions with MPI-1 routines. This may have an effect on the overall speed and memory usage because the MPI-2 extensions represent more general communication routines shifting efficiency handling and management toward the MPI-level while the older routines leave most of it at the user level.

On the Beowulf cluster we were able to use the PGF90 compiler for parallel STEM-III and on the Sun Enterprise we used the FORTRAN90 [9] compiler Sun F90.

4.2 MPI Versions

While we only used MPICH for the parallelization, we were able to test the communication library using MPICH and Sun MPI. While MPICH does not yet implement MPI-2 functions such as `MPI_TYPE_CREATE_RESIZED`, `MPI_TYPE_CREATE_HVECTOR` and `MPI_TYPE_GET_EXTENT`, Sun MPI provides most of the functions described in the MPI-1 and MPI-2 standard. Therefore our communication library provides two versions of the file implementing the communication data types where these functions are used. LAM MPI could have been used in order to evade rewriting these functions, however providing compatibility with MPICH, a highly used MPI-implementation, was more important to ensure portability of the library.

We have appended one routine for the shuffling that does not work on any of the MPI-implementations used, because it includes the MPI-2 function ALLTOALLW described in the standard that is not yet supported. Future use of that function could increase the efficiency of the communication during each time step by decreasing the shuffling time resulting in high overhead decrease.

4.3 Communication

Efficient implementation of the communication routines have a major impact on the parallelization speedup as well as scalability. A few different versions for each communication routine were implemented in order to provide high efficiency independent from the architecture used. The different impacts of start up cost versus transmission cost were measured by simply varying the number of messages sent between two processors during one time step.

Communication can be divided into three parts: distribution, shuffling and gathering. All of the arrays described in section 3.1.2 on page 13 as well as some constants need to be distributed from the master to all the workers. Data updates only take place on the concentration array of STEM, thereby seriously decreasing the communication requirements during computation time of the parallel version. While the input data can be distributed once to all the processes, only the concentration array for the chemical species needs to be exchanged during the computation, this exchange is also called shuffling because the data is shuffled from one partitioning into another. In the end of the simulation only the concentration array data need to be gathered at the master in order to create the output file.

Figure 4.3 on the next page, an upshot screen shot, illustrates the different communication steps that take place during simulation. In the beginning you can see the distribution from the master to the 4 workers. Then the master is done and waits to gather the final result. The workers compute the steps as described before and have to reshuffle 6 times before the final result can be gathered, first (right after the distribution) from X to Y or H to V, respectively. The next shuffling can be seen after 400 seconds, from Y to X right back to Y (H to V and back respectively). Another double

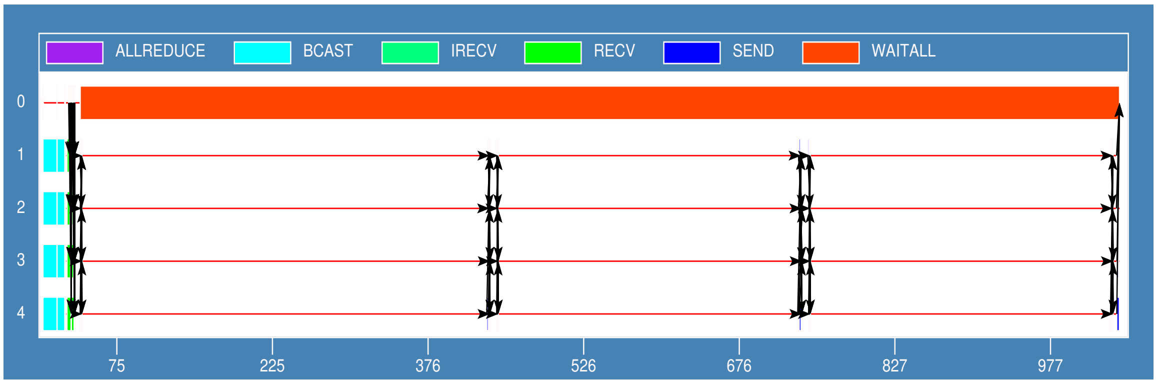

Figure 4.3: Communication during simulation on 5 processors (4 workers), using upshot

reshuffle takes place after 700 seconds and in the end, before gathering, one last reshuffling from Y to X occurs (V to H respectively).

4.4 Grid Dependencies of the Parallelization

The grid size for the test case used in our experiments was:

$$(Nx, Ny, Nz, Ns) = (61, 49, 20, 80)$$

As can be seen here a XY-partitioning can make use of at most 49 processors for X-slice and 61 processors for Y-slice computations, while a HV-partitioning is limited to 20 processors for H-slice computations, but can use up to 2989 processors working on one V-column each. Inherent load balancing only takes place if each processor has more than one X/Y/H-slice or V-column. Therefore the number of useful processors for XY-partitioning is lower than 30, while knowing that the chemistry calculations use most of the computation time, applying more than 500 processors to HV-partitioning still provides inherent load balancing effects.

Chapter 5

The Communication Library

This chapter presents the implementation of communication routines used for parallelizing STEM-III on the computational environment as described in the previous chapter. We explain our implementation strategies and compare the library performance on two different architectures. Detailed descriptions of the implemented functions can be found in the reference manual located in the appendix of this document.

5.1 Overview

Our goal was to provide a communication library for air quality models that can be used for certain parallelization strategies and different architectures. We have developed a library for communication of all the different data types between the processors.

The communication routines are architecture independent but need an MPI-implementation and a FORTRAN90 compiler. In table 5.1 on the following page the different systems our library has been tested on are described.

In the following we name the main assumptions for application of our library to a parallelization strategy of an Air Quality Model.

- The parallelization is conducted on a uniform parallel machine.
- We utilize a master-worker strategy where
  - the master handles input and output,
  - the master distributes data to the workers,
  - workers are responsible for local simulation and
  - the master gathers the data in the end.

Operating Systems	Linux	Sun OS
MPI-versions	MPICH	Sun MPI
Compilers	PGF90	Sun F90

Table 5.1: Computational environments

- Use XY or HV domain decomposition with the given or any user defined mapping.

These assumptions lead to advantages as well as disadvantages. The main advantages are:

- The reading of input and writing of output can be overlapped with computation (not yet implemented).
- The domain decomposition permits application of implicit as well as explicit transportation schemes.
- The structure of the distributed arrays is maintained locally.
- Reshuffling of data is only necessary once per time step and
- unemployment of the master allows for dynamic load balancing managing at the master (currently not implemented).

A few disadvantages were accepted when choosing the implementation strategy:

- Input and output is left serialized,
- the number of workers is limited by the grid size dimension and
- data movement during each time step is a complete exchange of data between all workers.

Using an explicit transportation scheme can decrease the data exchange needs during simulation. However, we wanted to apply a general scheme without limitations to any scientific computation model. Furthermore our timings show that the overhead due to the shuffling is only a small percentage of the overall communication overhead.

5.2 Communication Routines

Communication for parallelization of AQMs can be divided into three parts: distribution, shuffling and gathering. The effectivity of the parallelization depends on each of these routines since communication represents the overhead of the parallelization.

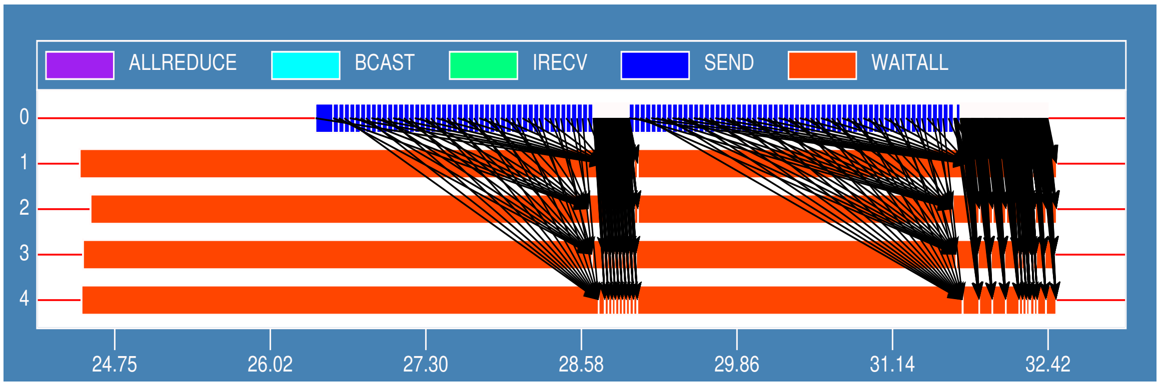

Figure 5.1: Distribution of all arrays using version 1 from the master to 4 workers

5.2.1 Distribution

All data types need to be distributed from the master that reads the input files to all the working processors, depending on the partitioning of the model. Four different distributions are provided: X-slice distribution, Y-slice distribution, H-slice distribution, V-column distribution. There exist three different routines for each of the data type distributions:

Version 1 is a routine distributing the arrays slice by slice or column by column, respectively, from the master to the participating processors.

Version 2 implements a routine distributing all the data needed on each processor in one message from the master to that processor and

Version 3 specifies a routine using the MPI-function SCATTER to distribute the data.

Version 1 distribution is illustrated in figure 5.1. The data is taken from XY-distribution, however HV-distribution results in a similar picture. The full distribution is partitioned into two parts, X-distribution and Y-distribution.

Version 2 distribution can be viewed in figure 5.2 on the following page. Again the data is taken from XY-distribution, here HV-distribution results in the same graphic but on different time lines. Graphs of version 3 distribution were not included since messages are not displayed in these graphs, because collective operations are intractable to `upshot`.

The speed of these routines depends on the MPI implementation as well as on the architecture used. While message passing machines usually have a high start-up cost (latency) in order to send messages and therefore usually show better results by sending few messages with high amount of data, shared memory machines can show good results by sending many messages with few information.

Another important factor is the creation of messages. To send a slice of the array we need to create an MPI-type that represents the offsets and positions of the slice to be sent in the large

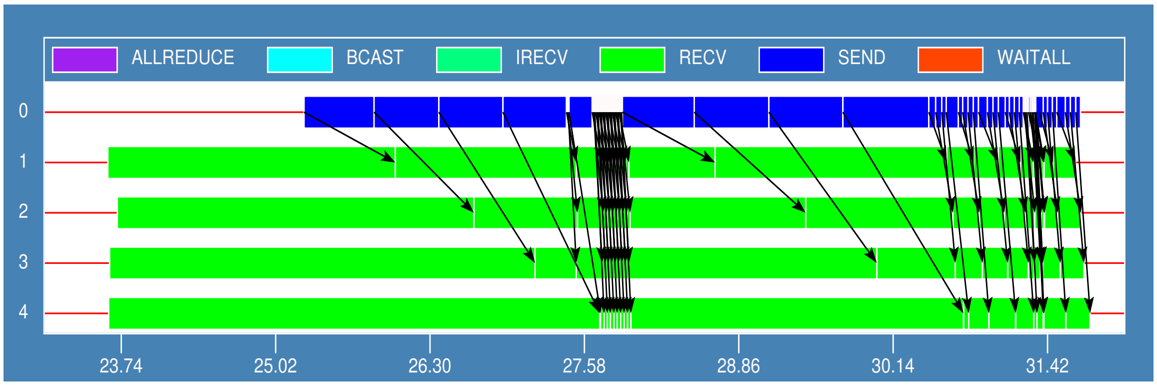

Figure 5.2: Version 2 distribution of all arrays on 5 processors (4 workers)

array. The data for one slice is spread out over manageable parts of the global array. By sending all information needed at one process, data spread out over the full cube needs to be arranged into one message by creating the associated MPI-type. The handling of these data types is much easier while the hidden copying done in the MPI-layer may result in time factors that reduce the effectivity gain of the smaller amount of messages that have to be sent.

Distribution routines for 2D, 2DN, 3D, BD and 4D data types as described in section 3.1.2 on page 13 are provided, each implemented in the 3 versions described above. An exception is the 2DN horizontal HV-distribution that is not necessary because 2DN arrays are not used for horizontal transportation. Also 2D data arrays do not need to be distributed for the horizontal part of the HV-distribution, we still implemented one routine using MPI_BROADCAST (no other versions are necessary).

5.2.2 Gathering

In the end of the calculations done by the working processors the final data need to be gathered by the Master. Since the only data important for the output file is the concentration array, we only need to provide the gathering routines for the 4D data types. Just like the distribution, gathering depends on the four different partitionings described in this document, therefore four different routines are provided: X-slice gathering, Y-slice gathering, H-slice gathering and V-column gathering. Again each of these routines is implemented in three different ways:

Version 1 using one message for each slice or column to be gathered,

Version 2 using one message for each processor that data is gathered from and

Version 3 using the GATHER function provided by MPI.

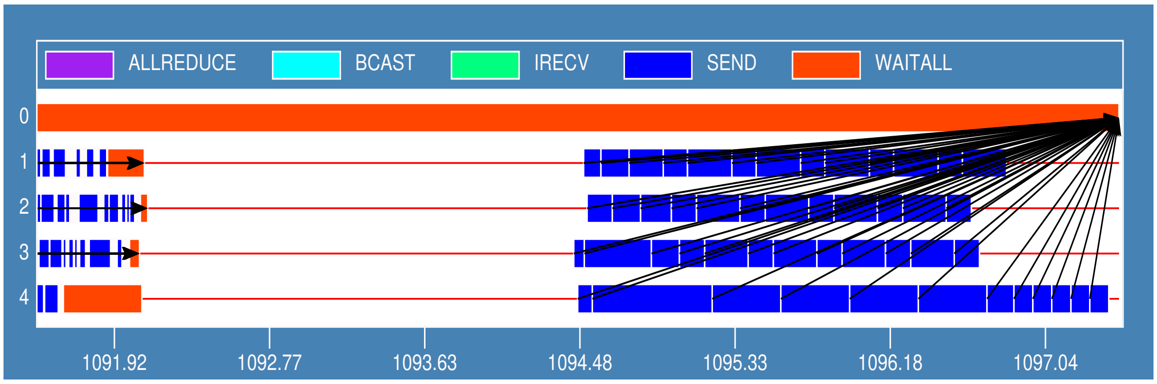

Figure 5.3: Version 1 gathering of the 4D array on 5 processors (4 workers)

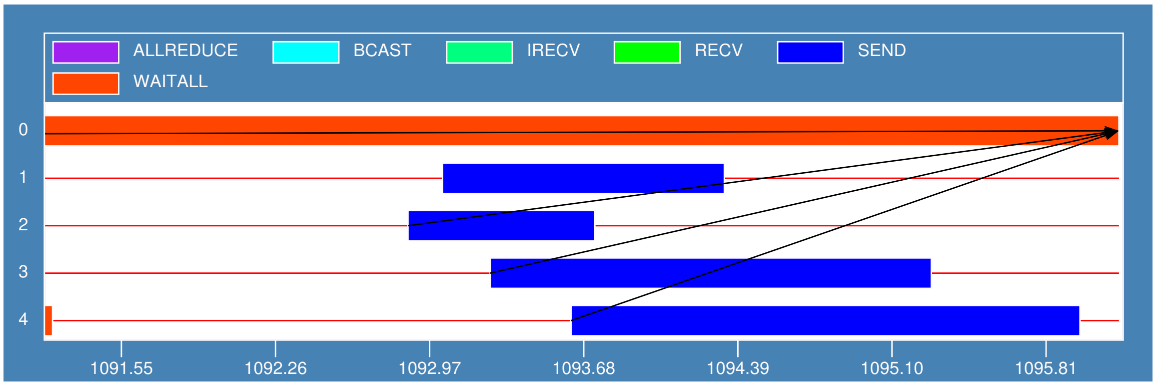

Figure 5.4: Version 2 gathering of the 4D array on 5 processors (4 workers)

The speed of the routines again depends on the architecture as well as the MPI-implementation used.

Version 1 gathering is illustrated in figure 5.3 and can be compared to figure 5.4, the graph of version 2 gathering. Again the illustration is taken from XY-partitioning (X-partitioning in this case) as can be verified when counting the messages sent from each processor, since we only have 20 vertical layers, each processor only sends 5 messages in HV-partitioning.

5.2.3 Repartitioning

In order to repartition the model during the computation two different routines are supplied, one to shuffle data from X-slices to Y-slices and and the other to shuffle the data back, from H-slices to V-columns and back respectively. Again the reshuffling methods are only needed for the 4D arrays that hold the concentrations of the chemical species. Reshuffling is an ALLTOALL communication, each worker needs to exchange data with every worker. These routines were implemented using different methods.

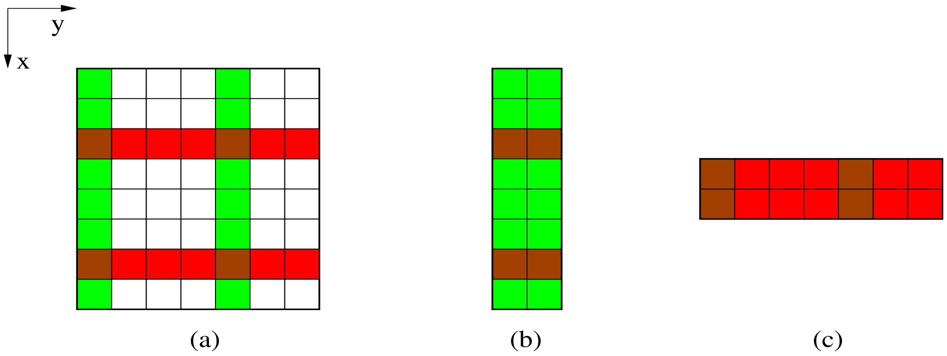

Figure 5.5: Location of data in the arrays - (a) global, (b) local x, (c) local y

For XY-shuffling the following 3 versions were implemented:

Version 1 sends many messages, for each local slice one message to each worker,

Version 2 creates one message including all information that needs to be exchanged between two processors and

Version 3 copies the data to a buffer and uses the ALLTOALLV function provided by MPI.

The MPI ALLTOALL-functions can only be used on a buffer that holds the information that have to be exchanged in a certain form, therefore we need to copy the data from the array at the processor into that buffer. The speed of that routine will only depend on the architecture. The routines using copying are provided because there exist architectures consisting of nodes with high computational power and using slow communication networks, therefore the speed of the routine would depend mainly on the communication of the data and therefore the possibility is given that the overall time of the function is less than one of the other routines.

Figure 5.5 illustrates the exchange necessary between two workers in XY domain decomposition. The first picture represents a top view of the cube model. When using 4 workers, worker 3 gets the red X-slices and worker 1 the green Y-slices. The data exchanged between these two workers during reshuffling is marked brown. Figure 5.5 (b) displays the local data at worker 1, figure 5.5 (c) illustrates the local data at worker 3. As can be seen, the data to be exchanged is arranged in a row or column at one processor and with and offset in row or column at the other.

HV-shuffling routines are of a different format as described in the following:

Version 1 copies the data to a buffer and sends one message to each worker including all data for that processor,

Version 2 creates one message for each worker using the defined data types and

Version 3 copies data to a buffer and uses the ALLTOALL function provided by MPI.

The different implementations for HV and XY-shuffling in version 1 are due to the much larger amount of V-columns than Y-slices. Sending one message for each V-column to each worker (as has been done for each Y-slice in the XY-shuffling) would result in very many short messages (including only a few concentration fields) and measurements showed the this leads to high transfer times.

5.3 Building Array Structures for Distribution

To distribute the large arrays efficiently and consistently from the master to the workers, MPI data structures had to be implemented. These fully depend on the kind of distribution to be achieved. But there are two ways to build them and some interesting timing results were discovered. To build large data structures one can use already implemented data structures, e.g. taking a slice and putting it together with other slices that have to be transferred to one processor. This results in slow array access, therefore the implemented arrays using this version are called “slow”. A second way is to recreate the full data structure completely, going through the dimensions one by one in order to put the necessary data together (“fast” arrays). The latter achieved better communication performance, the arrays are send and received faster. This is due to the copying and handling within the MPI layer.

Figures 5.6 on the following page, 5.7 on page 36, 5.8 on page 37 and 5.9 on page 38 show the time differences when using arrays put together using slices versus new built structures. Interestingly for the smaller structures like 2D, 2DN, 3D and BD arrays the differences we can see (figures 5.6, 5.8) do not depend on the different array implementations but on run time environment differences, because the times MPI-buffering and caching is unrecognizable in comparison to latency and bandwidth factors. However, on the larger 4D arrays (figures 5.7, 5.9) the difference clearly exists.

We can see differences in the handling of these data types in version 2 and 3, only the shuffling in version 3 does not depend on the array implementation at all as can be seen in the associated picture in figure 5.9, because the collective version does not use the structures, but copies parts of the local data right into a new buffer that is sent using an MPI ALLTOALL-function.

5.4 Using the Library

The library presented includes STEM-III specific code that can serve as an example for implementation of a parallel AQM. However, the main library functions are straightforward communication routines found in `mpi_commlibrary.f`. The version 1 routines represent the most general communication routines, both other versions heavily depend on the mapping used and the data types declared based on that mapping. In order to use these routines the mapping function has to be implemented,

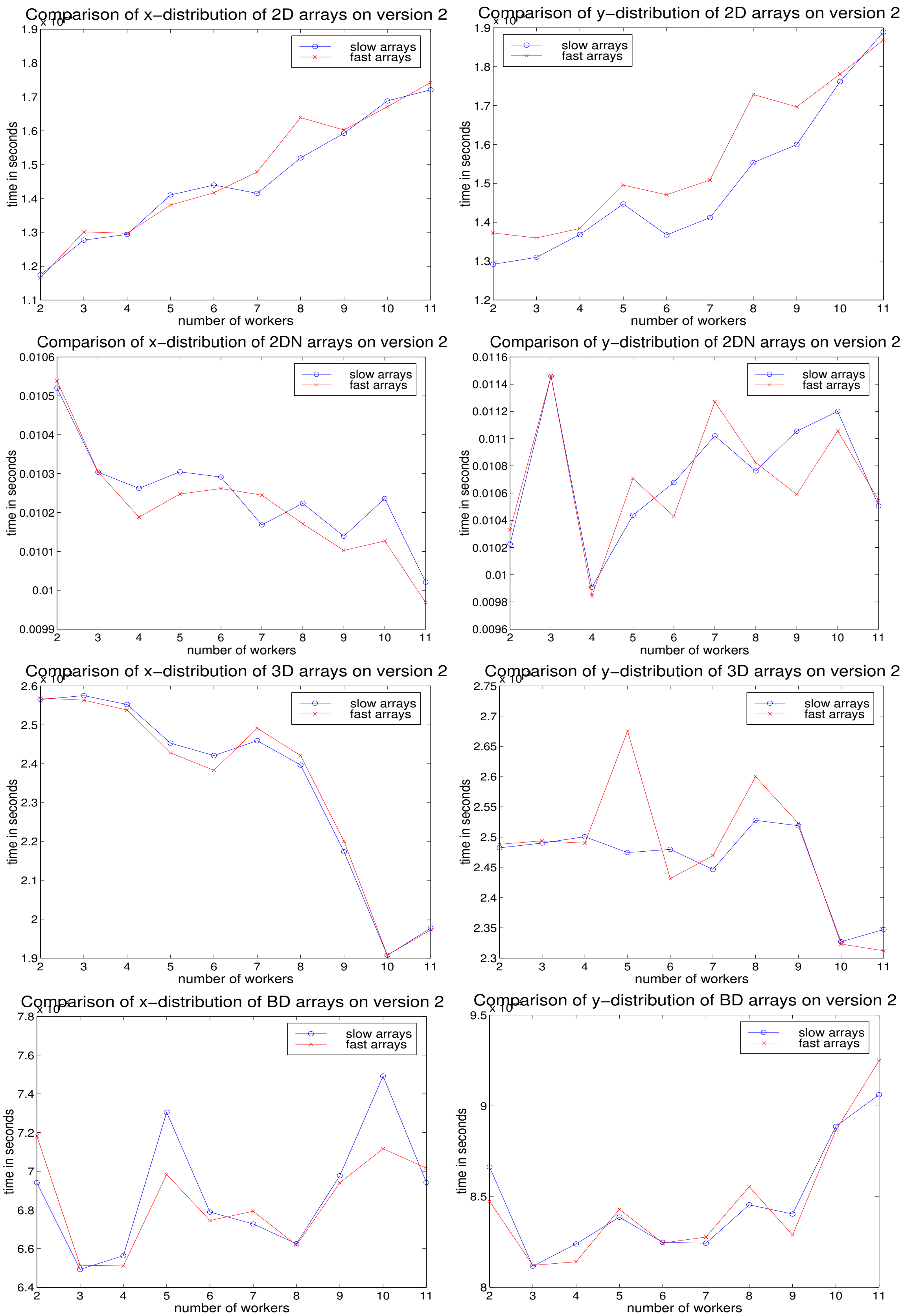

Figure 5.6: Time comparisons of slow and fast array implementations for 2D, 2DN, 3D and BD version 2 XY-distribution on the Sun Enterprise

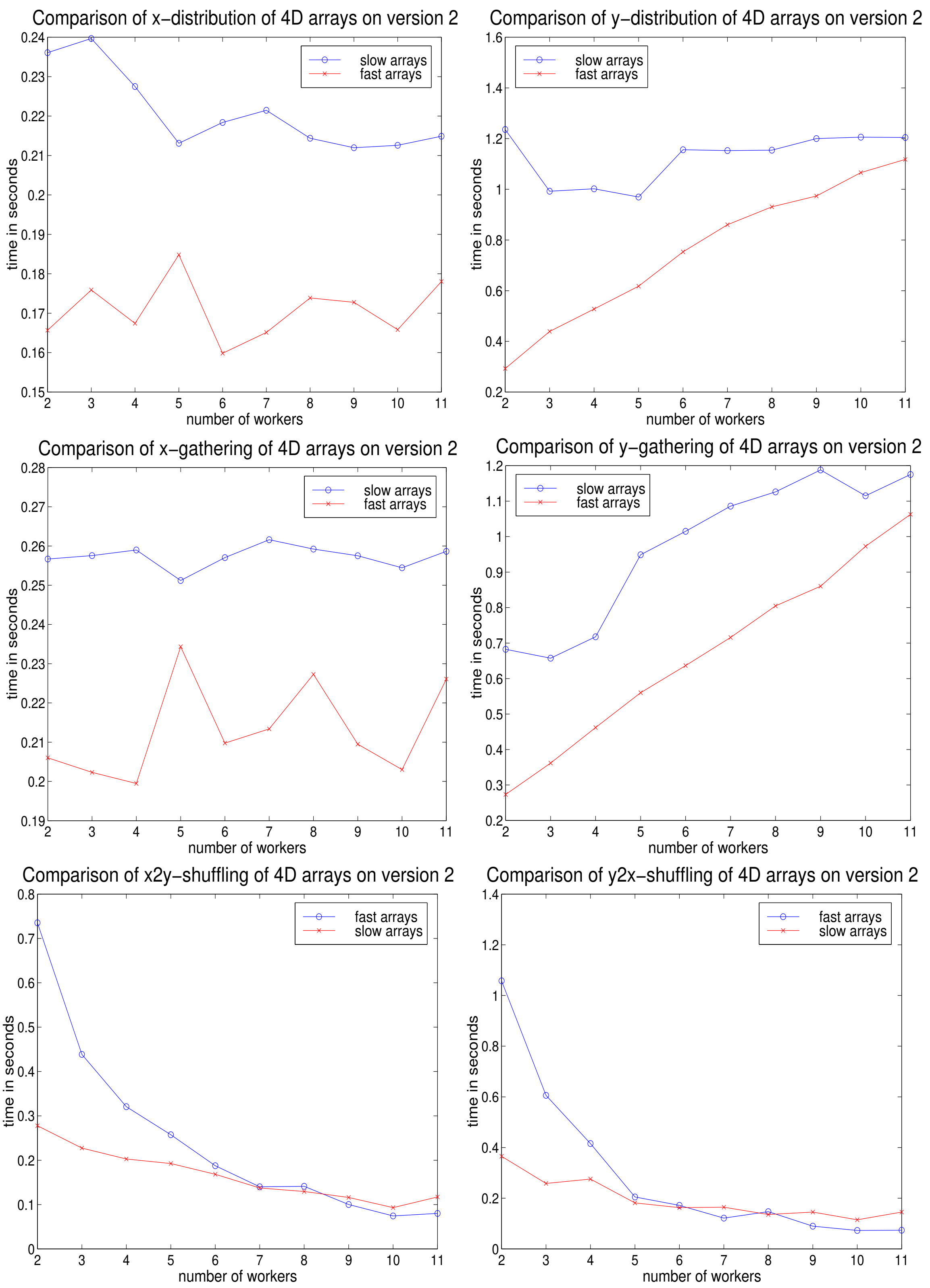

Figure 5.7: Time comparisons of slow and fast array implementations for 4D version 2 routines using XY-partitioning on the Sun Enterprise

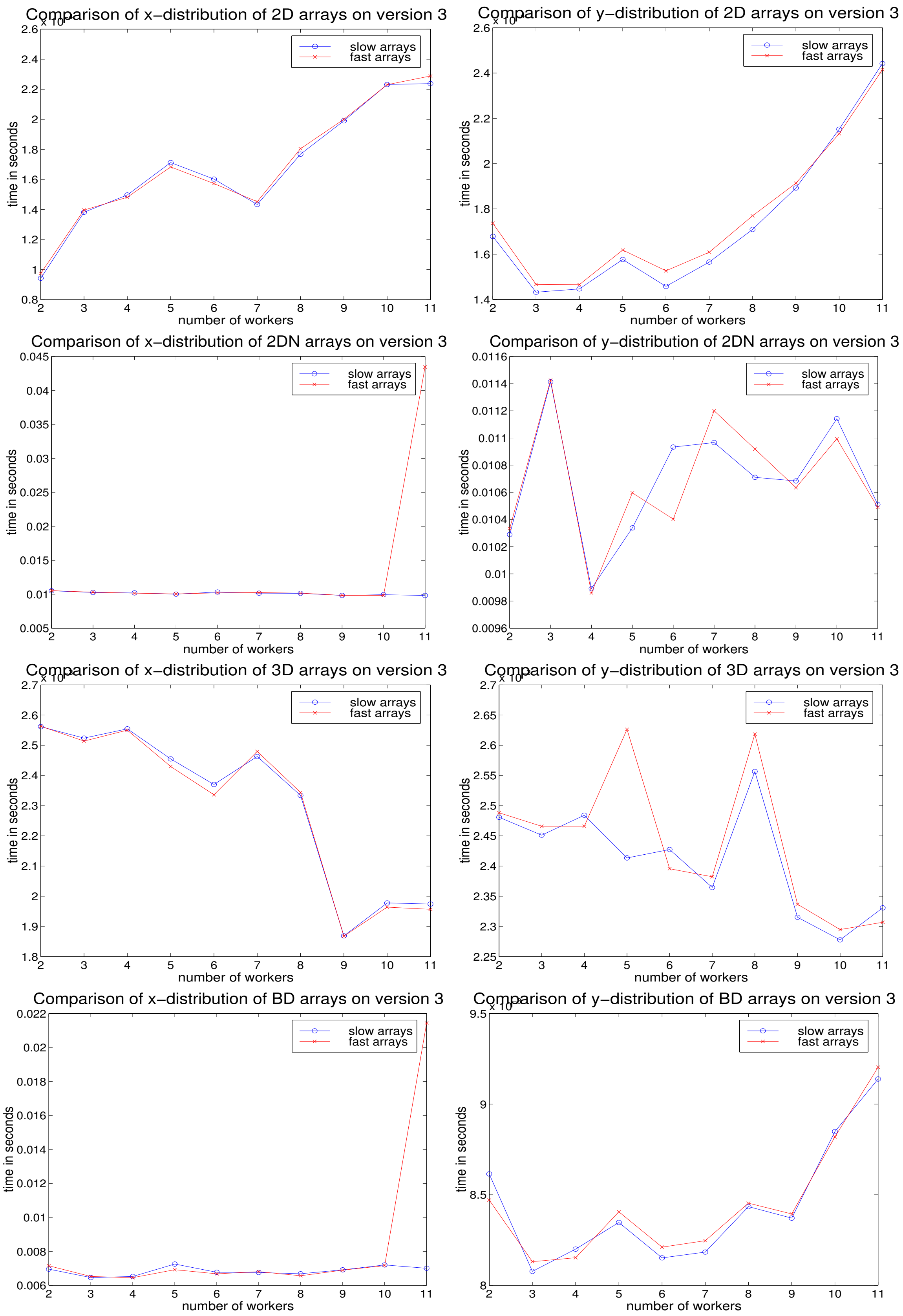

Figure 5.8: Time comparisons of slow and fast array implementations for 2D, 2DN, 3D and BD version 3 XY-distribution on the Sun Enterprise

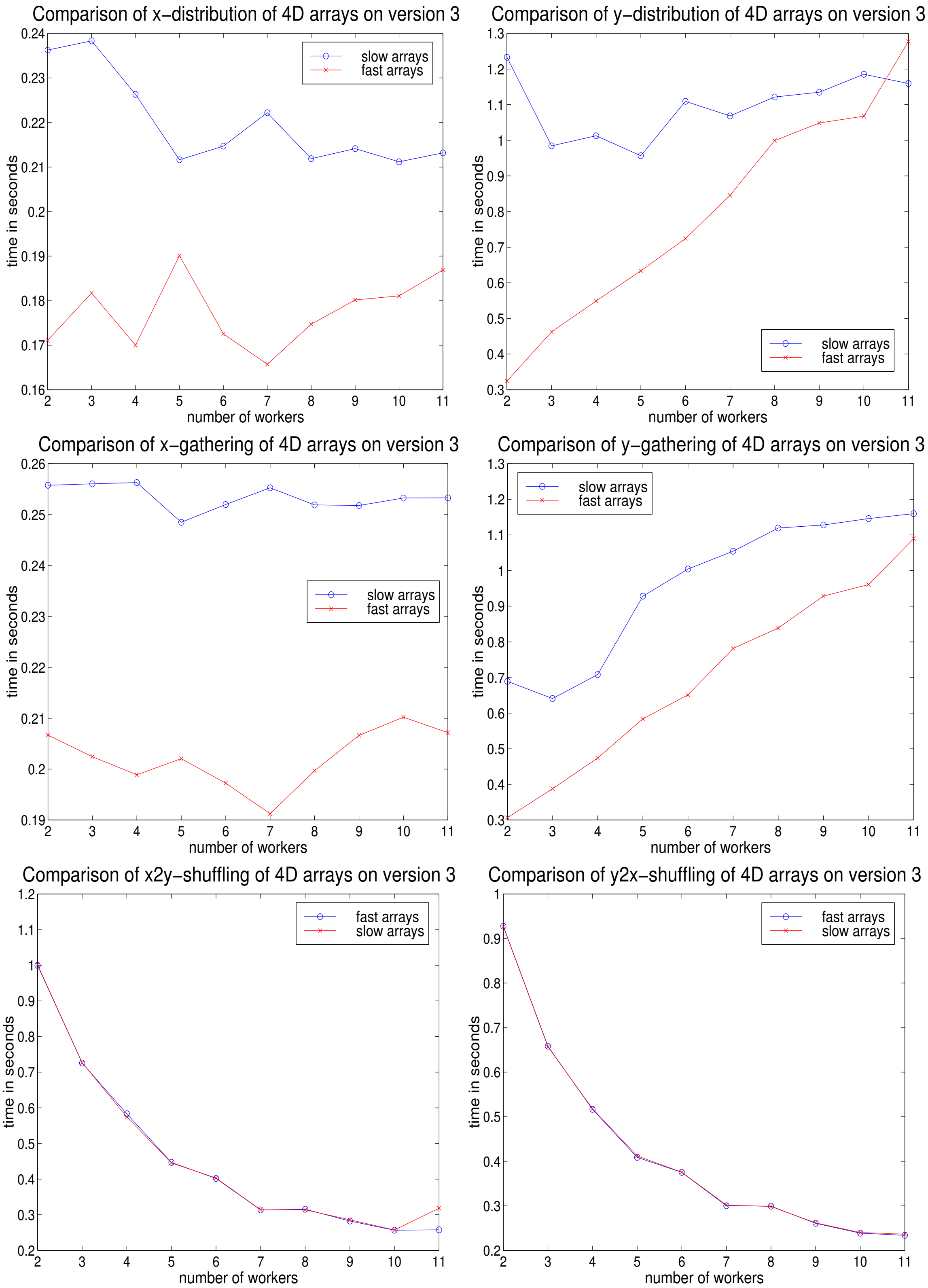

Figure 5.9: Time comparisons between slow and fast array implementations for 4D version 3 routines using XY-partitioning on the Sun Enterprise

example mappings can be found in `mpi_util.f` or `mpi_util_mpich.f`. These files also implement the communication data types used, it is strongly recommended to keep the given mappings, thereby enabling full capacity of the library - any change in mapping requires changes in the data types.

A make file is included in order to compile the STEM-III code. Table 5.2 lists the different make options, the serial STEM-III code as well as the parallel extensions have to be installed as described in the README and INSTALL files.

In order to change or extend the routines in the library please pay attention to the implemented data structures. Version 1 routines (standard routines) are dependent on slice or column structures only. Versions 2 and 3 have a strong dependency on the mapping used, their communication structures are sets of slices or columns that are regularly distributed over the large arrays. As soon as a different mapping is used these distributions, implemented in offsets and displacements, have to be updated in order for version 2 or 3 to work.

In the current STEM-III code there is no need for distribution of liquid and particulate matter arrays, but only the array storing gas phase chemicals. In order to distribute, gather and shuffle more than just the one concentration array, new data types have to be declared for the other structures in case they have a different size. If the other concentrations are stored in similar arrays our implemented routines can be used without changing data structures.

5.4.1 File Structure

The library is partitioned into a number of files. These files can be divided into two categories: STEM specific files and general files. The STEM specific files can serve as example files to document the necessary function structure that is used for parallelization. Files belonging into this category are: `mpi_communication.f`, `mpi_memalloc`, `mpi_aq_driver_xy.f` and `mpi_aq_driver_hv.f`.

The rest of the files seen in table 5.3 are general files. As main component of the library, `mpi_commlibrary.f` encloses the major level of communication routines used in the parallelization.

Option	Description
XY	compile code for XY-partitioning and build parallel STEM-III XY-executable
XYset	compile code for XY-partitioning and build XY-setup executable
HV	compile code for HV-partitioning and build parallel STEM-III HV-executable
HVset	compile code for HV-partitioning and build HV-setup executable
serial	compile serial code and build STEM-III serial executable
all	make all the above
clean	remove all *.o files

Table 5.2: Makefile options

File	Description
mpi_xy_setup.f	setup program for XY-partitioning
mpi_hv_setup.f	setup program for HV-partitioning
mpi_util.f	communication data types using many MPI-2 functions
mpi_util_mpich.f	communication data types using functions implemented by MPICH
mpi_communication.f	STEM-III specific communication functions
mpi_commlibrary.f	general communication library
mpi_memalloc.f	general memory allocation functions
mpi_aq_driver_xy	parallel driver using XY-partitioning
mpi_aq_driver_hv	parallel driver using HV-partitioning

Table 5.3: List of library files

Here routines for each of the data structures used in air quality models are implemented and therefore this file is the main result of our work. The description of the partition mapping and definitions of necessary global variables can be found in `mpi_util.f` and `mpi_util_mpich.f`. Here the number of workers for any partitioning is decided on and most of the global variables necessary for parallelization are initialized using the subroutines as described the appendix. The reader is welcome to implement new partitionings, however in order to use the communication library to its full extent the routines have to access and initialize the same global variables, therefore our routines should be used as examples.

The setup files (`mpi_xy_setup.f, mpi_hv_setup.f`) do not add to the library functionality, but they are necessary to chose the best and fastest implemented communication version for the architecture the AQM is parallelized on. The setup routines time each communication version and write the result into an output file called `XY_versions` and `HV_versions`. One of these files is read in during initialization of the associated partition where the communication versions are chosen.

For the different parallelization strategies different files are used. The main programs are started in `mpi_aq_driver_xy.f` and `mpi_aq_driver_hv.f`. Both files use the same function calls but different modules since the modules are set up dependent on the partition. Therefore the XY-partitioning only calls XY-modules and the HV-partitioning calls HV-modules while both modules may contain the same function names, however different parameters are used.

In table 5.4 on the next page the division of the modules into above named files is given. Note that we have collected the pairwise associated modules together in one file.

In order to apply a new partitioning one can either introduce a new module or change one of the given ones to that new partition strategy with respect to the module interrelation as described in the next section.

Module	File location
`XYParallelCommunication`	mpi_communication.f
`XYParallelMemAlloc`	mpi_memalloc.f
`XYCommDataTypes`	mpi_util.f mpi_util_mpich.f
`XYCommunicationLibrary`	mpi_commlibrary.f
`XYParallelDataMap`	mpi_util.f mpi_util_mpich.f
`HVParallelCommunication`	mpi_communication.f
`HVParallelMemAlloc`	mpi_memalloc.f
`HVCommDataTypes`	mpi_util.f mpi_util_mpich.f
`HVCommunicationLibrary`	mpi_commlibrary.f
`HVParallelDataMap`	mpi_util.f mpi_util_mpich.f

Table 5.4: File location of the modules

The given library functions account for distribution of the necessary arrays as well as data exchange during time steps and gathering in the end. Usually a few more constants and 1-dimensional arrays need to be broadcasted as can be seen in `mpi_communication.f`. Here STEM-III specific routines are implemented as well as the choice of library routines.

5.4.2 Module Interrelation

From a user perspective only one module needs to be known: the `ParallelCommunication`. It includes all the other modules using the FORTRAN `use`-statement. In order to build a parallelization only this module has to be included while functions from other modules still can be called.

Figure 5.10 displays the interrelation. By describing the mapping module `ParallelDataMap` is of high importance to any routine participating in communication. While module `CommDataTypes` is only needed in the actual communication routines given in `CommunicationLibrary`, `ParallelMem-Alloc` is a stand-alone-module, its functions can be directly called by the user and are not required in any other routines.

A few dependencies between modules have to be described. As `ParallelDataMap` specifies the partitioning, the global variables initialized here are to be used by the communication routines. However, general mappings have not been used for the whole communication library. In `CommunicationLibrary` only the version 1 (_v1-functions) uphold the mapping generality. As can be seen when looking at the `CommDataTypes`-module, the communication data structures describe slices, columns and sets of slices or columns. The latter are implemented in a mapping-dependent fashion, sets of slices or columns have to be formed generally by combining the data local to one processor in an abstract way. When using an irregular mapping we lack the ability to construct these

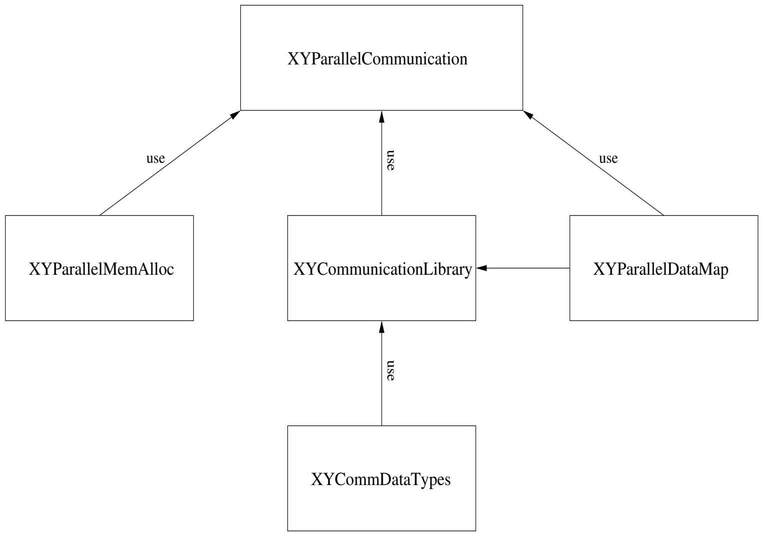

Figure 5.10: XY-module interrelation

Variable name	Description
Nprocs	number of processors available
NXworkers	number of workers for X-partitioning
NYworkers	number of workers for Y-partitioning
MyId	processor id
Master	master processor (processor with id 0)
XWorker	true for all processors with id less or equal NXworkers (excluding the master)
YWorker	true for all processors with id less or equal NYworkers (excluding the master)

Table 5.5: Global variables for XY-partitioning

sets to match the communication needs, therefore only the routines sending slice by slice or column by column (version 1 routines) work for any general mapping, while other routines are ignored.

5.4.3 Global Variables

A major impact on understanding of the main program is based on the global variables used. These are declared in the `ParallelDataMap`-module, in table 5.5 only the variables for XY-partitioning are displayed.

To parallelize an existing AQM program a profound understanding of the partitioning strategy is required. The AQM program includes routines that require a certain layout of the data, this has to be included in the partition considerations. The global variables listed in table 5.5 help to create an environment where partitioning and repartitioning schemes can be applied.

5.5 Library Behavior on the Beowulf Cluster

In the following the behavior of the library functions on the Beowulf cluster are described. We have built a setup program that tests all of the functions on some specified number of workers. It is used to initialize the fastest versions of the communication for the underlying architecture, the chosen grid size and the number of nodes specified. The setup program can also be used to time the communication routines, comparing results when applying different parameters. The setup program has been used to report the timings in this section. It executes each communication routine 10 times and reports the fastest result. We created a script running the setup program for every number of workers possible, ranging from 3 to 61 workers for XY-partitioning and from 3 to 64 workers for HV-partitioning. Other features of the setup program is checking the communication routines for errors in distribution; when applying a user implemented mapping it can be checked for possible programming errors in the book keeping process. In the end the setup program writes the version file (`XY_version` or `HV_version`), that names the fastest version of each routine. These features and the many synchronizations resulted in a overall run time of 7 hours and 30 minutes for the XY-mapping over the range of processors applied.

The neighboring table displays the input and architecture parameters used for the presented measurements. Our measurements have taken place during normal usage of the cluster, i.e. some data inconsistency may be explained by other users influences on the network.

grid size	(61,49,20,80)
MPI-version	MPICH

Table 5.6: Setup parameters for Beowulf cluster

For distribution and gathering global times are reported, by measuring time in between a synchronized start of the routine (using a barrier) and until each of the processors finished transmission. Therefore these global times reported are the ideal times when all processors start the routines at once. During real simulations these times will change since some processors start the routine but stay idle until receiving their from other processes that have not entered the routine yet.

The version 2 and version 3 distribution of 2D arrays as seen in figure 5.11 on the next page increase with higher number of processors involved in the communication. This can be explained with the start-up cost of messages that is higher than the transfer cost of these small array structures. Version 1 of this routine takes a much higher amount of time compared to the other versions but keeps leveled due to the static number of messages sent (one for each X or Y-slice). The timings for 4 and 8 processors using version 1 may show influences of other users or the operating system on the communication, when taking in consideration that the communication times are very small and already tiny delays can introduce these peaks. Clearly the latency has high influence on the distribution time for 2D arrays because of the small message sizes.

Figure 5.12 on page 45 presents the collected measurements for distribution of 2DN arrays into XY-partitioning. We believe the size of transferred data to be the reason for the leveled times.

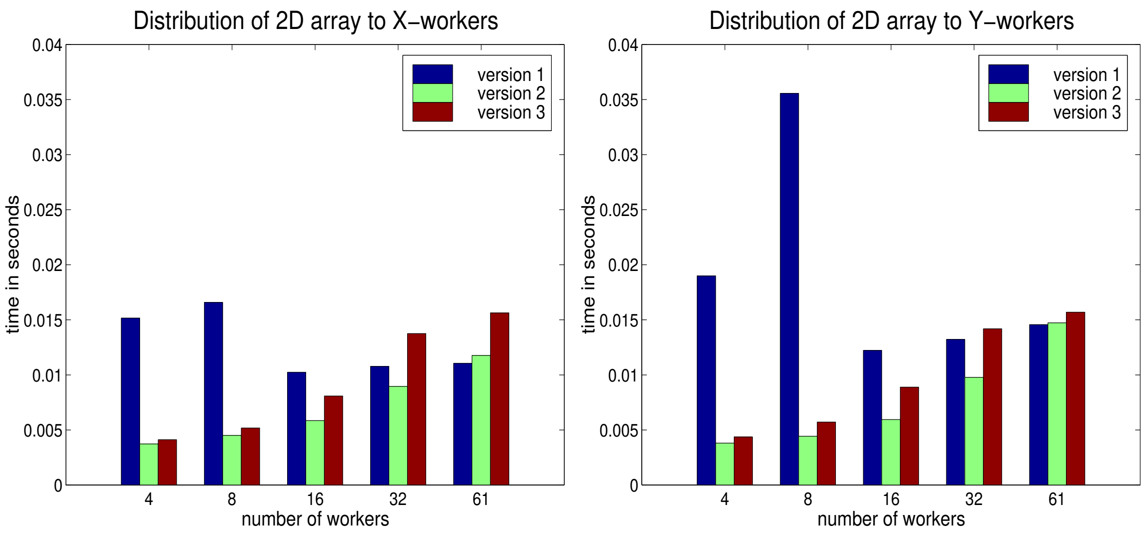

Figure 5.11: Global times of 2D XY-distribution

The small differences of global distribution times using different processors can almost be ignored. However a difference between X and Y-distribution can already be recognized, the structure necessary for Y-distribution seems to require more decomposition time. Detailed investigations showed that cache misses are responsible for these differences. When accessing data in the global array in X-slices, data is located closer together. For Y-slices neighboring concentration fields are separated by Nx intermediate fields in the global array resulting in more cache misses. The effects can also be seen on any of the larger arrays in this section. While for the global finish time version 1 seems to be the fastest of the three implementations, we realized that most decisions of our setup program favored version 2 over version 1 due to the different timings at all processors involved in the routine. The decision for the fastest version is based on the finish time at each processor and the global time on the master, however providing equal opportunities to each processor to influence the overall decision. If version 2 is favored, clearly more processors finished earlier when using single messages (version 2), therefore the global times reported here are only part of the whole picture. We have to keep that in mind when comparing different versions.

Distribution of 3D arrays (figure 5.13 on page 46 is clearly faster than that of 2DN arrays since the third dimension (z for 3D) is smaller than the species dimension (see table 5.6 on the preceding page). However the behavior is similar to 2DN arrays, again a leveled distribution time is found throughout the number of processors used. Also version 2 is preferred in the decision over version 1 due to the difference between global and local finish time mentioned above.

BD arrays (figure 5.13 on page 46 are of same magnitude as 2DN arrays and therefore behave similarly. Interestingly version 3 is never chosen by the setup program for any number of workers while in 2DN and 3D there were a few “exceptions” for certain numbers of workers tested. Most

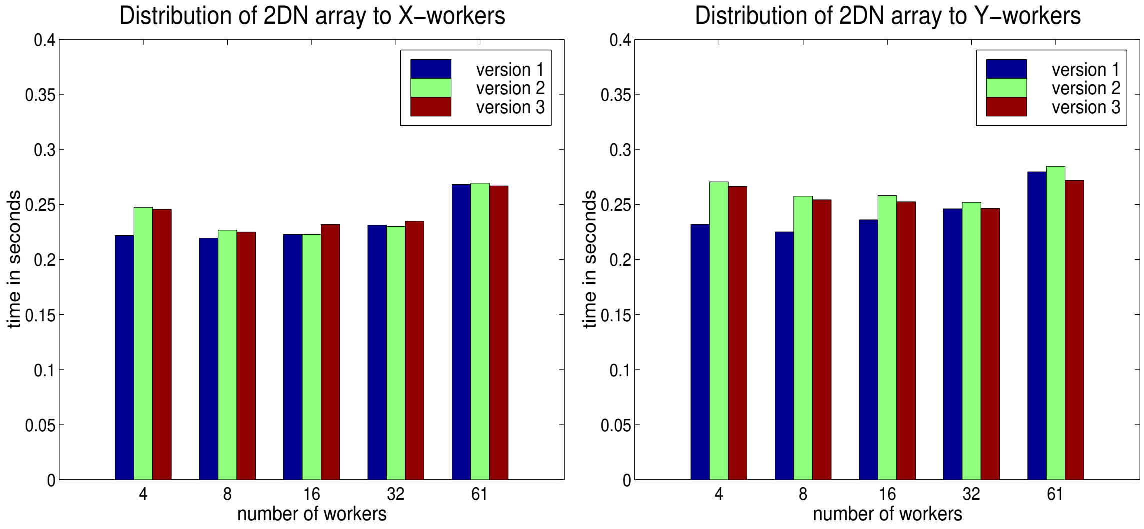

Figure 5.12: Global times of 2DN XY-distribution

decision favor version 2 again. There is no formation to be seen when the number of processors increase, the decision fluctuates between version 1 and 2.

Most impact on performance is founded on the 4D distribution since it takes the largest amount of time. As can be seen in figure 5.14 on page 47 the time of the routine depends on the number of processors. While during Y-distributions the finish times between versions vary, X-distribution does not show much difference. As before the decisions of the setup program vary between version 1 and 2, here giving no consistent preference to one of them.

Interestingly 4D gathering as seen in figure 5.14 does not depend on the number of workers involved like its opposite, the distribution. The times seem somewhat leveled and for the first time we can see a major difference between X and Y-partitioning. The majority of the decisions made by the setup program favor version 1. However, version 3 is in second place and succeeds more often than in other cases. The poor performance of version 2 stays unexplained. In case of gathering the global time is only measured at the master, none of the workers gets to vote.

For shuffling only local times on processor 1 are reported in figure 5.14, because no application will ever achieve ideal performance. Load imbalances and different computation start times after distribution will be the reason for idle times until messages are received. Again interesting observations can be made by comparing times for different worker numbers. When using few workers the full amount of data needs to be reshuffled using few messages of large size while when employing many workers many small messages are send in between them. However these many messages can be overlapped since it is an all-to-all communication, where independent pairs of processors can communicate with each other at the same time. Therefore the number of workers influences the finish time.

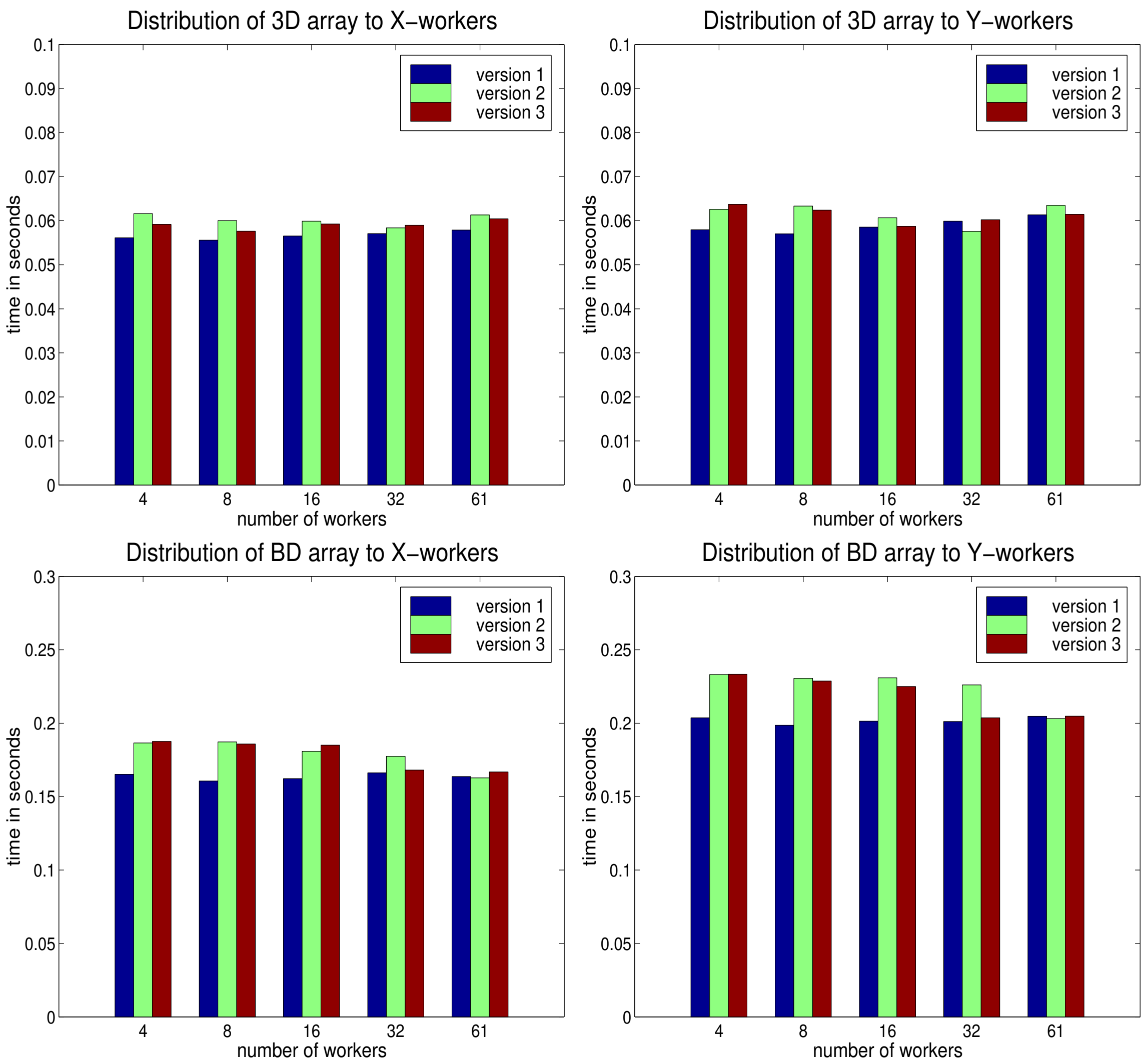

Figure 5.13: Global times of 3D and BD XY-distribution

As for HV-partitioning we have to view almost the same procedures. Results can be found in figures 5.15 on page 48 and 5.16 on page 49. There are no H-distribution routines for 2D arrays or 2DN-arrays, because they are not necessary for the horizontal transport performed at that stage.

Important interpretations of the diagrams are the overhead of V-column-distribution of the smaller arrays in version 1 since each column is send in a single message sometimes only consisting of a few real numbers. Therefore the start-up time for the messages to be sent is disproportionate to the message transfer time.

The measurements conducted for the communication routines have been used in chapter 6 in order to predict the communication overhead for the parallelization of STEM-III. The diagrams are presented to enable the reader to reconstruct our predictions.

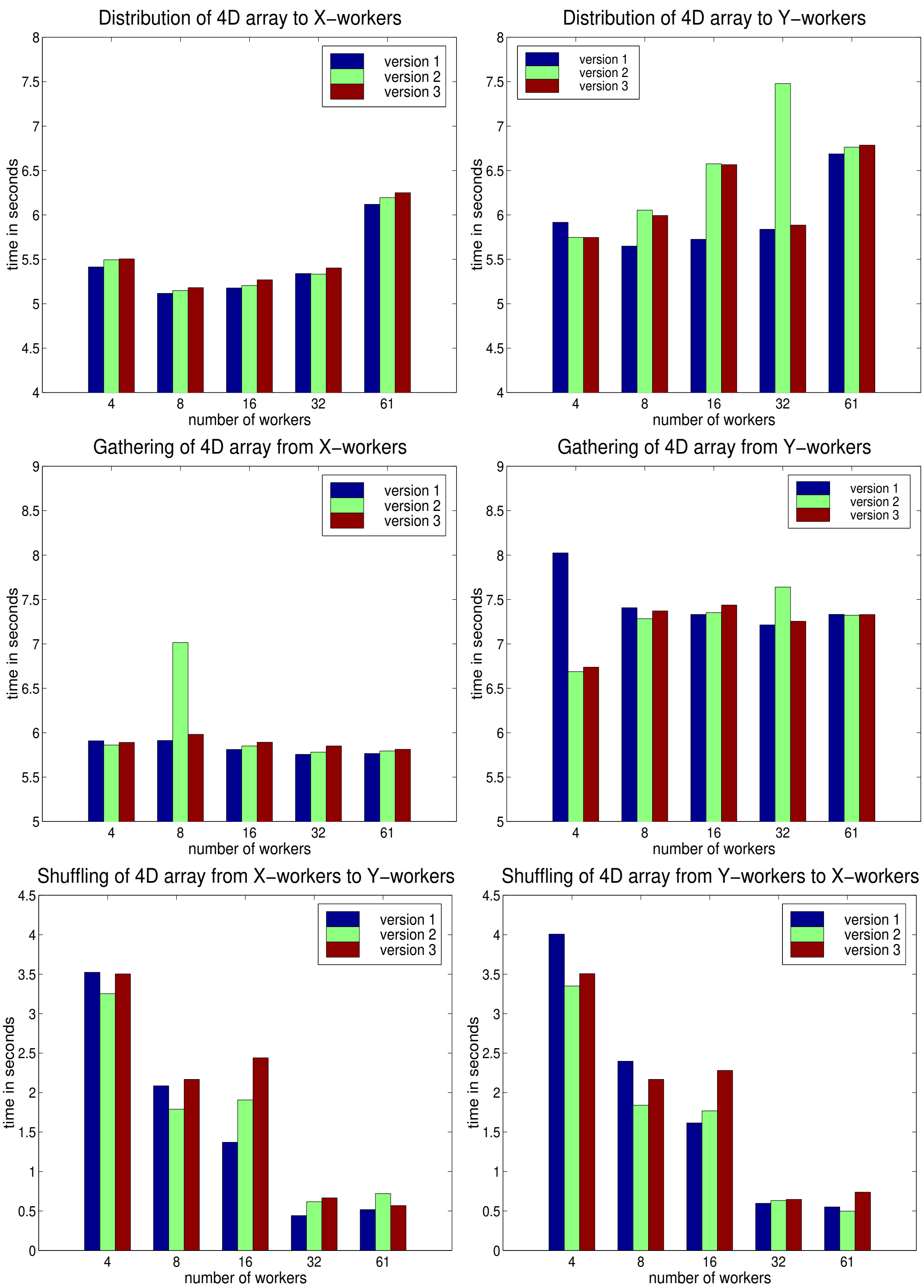

Figure 5.14: Global times of 4D XY-distribution and gathering and local times of 4D XY-shuffling on processor 1

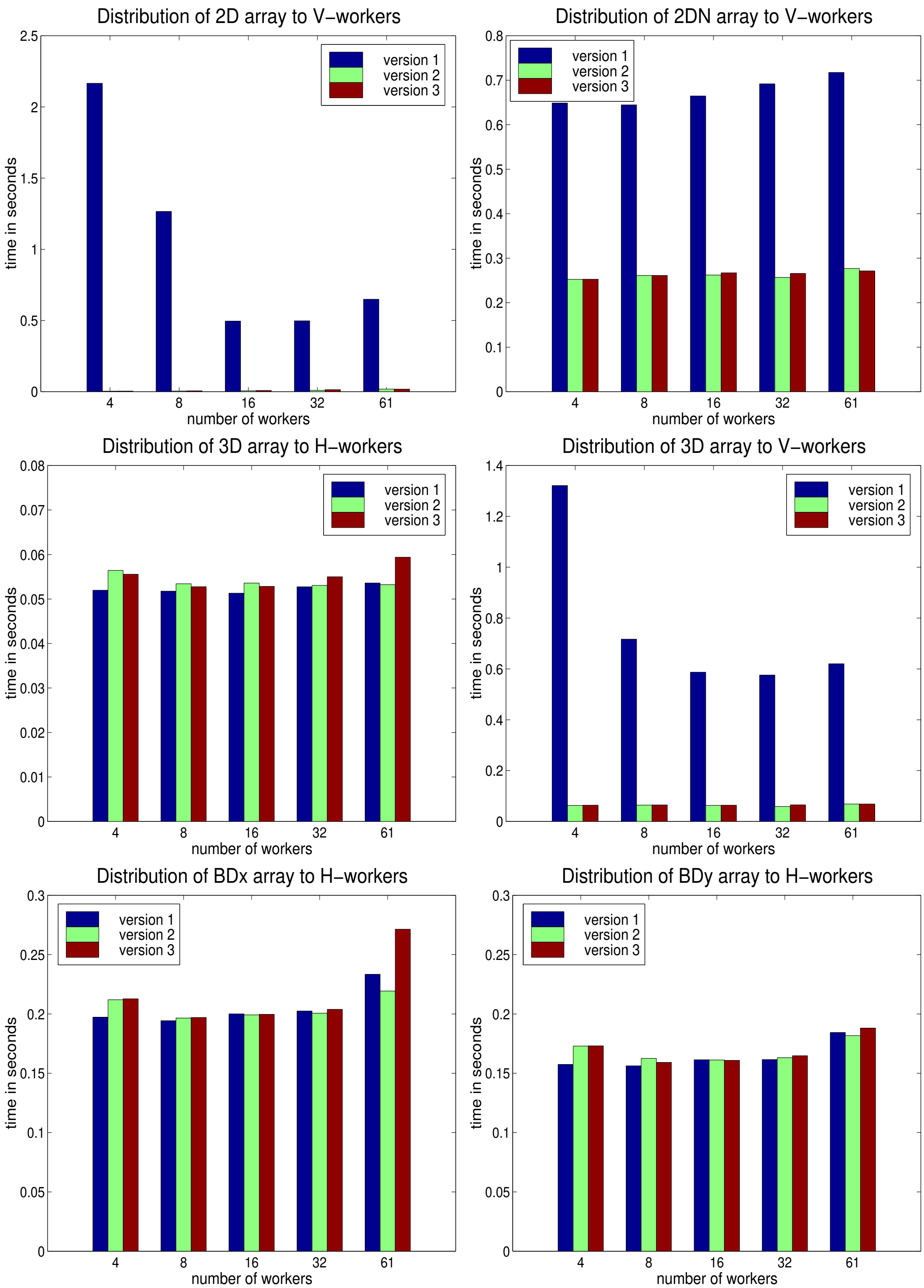

Figure 5.15: Global times of 2D, 2DN, 3D and BD HV-distribution on the Beowulf cluster

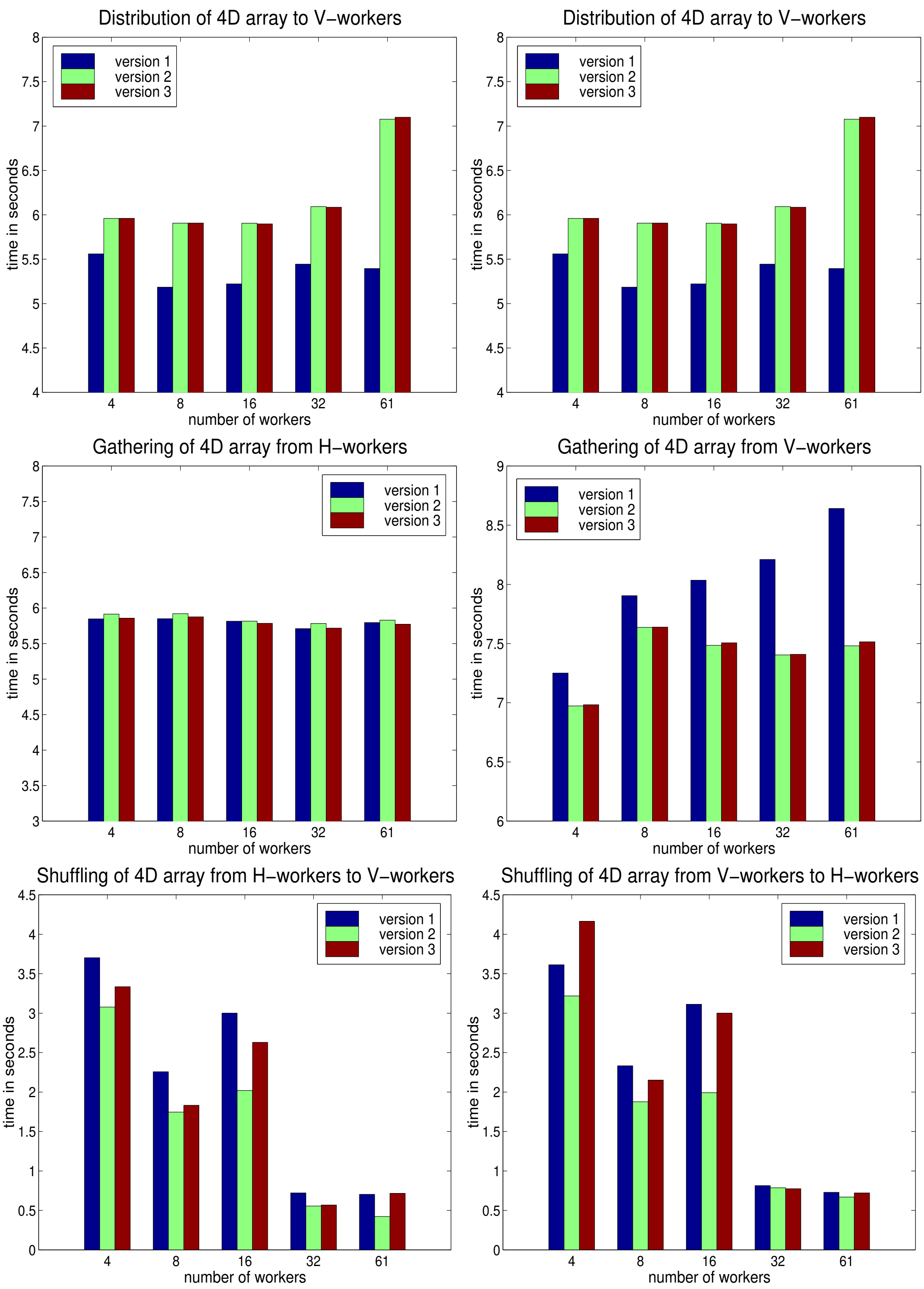

Figure 5.16: Global times of 4D HV-distribution, 4D HV-gathering and local times of 4D HV-shuffling on the Beowulf cluster

5.6 Library Behavior on the Sun Enterprise

Testing the communication on the Sun Enterprise was very exciting because the communication times are much smaller. This fact and the speed of the Sun F90 compiler at the Sun encouraged the decision to make it the development architecture for our communication library. The reason for the fast communication is the shared memory on the sun, the arrays do not need to be transmitted through a network but are copied from one memory position to another.

In figures 5.17 on the next page and 5.18 on page 52 the communication times for XY-partitioning on the Sun are displayed. These results serve for comparison purposes of timings on Sun and Beowulf. We skipped the HV-partitioning timings because the results are close to the XY-times. The latency on the Sun Enterprise is much lower than on the cluster. It still affects the communication using small messages as can be seen in the 2D distribution; sending many messages using version 1 is inferior to sending few compact messages (version 2 and 3). But already during 2DN distribution the latency plays an inferior role. The MPI-copying and buffering in order to send single large messages using version 2 and 3 takes more time than just sending many small messages using version 1. We can repeat this explanation for 3D distribution, as the size of the single large messages decreases (8 or 11 workers) the distribution time levels out with the version 1 distribution.

On 4D arrays the latency versus bandwidth effects can be seen again. The computational time for MPI-caching and buffering pays off during X-distribution, because the full bandwidth can be used for the long messages. In Y-distribution this is not the case, the layout of Y-slices in the global structures is too expensive and results in high transfer time using 11 workers. The difference of distribution into X or Y-partitioning can already be seen when looking at BD distribution, where the vertical scales display different numbers. For smaller arrays no time difference can be found while comparing partitionings.

Sun MPI implemented a few MPI-2 functions not available in the MPICH implementation on the cluster. In our library they have been used for creating the communication data types for version 2 and 3. We wanted to know if it makes a difference to use these MPI-extensions. After comparing the timings we have to conclude that the MPI-2 implementations do not introduce higher efficiency or speed on the Sun, only the programmer benefits from the more general functionality. However, this can be different on other architectures or MPI-implementations, therefore both function calls are provided in separate files.

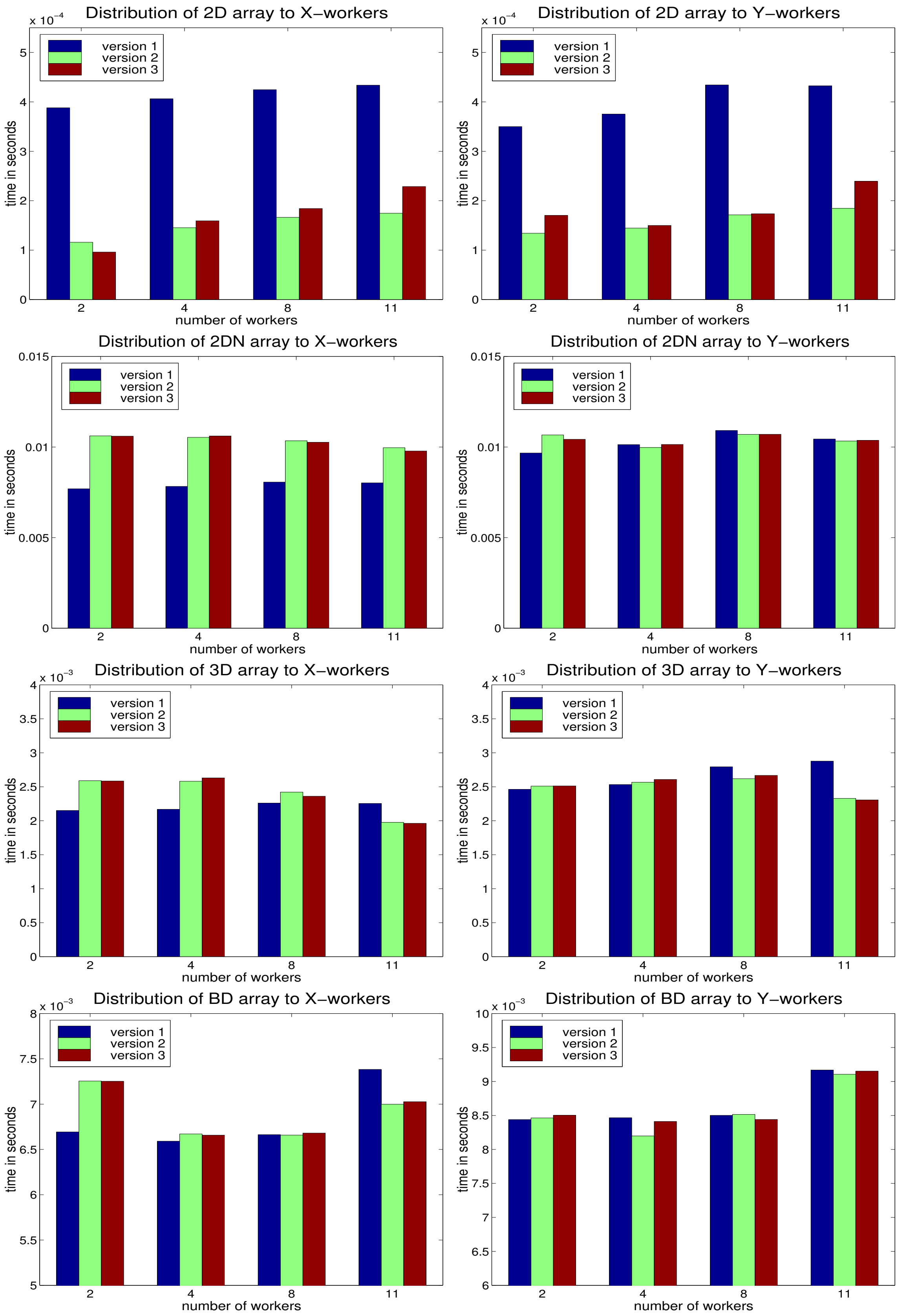

Figure 5.17: Global times of 2D, 2DN, 3D and BD XY-distribution on the Sun Enterprise

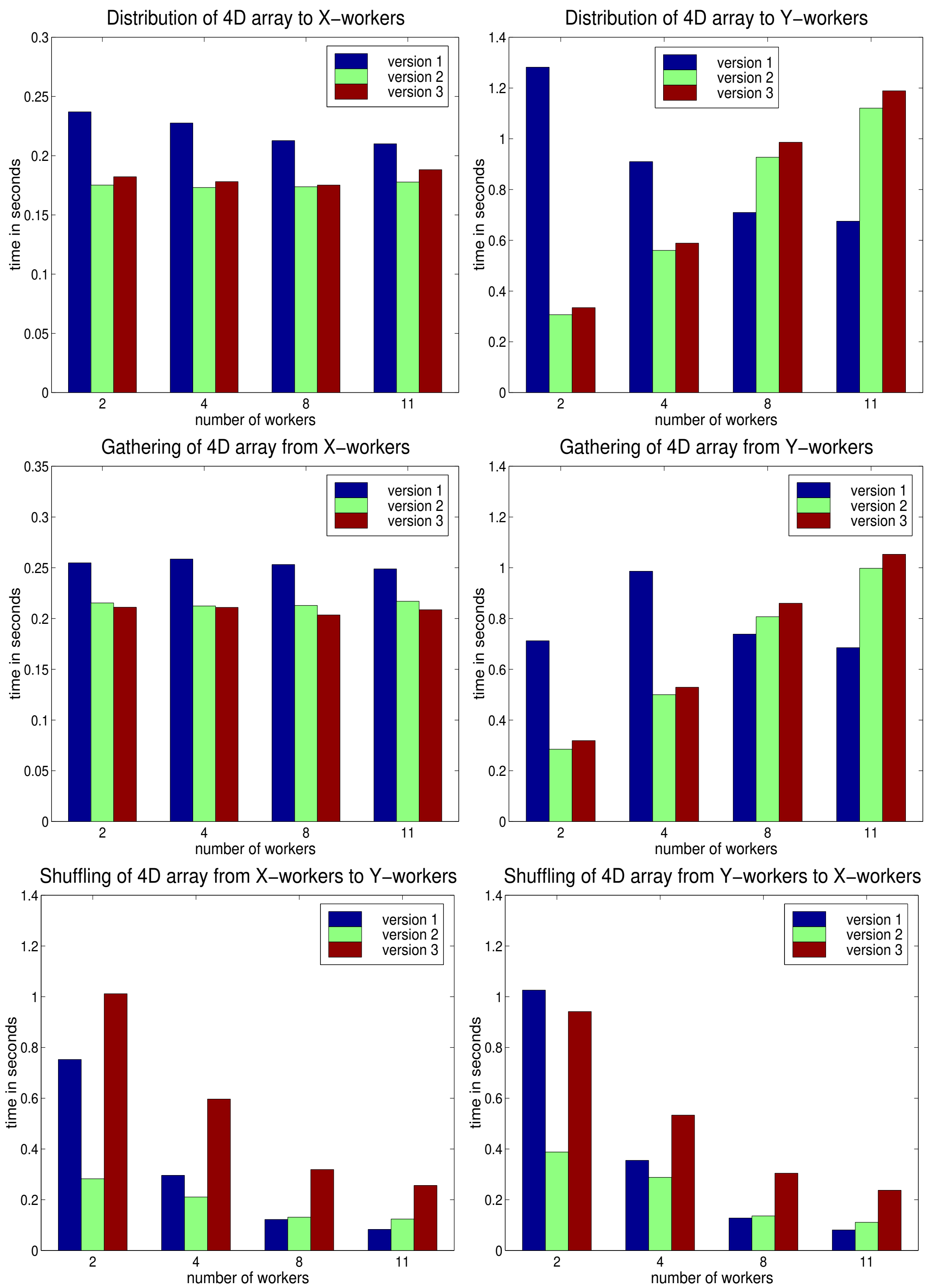

Figure 5.18: Global times of 4D XY-distribution, 4D XY-gathering and local times of 4D XY-shuffling on the Sun Enterprise

Chapter 6

Results

This chapter presents time measurements of the parallel version of STEM-III, based on the communication library presented in the last chapter. Experiments were conducted using a test case representing the simulation of an one hour pollution episode from May, 1998.

6.1 Predicted and Measured Run Times

The run time of the parallel code is affected by many different factors. The following formula can be used to predict parallel run times after determination of the included parameters:

$$t_{par} = \frac{t_{ser} - t_{io}}{N} + t_{io} + t_{comm}(N) \tag{6.1}$$

t_{ser}	- serial time
t_{par}	- parallel time
t_{io}	- serial time spent for input/output that is not parallelizable
t_{comm}	- communication overhead added in parallelization
N	- number of workers

We can split up the communication overhead into three parts, the distribution time, the shuffle time and the gather time, all depending on the number of workers used as well as on the architecture.

$$t_{comm}(N) = t_{distrib}(N) + t_{shuffle}(N) + t_{gather}(N) \tag{6.2}$$

The communication overhead can be divided into two categories:

- The distribution of the input data from the master to the workers and the gathering of the output data at the master and
- the communication between workers for reshuffling of data.

Both categories have a complex dependency on the number of workers. For distribution and gathering data, the master has to send or receive a number of messages proportional to the number of workers. The message length decreases as the number of workers increases. Gathering and distribution have to take place in an ordered fashion at the master, no overlapping is possible. Shuffling between the workers on the other hand has no single point of connection. Each processor needs to send messages to all the other workers, while the message length decreases as the number of workers increases. The amount of data sent in a shuffling procedure is the same as in the distribution of a 4D array, but split into messages of much smaller size in between the workers. In this case overlapping is possible as communication between independent worker pairs can occur at the same time up to the limitations of the switch connecting all nodes on the Beowulf cluster.

Note that there are some parallelization times that have not been included into the time measurements. First is the setup time for MPI that varies with the operating system used as well as the number of processors. We also did not include the times for setting up the mapping and allocation of local arrays, that is further overhead added to the parallelization time, but partially overlaps with the reading of the input file at the master.

Serial time	4480 sec
Input/Output time	26 sec

Table 6.1: Serial code constants for one hour of simulation time

Measurements have given us the environment dependent constant results for our specialized input file as displayed in table 6.1. These parameters do not depend on domain decomposition. The input file used for testing the parallel version as well as conducting the time measurements is a striped down and simplified version of the normally used input files. It includes data for one hour of simulation time. Boundary data and emission arrays are set to zero for simplification purposes. As can be seen in table 6.1 serial computation lasts for about 1 hour and 15 minutes, representing a simulation time to real time ratio of less than 1:1. The serial time was measured on the master node of the Beowulf cluster to be compared to the parallel times.

In case of our test simulations the time used for input/output accounts for less than 0.6 percent of the serial time, an exceptional low amount compared to up to 3 percent as illustrated in figure 3.2 on page 15. As the input is only read once in the beginning of the simulation that time cannot be overlapped with other computation.

In the following we describe the domain decomposition dependent factors influencing the time measurements.

6.1.1 XY-partitioning

As has been mentioned, the communication time depends on the number of workers. It can be split up into the three different categories: distribution, gathering and shuffling. The worker-dependency explains the ranges given in table 6.2 on the next page, where the first number was measured using

Communication	Array	X-slice	Y-slice
Distribution	2D	0.005 – 0.01	0.005 – 0.01
	2DN	0.2 – 0.25	0.2 – 0.25
	3D	0.06	0.06
	BD	0.15	0.2
	4D	5 – 6	5.5 – 7
Gathering	4D	6	6
Shuffling	4D	3.5 – 0.5	3.5 – 0.5

Table 6.2: Time range estimates (in seconds) for minimal and maximal number of workers using XY-partitioning

the smallest and the second number using the largest possible number of workers (here 61).

Distribution as well as gathering take place once during a simulation. In the beginning we have to distribute the following number of arrays into X-partitioning: one 4D, two 3D and one BD array. Into Y-partitioning we have to distribute: one 4D, ten 3D, four 2DN, four 2D and one BD array. Therefore we can calculate the minimal and maximal distribution time using table 6.2:

$$
\begin{aligned}
t_{distrib}^{min} &= tx_{4D}^{min} + 2tx_{3D}^{min} + tx_{BD}^{min} \ + \ ty_{4D}^{min} + 10ty_{3D}^{min} + 4ty_{2DN}^{min} + 4ty_{2D}^{min} + ty_{BD}^{min} && (6.3)\\
&= \underbrace{5 + 2 \cdot 0.06 + 0.15}_{\text{into X-slices}} \ + \ \underbrace{5.5 + 10 \cdot 0.06 + 4 \cdot 0.2 + 4 \cdot 0.005 + 0.2}_{\text{into Y-slices}} && (6.4)\\
&= 12.39 \text{ sec} && (6.5)\\
t_{distrib}^{max} &= tx_{4D}^{max} + 2tx_{3D}^{max} + tx_{BD}^{max} \ + \ ty_{4D}^{max} + 10ty_{3D}^{max} + 4ty_{2DN}^{max} + 4ty_{2D}^{max} + ty_{BD}^{min} && (6.6)\\
&= \underbrace{6 + 2 \cdot 0.06 + 0.15}_{\text{into X-slices}} \ + \ \underbrace{7 + 10 \cdot 0.06 + 4 \cdot 0.25 + 4 \cdot 0.01 + 0.2}_{\text{into Y-slices}} && (6.7)\\
&= 15.11 \text{ sec} && (6.8)
\end{aligned}
$$

Gathering is done in the end from X-partitioning on the 4D arrays only. Shuffling has to be done three times from X to Y and back. All approximations used for our predictions can be found in table 6.3.

Communication	Time
Distribution	12 – 15 sec
Gathering	6 sec
Shuffling	21 - 3 sec

Table 6.3: Communication overhead for parallel STEM-III using XY-partitioning

Distribution ranges from 12 to 15 seconds and shuffling from 21 to 3 seconds when using 2 to 61 workers. As can be summarized the communication time for our specific layout ranges from 39 to 24 seconds on the bandwidth of 2 to 61 workers.

After establishing all the approximations of constants and parameters for our predictions we can now display the resulting graphs. Figure 6.2 on page 57 presents the predictions for XY domain decomposition. The blue line represents ideal speedup where every part of the serial code

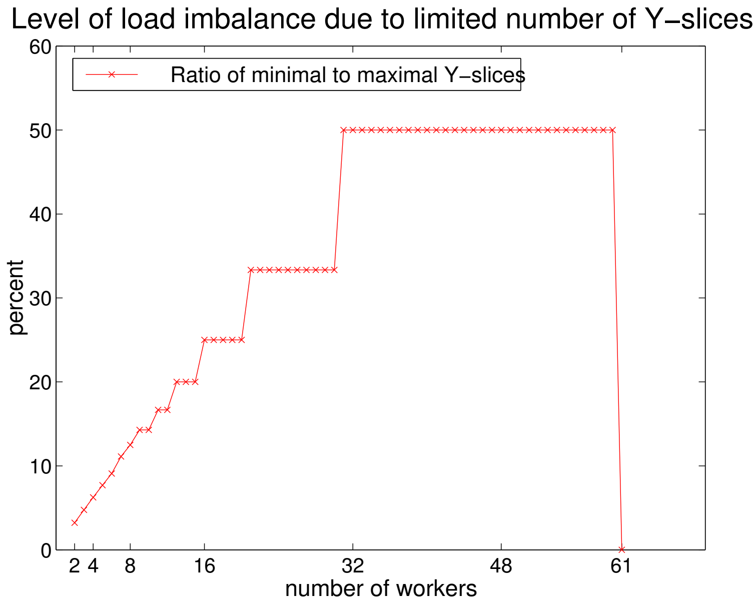

Figure 6.1: Level of load imbalance for XY-partitioning due to uneven distribution of Y-slices

is parallelizeable and no overhead is introduced. The green line is associated with our calculated prediction using the measurements from table 6.1 and equation 6.1. However, we know that due to the coarse partitioning the overall time will depend on the slowest worker, the worker that has the largest number of slices. Splitting up 61 Y-slices will most of the time lead to uneven distribution as presented in figure 6.1. Using these results we were able to predict the cascaded red graph in figure 6.2.

Note that XY-partitioning does not need to employ more than 61 workers because of the limiting grid size in x-dimension. Any more workers used will stay idle during the computation. This can be seen in figure 6.2 in the predicted horizontal line from 61 to 64 workers.

When comparing the predictions with the measurements in figure 6.3 on the following page our previous calculations and approximations should be reconsidered. However, as can be seen in figure 6.7 on page 62, our distribution predictions are below the real distribution times because measurements were taken under ideal situations. Same is true for shuffling and gathering times. In addition load imbalances have been introduced as showed in section 6.3 on page 63, therefore the parallelization measurements should be well below our prediction line. However, as said before, cache effects play an important role during parallelization, the application of modules on local data introduces super linear speedup and explains our measured results.

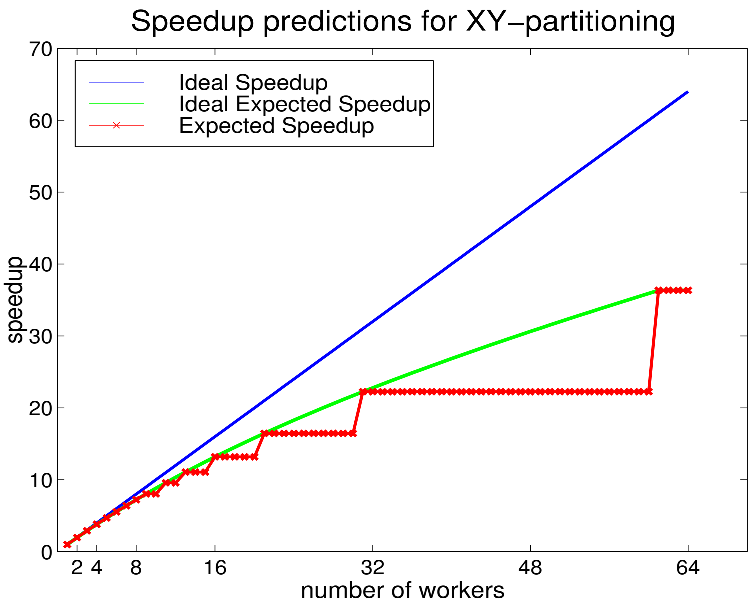

Figure 6.2: Ideal and predicted speedup for XY-partitioning

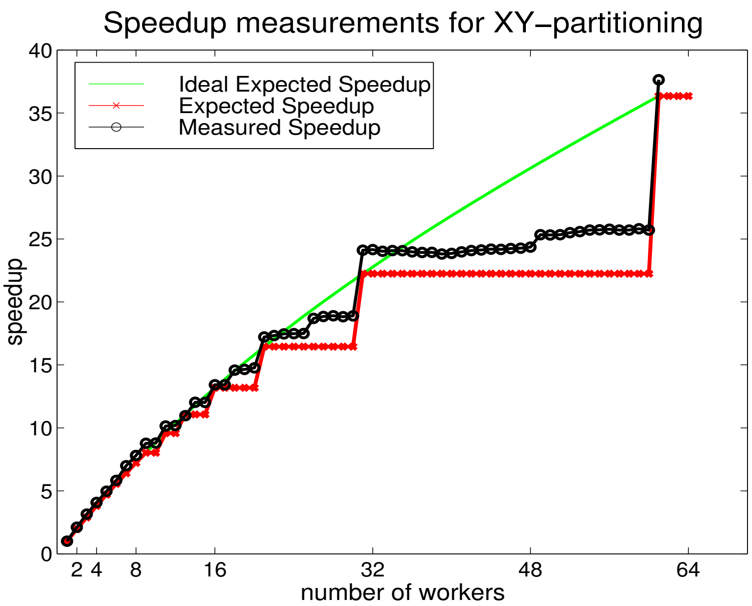

Figure 6.3: Predictions versus real measurements on XY-partitioning

Communication	Array	H-slice	V-column
Distribution	2D		0.005 – 0.02
	2DN		0.25
	3D	0.06	0.06
	BD	0.2	
	4D	6 – 7	6 – 7
Gathering	4D	6	7 – 7.5
Shuffling	4D	3 – 0.5	3.5 – 0.7

Table 6.4: Time range estimates (in seconds) for different number of workers using HV-partitioning

6.1.2 HV-partitioning

In difference to the XY-partitioning the HV-partition can employ up to 2989 workers. That is the number of existing V-columns in our given grid dimensions. Since the Beowulf has only 64 nodes we will limit the prediction graph to 64 workers.

Again the distribution as well as the gathering occur once during a simulation. In the beginning the following number of arrays have to be distributed into H-partitioning: one 4D, three 3D and two BD arrays. As into V-partitioning there has to be distributed: one 4D, eight 3D, four 2DN and four 2D arrays. Therefore we can calculate the minimal and maximal distribution time using table 6.4:

$$
\begin{aligned}
t_{distrib}^{min} &= th_{4D}^{min} + 3th_{3D}^{min} + 2th_{BD}^{min} \;+\; tv_{4D}^{min} + 8tv_{3D}^{min} + 4tv_{2DN}^{min} + 4tv_{2D}^{min} && (6.9)\\
&= \underbrace{6 + 3 \cdot 0.06 + 2 \cdot 0.2}_{\text{into H-slices}} \;+\; \underbrace{6 + 8 \cdot 0.06 + 4 \cdot 0.25 + 4 \cdot 0.005}_{\text{into V-columns}} && (6.10)\\
&= 14.08 \text{ sec} && (6.11)\\
t_{distrib}^{max} &= th_{4D}^{max} + 3th_{3D}^{max} + 2th_{BD}^{max} \;+\; tv_{4D}^{max} + 8tv_{3D}^{max} + 4tv_{2DN}^{max} + 4tv_{2D}^{max} && (6.12)\\
&= \underbrace{7 + 3 \cdot 0.06 + 2 \cdot 0.2}_{\text{into H-slices}} \;+\; \underbrace{7 + 8 \cdot 0.06 + 4 \cdot 0.25 + 4 \cdot 0.02}_{\text{into V-columns}} && (6.13)\\
&= 16.14 \text{ sec} && (6.14)
\end{aligned}
$$

Gathering is done in the end from H-partitioning on the 4D arrays only. Shuffling has to be done three times from H to V and back. All approximations used for our predictions can be found in table 6.5 on the next page.

Communication time for the HV-mapping therefore takes 39.5 to 25.6 seconds on the bandwidth of 2 to 64 workers. We have implemented two different HV-mappings in order to compare load balancing capabilities but provide one prediction curve for both of them. As can be seen in figure 6.5 on page 60 in both simulations the speedup outperforms the predictions. Again this is due to super linear speedup as result of cache effects. The red line in figure 6.5 presents the irregular HV2-mapping outperforming the regular HV-mapping in some cases. The improvement is even more

Communication	Time
Distribution	14 – 16 sec
Gathering	6 sec
Shuffling	19.5 - 3.6 sec

Table 6.5: Communication times for parallel STEM-III using HV-partitioning

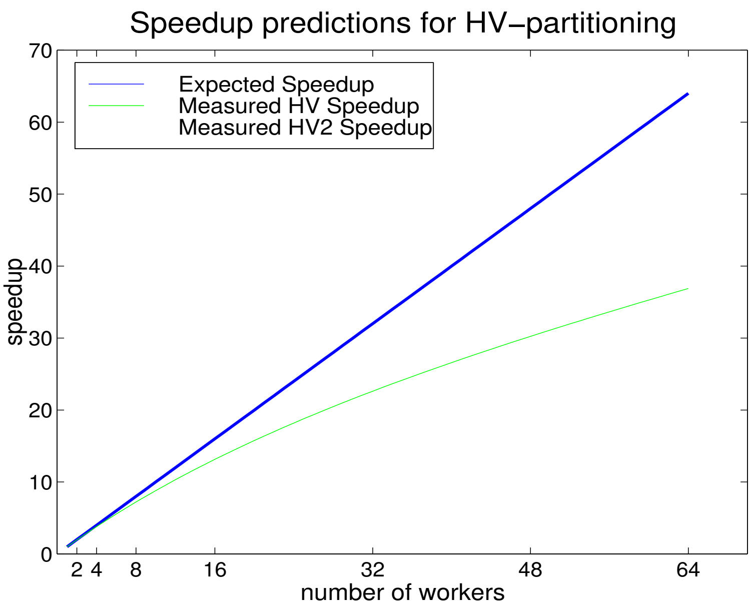

Figure 6.4: Ideal and predicted speedup for HV-partitioning

impressive when inspecting the distribution timings in figure 6.6 on the next page. Here we can find a major drawback of the irregular HV2-mapping. The distribution time is much higher because the irregular mapping cannot use the cache as effectively during distribution and MPI-level copying and buffering of the array parts. Therefore the outperformance has to result from better overall load balance on the irregular mapping. However, the general behavior of both schemes is very similar and we favor the regular mapping because it permits the application of the different versions providing better architecture independent performance.

6.2 Comparison of XY and HV Domain Decomposition

HV-partitioning outperforms XY-mapping due to the finer granularity used. Looking at figure 6.8 on page 62 there is one single point where XY-partitioning outmatches HV-partition. Using 62 processors XY domain decomposition reached the limit of applicable processors. The 61 workers achieve the higher performance due to limitations of the HV-overhead. Distribution, as seen in

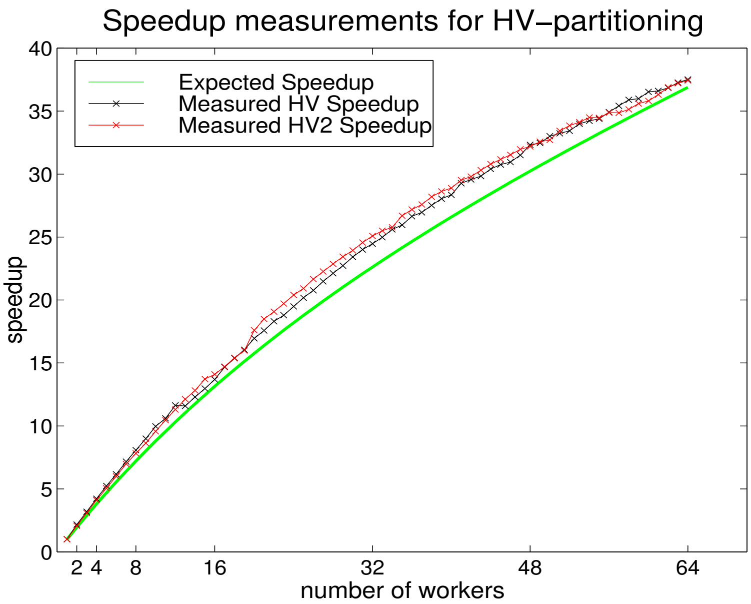

Figure 6.5: Predictions versus real measurements on HV-partitioning

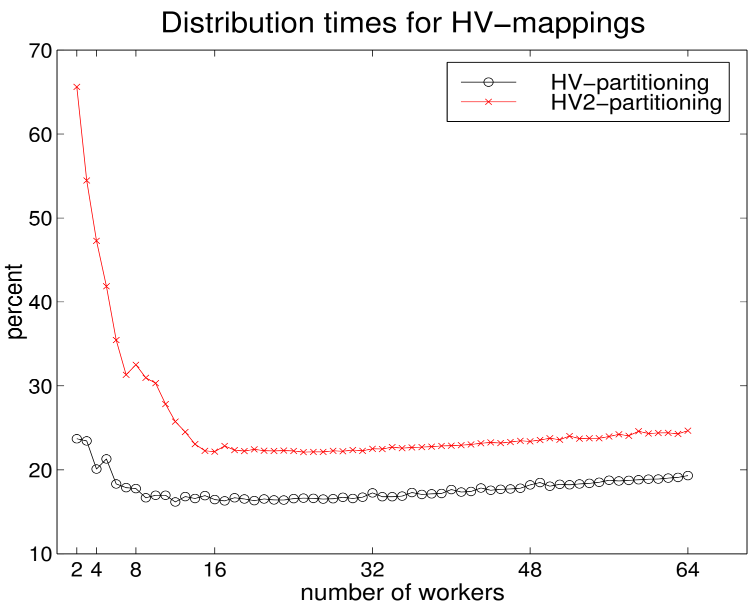

Figure 6.6: Distribution times for HV-mappings

Workers	XY-partitioning	HV-partitioning	HV2-partitioning
2	2120.55	2070.67	2166.90
4	1097.47	1061.96	1091.54
8	573.77	554.64	573.07
16	333.72	327.40	317.82
32	185.50	183.11	178.63
61	119.00	122.41	123.38
64		119.44	119.69
128		97.27	97.52

Table 6.7: Simulation times (in seconds) of the parallelization of STEM-III on the Beowulf cluster

figure 6.7 on the next page, takes more time for the HV-partitioning. As HV-shuffling and gathering are superior to XY-communication, the global overhead pushes HV-partitioning into the second place. Further, during HV-partitioning the transport in X and in Y-direction takes place on only 20 workers as we have a grid limitation on H-partitioning. When using XY-partitioning there are 49 workers available for transport in x-direction and 61 processors for transport in y-direction. This explains the better performance of XY-partitioning when using 61 workers. However, the ability to apply more processors to the calculation presents an advantage of HV over XY-partitioning.

As can be seen in figure 6.9 on page 63 we kept the efficiency of the parallelization above 50 percent up to 64 workers for the HV-schemes. Only the XY-scheme efficiency drops down due to the load imbalance that can already be predicted using figure 6.1 and therefore be avoided by applying the next efficient number of processors (31 or 61).

Communication	Array	H	V
Distribution	2D		0.03
	2DN		0.25
	3D	0.06	0.07
	BD	0.3	
	4D	8.5	9.5
Gathering	4D	5.8	7.8
Shuffling	4D	0.4	1

Table 6.6: Measured communication times using HV-partitioning on 129 processors (128 workers)

In table 6.7 we present selected timings for our domain decomposition schemes. We have been able to include measurements with 128 workers since the Beowulf cluster provides 64 double processor nodes. The shuffling overhead will increase and should be about twice as high as using 64 processors because on each node the shuffling communication has to be serialized for the two present workers. Distribution and gathering already exists in serialized form due to the master-worker approach. Detailed measurements can be found in table 6.6. The predicted time for 128 workers is about 82 seconds. As we can see the prediction is not reached while efficiency drops down to 36 percent.

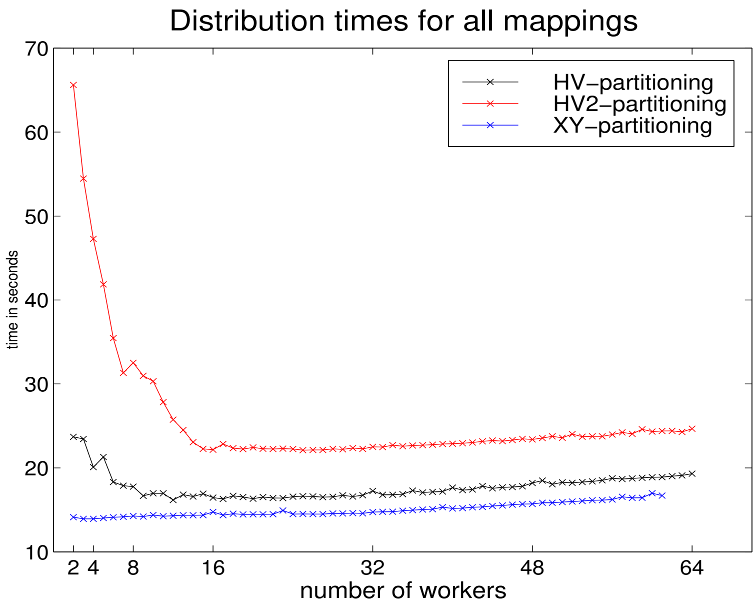

Figure 6.7: Distribution times for XY and HV-mappings

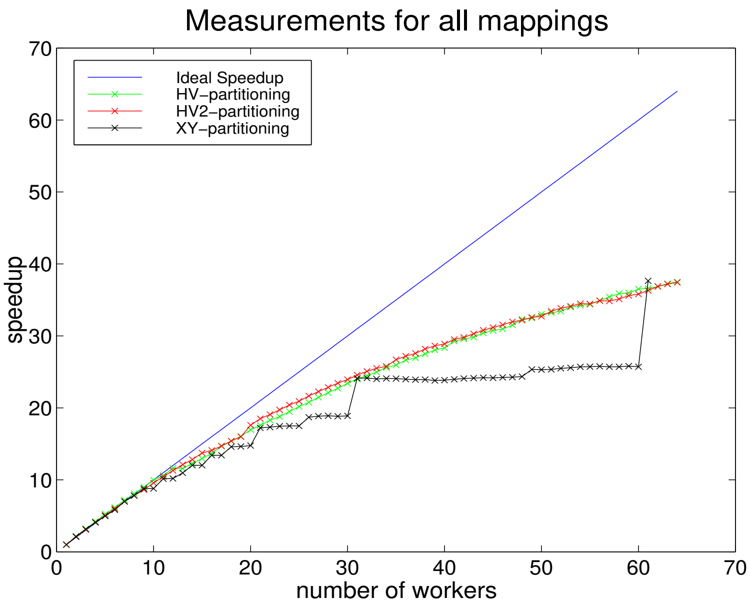

Figure 6.8: Speedup comparison of XY and HV domain decompositions

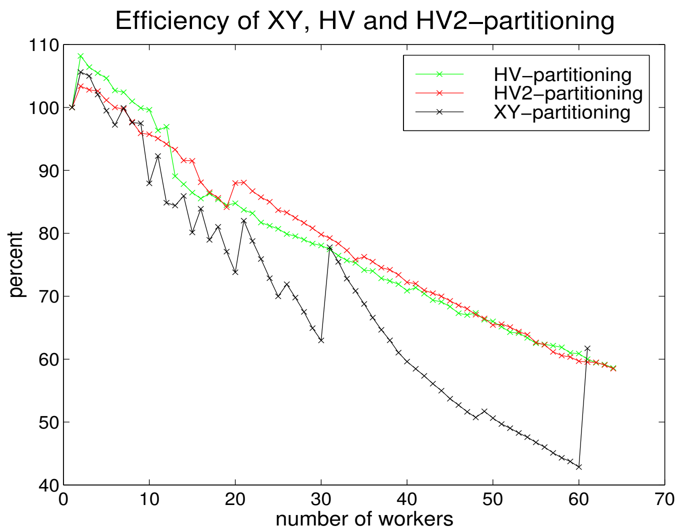

Figure 6.9: Efficiency comparison of XY and HV domain decompositions

6.3 Load Imbalance

Load imbalance has a major impact on the timings presented in the previous sections. The amount of load imbalance is presented in the figures 6.10 on the next page and 6.11 on page 65 using upshot. In figure 6.10 you can see the STEM-III simulation applied to 21 processors. The more detailed zoom on the lower half of the page displays the synchronizing procedure at each time step when reshuffling has to take place. To finish a reshuffling procedure each process involved has to receive at least one message from every other process. In this case, on XY-partitioning, all processes wait for worker 1 to finish since it has one more slice than the others. By analyzing the waiting time for workers 2 to 20 it can be discovered that there is a load imbalance of up to 3 seconds in between workers with the same number of Y-slices.

Surveying figure 6.11 we discovered that the load imbalance on HV-partitioning before the synchronizing reshuffling process is about 1.8 seconds. Therefore HV-partitioning cuts down on the load imbalance by a lot. When reviewing HV2-partitioning using upshot, the same picture as 6.11 appears with a slightly different time scale. However, when looking at the timings in the previous section we know that load balancing is more efficient in HV2 for selected numbers of processors, but as we use more processors the HV load balancing improves compared to the HV2 scheme.

Using these results we can declare the superiority of HV-partitioning over XY-partitioning. Clearly there still exists load imbalance but our domain decomposition strategy minimized it to less than 2.5 percent of the computation time in the case of application of 21 processors. Therefore the static load balancing scheme affected by the mapping strategy shows close to optimal results. Comparing these results to a dynamic load balancing scheme is left for future work.

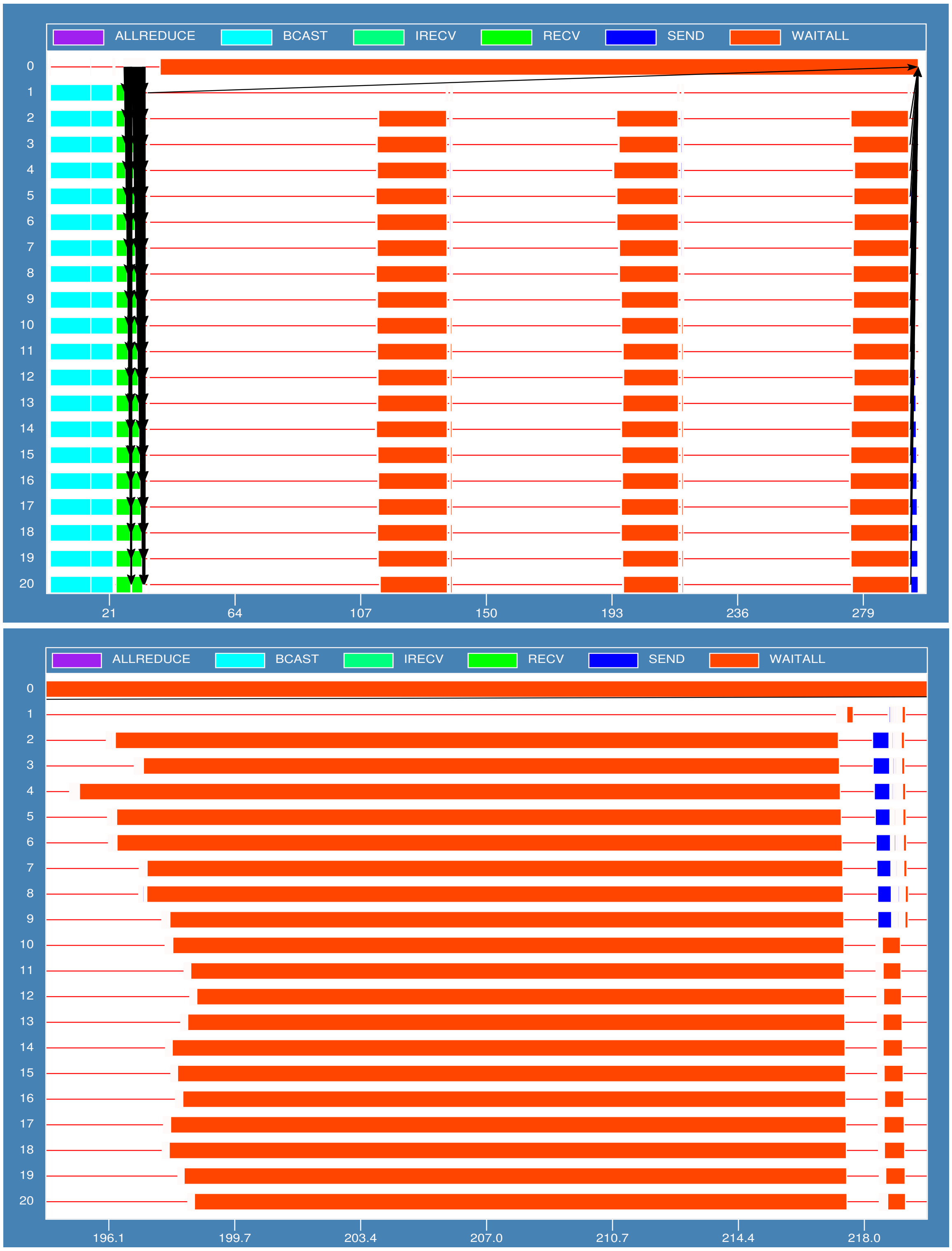

Figure 6.10: Load imbalance on XY-partitioning, full view and close up

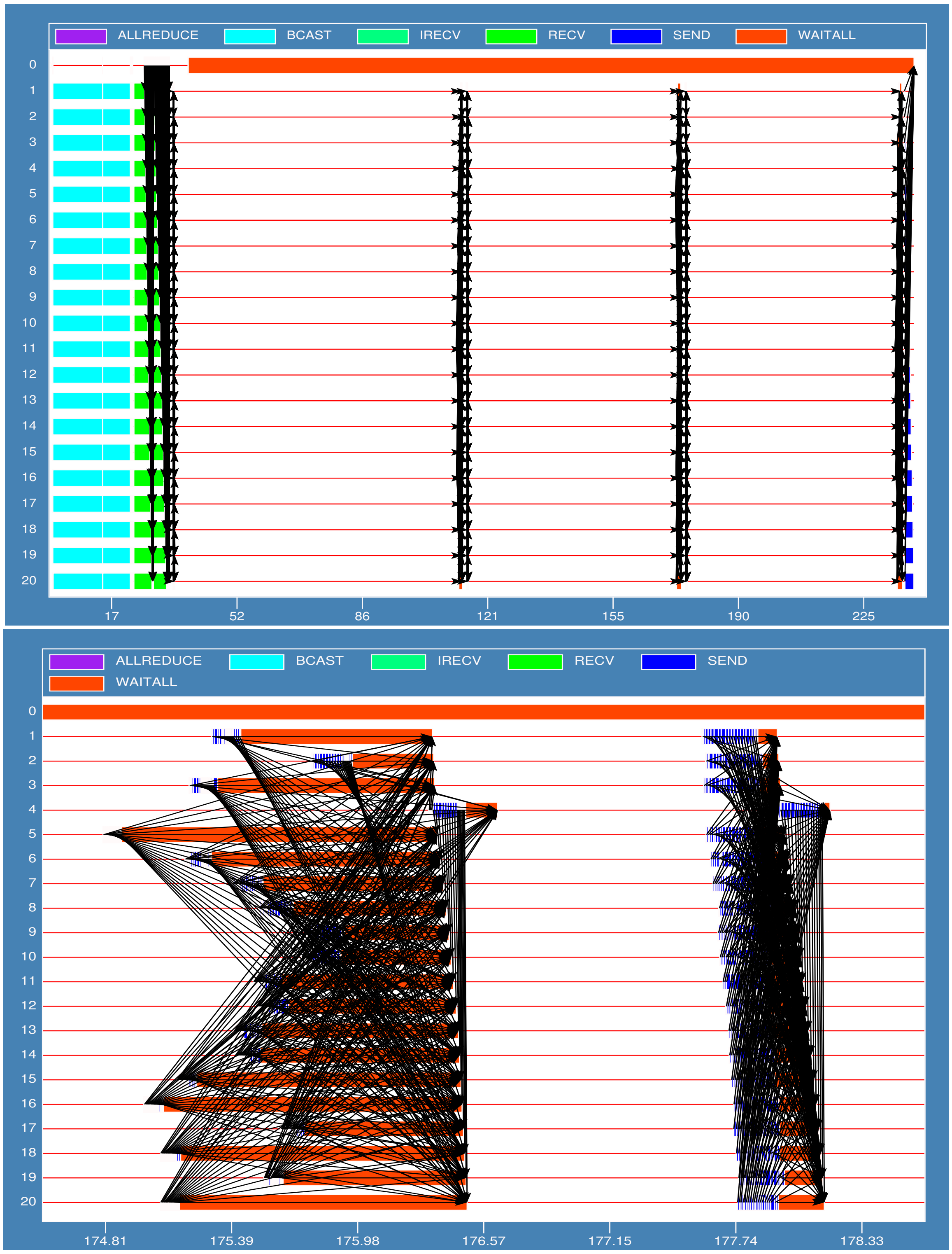

Figure 6.11: Load imbalance on HV-partitioning, full view and close up

6.4 Quality Control

In order to determine differences in the results thorough comparisons were performed.

Quality control of the communication library is the first and most important step. The setup program for the communication library includes an error control for every routine implemented. For each communication the data at the master are compared with the local data at each processor to make sure the transmission has taken place without loss of precision. Running the setup program on the different environments and with different grid sizes and numbers of processors involved made sure that the communication routines do not introduce any errors during communication.

After asserting the consistency of all the communication routines the next level of fault diagnostics and quality control is performed.

Comparing the results of parallel STEM-III with the serial version maximal relative errors in magnitude of 10^{-4} and average errors in magnitude of 10^{-7} were detected. Minimally we get four accurate digits and on average we obtain 7 accurate digits. When investigating the code in depth the Z-transportation was found to be the source of error. At a certain point in the time stepping algorithm (usually not during the first time step) the Z-transport delivers wrong concentration results for the first chemical species (NO) for XY-partitioning. The erroneous (compared to the serial version) concentrations result from an inner loop during Z-transport that is assigned a different number of iterations during computation in the parallel code than in the serial simulation. This is true for 1 to 3 different concentrations on the top boundary of the grid. However these erroneous concentrations are spread out over a larger area of the grid due to the following chemical reaction simulations and transportation in other directions.

In order to solve this problem in the parallel code more time has to be spent on investigations of the serial source code. At this time we cannot trace the error, however this error will be investigated in cooperation with implementers of the serial code. The timing results will not be affected by the differences in calculations, the communication library is not believed to be causing the errors, as result the problem has to be located in the serial code that appears to be due to grid size differences when working on the local data of a worker compared to the full grid arrays at only one processor during the serial simulation.

Chapter 7

Conclusions and Future Work

After presenting results in the last chapter, here the findings of this work are summarized and directions for future work are presented.

7.1 Conclusions

The presented communication library provides routines to parallelize air quality models using structured grids. It implements a master-worker strategy and XY and HV domain decompositions. The use of the MPI message-passing layer for communication creates the premise for portability. The library is an architecture independent implementation providing fast and effective functions for a wide range of architectures. This work compared the use of different environments, the Sun Enterprise 4500 using Sun MPI and the Beowulf cluster at MTU using MPICH. The domain decompositions represent a static load balancing approach that performs best when used on uniform parallel machines because the problem is divided into close to equal load shares for each worker involved.

Application of the library to STEM-III provides an example parallelization for users. It shows the performance and usability of the library routines that represent the overhead computations introduced for the parallelization. The run time for the simulation was reduced from 1 hour and 15 minuted down to less than 2 minutes using 64 workers for both decomposition strategies. It is recommended to use the HV-partitioning in order to lower the level of load imbalance for any number of workers applied and to decrease the influence of grid size on the maximal number of workers applicable.

7.2 Future Work

In our research inherent static load balancing approaches were investigated, thereby focusing on selected mappings and appropriate distribution of the data. Beyond that there is the possibility to introduce dynamic load balancing strategies. Although the scope of our work did not allow further investigation it nevertheless would be interesting to compare the overhead introduced by dynamic load balancing with the gained effectivity of the parallelization. We believe that the HV-partitioning can compete with a dynamic load balancing approach on uniform parallel machines and may even outperform it in the cost factor of invested work. Dynamic load balancing has the advantage to efficiently make the parallelization applicable to non-uniform networks of workstations. This would add to the portability of the communication library.

The Argonne National Laboratory has developed a *Runtime System Library (RSL)* [14, 13] mainly for the parallelization of regular-grid atmospheric models with nesting. It could also be used to parallelize AQMs by using an explicit transportation scheme. In our case the nesting would not be used since we focus on uniform structured grids. Application of the RSL to AQMs would simplify the load balance investigation since RSL contains load balancing functions that can be applied to the simulation. Utilizing RSL has the disadvantage of adding another layer to the already multi-layered application, however effectivity of dynamic load balancing can be researched and the overall gains or losses may be verified without large effort. Further investigations could include implementing library routines for block matrix distribution, gathering and reshuffling, where only explicit transportation schemes can be applied, as an addition to our communication library.

In order to reduce the communication overhead input and output routines can be parallelized, implementing new input and output modules closely related to the serial versions. Advantages of this idea are

- further parallelization of serial parts, introducing higher efficiency and
- no distribution and gathering is necessary, thereby decreasing communication overhead.

However, there are two major disadvantages of this approach. First there is no general parallelization for every model. The input and output modules of the AQM have to be examined and therefore the solution will be model dependent. Second not every architecture can be used. Due to the large size of the input and output files storage only available to the master node on our Beowulf cluster had to be used. Still, the impact of this approach on the speedup and efficiency can be huge and is worth investigating.

The library has to be tested on different architectures using various compilers and MPI-implementations to show its platform independence. It needs to be updated and maintained in the future to provide usefulness. Our results witness the advantages of applying the library to other air quality models in the future.

Acknowledgments

Special thanks go to A. Sandu, the advisor for this project, for his presence and support. He has been there during the most difficult times and pointed me to ways to solve the problems. I am thankful for the many helpful discussions and his patience in the iterative process of composing this document. J. Mayo, S. Seidel and D. Hand have provided helpful comments and suggestions as members of the project committee. I am very thankful for introduction into the parallel area by B. VanVoorst, who taught the Parallel Algorithms class at MTU in spring 2001, where we learned about parallel architectures, MPI and tools for parallel systems.

This research was supported by Michigan Technological University and NSF CAREER award ACI-0093139. The Center for Global and Regional Environmental Research at the University of Iowa provided the serial source code and test input files for the parallelization. We want to express our gratefulness to the CS department at MTU as well as the Center for Experimantal Computations (CEC) for access to the Beowulf cluster and Sun Enterprise. The author thanks J. Duff and D. Torrey for maintenance of the Beowulf cluster and Sun Enterprise, respectively. Further thanks go to all the people who answered my questions about MPI-irregularities during implementation of the library.

Appendix A

STEM-III Specific Functions

This part describes the STEM-III specific functions and their parameters. It can serve as an example on how to implement parallel versions of other AQMs using our library, because understanding of routines gives an insight on how the library functions can be used.

A.1 XY-Partitioning

MemAlloc

Allocates the memory for the arrays specified.

Syntax

```
MemAlloc(Nx,Ny,Nz,N_gas,N_liq,N_part,
         sg,sl,sp,u,v,w,kh,kv,t,dz,wc,wr,
         sprc,rvel,sx,sy,sz,q,em,vg,fz,hdz,h,tlon,tlat
```

```
  INTEGER Nx               !  global no. of grid points in x-dim
  INTEGER Ny               !  global no. of grid points in y-dim
  INTEGER Nz               !  global & local no. of grid points in z-dim
  INTEGER N_gas            !  global no. of gas species dim
  INTEGER N_liq            !  global no. of liquid species dim
  INTEGER N_part           !  global no. of particle species dim
  REAL sg(Nx,Ny,Nz,N_gas)  !  global 4D gas conc array
  REAL sl(Nx,Ny,Nz,N_liq)  !  global 4D liquid conc array
  REAL sp(Nx,Ny,Nz,N_part) !  global 4D particle conc array
  REAL u(Nx,Ny,Nz)         !  global 3D wind field component array in x-dir
  REAL v(Nx,Ny,Nz)         !  global 3D wind field component array in y-dir
  REAL w(Nx,Ny,Nz)         !  global 3D wind field component array in z-dir
  REAL kh(Nx,Ny,Nz)        !  global 3D hor. diffusitivity array
  REAL kv(Nx,Ny,Nz)        !  global 3D ver. diffusitivity array
  REAL t(Nx,Ny,Nz)         !  global 3D global time array
  REAL dz(Nx,Ny,Nz)        !  global 3D ver. resolution array
  REAL wc(Nx,Ny,Nz)        !  global 3D photolysis rate data array
  REAL wr(Nx,Ny,Nz)        !  global 3D photolysis rate data array
  REAL sprc(Nx,Ny)         !  global 2D unknown array
  REAL rvel(Nx,Ny,Nz)      !  global 3D removal velocity array
  REAL sx(Nx,Nz,2,N_gas)   !  global BD conc array for east-west boundaries
  REAL sy(Ny,Nz,2,N_gas)   !  global BD conc array for north-south boundaries
```

```
REAL sz(Nx,Ny,N_gas)      !  global 2DN conc array for Z-transport
REAL q(Nx,Ny,N_gas)       !  global 2DN unknown array for Z-transport
REAL em(Nx,Ny,Nz,N_gas)   !  global 4D emission rates array
REAL vg(Nx,Ny,N_gas)      !  global 2DN unknown array for Z-transport
REAL fz(Nx,Ny,N_gas)      !  global 2DN unknown array for input call
REAL hdz(Nx,Ny,Nz)        !  global 3D unknown array for chemistry calcs
REAL h(Nx,Ny)             !  global 2D top layer height array
REAL tlon(Nx,Ny)          !  global 2D unknown array for chemistry calcs
REAL tlat(Nx,Ny)          !  global 2D unknown array for chemistry calcs
```

Details

This routine allocates the memory for all the arrays specified. It can be used to allocate the global as well as the local arrays because it uses the input variables x, y, z and N_gas that can be varied in order to allocate the necessary arrays. Therefore workers can use the routine with specifying local names of the arrays and the local values for x, y, z and N_gas.

Location

mpi_memalloc.f
Module: `XYParallelMemAlloc`

See Also

MinimAlloc (page 71), MemAlloc for HV-partitioning (page 77)

MinimAlloc

Allocates minimal memory for the arrays specified.

Syntax

```
MinimAlloc(sg,sl,sp,u,v,w,kh,kv,t,dz,wc,wr,
         sprc,rvel,sx,sy,sz,q,em,vg,fz,hdz,h,tlon,tlad)

  REAL sg(1,1,1,1)     !  global 4D gas conc array
  REAL sl(1,1,1,1)     !  global 4D liquid conc array
  REAL sp(1,1,1,1)     !  global 4D particle conc array
  REAL u(1,1,1)        !  global 3D wind field component array in x-dir
  REAL v(1,1,1)        !  global 3D wind field component array in y-dir
  REAL w(1,1,1)        !  global 3D wind field component array in z-dir
  REAL kh(1,1,1)       !  global 3D hor. diffusitivity array
  REAL kv(1,1,1)       !  global 3D ver. diffusitivity array
  REAL t(1,1,1)        !  global 3D global time array
  REAL dz(1,1,1)       !  global 3D ver. resolution array
  REAL wc(1,1,1)       !  global 3D photolysis rate data array
  REAL wr(1,1,1)       !  global 3D photolysis rate data array
  REAL sprc(1,1)       !  global 2D unknown array
  REAL rvel(1,1,1)     !  global 3D removal velocity array
  REAL sx(1,1,2,1)     !  global BD conc array for east-west boundaries
  REAL sy(1,1,2,1)     !  global BD conc array for north-south boundaries
  REAL sz(1,1,1)       !  global 2DN conc array for Z-transport
  REAL q(1,1,1)        !  global 2DN unknown array for Z-transport
  REAL em(1,1,1,1)     !  global 4D emission rates array
```

```
REAL vg(1,1,1)       !  global 2DN unknown array for Z-transport
REAL fz(1,1,1)       !  global 2DN unknown array for input call
REAL hdz(1,1,1)      !  global 3D unknown array for chemistry calcs
REAL h(1,1)          !  global 2D top layer height array
REAL tlon(1,1)       !  global 2D unknown array for chemistry calcs
REAL tlat(1,1)       !  global 2D unknown array for chemistry calcs
```

Details

This routine allocates minimal memory for all the arrays specified. It can be used on the master to allocate the X and Y-worker specific variables not to be nil pointers as well as on the workers to minimally allocate the master's and unnecessary worker specific variables in order to reduce run time errors with nil pointers.

Location

mpi_memalloc.f
Module: `XYParallelMemAlloc`

See Also

MemAlloc (page 70), MinimAlloc for HV-partitioning (page 78)

distrib_xy

Distribute all arrays required in the model into their X and Y-slice formats.

Syntax

```
distrib_xy(Nx,Ny,Nz,Nxloc,Nyloc,N_gas,N_liq,N_part,
          sg,sg_x,sg_y,sl,sl_x,sl_y,sp,sp_x,sp_y,
          u,v,w,u_x,v_x,w_x,u_y,v_y,w_y,
          kh,kv,t,kh_x,kv_x,t_x,kh_y,kv_y,t_y,
          wc,wc_x,wc_y,wr,wr_x,wr_y,rvel,rvel_x,rvel_y,
          q,q_x,q_y,sprc,sprc_x,sprc_y,em,em_x,em_y,
          vg,vg_x,vg_y,fz,fz_x,fz_y,hdz,hdz_x,hdz_y,
          sx,sx_y,sy,sy_x,sz,sz_x,sz_y,h,h_x,h_y,
          tlon,tlon_x,tlon_y,tlat,tlat_x,tlat_y,dz,dz_x,dz_y)

INTEGER Nx                  !  global no. of grid points in x-dim
INTEGER Ny                  !  global no. of grid points in y-dim
INTEGER Nz                  !  global & local no. of grid points in z-dim
INTEGER Nxloc               !  local no. of grid points in x-dim
INTEGER Nyloc               !  local no. of grid points in y-dim
INTEGER N_gas               !  local & global no. of gas species dim
INTEGER N_liq               !  local & global no. of liquid species dim
INTEGER N_part              !  local & global no. of particle species dim
REAL sg(Nx,Ny,Nz,N_gas)     !  global 4D gas conc array
REAL sg_x(Nx,Nyloc,Nz,N_gas) !  local 4D gas conc array at X-worker
REAL sg_y(Nxloc,Ny,Nz,N_gas) !  local 4D gas conc array at Y-worker
REAL sl(Nx,Ny,Nz,N_liq)     !  global 4D liquid conc array
REAL sl_x(Nx,Nyloc,Nz,N_gas) !  local 4D liquid conc array at X-worker
REAL sl_y(Nxloc,Ny,Nz,N_gas) !  local 4D liquid conc array at Y-worker
```

```
REAL sp(Nx,Ny,Nz,N_part)      !  global 4D particle conc array
REAL sp_x(Nx,Nyloc,Nz,N_gas)  !  local 4D particle conc array at X-worker
REAL sp_y(Nxloc,Ny,Nz,N_gas)  !  local 4D particle conc array at Y-worker
REAL u(Nx,Ny,Nz)              !  global 3D wind field component array in x-dir
REAL v(Nx,Ny,Nz)              !  global 3D wind field component array in y-dir
REAL w(Nx,Ny,Nz)              !  global 3D wind field component array in z-dir
REAL u_x(Nx,Nyloc,Nz)         !  local 3D wind field component array in x-dir
                              !  at X-worker
REAL v_x(Nx,Nyloc,Nz)         !  local 3D wind field component array in y-dir
                              !  at X-worker
REAL w_x(Nx,Nyloc,Nz)         !  local 3D wind field component array in z-dir
                              !  at X-worker
REAL u_y(Nxloc,Ny,Nz)         !  local 3D wind field component array in x-dir
                              !  at Y-worker
REAL v_y(Nxloc,Ny,Nz)         !  local 3D wind field component array in y-dir
                              !  at Y-worker
REAL w_y(Nxloc,Ny,Nz)         !  local 3D wind field component array in z-dir
                              !  at Y-worker
REAL kh(Nx,Ny,Nz)             !  global 3D hor. diffusitivity array
REAL kv(Nx,Ny,Nz)             !  global 3D ver. diffusitivity array
REAL t(Nx,Ny,Nz)              !  global 3D global time array
REAL kh_x(Nx,Nyloc,Nz)        !  local 3D hor. diffusitivity array at X-worker
REAL kv_x(Nx,Nyloc,Nz)        !  local 3D ver. diffusitivity array at X-worker
REAL t_x(Nx,Nyloc,Nz)         !  local 3D global time array at X-worker
REAL kh_y(Nxloc,Ny,Nz)        !  local 3D hor. diffusitivity array at Y-worker
REAL kv_y(Nxloc,Ny,Nz)        !  local 3D ver. diffusitivity array at Y-worker
REAL t_y(Nxloc,Ny,Nz)         !  local 3D global time array at Y-worker
REAL wc(Nx,Ny,Nz)             !  global 3D photolysis rate data array
REAL wc_x(Nx,Nyloc,Nz)        !  local 3D photolysis rate data array at X-worker
REAL wc_y(Nxloc,Ny,Nz)        !  local 3D photolysis rate data array at Y-worker
REAL wr(Nx,Ny,Nz)             !  global 3D photolysis rate data array
REAL wr_x(Nx,Nyloc,Nz)        !  local 3D photolysis rate data array at X-worker
REAL wr_y(Nxloc,Ny,Nz)        !  local 3D photolysis rate data array at Y-worker
REAL rvel(Nx,Ny,Nz)           !  global 3D removal velocity array
REAL rvel_x(Nx,Nyloc,Nz)      !  local 3D removal velocity array at X-worker
REAL rvel_y(Nxloc,Ny,Nz)      !  local 3D removal velocity array at Y-worker
REAL q(Nx,Ny,N_gas)           !  global 2DN unknown array for Z-transport
REAL q_x(Nx,Nyloc,Nz)         !  local 2DN unknown array at X-worker
REAL q_y(Nxloc,Ny,Nz)         !  local 2DN unknown array at Y-worker
REAL sprc(Nx,Ny)              !  global 2D unknown array
REAL sprc_x(Nx,Nyloc)         !  local 2D unknown array at X-worker
REAL sprc_y(Nxloc,Ny)         !  local 2D unknown array at Y-worker
REAL em(Nx,Ny,Nz,N_gas)       !  global 4D emission rates array
REAL em_x(Nx,Nyloc,Nz,N_gas)  !  local 4D emission rates array at X-worker
REAL em_y(Nxloc,Ny,Nz,N_gas)  !  local 3D emission rates array at Y-worker
REAL vg(Nx,Ny,N_gas)          !  global 2DN deposition velocity array
REAL vg_x(Nx,Nyloc,Nz)        !  local 2DN deposition velocity array at X-worker
REAL vg_y(Nxloc,Ny,Nz)        !  local 2DN deposition velocity array at Y-worker
REAL fz(Nx,Ny,N_gas)          !  global 2DN unknown array for input call
REAL fz_x(Nx,Nyloc,Nz)        !  local 2DN unknown array at X-worker
REAL fz_y(Nxloc,Ny,Nz)        !  local 2DN unknown array at Y-worker
REAL hdz(Nx,Ny,Nz)            !  global 3D unknown array for chemistry calcs
REAL hdz_x(Nx,Nyloc,Nz)       !  local 3D unknown array at X-worker
REAL hdz_y(Nxloc,Ny,Nz)       !  local 3D unknown array at Y-worker
```

```
REAL sx(Ny,Nz,2,N_gas)        !  global BD conc array for east-west boundaries
REAL sx_x(Nyloc,Nz,2,N_gas)   !  local BD conc array at X-worker
REAL sy(Nx,Nz,2,N_gas)        !  global BD conc array for north-south boundaries
REAL sy_y(Nxloc,Nz,2,N_gas)   !  local BD conc array at Y-worker
REAL sz(Nx,Ny,N_gas)          !  global 2DN conc array for Z-transport
REAL sz_x(Nx,Nyloc,Nz)        !  local 2DN conc array at X-worker
REAL sz_y(Nxloc,Ny,Nz)        !  local 2DN conc array at Y-worker
REAL h(Nx,Ny)                 !  global 2D top layer height array
REAL h_x(Nx,Nyloc)            !  local 2D top layer height array at X-worker
REAL h_y(Nxloc,Ny)            !  local 2D top layer height array at Y-worker
REAL tlon(Nx,Ny)              !  global 2D unknown array for chemistry calcs
REAL tlon_x(Nx,Nyloc)         !  local 2D unknown array at X-worker
REAL tlon_y(Nxloc,Ny)         !  local 2D unknown array at Y-worker
REAL tlat(Nx,Ny)              !  global 2D unknown array for chemistry calcs
REAL tlat_x(Nx,Nyloc)         !  local 2D unknown array at X-worker
REAL tlat_y(Nxloc,Ny)         !  local 2D unknown array at Y-worker
REAL dz(Nx,Ny,Nz)             !  global 3D ver. resolution array
REAL dz_x(Nx,Nyloc,Nz)        !  local 3D ver. resolution array at X-worker
REAL dz_y(Nxloc,Ny,Nz)        !  local 3D ver. resolution array at Y-worker
```

Details

All arrays used by the model for X, Y, Z-transport and chemistry calculations are distributed in XY-partitioning using the associated library functions for each array. If further arrays and data are used for the serial model during any of above mentioned computations they will need to be added to this function.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

distrib_hv (page 79)

int_distrib

Distribute several integer variables from the master to all workers.

Syntax

```
int_distrib(numl,nbin,xtrn,ytrn,ztrn,rxng,rxnl,num,mdt,idate,iend)
```

```
INTEGER numl(3,4)         !
INTEGER nbin              !
INTEGER Nxtrn             !  decision flag for X-transport
INTEGER Nytrn             !  decision flag for Y-transport
INTEGER Nztrn             !  decision flag for Z-transport
INTEGER rxng              !  decision flag for reaction calcs
INTEGER rxnl              !
INTEGER num               !
INTEGER mdt               !  no. of inner loop iterations
INTEGER idate(3)          !  date of simulation data
INTEGER iend              !  no. of outer loop iterations
```

Details

This routine distributes several integer variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers. Integer values that may change after the 6-hour interval when new data is read in and therefore need to be redistributed are included in this routine.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

int_distrib for HV-partitioning (page 80)

int_distrib1

Distribute further integer variables from the master to all workers.

Syntax

```
int_distrib1(iend,numl)

  INTEGER iend              !  no. of outer loop iterations
  INTEGER numl(3,4)         !
```

Details

This routine distributes several global known integer variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers. Most of them are index values for certain species that are used very often like water (H_2O), air, Oxygen (O_2) and Nitrogen (N_2) as well as other indices. All variables part of this routine are set in the beginning of the simulation and do not change at any time during computations.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

int_distrib1 for HV-partitioning (page 81)

real_distrib

Distribute several real variables from the master to all workers.

Syntax

```
real_distrib(Nx,Ny,Nz,Ns,dx,dy,sigmaz,dht,baseh,rmw,dt,ut)
```

```
INTEGER Nx                  !  global no. of grid points in x-dim
INTEGER Ny                  !  global no. of grid points in y-dim
INTEGER Nz                  !  global no. of grid points in z-dim
INTEGER Ns                  !  global no. of chemical species dim
REAL dx(Nx)                 !
REAL dy(Ny)                 !
REAL sigmaz(Nz)             !
REAL dht                    !
REAL baseh                  !
REAL rmw(Ns)                !
REAL dt                     !
REAL ut                     !
```

Details

This routine distributes several real variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

real_distrib for HV-partitioning (page 82)

---

cmm_distrib

Distribute all common block data from the master to all workers.

Syntax

`cmm_distrib()`

Details

This routine distributes all common block data from the master to the workers by copying the values to several buffers and broadcasting these buffers to all workers.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

cmm_distrib for HV-partitioning (page 82)

A.2 HV-Partitioning

MemAlloc

Allocates the memory for the arrays specified.

Syntax

```
MemAlloc(Nx,Ny,Nz,N_gas,N_liq,N_part,
         sg,sl,sp,u,v,w,kh,kv,t,dz,wc,wr,
         sprc,rvel,sx,sy,sz,q,em,vg,fz,hdz,h,tlon,tlat

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER N_gas              !  global no. of gas species dim
  INTEGER N_liq              !  global no. of liquid species dim
  INTEGER N_part             !  global no. of particle species dim
  REAL sg(Nx,Ny,Nz,N_gas)    !  global 4D gas conc array
  REAL sl(Nx,Ny,Nz,N_liq)    !  global 4D liquid conc array
  REAL sp(Nx,Ny,Nz,N_part)   !  global 4D particle conc array
  REAL u(Nx,Ny,Nz)           !  global 3D wind field component array in x-dir
  REAL v(Nx,Ny,Nz)           !  global 3D wind field component array in y-dir
  REAL w(Nx,Ny,Nz)           !  global 3D wind field component array in z-dir
  REAL kh(Nx,Ny,Nz)          !  global 3D hor. diffusitivity array
  REAL kv(Nx,Ny,Nz)          !  global 3D ver. diffusitivity array
  REAL t(Nx,Ny,Nz)           !  global 3D global time array
  REAL dz(Nx,Ny,Nz)          !  global 3D ver. resolution array
  REAL wc(Nx,Ny,Nz)          !  global 3D photolysis rate data array
  REAL wr(Nx,Ny,Nz)          !  global 3D photolysis rate data array
  REAL sprc(Nx,Ny)           !  global 2D unknown array
  REAL rvel(Nx,Ny,Nz)        !  global 3D removal velocity array
  REAL sx(Nx,Nz,2,N_gas)     !  global BD conc array for east-west boundaries
  REAL sy(Ny,Nz,2,N_gas)     !  global BD conc array for north-south boundaries
  REAL sz(Nx,Ny,N_gas)       !  global 2DN conc array for Z-transport
  REAL q(Nx,Ny,N_gas)        !  global 2DN unknown array for Z-transport
  REAL em(Nx,Ny,Nz,N_gas)    !  global 4D emission rates array
  REAL vg(Nx,Ny,N_gas)       !  global 2DN unknown array for Z-transport
  REAL fz(Nx,Ny,N_gas)       !  global 2DN unknown array for input call
  REAL hdz(Nx,Ny,Nz)         !  global 3D unknown array for chemistry calcs
  REAL h(Nx,Ny)              !  global 2D top layer height array
  REAL tlon(Nx,Ny)           !  global 2D unknown array for chemistry calcs
  REAL tlat(Nx,Ny)           !  global 2D unknown array for chemistry calcs
```

Details

This routine allocates the memory for all the arrays specified. It can be used to allocate the global as well as the local arrays because it uses the input variables x, y, z, and N_gas that can be varied in order to allocate the necessary arrays. Therefore workers can use the routine with specifying local names of the arrays and the local values for x, y, z, and N_gas.

Location

mpi_memalloc.f
Module: `HVParallelMemAlloc`

See Also

MinimAlloc (page 78), MemAlloc for XY-partitioning (page 70)

MinimAlloc

Allocates minimal memory for the arrays specified.

Syntax

```
MinimAlloc(sg,sl,sp,u,v,w,kh,kv,t,dz,wc,wr,
          sprc,rvel,sx,sy,sz,q,em,vg,fz,hdz,h,tlon,tlad)

  REAL sg(1,1,1,1)     !  global 4D gas conc array
  REAL sl(1,1,1,1)     !  global 4D liquid conc array
  REAL sp(1,1,1,1)     !  global 4D particle conc array
  REAL u(1,1,1)        !  global 3D wind field component array in x-dir
  REAL v(1,1,1)        !  global 3D wind field component array in y-dir
  REAL w(1,1,1)        !  global 3D wind field component array in z-dir
  REAL kh(1,1,1)       !  global 3D hor. diffusitivity array
  REAL kv(1,1,1)       !  global 3D ver. diffusitivity array
  REAL t(1,1,1)        !  global 3D global time array
  REAL dz(1,1,1)       !  global 3D ver. resolution array
  REAL wc(1,1,1)       !  global 3D photolysis rate data array
  REAL wr(1,1,1)       !  global 3D photolysis rate data array
  REAL sprc(1,1)       !  global 2D unknown array
  REAL rvel(1,1,1)     !  global 3D removal velocity array
  REAL sx(1,1,2,1)     !  global BD conc array for east-west boundaries
  REAL sy(1,1,2,1)     !  global BD conc array for north-south boundaries
  REAL sz(1,1,1)       !  global 2DN conc array for Z-transport
  REAL q(1,1,1)        !  global 2DN unknown array for Z-transport
  REAL em(1,1,1,1)     !  global 4D emission rates array
  REAL vg(1,1,1)       !  global 2DN unknown array for Z-transport
  REAL fz(1,1,1)       !  global 2DN unknown array for input call
  REAL hdz(1,1,1)      !  global 3D unknown array for chemistry calcs
  REAL h(1,1)          !  global 2D top layer height array
  REAL tlon(1,1)       !  global 2D unknown array for chemistry calcs
  REAL tlat(1,1)       !  global 2D unknown array for chemistry calcs
```

Details

This routine allocates minimal memory for all the arrays specified. It can be used on the master to allocate the H and V-worker specific variables not to be nil pointers as well as on the workers to allocate the master's and other worker specific variables in order to reduce run time errors with nil pointers.

Location

mpi_memalloc.f
Module: `HVParallelMemAlloc`

See Also

MemAlloc (page 77), MinimAlloc for XY-partitioning (page 71)

distrib_hv

Distribute all arrays required in the model into their H-slice and V-column formats.

Syntax

```
distrib_hv(Nx,Ny,Nz,Nzloc,Ncloc,N_gas,N_liq,N_part,
           sg,sg_x,sg_y,sl,sl_x,sl_y,sp,sp_x,sp_y,
           u,v,w,u_x,v_x,w_x,u_y,v_y,w_y,
           kh,kv,t,kh_x,kv_x,t_x,kh_y,kv_y,t_y,
           wc,wc_x,wc_y,wr,wr_x,wr_y,rvel,rvel_x,rvel_y,
           q,q_x,q_y,sprc,sprc_x,sprc_y,em,em_x,em_y,
           vg,vg_x,vg_y,fz,fz_x,fz_y,hdz,hdz_x,hdz_y,
           sx,sx_y,sy,sy_x,sz,sz_x,sz_y,h,h_x,h_y,
           tlon,tlon_x,tlon_y,tlat,tlat_x,tlat_y,dz,dz_x,dz_y)
```

```
   INTEGER Nx                    !  global no. of grid points in x-dim
   INTEGER Ny                    !  global no. of grid points in y-dim
   INTEGER Nz                    !  global & local no. of grid points in z-dim
   INTEGER Nzloc                 !  local no. of grid points in z-dim
   INTEGER Ncloc                 !  local no. of ver. columns
   INTEGER N_gas                 !  local & global no. of gas species dim
   INTEGER N_liq                 !  local & global no. of liquid species dim
   INTEGER N_part                !  local & global no. of particle species dim
   REAL sg(Nx,Ny,Nz,N_gas)       !  global 4D gas conc array
   REAL sg_x(Nx,Ny,Nzloc,N_gas)  !  local 4D gas conc array at H-worker
   REAL sg_y(1,Ncloc,Nz,N_gas)   !  local 4D gas conc array at V-worker
   REAL sl(Nx,Ny,Nz,N_liq)       !  global 4D liquid conc array
   REAL sl_x(Nx,Ny,Nzloc,N_gas)  !  local 4D liquid conc array at H-worker
   REAL sl_y(1,Ncloc,Nz,N_gas)   !  local 4D liquid conc array at V-worker
   REAL sp(Nx,Ny,Nz,N_part)      !  global 4D particle conc array
   REAL sp_x(Nx,Ny,Nzloc,N_gas)  !  local 4D particle conc array at H-worker
   REAL sp_y(1,Ncloc,Nz,N_gas)   !  local 4D particle conc array at V-worker
   REAL u(Nx,Ny,Nz)              !  global 3D wind field component array in x-dir
   REAL v(Nx,Ny,Nz)              !  global 3D wind field component array in y-dir
   REAL w(Nx,Ny,Nz)              !  global 3D wind field component array in z-dir
   REAL u_h(Nx,Ny,Nzloc)         !  local 3D wind field component array in x-dir
                                 !  at H-worker
   REAL v_h(Nx,Ny,Nzloc)         !  local 3D wind field component array in y-dir
                                 !  at H-worker
   REAL w_v(1,Ncloc,Nz)          !  local 3D wind field component array in z-dir
                                 !  at V-worker
   REAL kh(Nx,Ny,Nz)             !  global 3D hor. diffusitivity array
   REAL kv(Nx,Ny,Nz)             !  global 3D ver. diffusitivity array
   REAL t(Nx,Ny,Nz)              !  global 3D global time array
   REAL kh_h(Nx,Ny,Nzloc)        !  local 3D hor. diffusitivity array at H-worker
   REAL kv_v(1,Ncloc,Nz)         !  local 3D ver. diffusitivity array at V-worker
   REAL t_v(1,Ncloc,Nz)          !  local 3D global time array at V-worker
   REAL wc(Nx,Ny,Nz)             !  global 3D photolysis rate data array
   REAL wc_v(1,Ncloc,Nz)         !  local 3D photolysis rate data array at V-worker
   REAL wr(Nx,Ny,Nz)             !  global 3D photolysis rate data array
```

```
REAL wr_v(1,Ncloc,Nz)        !  local 3D photolysis rate data array at V-worker
REAL rvel(Nx,Ny,Nz)          !  global 3D removal velocity array
REAL rvel_v(1,Ncloc,Nz)      !  local 3D removal velocity array at V-worker
REAL q(Nx,Ny,N_gas)          !  global 2DN unknown array for Z-transport
REAL q_v(1,Ncloc,Nz)         !  local 2DN unknown array at V-worker
REAL sprc(Nx,Ny)             !  global 2D unknown array
REAL sprc_v(1,Ncloc)         !  local 2D unknown array at V-worker
REAL em(Nx,Ny,Nz,N_gas)      !  global 4D emission rates array
REAL em_v(1,Ncloc,Nz,N_gas)  !  local 3D emission rates array at V-worker
REAL vg(Nx,Ny,N_gas)         !  global 2DN deposition velocity array
REAL vg_v(1,Ncloc,Nz)        !  local 2DN deposition velocity array at V-worker
REAL fz(Nx,Ny,N_gas)         !  global 2DN unknown array for input call
REAL fz_v(1,Ncloc,Nz)        !  local 2DN unknown array at V-worker
REAL hdz(Nx,Ny,Nz)           !  global 3D unknown array for chemistry calcs
REAL hdz_v(1,Ncloc,Nz)       !  local 3D unknown array at V-worker
REAL sx(Nx,Nz,2,N_gas)       !  global BD conc array for east-west boundaries
REAL sx_h(Nx,Nzloc,2,N_gas)  !  local BD conc array at H-worker
REAL sy(Ny,Nz,2,N_gas)       !  global BD conc array for north-south boundaries
REAL sy_h(Ny,Nzloc,2,N_gas)  !  local BD conc array at H-worker
REAL sz(Nx,Ny,N_gas)         !  global 2DN conc array for Z-transport
REAL sz_v(1,Ncloc,Nz)        !  local 2DN conc array at V-worker
REAL h(Nx,Ny)                !  global 2D top layer height array
REAL h_v(1,Ncloc)            !  local 2D top layer height array at V-worker
REAL tlon(Nx,Ny)             !  global 2D unknown array for chemistry calcs
REAL tlon_v(1,Ncloc)         !  local 2D unknown array at V-worker
REAL tlat(Nx,Ny)             !  global 2D unknown array for chemistry calcs
REAL tlat_v(1,Ncloc)         !  local 2D unknown array at V-worker
REAL dz(Nx,Ny,Nz)            !  global 3D ver. resolution array
REAL dz_v(1,Ncloc,Nz)        !  local 3D ver. resolution array at V-worker
```

Details

All arrays used by the model for X, Y, X-transport and chemistry calculations are distributed in HV-partitioning using the associated library functions for each array. If further arrays and data are used for the serial model during any of above mentioned computations they will need to be added to this function.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

distrib_xy (page 72)

int_distrib

Distribute several integer variables from the master to all workers.

Syntax

```
int_distrib(numl,nbin,xtrn,ytrn,ztrn,rxng,rxnl,num,mdt,idate,iend)
```

```
INTEGER numl(3,4)          !
INTEGER nbin               !
INTEGER Nxtrn              !  decision flag for X-transport
INTEGER Nytrn              !  decision flag for Y-transport
INTEGER Nztrn              !  decision flag for Z-transport
INTEGER rxng               !  decision flag for reaction calcs
INTEGER rxnl               !
INTEGER num                !
INTEGER mdt                !  no. of inner loop iterations
INTEGER idate(3)           !  date of simulation data
INTEGER iend               !  no. of outer loop iterations
```

Details

This routine distributes several integer variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers. Integer values that may change after the 6-hour interval when new data is read in and therefore need to be redistributed are included in this routine.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

int_distrib for XY-partitioning (page 74)

int_distrib1

Distribute further integer variables from the master to all workers.

Syntax

```
int_distrib1(iend,numl)

INTEGER iend               !  no. of outer loop iterations
INTEGER numl(3,4)          !
```

Details

This routine distributes several global known integer variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers. All variables part of this routine are set in the beginning of the simulation and do not change at any time during computations.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

int_distrib1 for XY-partitioning (page 75)

real_distrib

Distribute several real variables from the master to all workers.

Syntax

```
real_distrib(Nx,Ny,Nz,Ns,dx,dy,sigmaz,dht,baseh,rmw,dt,ut)

   INTEGER Nx                 !  global no. of grid points in x-dim
   INTEGER Ny                 !  global no. of grid points in y-dim
   INTEGER Nz                 !  global no. of grid points in z-dim
   INTEGER Ns                 !  global no. of chemical species dim
   REAL dx(Nx)                !
   REAL dy(Ny)                !
   REAL sigmaz(Nz)            !
   REAL dht                   !
   REAL baseh                 !
   REAL rmw(Ns)               !
   REAL dt                    !
   REAL ut                    !
```

Details

This routine distributes several real variables from the master to all workers by copying the values to a buffer and broadcasting that buffer to all workers.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

real_distrib for XY-partitioning (page 75)

cmm_distrib

Distribute all common block data from the master to all workers.

Syntax

`cmm_distrib()`

Details

This routine distributes all common block data from the master to the workers by copying the values to several buffers and broadcasting these buffer to all workers.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

cmm_distrib for XY-partitioning (page 76)

Appendix B

General Communication Functions

The following general routines need to be included in order to use the communication library. While functions described in the last chapter were STEM-III specific, routines illustrated in this chapter are the general basis of the library.

B.1 XY-Partitioning

init_xy

Initialize and set up the communication for XY-partitioning.

Syntax

```
init_xy(Nx,Ny,Nz,Nxloc,Nyloc,Ns)

  INTEGER Nx              !  global no. of grid points in x-dim
  INTEGER Ny              !  global no. of grid points in y-dim
  INTEGER Nz              !  global no. of grid points in z-dim
  INTEGER Nxloc           !  local no. of grid points in x-dim on Y-workers
  INTEGER Nyloc           !  local no. of grid points in y-dim on X-workers
  INTEGER Ns              !  no. of chemical species to be monitored
```

Details

This routine initializes the parallel communication for XY-partitioning by calling the three later described functions `init_xy_versions`, `CreateMap` and `CreateCommDataTypes`. Calling this routine instead of the others makes sure that the right modules are included.

Location

mpi_communication.f
Module: `XYParallelCommunication`

See Also

init_xy_versions (page 84), CreateMap (page 83), CreateCommDataTypes (page 85)

CreateMap

Create the mapping between X and Y-slices and processors as well as initializing global variables.

Syntax

```
CreateMap(Nx,Ny,Nxloc,Nyloc)

  INTEGER Nx             ! global no. of grid points in x-dim
  INTEGER Ny             ! global no. of grid points in y-dim
  INTEGER Nxloc          ! local no. of grid points in x-dim on Y-workers
  INTEGER Nyloc          ! local no. of grid points in y-dim on X-workers
```

Details

This routine is part of the module ParallelDataMap. It assigns the processors to be X-workers and Y-workers, respectively and keeps book about the assignments of the global X and Y-slices to these workers. This includes the number of X and Y-slices assigned to each worker, their global and local id, as well as the owner relationship to each X and Y-slice.

Most global variables are set in this routine: the number of X and Y-workers, the size of the local arrays at the workers as well as mapping variables. The provided routine can serve as an example on how to set up a different mapping. We have used the following global mapping variables:

`owner_of_xslice(1..Ny)`	specifies the process that gets the X-slice
`owner_of_yslice(1..Nx)`	specifies the process that gets the Y-slice
`no_of_xslices(p)`	number of X-slices owned by process p where $p \in \{1, \ldots, \texttt{NXworkers}\}$
`no_of_yslices(p)`	number of Y-slices owned by process p where $p \in \{1, \ldots, \texttt{NYworkers}\}$
`local_xslice_id(1..Ny)`	local index of the global X-slice (index seen by the process which owns it)
`local_yslice_id(1..Nx)`	local index of the global Y-slice (index seen by the process which owns it)
`global_xslice_id(p,1..Nyloc)`	global index of a local X-slice on processor p
`global_yslice_id(p,1..Nxloc)`	global index of a local Y-slice on processor p
`owned_xslices(p,1..no_of_xslices(p))`	global X-slices owned by worker p
`owned_yslices(p,1..no_of_yslices(p))`	global Y-slices owned by worker p

Location

mpi_util.f
Module: `XYParallelDataMap`

See Also

CreateMap for HV-partitioning (page 97)

init_xy_versions

Initialize the communication versions to be used.

Syntax

```
init_xy_versions()
```

Details

This routine reads the file created by the setup program that specifies the fastest version numbers for the communication routines on the specific architecture using a set grid resolution and given number of processors. If the setup program has not been run the default version (version 1) will be used.

To decide on the versions during runtime either a few subroutines have to be provided that map the integers specified by the setup program (1, 2, 3) to the communication routine version. Another approach is hardcoding the best routines for the used architecture in the AQM-specific functions after testing the speed of each version using the setup programs.

Location

mpi_communication.f
Module: `XYParallelCommunication`

CreateCommDataTypes

Creates the data types used by the library routines for communication.

Syntax

```
CreateCommDataTypes(Nx,Ny,Nz,Nxloc,Nyloc,Ns)

  INTEGER Nx              !  global no. of grid points in x-dim
  INTEGER Ny              !  global no. of grid points in y-dim
  INTEGER Nxloc           !  local no. of grid points in x-dim on Y-workers
  INTEGER Nyloc           !  local no. of grid points in y-dim on X-workers
  INTEGER Nz              !  global no. of grid points in z-dim
  INTEGER Ns              !  no. of chemical species to be monitored
```

Details

This routine creates the worker group used for communication during shuffling as well as the communication data types for each of the array types in the air quality model. There have to be slice-types to do the slice-communication in distribution and gathering. Other data types created are the combined slices for the single message distribution and gathering as well as the combined data to be sent during shuffling when using one message only. These data types represent parts of an array that can be located anywhere in the array specified by count numbers and displacements.

Most of these data types depend on the mapping function described previously. The implementation can serve as an example on how to define these data types when using different mapping routines. Be aware that the combining of X and Y-slices to one data type for single message communication may not be possible if the mapping is irregular! In that case only the first version of communication routines can be used that distribute and gather the X and Y-slices slice by slice.

Location

mpi_util.f
Module: `XYCommDataTypes`

See Also

CreateCommDataTypes for HV-partitioning (page 98)

B.2 XY Library

distrib_x_4D

Distribute a 4D array in X-slice partitioning from the master to the X-workers.

Syntax

```
distrib_x_4D_v[1|2|3](Nx,Ny,Nz,Nyloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global & local no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nyloc              !  local no. of grid points in y-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(Nx,Nyloc,Nz,Ns)  !  local 4D array on each X-worker
```

Details

These routines distribute a global 4D array at the master into X-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Ny messages

v2 - distribution sending one message to each X-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary`

See Also

distrib_y_4D (page 86), distrib_h_4D (page 99)

distrib_y_4D

Distribute a 4D array in Y-slice partitioning from the master to the Y-workers.

Syntax

```
distrib_y_4D_v[1|2|3](Nx,Ny,Nz,Nxloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global & local no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nxloc              !  local no. of grid points in x-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(Nxloc,Ny,Nz,Ns)  !  local 4D array on each Y-worker
```

Details

These routines distribute a global 4D array at the master into Y-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using x messages

v2 - distribution sending one message to each Y-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary`

See Also

distrib_x_4D (page 86), distrib_v_4D (page 99)

gather_x_4D

Gather a 4D array at the master from X-slice partitioning at the X-workers.

Syntax

```
gather_x_4D_v[1|2|3](Nx,Ny,Nz,Nyloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global & local no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nyloc              !  local no. of grid points in y-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(Nx,Nyloc,Nz,Ns)  !  local 4D array on each X-worker
```

Details

These routines gather the local 4D arrays at the X-workers into the global 4D array at the master. The following three versions are implemented:

v1 - master gathering, receiving slice by slice in Ny messages

v2 - master gathering, receiving one message from each X-worker combining the local slices if more than one

v3 - using two calls to `MPI_GATHERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

mpi_commlibrary.f
Module: XYCommunicationLibrary

See Also

gather_y_4D (page 88), gather_h_4D (page 100)

---

gather_y_4D

Gather a 4D array at the master from Y-slice partitioning at the Y-workers.

Syntax

```
gather_y_4D_v[1|2|3](Nx,Ny,Nz,Nxloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global & local no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nxloc              !  local no. of grid points in x-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(Nxloc,Ny,Nz,Ns)  !  local 4D array on each Y-worker
```

Details

These routines gather the local 4D arrays at the Y-workers into the global 4D array at the master. The following three versions are implemented:

v1 - master gathering, receiving slice by slice in x messages

v2 - master gathering, receiving one message from each Y-worker combining the local slices if more than one

v3 - using two calls to MPI_GATHERV

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function CreateCommDatatypes and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

mpi_commlibrary.f
Module: XYCommunicationLibrary

See Also

gather_x_4D (page 87),gather_v_4D (page 101)

---

shuffle_x2y_4D

Shuffle 4D arrays from X-partitioning on X-workers to Y-partitioning on Y-workers.

Syntax

```
shuffle_x2y_4D_v[1|2|3](Nx,Ny,Nz,Nxloc,Nyloc,Ns,alocx,alocy)

  INTEGER Nx                  !  global no. of grid points in x-dim
  INTEGER Ny                  !  global no. of grid points in y-dim
  INTEGER Nz                  !  global & local no. of grid points in z-dim
  INTEGER Nxloc               !  local no. of grid points in x-dim
  INTEGER Nyloc               !  local no. of grid points in y-dim
  INTEGER Ns                  !  local & global no. of chemical species dim
  REAL alocx(Nx,Nyloc,Nz,Ns) !  local 4D array on each X-worker
  REAL alocy(Nxloc,Ny,Nz,Ns) !  local 4D array on each Y-worker
```

Details

These routines rearrange the local 4D in X-partitioning at the X-workers to Y-partitioning at the Y-workers. The following three versions are implemented:

v1 - distribution sending slice-part by slice-part using Nyloc messages on each X-worker

v2 - distribution sending one message from each X-worker to each Y-worker combining the necessary data

v3 - using copying and `MPI_ALLTOALL`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` shuffle_y2x_4D (page 89), shuffle_h2v_4D (page 101)

shuffle_y2x_4D

Shuffle 4D arrays from Y-partitioning on Y-workers to X-partitioning on X-workers.

Syntax

```
shuffle_y2x_4D_v[1|2|3](Nx,Ny,Nz,Nxloc,Nyloc,Ns,alocx,alocy)

  INTEGER Nx                  !  global no. of grid points in x-dim
  INTEGER Ny                  !  global no. of grid points in y-dim
  INTEGER Nz                  !  global & local no. of grid points in z-dim
  INTEGER Nxloc               !  local no. of grid points in x-dim
  INTEGER Nyloc               !  local no. of grid points in y-dim
  INTEGER Ns                  !  local & global no. of chemical species dim
  REAL alocx(Nx,Nyloc,Nz,Ns) !  local 4D array on each X-worker
  REAL alocy(Nxloc,Ny,Nz,Ns) !  local 4D array on each Y-worker
```

Details

These routines rearrange the local 4D in Y-partitioning at the Y-workers to X-partitioning at the X-workers. The following three versions are implemented:

v1 - distribution sending slice-part by slice-part using Nxloc messages on each Y-worker v2 - distribution sending one message from each Y-worker to each X-worker combining the necessary data

v3 - using copying and `MPI_ALLTOALL`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` shuffle_x2y_4D (page 88), shuffle_v2h_4D (page 102)

---

distrib_x_2D

Distribute a 2D array in X-slice partitioning from the master to the X-workers.

Syntax

`distrib_x_2D_v[1|2|3](Nx,Ny,Nyloc,aglob,aloc)`

```
INTEGER Nx                ! global & local no. of grid points in x-dim
INTEGER Ny                ! global no. of grid points in y-dim
INTEGER Nyloc             ! local no. of grid points in y-dim
REAL aglob(Nx,Ny)         ! global 2D array
REAL aloc(Nx,Nyloc)       ! local 2D array on each X-worker
```

Details

These routines distribute a global 2D array at the master into X-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Ny messages

v2 - distribution sending one message to each X-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_y_2D (page 90), distrib_h_2D (page 103)

---

distrib_y_2D

Distribute a 2D array in Y-slice partitioning from the master to the Y-workers.

Syntax

`distrib_y_2D_v[1|2|3](Nx,Ny,Nxloc,aglob,aloc)`

```
INTEGER Nx                ! global no. of grid points in x-dim
INTEGER Ny                ! global & local no. of grid points in y-dim
INTEGER Nxloc             ! local no. of grid points in x-dim
REAL aglob(Nx,Ny)         ! global 2D array
REAL aloc(Nxloc,Ny)       ! local 2D array on each Y-worker
```

Details

These routines distribute a global 2D array at the master into Y-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Nx messages

v2 - distribution sending one message to each Y-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_x_2D (page 90), distrib_v_2D (page 103)

distrib_x_2DN

Distribute a 2DN array in X-slice partitioning from the master to the X-workers.

Syntax

```
distrib_x_2DN_v[1|2|3](Nx,Ny,Nyloc,Ns,aglob,aloc)
```

```
INTEGER Nx                ! global & local no. of grid points in x-dim
INTEGER Ny                ! global no. of grid points in y-dim
INTEGER Nyloc             ! local no. of grid points in y-dim
INTEGER Ns                ! local & global no. of chemical species dim
REAL aglob(Nx,Ny,Ns)      ! global 2DN array
REAL aloc(Nx,Nyloc,Ns)    ! local 2DN array on each X-worker
```

Details

These routines distribute a global 2DN array at the master into X-slice partitioning to the workers. A 2DN array is a 3D array in the dimensions x, y, and number of chemical species. The following three versions are implemented:

v1 - distribution sending slice by slice using Ny messages

v2 - distribution sending one message to each X-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_y_2DN (page 92)

distrib_y_2DN

Distribute a 2DN array in Y-slice partitioning from the master to the Y-workers.

Syntax

```
distrib_y_2DN_v[1|2|3](Nx,Ny,Nxloc,Ns,aglob,aloc)

  INTEGER Nx               !  global no. of grid points in x-dim
  INTEGER Ny               !  global & local no. of grid points in y-dim
  INTEGER Nxloc            !  local no. of grid points in x-dim
  INTEGER Ns               !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Ns)     !  global 2DN array
  REAL aloc(Nxloc,Ny,Ns)   !  local 2DN array on each Y-worker
```

Details

These routines distribute a global 2DN array at the master into Y-slice partitioning to the workers. A 2DN array is a 3D array in the dimensions x, y, and number of chemical species. The following three versions are implemented:

v1 - distribution sending slice by slice using Nx messages

v2 - distribution sending one message to each Y-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_x_2DN (page 91), distrib_v_2DN (page 104)

distrib_x_3D

Distribute a 3D array in X-slice partitioning from the master to the X-workers.

Syntax

```
distrib_x_3D_v[1|2|3](Nx,Ny,Nz,Nyloc,aglob,aloc)

  INTEGER Nx               !  global & local no. of grid points in x-dim
  INTEGER Ny               !  global no. of grid points in y-dim
  INTEGER Nz               !  global & local no. of grid points in z-dim
  INTEGER Nyloc            !  local no. of grid points in y-dim
  REAL aglob(Nx,Ny,Nz)     !  global 3D array
  REAL aloc(Nx,Nyloc,Nz)   !  local 3D array on each X-worker
```

Details

These routines distribute a global 3D array at the master into X-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Ny messages

v2 - distribution sending one message to each X-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_y_3D (page 93), distrib_h_3D (page 105)

distrib_y_3D

Distribute a 3D array in Y-slice partitioning from the master to the Y-workers.

Syntax

```
distrib_y_3D_v[1|2|3](Nx,Ny,Nz,Nxloc,aglob,aloc)

  INTEGER Nx                !  global no. of grid points in x-dim
  INTEGER Ny                !  global & local no. of grid points in y-dim
  INTEGER Nz                !  global & local no. of grid points in z-dim
  INTEGER Nxloc             !  local no. of grid points in x-dim
  REAL aglob(Nx,Ny,Nz)      !  global 3D array
  REAL aloc(Nxloc,Ny,Nz)    !  local 3D array on each Y-worker
```

Details

These routines distribute a global 3D array at the master into Y-slice partitioning to the workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Nx messages

v2 - distribution sending one message to each Y-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_x_3D (page 92), distrib_v_3D (page 106)

distrib_x_BD

Distribute a BD array in X-slice partitioning from the master to the X-workers.

Syntax

```
distrib_x_BD_v[1|2|3](Ny,Nz,Nyloc,Ns,aglob,aloc)

  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nyloc              !  local no. of grid points in y-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Ny,Nz,2,Ns)     !  global BD array
  REAL aloc(Nyloc,Nz,2,Ns)   !  local BD array on each X-worker
```

Details

These routines distribute a global BD array at the master into X-slice partitioning to the workers. A BD array is a 4D array in the dimensions x, z, 2, and number of chemical species or y, z, 2, and number of chemical species that contains data about each chemical species on the east and west or north and south boundaries of the model. For X-slices we will need the east and west boundaries for each slice. The following three versions are implemented:

v1 - distribution sending slice by slice using Ny messages

v2 - distribution sending one message to each X-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_y_BD (page 94), distrib_xh_BD (page 106)

distrib_y_BD

Distribute a BD array in Y-slice partitioning from the master to the Y-workers.

Syntax

```
distrib_y_BD_v[1|2|3](Nx,Nz,Nxloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Nxloc              !  local no. of grid points in x-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Nz,2,Ns)     !  global BD array
  REAL aloc(Nxloc,Nz,2,Ns)   !  local BD array on each Y-worker
```

Details

These routines distribute a global BD array at the master into Y-slice partitioning to the workers. A BD array is a 4D array in the dimensions x, z, 2, and number of chemical species or y, z, 2, and number of chemical species that contains data about each chemical species on the east and west or north and south boundaries of the model. For Y-slices we will need the north and south boundaries for each slice. The following three versions are implemented:

v1 - distribution sending slice by slice using Nx messages

v2 - distribution sending one message to each Y-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `XYCommunicationLibrary` distrib_x_BD (page 94), distrib_yh_BD (page 107)

B.3 HV-Partitioning

init_hv

Initialize and set up the communication for HV-partitioning.

Syntax

```
init_hv(Nx,Ny,Nz,Nzloc,Ncloc,Ns)

  INTEGER Nx              !  global no. of grid points in x-dim
  INTEGER Ny              !  global no. of grid points in y-dim
  INTEGER Nz              !  global no. of grid points in z-dim
  INTEGER Nzloc           !  local no. of grid points in z-dim on H-workers
  INTEGER Ncloc           !  local no. of V-columns on V-workers
  INTEGER Ns              !  no. of chemical species to be monitored
```

Details

This routine initializes the parallel communication for HV-partitioning by calling the three later described functions `init_hv_versions`, `CreateMap` and `CreateCommDataTypes`. Calling this routine instead of the others makes sure that the right modules are included.

Location

mpi_communication.f
Module: `HVParallelCommunication`

See Also

init_hv_versions (page 96), CreateMap (page 97), CreateCommDataTypes (page 98)

init_hv_versions

Initialize the communication versions to be used.

Syntax

```
init_xy_versions()
```

Details

This routine reads the file created by the setup program that specifies the fastest version numbers for the communication routines on the specific architecture using a set grid resolution and given number of processors. If the setup program has not been run the default version (version 1) will be used.

To decide on the versions during runtime either a few subroutines have to be provided that map the integers specified by the setup program (1, 2, 3) to the communication routine version. Another approach is hardcoding the best routines for the used architecture in the AQM-specific functions after testing the speed of each version using the setup programs.

Location

mpi_communication.f
Module: `HVParallelCommunication`

CreateMap

Create the mapping between H-slices and V-columns and processors as well as important global variables.

Syntax

```
CreateMap(Nx,Ny,Nz,Nzloc,Ncloc)

  INTEGER Nx              !  global no. of grid points in x-dim
  INTEGER Ny              !  global no. of grid points in y-dim
  INTEGER Nz              !  global no. of grid points in z-dim
  INTEGER Nzloc           !  local no. of grid points in z-dim on H-workers
  INTEGER Ncloc           !  local no.   V-columns on V-workers
```

Details

This routine is part of the module ParallelDataMap. It assigns the processors to be H-workers and V-workers, respectively and keeps book about the assignments of the global H-slices and V-columns to these workers. This includes the number of H-slices and V-columns assigned to each worker, their global and local id, as well as the owner relationship to each H-slice and V-column.

Most global variables are set in this routine: the number of H and V-workers, the size of the local arrays at the workers as well as mapping variables. The provided routine can serve as an example on how to set up a different mapping. We have used the following global mapping variables:

`owner_of_hslice(1..Nz)`	specifies the process that gets the H-slice
`owner_of_vcol(1..Nx·Ny)`	specifies the process that gets the V-column
`no_of_hslices(p)`	number of H-slices owned by process p where $p \in \{1, \ldots, \texttt{NHworkers}\}$
`no_of_vcols(p)`	number of V-columns owned by process p where $p \in \{1, \ldots, \texttt{NVworkers}\}$
`local_hslice_id(1..Nz)`	local index of the global H-slice (index seen by the process which owns it)
`local_vcol_id(1..Nx·Ny)`	local index of the global V-column (index seen by the process which owns it)
`global_hslice_id(p,1..Nzloc)`	global index of a local H-slice on processor p
`global_vcol_id(p,1..Ncloc)`	global index of a local V-column on processor p
`owned_hslices(p,1..no_of_hslices(p))`	global H-slices owned by worker p
`owned_vcols(p,1..no_of_vcols(p))`	global V-columns owned by worker p
`planar_vcol_id(1..Nx·Ny,1..2)`	global x and y index of V-column id
`linear_vcol_id(1..Nx,1..Ny)`	global V-column id for column at given position in the global array

Location

mpi_util.f
Module: `HVParallelDataMap`

See Also

CreateCommDataTypes for XY-partitioning (page 85)

CreateCommDataTypes

Creates the data types used by the library routines for any communication.

Syntax

```
CreateCommDataTypes(Nx,Ny,Nz,Nzloc,Ncloc,Ns)

  INTEGER Nx              !  global no. of grid points in x-dim
  INTEGER Ny              !  global no. of grid points in y-dim
  INTEGER Nz              !  global no. of grid points in z-dim
  INTEGER Nzloc           !  local no. of grid points in z-dim on H-workers
  INTEGER Ncloc           !  local no. of V-columns on V-workers
  INTEGER Ns              !  no. of chemical species to be monitored
```

Details

This routine creates the worker group used for communication during shuffling as well as the communication data types for each of the array types in the air quality model. There have to be slice-types and column-types to do the slice- and column-communication in distribution and gathering. Other data types created are the combined slices and columns for the single message distribution and gathering as well as the combined data to be sent during shuffling when using one message only. These data types represent parts of an array that can be located anywhere in the array specified by count numbers and displacements.

Most of these data types depend on the mapping function described above. The implementation can serve as an example on how to define these data types when using different mapping routines. Be aware that the combining of H-slices or V-columns to one data type for single message communication may not be possible if the mapping is irregular! In that case only the first version of communication routines can be used that distribute and gather the H-slices and V-columns slice by slice and column by column respectively.

Location

mpi_util.f
Module: `HVCommDataTypes`

See Also

CreateCommDataTypes for XY-partitioning (page 85)

B.4 HV Library

distrib_h_4D

Distribute a 4D array in H-slice partitioning from the master to the H-workers.

Syntax

```
distrib_h_4D_v[1|2|3](Nx,Ny,Nz,Nzloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global & local no. of grid points in x-dim
  INTEGER Ny                 !  global & local no. of grid points in y-dim
  INTEGER Nz                 !  global no. of grid points in z-dim
  INTEGER Nzloc              !  local no. of grid points in z-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(Nx,Ny,Nzloc,Ns)  !  local 4D array on each H-worker
```

Details

These routines distribute a global 4D array at the master into H-slice partitioning to the H-workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Nz messages

v2 - distribution sending one message to each H-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_v_4D (page 99), distrib_x_4D (page 86)

distrib_v_4D

Distribute a 4D array in V-column partitioning from the master to the V-workers.

Syntax

```
distrib_v_4D_v[1|2|3](Nx,Ny,Nz,Ncloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in xy-dim
  INTEGER Ncloc              !  local no. of grid points in v-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(1,Ncloc,Nz,Ns)   !  local 4D array on each V-worker
```

Details

These routines distribute a global 4D array at the master into V-column partitioning to the V-workers. The following three versions are implemented:

v1 - distribution sending column by column using Nx·Ny messages

v2 - distribution sending one message to each V-worker combining the columns if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_h_4D (page 99), distrib_y_4D (page 86)

gather_h_4D

Gather a 4D array at the master from H-slice partitioning at the H-workers.

Syntax

```
gather_h_4D_v[1|2|3](Nx,Ny,Nz,Nzloc,Ns,aglob,aloc)

  INTEGER Nx                  !  global & local no. of grid points in x-dim
  INTEGER Ny                  !  global & local no. of grid points in y-dim
  INTEGER Nz                  !  global no. of grid points in z-dim
  INTEGER Nzloc               !  local no. of grid points in z-dim
  INTEGER Ns                  !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)     !  global 4D array
  REAL aloc(Nx,Ny,Nzloc,Ns)   !  local 4D array on each H-worker
```

Details

These routines gather the local 4D arrays at the H-workers into the global 4D array at the master. The following three versions are implemented:

v1 - master gathering, receiving slice by slice in Nz messages

v2 - master gathering, receiving one message from each H-worker combining the local slices if more than one

v3 - using two calls to `MPI_GATHERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

gather_v_4D (page 101), gather_x_4D (page 87)

---

gather_v_4D

Gather a 4D array at the master from V-column partitioning at the V-workers.

Syntax

```
gather_v_4D_v[1|2|3](Nx,Ny,Nz,Ncloc,Ns,aglob,aloc)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global & local no. of grid points in z-dim
  INTEGER Ncloc              !  local no. of grid points in xy-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aglob(Nx,Ny,Nz,Ns)    !  global 4D array
  REAL aloc(1,Ncloc,Nz,Ns)   !  local 4D array on each V-worker
```

Details

These routines gather the local 4D arrays at the V-workers into the global 4D array at the master. The following three versions are implemented:

v1 - master gathering, receiving column by column in Nx·Ny messages

v2 - master gathering, receiving one message from each V-worker combining the local slices if more than one

v3 - using two calls to `MPI_GATHERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

gather_h_4D (page 100), gather_y_4D (page 88)

---

shuffle_h2v_4D

Shuffle 4D arrays from h-partitioning on H-workers to v-partitioning on V-workers.

Syntax

```
shuffle_h2v_4D_v[1|2|3](Nx,Ny,Nz,Nzloc,Ncloc,Ns,aloch,alocv)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global no. of grid points in z-dim
  INTEGER Nzloc              !  local no. of grid points in z-dim
  INTEGER Ncloc              !  local no. of grid points in xy-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aloch(Nx,Ny,Nzloc,Ns) !  local 4D array on each H-worker
  REAL alocv(1,Ncloc,Nz,Ns)  !  local 4D array on each V-worker
```

Details

These routines rearrange the local 4D in h-partitioning at the H-workers to v-partitioning at the V-workers. The following three versions are implemented:

v1 - distribution using copying into a buffer and sending blocks, one message from each H-worker to each V-worker

v2 - distribution sending one message from each H-worker to each V-worker combining the necessary data

v3 - using copying and `MPI_ALLTOALL`

Note that the first and third routines are independent of the mapping while the second routine depends on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

shuffle_v2h_4D (page 102), shuffle_x2y_4D (page 88)

---

shuffle_v2h_4D

Shuffle 4D arrays from v-partitioning on V-workers to h-partitioning on H-workers.

Syntax

```
shuffle_v2h_4D_v[1|2|3](Nx,Ny,Nz,Nzloc,Ncloc,Ns,aloch,alocv)

  INTEGER Nx                 !  global no. of grid points in x-dim
  INTEGER Ny                 !  global no. of grid points in y-dim
  INTEGER Nz                 !  global no. of grid points in z-dim
  INTEGER Nzloc              !  local no. of grid points in h-dim
  INTEGER Ncloc              !  local no. of grid points in xy-dim
  INTEGER Ns                 !  local & global no. of chemical species dim
  REAL aloch(Nx,Ny,Nzloc,Ns) !  local 4D array on each H-worker
  REAL alocv(1,Ncloc,Nz,Ns)  !  local 4D array on each V-worker
```

Details

These routines rearrange the local 4D in v-partitioning at the V-workers to h-partitioning at the H-workers. The following three versions are implemented:

v1 - distribution using copying into a buffer and sending blocks, one message from each V-worker to each H-worker

v2 - distribution sending one message from each V-worker to each H-worker combining the necessary data

v3 - using copying and `MPI_ALLTOALL`

Note that the first and third routines are independent of the mapping while the second routine depends on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

shuffle_h2v_4D (page 101), shuffle_y2x_4D (page 89)

distrib_h_2D

Distribute a 2D array in H-slice partitioning from the master to the H-workers.

Syntax

```
distrib_h_2D_v[1|2|3](Nx,Ny,aglob)

  INTEGER Nx              !  global & local no. of grid points in x-dim
  INTEGER Ny              !  global & local no. of grid points in y-dim
  REAL aglob(Nx,Ny)       !  global & local 2D array
```

Details

This routine distributes a global 2D array at the master into H-slice partitioning to the H-workers using broadcast. Currently there are no 2D arrays that have to be distributed in H-slice form, but we provided an implementation.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_v_2D (page 103), distrib_x_2D (page 90)

distrib_v_2D

Distribute a 2D array in V-column partitioning from the master to the V-workers.

Syntax

```
distrib_v_2D_v[1|2|3](Nx,Ny,Ncloc,aglob,aloc)
```

```
INTEGER Nx              !  global no. of grid points in x-dim
INTEGER Ny              !  global no. of grid points in y-dim
INTEGER Ncloc           !  local no. of grid points in v-dim
REAL aglob(Nx,Ny)       !  global 2D array
REAL aloc(1,Ncloc)      !  local 2D array on each V-worker
```

Details

These routines distribute a global 2D array at the master into V-column partitioning to the V-workers. The following three versions are implemented:

v1 - distribution sending column by column using Nx·Ny messages

v2 - distribution sending one message to each V-worker combining the columns if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_h_2D (page 103), distrib_y_2D (page 90)

distrib_v_2DN

Distribute a 2DN array in V-column partitioning from the master to the V-workers.

Syntax

```
distrib_v_2DN_v[1|2|3](Nx,Ny,Ncloc,Ns,aglob,aloc)
```

```
INTEGER Nx              !  global no. of grid points in x-dim
INTEGER Ny              !  global no. of grid points in y-dim
INTEGER Ncloc           !  local no. of grid points in v-dim
INTEGER Ns              !  local & global no. of chemical species dim
REAL aglob(Nx,Ny,Ns)    !  global 2DN array
REAL aloc(1,Ncloc,Ns)   !  local 2DN array on each V-worker
```

Details

These routines distribute a global 2DN array at the master into V-column partitioning to the V-workers. A 2DN array is a 3D array in the dimensions x, y, and number of chemical species. The following three versions are implemented:

v1 - distribution sending column by column using Nx·Ny messages

v2 - distribution sending one message to each V-worker combining the columns if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_y_2DN (page 92)

distrib_h_3D

Distribute a 4D array in H-slice partitioning from the master to the H-workers.

Syntax

```
distrib_h_4D_v[1|2|3](Nx,Ny,Nz,Nzloc,aglob,aloc)

  INTEGER Nx                ! global & local no. of grid points in x-dim
  INTEGER Ny                ! global & local no. of grid points in y-dim
  INTEGER Nz                ! global no. of grid points in z-dim
  INTEGER Nzloc             ! local no. of grid points in z-dim
  REAL aglob(Nx,Ny,Nz)      ! global 3D array
  REAL aloc(Nx,Ny,Nzloc)    ! local 3D array on each H-worker
```

Details

These routines distribute a global 3D array at the master into H-slice partitioning to the H-workers. The following three versions are implemented:

v1 - distribution sending slice by slice using Nz messages

v2 - distribution sending one message to each H-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_v_3D (page 106), distrib_x_3D (page 92)

distrib_v_3D

Distribute a 3D array in V-column partitioning from the master to the V-workers.

Syntax

```
distrib_v_3D_v[1|2|3](Nx,Ny,Nz,Ncloc,aglob,aloc)
```

```
INTEGER Nx              !  global no. of grid points in x-dim
INTEGER Ny              !  global no. of grid points in y-dim
INTEGER Nz              !  global & local no. of grid points in xy-dim
INTEGER Ncloc           !  local no. of grid points in v-dim
REAL aglob(Nx,Ny,Nz)    !  global 3D array
REAL aloc(1,Ncloc,Nz)   !  local 3D array on each V-worker
```

Details

These routines distribute a global 3D array at the master into V-column partitioning to the V-workers. The following three versions are implemented:

v1 - distribution sending column by column using Nx·Ny messages

v2 - distribution sending one message to each V-worker combining the columns if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore ccan not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_h_3D (page 105), distrib_y_3D (page 93)

distrib_xh_BD

Distribute a BDx array in H-slice partitioning from the master to the H-workers.

Syntax

```
distrib_xh_BD_v[1|2|3](Nx,Nz,Nzloc,Ns,aglob,aloc)
```

```
INTEGER Nx                  !  global & local no. of grid points in x-dim
INTEGER Nz                  !  global no. of grid points in z-dim
INTEGER Nzloc               !  local no. of grid points in z-dim
INTEGER Ns                  !  local & global no. of chemical species dim
REAL aglob(Nx,Nz,2,Ns)      !  global BDx array
REAL aloc(Nx,Nzloc,2,Ns)    !  local BDx array on each H-worker
```

Details

These routines distribute a global BDx array at the master into H-slice partitioning to the H-workers. A BDx array is a 4D array in the dimensions x, z, 2, and number of chemical species that contains data about each chemical species on the east and west boundaries of the model. The following three versions are implemented:

v1 - distribution sending slice by slice using Nz messages

v2 - distribution sending one message to each H-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_yh_BD (page 107), distrib_x_BD (page 94)

distrib_yh_BD

Distribute a BDy array in H-slice partitioning from the master to the H-workers.

Syntax

```
distrib_yh_BD_v[1|2|3](Ny,Nz,Nzloc,Ns,aglob,aloc)
```

```
INTEGER Ny                  !  global & local no. of grid points in y-dim
INTEGER Nz                  !  global no. of grid points in z-dim
INTEGER Nzloc               !  local no. of grid points in z-dim
INTEGER Ns                  !  local & global no. of chemical species dim
REAL aglob(Ny,Nz,2,Ns)      !  global BDy array
REAL aloc(Ny,Nzloc,2,Ns)    !  local BDy array on each H-worker
```

Details

These routines distribute a global BDy array at the master into H-slice partitioning to the H-workers. A BDy array is a 4D array in the dimensions y, z, 2, and number of chemical species that contains data about each chemical species on the north and south boundaries of the model. The following three versions are implemented:

v1 - distribution sending slice by slice using Nz messages

v2 - distribution sending one message to each H-worker combining the slices if more than one

v3 - using two calls to `MPI_SCATTERV`

Note that the first routine is independent of the mapping while the second and third routines depend on the data types declared for a certain mapping in the function `CreateCommDatatypes` and therefore can not be used with any other mapping unless the communication structures have been reimplemented.

Location

`mpi_commlibrary.f`
Module: `HVCommunicationLibrary`

See Also

distrib_xh_BD (page 106), distrib_y_BD (page 94)

B.5 Abbreviations in the Appendix

In the following we explain a few abbreviations used in the appendix.

no.	- number
calcs	- calculations
dim	- dimension
dir	- direction
conc	- concentration
hor.	- horizontal
ver.	- vertical

Bibliography

[1] G. R. Carmichael. Private communication.

[2] G. R. Carmichael, A. Sandu, F. A. Potra, V. Damian-Iordache, and M. Damian-Iordache. The current state and the future directions in air quality modeling. *SAMS*, 25:75–105, 1996.

[3] G. R. Carmichael, A. Sandu, C. H. Song, S. HE, M. J. Phandis, D. Daescu, V. Damian-Iordache, and F. A. Potra. *Computational challenges of modeling interactions between aerosol and gas phase processes in large scale air pollution models*, pages 99–136. Kluwer Publishers, 1999.

[4] S.Y. Cho, G. Calori, and G.R. Carmichael. User's guide for air quality modeling system (AQuMS). Documentation for STEM program.

[5] D. Dabdub. Private communication.

[6] D. Dabdub and R. Manohar. Performance and portability of an air quality model. *Parallel Computing*, 23(14):2187–2200, 1997.

[7] H. Elbern. Parallelization and load balancing of a comprehensive atmospheric chemistry transport model. *Atm. Env.*, 31:3561–3574, 1997.

[8] H. Elbern. On the load balancing problem of comprehensive air quality models. *SAMS*, 32:31–56, 1998.

[9] T.M.R. Ellis, I. R. Philips, and T. M. Lahey. *FORTRAN 90 Programming.* Addison-Wesley, 1994.

[10] Center for Global and Regional Environmental Research at University of Iowa. Homepage. http://www.cgrer.uiowa.edu.

[11] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - The Complete Reference, Volume 2, The MPI Extensions*, volume 2. MIT Press, 1999.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin / Cummings, 1994.

[13] J. Michalakes. Reference manual for RSL.

[14] J. Michalakes. Rsl: A parallel runtime system library for regional atmospheric models with nesting.

[15] Office of Air Quality Planning and Standards. Homepage. http://www.epa.gov/oar/oaqps/.

[16] W. Owczarz and Z. Zlatev. Running a large-scale air pollution model on fast supercomputers. *Atmospheric Modelling*, to appear.

[17] P. S. Pacheco and Woo Chat Ming. MPI user guide in FORTRAN.

[18] A. Sandu. Running STEM-III with MPI on small Beowulf clusters. Private communication.

[19] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI - The Complete Reference, Volume 1, The MPI Core*, volume 1. MIT Press, Cambridge, MA, USA, second edition, 1999.

[20] A.S. Tomlin, S.Ghorai, G.Hart, and M.Berzins. 3-D adaptive unstructured meshes in air pollution modelling. *Environmental Management and Health*, volume 10, issue 4, 1999.

[21] B. VanVoorst and S. Seidel. Comparison of MPI implementations on a shared memory machine. *Lecture Notes in Computer Science*, 1586, 2000.