A HPX-based parallelization of a Navier-Stokes-solver

Lukas Troska

02. August 2016
Acknowledgment

I would like to express my gratitude to all the people who made it possible to complete this thesis. Firstly I thank Prof. Dr. Schweitzer for the extensive supervision as well as help and interesting discussions of questions and problems that arose during the writing. I also thank Prof. Dr. Griebel for the second opinion. Furthermore I want to express great appreciation to Patrick Diehl for the interesting and valuable conversations as well as continued support. Lastly I want to thank Alexander Rüttgers for providing a MPI implementation of the algorithm and Tobias Tersteegen for proofreading.
Abstract

The increase in computing power of modern high performance computing platforms and corresponding complexity in terms of programmability and utilization of available resources influences the design and implementation of numerical algorithms on these platforms. A novel approach under the name of High Performance ParalleX (HPX), a parallel C++11/14 compliant runtime system enabling fine-grained constraint-based parallelism as well as distributed operations, has been developed. To explore the employability of this system in the context of numerical simulations I present the implementation of a two-dimensional Navier-Stokes-solver and show the performance and scalability of its HPX-based parallelization.
Contents

1 Introduction 9
  1.1 German Summary .................................................. 11

2 The HPX runtime system 13
  2.1 Obstacles in the design of parallel applications ................. 13
  2.2 HPX design principles ............................................. 14
  2.3 Performance benchmarking ....................................... 16
  2.4 A parallel matrix vector multiplication in HPX ................. 18

3 The model problem - Navier-Stokes equations 23
  3.1 The mathematical setting .......................................... 23
  3.2 Discretization and algorithm ...................................... 24
  3.3 A parallel implementation in HPX ................................ 29

4 Performance benchmarking 33
  4.1 Preliminary considerations ........................................ 33
  4.2 Impact of data dependencies on scalability ..................... 39
  4.3 Flows in complex geometries ..................................... 51

5 Conclusion and outlook 77
  5.1 Conclusion .......................................................... 77
  5.2 Outlook ............................................................. 77

6 Appendix 79
  6.1 Code snippets ...................................................... 79

Bibliography 85
List of Figures

2.1 Dividing the multiplication into smaller tasks ........................................ 19
2.2 Strong scaling experiment for the matrix vector multiplication of a $18000 \times 18000$ matrix on an Intel(R) Xeon(R) E5-2620 v2 ................................................. 21
2.3 HPX version of the matrix vector multiplication with variable block size .... 22
3.1 Staggered grid ......................................................................................... 25
4.1 Specification of the Atacama cluster of the Institut für Numerische Simulation, Universität Bonn ................................................................. 33
4.2 Visualization of a solution for the driven cavity problem on a $128 \times 128$ grid at different time steps for Reynolds Number $Re = 1000$ ......................... 35
4.3 Comparison of scaling properties for the flow over a step for different Reynolds Numbers .................................................................................. 36
4.4 Comparison of scaling properties for the driven cavity problem for different numbers of time steps and solver iterations per time step ......................... 37
4.5 Comparison of HPX and MPI using the driven cavity problem .................. 39
4.6 Strong scaling experiment for the driven cavity problem using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster ............................................................ 42
4.7 Heat map for the idle rates of the driven cavity problem on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver ................................................................. 43
4.8 Strong scaling experiment for the driven cavity problem using a SOR solver for a fixed amount of time steps and solver iterations on the Atacama cluster ............................................................ 45
4.9 Heat map for the idle rates of the driven cavity problem on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a SOR solver ............................................................ 46
4.10 Average idle rates of the blockwise Jacobi and SOR method .................... 47
4.11 Ratio of the runtimes of the blockwise Jacobi and SOR method ................. 48
4.12 Ratio of the runtimes of the solvers when waiting for the residual to be computed to the runtime when using a cancellation token .................................... 50
4.13 The domain for the flow over a step ....................................................... 52
4.14 Visualization of the solution for the flow over a step on a $512 \times 128$ grid at different time steps for Reynolds Number $Re = 100$ .......................... 52
4.15 Strong scaling experiment for the flow over a step using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster .......................................................... 54

4.16 Heat map for the idle rates of the flow over a step on a $40320 \times 10080$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver ......................................................... 55

4.17 The domains with uniform obstacles ................................................................. 57

4.18 Visualization of the solution for the flow with uniform obstacles on a $630 \times 630$ grid at different time steps for Reynolds Number $Re = 1000$ ......................................................... 58

4.19 Strong scaling experiment for the flow through spaced out uniform obstacles using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster .......................................................... 60

4.20 Heat map for the idle rates of the flow through spaced out uniform obstacles on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver ......................................................... 61

4.21 Strong scaling experiment for the flow through tightly grouped uniform obstacles using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster .......................................................... 62

4.22 Heat map for the idle rates of the flow through tightly grouped uniform obstacles on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver ......................................................... 63

4.23 The domain for the porous medium .................................................................. 65

4.24 Visualization of the solution for the porous medium on part of the domain at different time steps for Reynolds Number $Re = 1000$ ......................................................... 66

4.25 Strong scaling experiment for the porous medium using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster .......................................................... 68

4.26 Heat map for the idle rates of the flow through a porous medium on a $20160 \times 20160$ grid in the tenth time step on 16 nodes (256 processors total) using a blockwise Jacobi solver ......................................................... 69

4.27 Heat map for the idle rates of the flow through a porous medium on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver ......................................................... 70

4.28 Visualization of the solution for the flow through narrow channels on part of the domain at different time steps for Reynolds Number $Re = 1000$ ......................................................... 71

4.29 The domain with narrow channels .................................................................. 72

4.30 Strong scaling experiment for the narrow channels using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster .......................................................... 74

4.31 Heat map for the idle rates of the flow through narrow channels on a $20160 \times 20160$ grid in the tenth time step on 4 nodes (64 processors total) using a blockwise Jacobi solver ......................................................... 75
4.3 Heat map for the idle rates of the flow through narrow channels on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.

List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Intel(R) Xeon(R) E5-2620 v2 specification</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Results for the parallel overheads for the matrix vector multiplication</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>AMD Phenom II X4 960T specification</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Parallel overheads for the finite difference method</td>
<td>34</td>
</tr>
</tbody>
</table>

List of Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Serial version of the matrix vector multiplication</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>&quot;Computing a matrix vector product using MPI&quot;</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>&quot;Barebone structure for HPX components representing a block of rows of a matrix&quot;</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>&quot;HPX action representing the matrix vector multiplication&quot;</td>
<td>81</td>
</tr>
<tr>
<td>6.4</td>
<td>&quot;Creating the HPX dependency tree for the matrix vector multiplication for one block&quot;</td>
<td>82</td>
</tr>
<tr>
<td>6.5</td>
<td>&quot;Sending and receiving data to and from the neighboring locality to the left using HPX&quot;</td>
<td>83</td>
</tr>
<tr>
<td>6.6</td>
<td>&quot;Asynchronously executing SOR for a single block using HPX&quot;</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Algorithms

3.1 Splitting algorithm in time step $n$ ........................................ 28
1 Introduction

In modern High Performance Computing (HPC) software development, developers are faced with problems concerning programmability and portability of software on large scale, sometimes heavily heterogeneous systems. The systems available for computation are getting larger and larger. Examples are the “JUQUEEN” supercomputer at the Jülich Supercomputing Centre\(^1\) with 458752 cores or “Titan” at Oak Ridge National Laboratory\(^2\) with 16688 CPUs with 16 cores each. The efficient implementation of numerical algorithms on these machines with traditional means such as the various versions of the Message Passing Interface such as MPI or OpenMPI requires a lot of expertise and knowledge about the problem domain. In addition, the rapidly increasing complexity in these platforms make portability of applications a concern.

The High Performance ParalleX (HPX) C++ runtime system tries to provide a novel approach to tackle these problems. It aims at a parallelization of algorithms through constraint-based synchronization in contrast to the more traditional fork-join parallelism. The use of executors and corresponding policies allows the system to hide implementation details such as thread scheduling. Since its programming model strictly conforms to the C++11/14 Standard, code written with the help of HPX is highly portable.

In this thesis I analyze the applicability and employability of the HPX runtime system in the context of numerical algorithms. To this end I implemented a two-dimensional Navier-Stokes-solver for incompressible fluids based on the algorithm presented in [GDN98], porting all the described concepts and parallelization techniques to use the facilities and tools provided by HPX while taking advantage of features like task stealing to enable localized work balancing.

In Chapter 2 the HPX runtime system is described in more detail. An overview of the challenges in designing parallel applications as well as the features provided by HPX to tackle these problems is given. In addition, two simple parallel matrix vector codes written with the use of MPI and HPX are compared in terms of their performance. Chapter 3 describes the mathematical setting of the Navier-Stokes model problem. The discretization using a finite differences approach on a staggered mesh is introduced. Next an overview of the implementation of this algorithm using the parallel facilities of the HPX runtime system is given. In Chapter 4 the setting for the analysis of the application in terms of performance

---

1\[^1\]http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Configuration/
2\[^2\]https://www.olcf.ornl.gov/titan/
on a high-performance cluster is given. In addition, a series of experiments with different geometries and simulation parameters is conducted to give insight into the scalability of the application.
1.1 German Summary


Häufig werden numerische Algorithmen durch das sogenannte “fork-join” Modell parallelisiert: Im ersten Schritt wird die Arbeit auf die Prozessoren aufgeteilt (zum Beispiel die Berechnung in einer for-Schleife). Danach werden alle Prozessoren synchronisiert, in dem am Ende der Schleife auf die Beendigung der Berechnung auf allen Knoten gewartet wird (“globale Barriere”).

Im Gegensatz hierzu verwendet das HPX C++-Laufzeitsystem ein Arbeitspakete basiertes Parallelisierungsmodell, in dem die Synchronisierung durch explizite Angabe der Datenabhängigkeiten geschieht. Ein Arbeitspaket ist hierbei eine Menge an auszuführenden Operationen, zum Beispiel die ersten zehn Iterationen einer for-Schleife. Durch diese Datenabhängigkeit können Berechnungen sofort ausgeführt werden, sobald die nötigen Daten zur Verfügung stehen. Es muss also nicht wie im “fork-join” Modell darauf gewartet werden, dass alle Berechnungen des vorherigen Schrittes abgeschlossen wurden.


Sobald die Berechnung des Residuums ergibt, dass die gewünschte Genauigkeit erreicht wurde, wird das Token ausgelöst und der Löser abgebrochen. Mit dieser Methode ist es möglich, dass der Löser der Berechnung des Residuums weit voraus ist. Daher müssen gegebenenfalls die Daten auf den, dem Residuum entsprechenden Stand zurückgesetzt werden.

Um die Performanz des mit HPX parallelisierten Algorithmus zu testen, wurden „strong scaling“-Experimente ausgeführt. Hierbei wird der gesamte Arbeitsaufwand konstant gehalten (in diesem Fall die Gittergröße) während die Rechenleistung, das heißt die Anzahl der Prozessoren, variiert wird. Diese Tests wurden zusätzlich auf Geometrien mit unterschiedlichen Eigenschaften, wie zum Beispiel einem porösem Medium oder einem schachbrettartigen Kanalsystem, ausgeführt.

2 The HPX runtime system

HPX (High Performance ParalleX) is a general purpose C++ runtime system, developed by the STE||AR Group\(^1\). Its aim is to provide a unified programming model for parallel and distributed applications of any scale. In this chapter an overview of the design principles behind HPX and the problems it is trying to address is given. Additionally, I implemented a parallel matrix vector multiplication code using both HPX and the Message Passing Interface (MPI) standard, which is used to showcase some of the differences between the two programming philosophies.

2.1 Obstacles in the design of parallel applications

Writing efficient code that scales well with the size of the problem as well as the amount of processing power available poses certain challenges. On one hand parallel applications are often highly specialized for the target hardware system. Consequently, running such an application on a different system (for example a system that uses a heterogeneous setup or accelerators) requires expertise and potentially a lot of work. On the other hand, one needs to take into account a set of problems that are inherent to parallel computing which the developers of HPX describe by the acronym SLOW (Starvation, Latency, Overhead, Waiting). Starvation occurs if the amount of work that can be executed concurrently is insufficient for enabling proper usage of computing resources. The term latency describes the inherent delay of using remote resources or services, for example accessing remote data. The third factor potentially affecting efficiency is overhead. Managing the parallel execution and resources adds additional work to the problem that is not present in a sequential version and may possibly affect performance. The last difficulty to overcome is waiting for contention resolution. Contention describes conflicts over a shared resource, for example multiple processes writing data to memory might result in a delay if bandwidth limits are exceeded.

\(^1\)stellar-group.org
2.2 HPX design principles

This section gives a quick overview of the design principles behind HPX. For a more detailed introduction see [Kai+14] and [And+11].

HPX extends the C++11 and C++14 standards to provide tools that try to mitigate or overcome the problems described in Section 2.1.

One of the main concepts is constraint-based synchronization. In most parallel applications synchronization is done using "global barriers". A global barrier is a step in the execution of parallel code that all threads or processes have to reach in order for the execution to continue. This means that some (sometimes even all but one) of the threads have to wait for the remaining threads to finish executing before continuing the computation. An example is the implicitly inserted global barrier after each loop that has been parallelized in MPI. This can lead to underuse of processing power. A different approach to this kind of synchronization is the constraint-based synchronization, where parallel operations can be executed as soon as all its preconditions are met. For example in the implementation of a Finite Differences method, the computation for the value of a cell in the next time step can begin as soon as all values of the previous time step needed by the stencil are set correctly. It is for example not necessary to wait for the boundary conditions to be set everywhere before applying a stencil to the cells in the interior of the domain that do not depend on the boundary values.

HPX provides three main facilities to enable constraint-based synchronization. The programmer can associate an action to a global function or member function, which allows asynchronous execution as well as execution on a remote locality (for the purpose of this thesis, a single locality corresponds to a single compute node on a system). Actions have a special return type, a future. This allows the code that schedules the execution of an action to refer to the result of the computation. When the result of the computation is needed, it can be retrieved from the future. If the execution of the action already finished the future returns the result immediately (its state is "ready"), otherwise the thread requesting the data will be suspended until the result becomes available. The future functionality described above is already available in the C++11 standard. HPX adds to the standard by providing ways to compose futures with so called "future continuations". The two most commonly used functions are future::then and dataflow. With future::then an action can be attached to the future. This means that as soon as the future becomes ready, the action will execute (a-)synchronously and the future is passed to the action as an argument. The function dataflow takes an action and an arbitrary amount of arguments. If some of the arguments are futures, dataflow will execute the action as soon as all futures become ready, passing along all other arguments.

Executing code that utilizes the constraint-based synchronization in HPX can be understood as a two step process: First the programmer creates a (data) dependency tree of the execution by composing futures using future::then and dataflow. In the second step the HPX runtime system traverses this tree, scheduling and executing each task as soon as the required dependencies
2.2 HPX design principles

are met. This also allows for runtime adaption of the specifics of the execution such as on which locality the computation should take place or how to schedule the tasks.

These concepts also help combat the latency which is inherent to distributed applications. A certain amount of latency is unavoidable due to physical limits, for example when sending data over a network. Thus in addition to the attempts of minimizing latency (using for example hierarchical memory architectures, caches or low latency networking) the runtime tries to hide latency. This is accomplished by enabling the use of fine-grained parallelism using lightweight tasks. Since many delays come from waiting for a resource to complete its operation, HPX tries to use the idle time to do useful work. For this the concept of a HPX task was developed. A task represents a small amount of parallel work, such as the execution of an action. Since the footprint of a task is small, changing contexts between waiting for resources to become available and execution is a cheap operation which does not cause a big overhead (see [Kai+14] and [Gru+15]).

The HPX runtime system uses an underlying thread scheduler to map the large number of lightweight tasks to HPX worker threads. Note that HPX worker threads (in the following sometimes only “threads”) are not necessarily in one-to-one correspondence with hardware threads. HPX worker threads are built on top of operating system (OS) threads to enable cheap context switches between tasks, since an OS thread context switch is an expensive operation ([Tan92]). The higher level concept of a worker thread allows mapping one or multiple worker threads to each OS thread. Each of these worker threads is then assigned one or more queues from which the tasks are taken. The specifics of the scheduling (for example in what fashion tasks are distributed or how many queues of different execution priorities are assigned to each thread) can be specified by different scheduler types such as the “priority local policy”, where each thread is assigned one high priority queue from which the work is taken first. Additionally there is one low priority queue from which tasks are scheduled when no other work is available. A second scheduler variant is the “static scheduling policy”, where only one queue is maintained for all threads. This offers the advantage that threads can “steal” the work from other local threads (HPX worker threads that are mapped to hardware threads on the same system or NUMA-domain) once it has completed all the work enqueued in its own queues. The goal is to mitigate the effect of (local) work imbalances.

Another factor for the performance of a parallel application is data placement. For example in the MPI programming model the programmer is responsible for proper distribution. This means that the programmer needs to take into account workload balance, which can be a difficult endeavor. To this end HPX provides data distribution policies and data migration operations. These are complemented by performance counters, an introspective system for measuring performance characteristics such as the idle rate of each processor or various networking information.
2.3 Performance benchmarking

To evaluate the performance of a parallel program and assess its behavior on different systems (for example systems varying in computational power) and for other problem sizes, one needs to define what it means for code to be "efficient" or "well performing". In this Section some of the most commonly used tools will be presented and used to conduct a performance comparison of the two variants of the matrix vector multiplication of Section 2.4.

2.3.1 Runtime

Runtime is the quantity essential to the metrics in the following sections.

**Definition 2.3.1 (Runtime)**

The runtime of a program is the passed time from the beginning of the execution until termination measured in seconds. Denote by $t_n(p)$ the runtime of a problem of size $n$ (for example parametrized by amount of work or grid size) on $p$ processors. If the problem size is fixed or can be deduced from the context, the index $n$ will be omitted.

Often runtime is averaged over a number of runs of the program to mitigate the influence of outside factors such as fluctuations of runtime on systems with time-sharing operating system kernels like Linux or systems with a CPU that uses dynamic frequency scaling. In addition, the first few iterations of the execution are usually ignored to mitigate problems such as data movement to the cache. This allows measurement of comparable data.

2.3.2 Throughput

A first (and arguably the most fundamental) metric for the performance of parallel software is throughput.

**Definition 2.3.2 (Throughput)**

Let $n$ be the amount of work done by a program, $p$ the amount of processors. Then the throughput is defined as the amount of work done per unit time $W_n(p) = \frac{n}{t_n(p)}$.

For example for the matrix vector multiplication in Section 2.4, the amount of work corresponds to the bytes processed, i.e. bytes read and written. That means for a square matrix with $m$ rows, the amount of work will be $(m^2 + 2m) \cdot \text{sizeof(double)}$ (reading and processing the matrix $A$ and vector $x$, as well as writing to the vector $b$), where sizeof(double) represents the platform specific size of a double precision floating point number in bytes.
2.3.3 Speedup and efficiency

A useful quantity that allows evaluation of the behavior of the program as a function of the number of processors is the speedup.

Definition 2.3.3 (Speedup)
The ratio of sequential runtime to runtime on \( p \) processors of an individual problem with amount of work \( n \), \( S_n(p) = \frac{t_n(1)}{t_n(p)} \), is called the speedup.

Note: Often the fastest available serial version of the code is used to compute the sequential runtime \( t_n(1) \).

Tied to the speedup is another metric that measures the utilization of the available resources:

Definition 2.3.4 (Efficiency)
The efficiency of a problem on \( p \) processors with problem size \( n \) is defined as the ratio \( E_n(p) = \frac{S_n(p)}{p} \).

For example if the program achieves a speedup of 3 by using 4 processors \( (S_n(4) = 3) \), its efficiency will be \( E_n(4) = \frac{3}{4} \) which can be interpreted as “25% of the compute resources have not been used efficiently”. Note that it is possible to achieve efficiency greater than 100%. Then the speedup is called super-linear.

2.3.4 Weak and strong scaling

Two of the most fundamental experiments to evaluate parallel applications are weak and strong scaling tests. When analysing the strong scaling characteristics of an application, one looks at how the runtime varies when the problem size is kept fixed and the number of processors varies. This gives insight in how the overheads of the parallelization behave as a function of compute power. In a strong scaling test and a 100% efficient code (efficient defined as in Definition 2.3.4) one would expect the runtime to behave as \( \frac{1}{p} \) with \( p \) processors. From this one can define “ideal scaling” by using the runtime for one processor \( t(1) \) as the baseline and assuming 100% efficiency, such that the runtime for an ideally scaling code is \( t_{\text{ideal}}(p) = \frac{t(1)}{p} \).

On the other hand, in a weak scaling test one fixes the amount of work per processor while varying the number of processors. This test is used to determine if the parallel overheads vary with the same order as the problem size. Note that ideal weak scaling is characterized by a constant runtime.
2.4 A parallel matrix vector multiplication in HPX

For a first overview of some of the differences between HPX and MPI, two programs that solve the equation $Ax = b$, where $A \in \mathbb{R}^{n^2}$ and $x \in \mathbb{R}^n$ are given and $b \in \mathbb{R}^n$ needs to be computed, were implemented. The basic, naïve algorithm consists of a doubly nested loop, first over all rows of $A$ and then over all entries of that row. The first program parallelizes this algorithm using the functionality provided by HPX, the second code uses MPI. The following sections describe the data storage, computation and communication used in the two approaches.

2.4.1 MPI implementation

In the MPI variant, all data is stored in arrays of datatype `double`. By arranging the matrix in row-major format and interpreting a vector with $n$ entries as a matrix with $n$ rows and one column, each compute node holds a block of rows of the matrix $A$ and blocks of the vector $x$. For simplicity it is assumed that the number of rows of $A$ is evenly divisible by the number of nodes such that the rows can be evenly distributed onto the nodes (e.g. for 15 rows and 3 nodes, the node with rank 0 receives the first five rows of $A$ and the first five entries of $x$ and so forth). Using this data layout, the computational step starts with a call to “MPI_Allgather” sending the local block of the vector $x$ to all other nodes such that the whole vector is available for the computation. Then, instead of using a doubly nested loop over all rows of $A$ and all entries in that row, only the rows in the local slice of the matrix are considered. Finally, each node has to send the resulting block of entries of $b$ to the root node to assemble the final result.

The code in Listing 6.1 shows an excerpt of the computation of the program described above.

2.4.2 HPX implementation

HPX provides the concept of a component, which is a class that can be used to represent local and remote data. Components provide a unified syntax for data access such that remote data can be treated the same way as local data. This is realized by an Active Global Adressing Space (AGAS) ([Kai+14]), letting the programmer write code that can be used on local and remote data since both are treated the same way. In this setting, instead of storing a block of rows in an array like in Section 2.4.1, each block of rows will be represented by a component. An excerpt of the code can be seen in Listing 6.2. This way, calling the `get_data` function of a block component will return a future containing a pointer to the requested chunk of the block of rows. If the data is local, no copying will happen, if it is remote the runtime system will serialize the data and send it to the locality requesting the data. In either case the execution is asynchronous which allows further computation to be performed while the data is sent.
2.4 A parallel matrix vector multiplication in HPX

The function that does the actual computation (shown in Listing 6.3) can be tied to the future containing the data by using the dataflow functionality, which will start the execution as soon as the data is received. Figure 2.1 shows an example for four localities. In this example, the rows colored blue are stored on locality 0, the red rows on locality 1 and so on. On each of the localities, the local set of rows is further split into blocks of rows of a given size. This block size parameter as well as the number of blocks on each locality are crucial to the performance, as will be shown in Section 2.4.3. The vector \( x \) is always stored in total on the root locality and distributed to the other localities whenever needed. This way the computation is split into 4 smaller computations that can be executed independently. The futures representing the results of these computations are then used to update the corresponding right hand side as soon as they become ready. This can be seen in Listing 6.4.

2.4.3 Benchmarking the applications

This Section showcases the application of the concepts described in Section 2.3 to the matrix vector multiplication codes described in the previous section.

2.4.3.1 Parallel overheads

Parallelization of an application has an inherent overhead. This overhead can be measured by comparing the runtime of the parallel application running on one processor to the runtime of a serial reference implementation of the algorithm. In the context of a matrix vector multiplication, I considered a simple implementation using a nested for loop. The main part of the program can be seen in Listing 2.1. I then ran the reference code as well as the MPI and HPX versions of the algorithm on a machine with an Intel(R) Xeon(R) E5-2620 v2 processor. Its specifications can be found in Table 2.1. The results are depicted in Table 2.2. Here, the factor column shows the factor by which the runtime increases when using the parallelized
Table 2.1: Intel(R) Xeon(R) E5-2620 v2 specification

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>2.1 GHz (2.6 turbo)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cores</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Threads</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>15 MB SmartCache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Results for the parallel overheads for the matrix vector multiplication

versions. The table shows that a small overhead was measurable for both versions. These overheads however are only in the range of one to two percent. Additionally, they decrease as the amount of work increases due to the ratio of the time spent on actual computations (that is the looping over the data) to the time spent on managing the parallelization getting skewed heavily towards computational work.

Listing 2.1: Serial version of the matrix vector multiplication

```
std::vector<double> A(rows * columns);
std::vector<double> x(rows);
std::vector<double> r(rows);
double* A_ptr = &A[0];
for (std::size_t i = 0; i < rows; ++i)
{
    double sum = 0;
    for (std::size_t j = 0; j < columns; ++j)
        sum += *(A_ptr++) * x[j];
r[i] = sum;
}
```

2.4.3.2 Scaling behavior

The strong scaling experiment was performed on a machine with an Intel(R) Xeon(R) E5-2620 v2 processor (specifications in Figure 2.1) with a matrix of size 18000 × 18000. Since the
2.4 A parallel matrix vector multiplication in HPX

Figure 2.2: Strong scaling experiment for the matrix vector multiplication of a $18000 \times 18000$ matrix on an Intel(R) Xeon(R) E5-2620 v2

The difference between the serial runtimes of the HPX and MPI codes to the plain C++ implementation was shown to be very small in Section 2.4.3.1 the efficiency of each application is calculated using the respective runtime on one processor. Figure 2.2a shows the speedup of both version. In addition the ideal speedup, which is the speedup that assumes a constant efficiency of 100% was calculated. In this experiment both parallelizations exhibit a very similar scaling behavior. Figure 2.2b depicts the efficiencies calculated from the speedup of Figure 2.2a. Note that the difference in efficiency is at most two percent points. Thus for a simple application like a parallel matrix vector multiplication one can expect the code using the task and constraint-based approach to perform similarly to the traditional fork-join-based parallelization model.

2.4.3.3 Impact of task size on performance

The next benchmark was done on a consumer work station containing an AMD Phenom II X4 960T and 8 GB of RAM. Its specification can be found in Table 2.3. The code was run for a matrix of size $8192 \times 8192$, which results in around 70 million values that are being operated on. The HPX version was run with a variable block size (that is number of rows per block) in such a way that the total number of rows was fixed at 8192. As can be seen in Figure 2.3a, for small block sizes the overhead of creating and executing a large amount of tasks dominates the runtime. This is due to the fine-grained division into many blocks leading to short task durations which do not amortize the overheads effectively. On the other hand for block sizes that are too large, not all processing power can be used, since one block corresponds to one task being executed. If there are less tasks than threads, some of the threads will idle, which
The HPX runtime system

<table>
<thead>
<tr>
<th>Clock Frequency</th>
<th>3 GHz (3.4 turbo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>4</td>
</tr>
<tr>
<td>Cache/Core</td>
<td>64 KB L1</td>
</tr>
<tr>
<td></td>
<td>512 KB L2</td>
</tr>
<tr>
<td>Shared Cache</td>
<td>6 MB</td>
</tr>
</tbody>
</table>

Table 2.3: AMD Phenom II X4 960T specification

![Runtime on an AMD Phenom II X4 960T](image1)

![Average Idle Rate on an AMD Phenom II X4 960T](image2)

Figure 2.3: HPX version of the matrix vector multiplication with variable block size

(a) Runtime on an AMD Phenom II X4 960T  (b) Average Idle Rate on an AMD Phenom II X4 960T

can be seen in Figure 2.3b. These figures clearly show a correlation of low idle-rate (which means high utilization of compute resources) to a shorter runtime.
3 The model problem - Navier-Stokes equations

In this section I will describe the setting in which a special case of the Navier-Stokes equations will be tackled. In addition, a basic overview of the discretization and algorithm used to solve this system of differential equations is given. In the last section, my implementation of the algorithm using the HPX framework is presented. The code is based on the algorithm presented in [GDN98], which I refer to for a more detailed description.

3.1 The mathematical setting

The Navier-Stokes equations are a system of equations used to describe the flow of a fluid in a domain $\Omega \subset \mathbb{R}^d$ for $d \in \{2, 3\}$ for time points $t \in [0, t_{\text{end}}]$. I will restrict myself to the case of incompressible fluids, that is fluids where the density is independent of the pressure. An example are fluids that have a constant density throughout the region $\Omega$,

$$\rho(x) = \rho_\infty = \text{const} \quad \forall x \in \Omega. \quad (3.1)$$

Then the flow can be represented by a set of partial differential equations:

**Definition 3.1.1 (Navier-Stokes Equations)**

Assume the fluid to be incompressible. Let $u : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}^d$ describe the velocity field and $p : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}$ the pressure of the fluid. In addition let $g : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}^d$ describe the body forces in the region. Then the system

$$\frac{\partial}{\partial t} u + (u \cdot \nabla) u + \nabla p = \frac{1}{Re} \Delta u + g \quad \text{in } \Omega \quad (3.2)$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega \quad (3.3)$$

is called the Navier-Stokes equations. The quantity $Re$ is called the Reynolds Number of the flow. It describes the ratio of inertial forces to viscous forces. Equation (3.2) is called the momentum equation and Equation (3.3) is referred to as the continuity equation.

**Note:** To be able to compute meaningful solutions from the Navier-Stokes equations one generally imposes initial as well as boundary conditions along the fixed boundary $\partial \Omega$, which will be described in more detail in Section 3.2.1.
3.1.1 Visualization

A helpful tool for the visualization of a solution in the two-dimensional case (which will be the focus of this work) is the so-called “stream function” $\psi$, which is defined by

$$\frac{\partial \psi(x, y)}{\partial x} := -v, \quad \frac{\partial \psi(x, y)}{\partial y} := u.$$ 

$\psi$ is well defined due to an integrability condition which follows directly from the continuity equation (3.3). The contour lines of the stream function are referred to as streamlines. Streamlines are curves that are tangential to the velocity field at each point. The proofs of these facts can be found in [GDN98].

3.2 Discretization and algorithm

To simplify the description of the discretization and to keep the implementation of the algorithm manageable, the special case of a two-dimensional, rectangular domain $\Omega = [0, a] \times [0, b] \subset \mathbb{R}^2$ is assumed. As will be seen later, this is not a substantial restriction as it still allows for complex domains using obstacle cells.

3.2.1 Discretization

3.2.1.1 Geometry

To allow the treatment of equations such as the Navier-Stokes equations in Definition 3.1.1 in a computer, a discrete version of the model has to be defined. Since a computer can only work with finite precision, a substitution for the derivatives has to be used. Also the domain cannot be resolved by a finite amount of points, thus an approximation has to be employed. To this end the finite difference method is used:

The basic idea behind the finite difference method is to define a grid on the domain and then replace the derivatives by difference quotients. More formally let $\Omega = [0, a] \times [0, b]$ be a rectangular domain. Now $\Omega$ is divided into a grid of $i_{\text{max}}$ cells in the $x$ direction and $j_{\text{max}}$ cells in the $y$ direction. Additionally, let $\delta x := \frac{a}{i_{\text{max}}}$, $\delta y := \frac{b}{j_{\text{max}}}$. $\delta x$ is the distance between grid lines parallel to the $y$-axis and $\delta y$ the distance between grid lines parallel to the $x$-axis.

Let $u$ be the component of the velocity field in the $x$ direction, $v$ the component in the $y$ direction. Then the $u$ will be computed in the centers of the right cell edges, $v$ in the centers of the upper cell edges and $p$ in the cell center. For example $u_{i,j}$ will be computed at the point $(i\delta x, (j - 0.5)\delta y)$. The values for the cell with indices $i$ and $j$ in a staggered grid can be seen in Fig. 3.1.
3.2 Discretization and algorithm

![Figure 3.1: Staggered grid](image)

The reason for a staggered grid is that in a grid where all three values are computed in the same point pressure oscillations can occur.

3.2.1.2 Derivatives

With this grid the first order spatial derivatives as well as the spatial derivatives in the diffusive terms \( \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 v}{\partial x^2}, \frac{\partial^2 v}{\partial y^2} \) can directly be replaced by finite difference quotients (in our case central differences) at the grid points. The spatial derivatives in the convective terms \( \frac{\partial (u^2)}{\partial x}, \frac{\partial (uv)}{\partial x}, \frac{\partial (uv)}{\partial y}, \frac{\partial (v^2)}{\partial y} \) however require values of the product that are not readily available with this grid. I followed [GDN98] and used a combination of a central difference quotient and donor cell discretization with a parameter to control the scheme.

Computing a difference quotient for a cell at the boundary may require values that lie outside of the domain. For example central difference for \( \frac{\partial u}{\partial x} \) at point \((i, j) = (0, 2)\) requires the values \( u_{-1,2}, u_{0,2}, u_{1,2}, \) where \( u_{-1,2} \) is not available. To solve this, an extra layer of cells around the domain is added. These “ghost cells” are additionally used to satisfy the boundary conditions in each time step. In this discretization I considered four different types of boundary conditions:

1. no-slip: The fluid is not allowed to move tangentially along the boundary as well as leave the domain through the boundary. The former is achieved by setting the tangential velocity component to 0. The latter condition implies that the normal derivative across the boundary is 0. This can be satisfied by discretizing the normal derivative using the ghost cells introduced above. For example at the right boundary the normal velocity component is \( u \). The normal derivative can be approximated by the difference quotient \( \frac{u_{i+1,j} - u_{i,j}}{dx} \). Requiring this term to be 0 yields \( u_{i+1,j} = u_{i,j} \).
2. free-slip: The fluid is not allowed to leave the domain through the boundary but has no frictional losses at the boundary, which means that the normal velocity component is set to 0 as well as the normal derivative of the tangential component. This is achieved similarly as above.

3. outflow: Both velocity components do not change value in normal direction to the boundary. This is again achieved similarly to above.

4. inflow: Both velocity components are set to a given value.

3.2.1.3 Complex geometries

To be able to deal with complex geometries, obstacle cells are introduced. Obstacle cells are cells in the interior of the domain at which no-slip conditions are imposed. That means the fluid is not allowed to flow through them but must instead pass around the obstacle. Cells on the interior that are not obstacle cells are called fluid cells (that is cells in which fluid is allowed to flow freely). For this implementation only obstacle cells which do not neighbor fluid cells on opposing sides are allowed to permit uniquely defined values for the obstacle cell. Thus each obstacle cells borders at most three fluid cells (namely across a corner of the cell). The velocity components of these cells are set analogously to the case of the no-slip boundary condition. For example for an obstacle cell that is adjacent to a fluid cell at the western and northern edge, the $u$ velocity component will be treated similarly to the eastern boundary strip and the $v$ velocity component similarly to the southern boundary strip.

3.2.2 Algorithm

With the spatial discretization of Section 3.2.1 the discrete version of the continuous Navier-Stokes equations (3.2) and (3.3) are

**Definition 3.2.1 (Discrete Navier-Stokes equations in component form)**

The discrete Navier-Stokes equations in component form on a staggered grid as outlined in 3.2.1 are given by

\[
\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x (3.4)
\]

\[
\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y (3.5)
\]

\[
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. (3.6)
\]
Note: [·] denotes the corresponding chosen finite difference quotient. Equation (3.4) has to be satisfied for all cells whose $u$-values lie in the interior of the domain, that is for all indices $(i, j)$ where $i = 1, \ldots, i_{\text{max}} - 1, j = 1, \ldots, j_{\text{max}}$. Similarly, Equation (3.5) has to be satisfied for all $v$-values in the interior of the domain. These can be found in the cells with index $(i, j)$, $i = 1, \ldots, i_{\text{max}}, j = 1, \ldots, j_{\text{max}} - 1$. The discrete continuity Equation (3.6) must be satisfied for all interior cells.

In the following equations the indices $i$ and $j$ will be omitted. All equations that are derived from each of the three equations (3.4), (3.5) and (3.6) are to be satisfied for the cells for which the original equation has to be satisfied.

To treat the remaining time derivative in the momentum equations, a forward difference quotient

\[ \frac{\partial u}{\partial t}(n+1) = \frac{u^{(n+1)} - u^{(n)}}{\delta t} \]  

(3.7)

is used. Here the exponent indicates the time step at which the function is to be evaluated. Substituting Equation (3.7) into equations (3.4) and (3.5) and solving for $u^{(n+1)}$ leads to

\[ u^{(n+1)} = u^{(n)} + \delta t\left( \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} + g_x \right) \]  

and

\[ v^{(n+1)} = v^{(n)} + \delta t\left( \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + g_y \right). \]  

(3.9)

Associating the pressure terms with time step $n + 1$ and the velocity terms with time step $n$ leads to a scheme that is implicit in the pressure and explicit in the velocities. To simplify notation, we collect all terms of Equation (3.8) that do not contain pressure terms into a variable $F$, and similarly all terms for $v$ from Equation (3.9) into a variable $G$:

\[ F^{(n)} := u^{(n)} + \delta t\left( \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} + g_x \right) \]  

(3.10)

\[ G^{(n)} := v^{(n)} + \delta t\left( \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + g_y \right). \]  

(3.11)

The equations (3.8) and (3.9) then read

\[ u^{(n+1)} = F^{(n)} - \delta t\frac{\partial p^{(n+1)}}{\delta x} \]  

(3.12)
and

\[ v^{(n+1)} = G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y}. \]  

(3.13)

Substituting (3.12) and (3.13) into the discrete continuity equation (3.6) leads to

**Definition 3.2.2 (Poisson equation for the pressure on a staggered grid)**

The Poisson equation for the pressure on a staggered grid is given by

\[ \frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\delta t} \left( \frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y} \right). \]  

(3.14)

For this equation a suitable solver has to be chosen. I implemented the method of successive over-relaxation (SOR) and a blockwise Jacobi method (see Section 3.3.3).

To be able to compute \( F^{(n)} \) and \( G^{(n)} \) in (3.12) and (3.13) as well as solve (3.14) for \( p^{(n+1)} \) in obstacle cells or cells bordering the boundary the corresponding values in these cells are set to the values of the neighboring fluid cell (or in the case of obstacle cells set to the average of all bordering fluid cells). This also means that the Poisson equation (3.14) only has to be solved in the fluid cells. Note that since the system matrix of (3.14) is singular due to the Neumann boundary conditions of the boundary value problem for the pressure, one can set the pressure in the boundary and obstacle cells like described above before each iteration to allow for a solution with physical values to exist (see [GDN98]).

Combining all ideas outlined above leads to the time stepping procedure in Algorithm 3.1.

**Algorithm 3.1** Splitting algorithm in time step \( n \)

- Set velocity values of boundary and obstacles.
- Compute \( F^{(n)} \) and \( G^{(n)} \) in fluid cells from equations (3.12) and (3.13).
- Compute the right hand side of the Poisson equation (3.14) in fluid cells.
- Solve the Poisson equation (3.14) in fluid cells.
- Update the velocity in fluid cells using equations (3.12) and (3.13).

Note that in order for this algorithm to be stable, \( \delta x, \delta y \) and \( \delta t \) must satisfy a Courant-Friedrichs-Lewy (CFL) condition.
3.3 A parallel implementation in HPX

In this section a high-level description of the steps required to implement the algorithm from Section 3.2.2 in the HPX runtime system is given. The actual code can be found on github\textsuperscript{1}.

3.3.1 Data structures

To enable parallelism the grid was divided into non overlapping tiles of the same size (in a regular rectangular pattern). Then the tiles are mapped to the localities available in the system in a one-to-one correspondence such that each locality only holds the data for one tile and is responsible for the computation on that part of the grid. The input of the geometry is done via a flag array that encodes the type of each cell and its neighboring cells using a bit field to allow for easy case distinction in the algorithm.

3.3.2 Communication

Since each locality holds only a part of the data, the outer layer of cells needs to be communicated with the neighboring localities. To achieve this an extra strip of ghost cells around the tile is added. To this end HPX provides an object called receive\_buffer, that allows the user to implement asynchronous communication of data. The buffer has member functions store and retrieve to handle the networking in an asynchronous way: If the data is retrieved from the buffer a future is returned. If the data was already stored into the buffer, this future is ready immediately and all continuations start executing. If not, the future will become ready as soon as the data is stored in the buffer. In Listing 6.5 the communication with the left neighbor is displayed. For sending data, the function takes in the current step, the variable (for example pressure or velocity) to be sent as well as a vector of futures that represent the results of the calculation of the data to be sent. As soon as all these computations are done (the future becomes ready) the data is sent. The receive function on the other hand will store the future that corresponds to receiving the data into a vector.

The advantage of using this approach is in the constraint-oriented programming philosophy of HPX described in Section 2.2: the buffer provides a futures that becomes ready as soon as the data from the other locality has been received. That means that each computation can start as soon as all its dependencies are met which eliminates the need to wait for all data to arrive, avoiding the need for a global barrier after each communication.

\textsuperscript{1}https://github.com/ltroska/nast_hpx
3.3.3 Solvers

To solve the Poisson equation for the pressure (3.14) the method of successive over-relaxation (SOR) and a blockwise Jacobi solver have been implemented. Since the pressure only has to be determined in fluid cells (boundary and obstacle cells are set to the average of the pressure values of all neighboring fluid cells), the solver only needs to be executed on these cells. Let

\[ F := \{(i, j) \in \{1, \ldots, i_{\text{max}}\} \times \{1, \ldots, j_{\text{max}}\} : \text{cell } (i, j) \text{ is a fluid cell}\} \]

denote the set of the indices of all fluids cells. Then the \( n \)-th iteration of the Jacobi method in time step \( m \) is given by

\[
p_{i,j}^{(n)} = \frac{(p_{i+1,j}^{(n-1)} + p_{i-1,j}^{(n-1)})(\Delta y)^2 + (p_{i,j+1}^{(n-1)} + p_{i,j-1}^{(n-1)})(\Delta x)^2 - (\Delta x \Delta y)^2 r_{i,j}}{2((\Delta x)^2 + (\Delta y)^2)} \quad \forall (i, j) \in F \tag{3.15}
\]

where \( p_{i,j}^{(0)} \) is the solution produced by the Jacobi method in time step \( m - 1 \) and \( r_{i,j} \) is the right hand side of the Poisson equation for the pressure (3.14) in time step \( m \).

With the same notation, the pressure in the \( n \)-th iteration of the SOR method in time step \( m \) is calculated by

\[
p_{i,j}^{(n)} = (1 - \omega)p_{i,j}^{(n-1)} + \omega \frac{(\Delta x \Delta y)^2}{2((\Delta x)^2 + (\Delta y)^2)} \left( \frac{p_{i+1,j}^{(n-1)} + p_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{p_{i,j+1}^{(n-1)} + p_{i,j-1}^{(n)}}{(\Delta y)^2} - r_{i,j} \right) \quad \forall (i, j) \in F \tag{3.16}
\]

where \( p_{i,j}^{(0)} \) is the solution produced by the SOR method in time step \( m - 1 \) and \( \omega \) is the relaxation parameter (which for this setting can be chosen as 1.7, see [GDN98]).

Both solvers have dependencies on the data that might stunt parallelism. In the Jacobi method the new value of a point can be computed as soon as all four neighboring values have been computed in the previous iteration. The SOR method can compute the new value of a point when the values of the left and bottom neighbor from the current iteration and the values of the top and right neighbor from the last iteration are available. Using the send and receive functions described in Section 3.3.2 the dependencies for both solvers can be represented by dataflow future continuations. An example for the SOR method can be seen in Listing 6.6. This results in a wavefront approach where each of the local grids starts the computation as soon as the computation of the neighboring node to the left and bottom is done. In contrast to the well known parallelization of the SOR method through a coloring scheme (such as Red-Black-SOR), the computed values equal the values computed by the serial version.

In addition both solvers have an implicit global barrier at the end of each iteration, namely testing if the residual is smaller than a given tolerance and then continuing or aborting,
depending on the result of the check. This might lead to some localities wasting processing power while waiting for other localities to finish, since work may not be evenly distributed (one locality might hold mostly obstacle cells, thus does not need to compute the solution of the Poisson equation on many fluid cells). To reduce this effect, the implemented solvers use a cancellation token. This cancellation token is implemented as a HPX component and allows localities to signal to other localities in which iteration the residual has dropped below the given tolerance, taking advantage of asynchronous execution of tasks. This way there is no need to wait for the residual to be computed, in essence synchronizing all localities, then checking the criterion for cancelling the solver before continuing to the next iteration. Instead, each iteration is started as soon as its dependencies are met (namely all neighboring localities have finished the last iteration) without waiting for the computation of the residual to complete. When the asynchronous task that computes the residual finishes, the cancellation token can be set, signalling all localities that the equation is solved to an acceptable accuracy. At this point, some localities might already be a few iterations ahead of the iteration to which the computed residual is associated, leading to a better utilization of compute resources.

3.3.4 Work balance

With a decomposition of a domain such as the one described in Section 3.2.1 and 3.3.1, imbalance in the amount of work that has to be performed on each node can arise. For example for the “flow over a step” problem (see Section 4.3.1), some nodes might hold a lot of obstacle cells, reducing the amount of work to be done drastically and thus having to wait for other nodes to finish computations before continuing with the next time step. This potentially leads to decreased scalability and bad performance.

In the model described in Section 3.2 the discretization and corresponding decomposition of cells into fluid, boundary and obstacle cells is static thus only needs to be performed once at the beginning of the simulation (unlike for example free boundary problems which necessitate a dynamic classification). This means that a good a-priori decomposition of the domain facilitates good performance and scalability. On the other hand this requires the user to perform said decomposition beforehand and individually for each domain.

HPX provides some tools that mitigate the impact of an uneven work distribution: Firstly, due to the task-based nature of the programming model the runtime system can perform task stealing. The runtime system maintains one or more queues for each HPX worker thread in which the tasks that are to be executed on that thread are inserted. These queues can be assigned different priorities, such that tasks in high priority queues are executed before other work is done. Each worker thread then takes a task from the corresponding queues. If the queues are empty, a worker thread can steal tasks from the queues of other worker threads that are on the same locality. This thread scheduling and stealing behavior can be specified by the user via thread scheduling policies, such as the default priority local scheduling policy or the static scheduling policy. In the priority local scheduling policy, one queue per operating
system thread as well as a total of one low priority thread is maintained. Each worker thread
pulls its work from the corresponding queue. Once these queues are empty, work is taken from
the low priority queue. The static scheduling policy also maintains one queue per operating
system thread, but the tasks are distributed using a round robin method and thread stealing
is disabled. Through the different choices of scheduling policies the work balancing can be
influenced locally.

If the load should be balanced globally, HPX provides a way to move data and corresponding
work. The method `components::migrate` allows HPX components to be moved to a different
locality. The advantage of this approach lies in the use of the global address space: migrating
a component does not change its global address such that the only addition to the code is a
call to `components::migrate`. All other functions using the migrated component such as functions
scheduling work to be performed or accessing the data do not have to be changed. On the
other hand one has to define a set of rules that choose a time point at which to migrate, as
well as the component to move and the target locality of the process.
4 Performance benchmarking

4.1 Preliminary considerations

To evaluate the performance of the HPX implementation several different experiments and comparisons have been conducted. All measurements were done on the Atacama cluster of the Institute for Numerical Simulation at the Rheinische Friedrich-Wilhelms-Universität Bonn. The specification and more details can be found in Figure 4.1 or on the website of the institute\(^1\). Note that in the Figures 4.13, 4.17, 4.23 and 4.29 depicting the layouts of the geometries, black pixels represent obstacle and boundary cells whereas white pixels represent fluid cells. In Figures 4.6, 4.15, 4.19, 4.21, 4.25 and 4.30 the graphs correspond to different grid sizes with varying amounts of local blocks. For example \(10080 \times 2520 (6 \times 6)\) denotes a grid that is 10080 cells wide and 2520 cells high where each node divides its local grid into 6 by 6 partitions of equal size.

To be able to draw conclusions from metrics such as speedup and efficiency one usually uses the fastest serial version of the algorithm as a baseline. Since running a parallelized implementation on one processor might not necessarily be the best performing implementation I measured the parallelization overhead of my implementation against the reference serial implementation NaSt2D\(^2\) developed by the research group of Prof. Dr. M. Griebel. Both

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>Dell PowerEdge M620</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>Intel(R) Xeon(R) CPU E5-2650 v2</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
</tr>
<tr>
<td>Hardware Threads</td>
<td>16</td>
</tr>
<tr>
<td>Cache</td>
<td>20 MB (SmartCache)</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>64 GB</td>
</tr>
<tr>
<td>Cluster Interconnect</td>
<td>Infiniband 56 Gb/sec (4X FDR)</td>
</tr>
</tbody>
</table>

**Figure 4.1:** Specification of the Atacama cluster of the Institut für Numerische Simulation, Universität Bonn

\(^1\)http://wissrech.ins.uni-bonn.de/research/himalaya/hardware.html  
\(^2\)http://wissrech.ins.uni-bonn.de/research/projects/NaSt2D/index.html
use the algorithm outlined in [GDN98]. For this comparison the driven cavity problem was considered.

The driven cavity problem (or lid-driven cavity problem) is a classical problem used to test and validate solvers for the Navier-Stokes equations. It is a simple two-dimensional problem that consists of a square domain and no-slip boundary conditions on three sides (Section 3.2.1). The boundary condition on the fourth side is set up to move the fluid in one direction along the boundary with a constant velocity and to not allow any fluid to penetrate the boundary, which means the tangential velocity component is set to a constant value and the normal component to 0. In this example, the top boundary will be moved to the right with a constant velocity of 1 unit, that is \( u(x) = 1 \) \( \forall x \in \Gamma_T \) where \( \Gamma_T \) denotes the top boundary of the domain. In the case of a staggered grid, this value can not be stored directly on the grid. Thus the \( u \)-value of the cell above the boundary (the “ghost cell”) is set in such a way that the average with the cell below the boundary is 1, since it lies directly on the boundary. Figure 4.2 shows the velocity field as well as the contour of the stream function (the stream lines) at various time steps for an experiment with Reynolds Number \( Re = 1000 \). One can clearly see the two characteristic eddies forming in the bottom corners. The solution shown for \( t = 25 \text{s} \) is an approximation to the steady state solution of the system.

The parallel overheads for different grid sizes for this problem can be found in Table 4.1. The factor describes the ratio of the runtime of the parallelized version to the runtime of the reference implementation. Consequently, if this factor is larger than 1 the serial reference implementation is faster and all measurements done in the following need to be discounted by that factor. The runtime is measured using the `time` command line tool.

Table 4.1 shows the runtimes as well as runtime ratios for different grid sizes for 10 time steps with 100 solver iterations each. The parallelization causes a slowdown of around 5% when using one processor. This means that if in the following the code is said to be 50% efficient one would need to take into account the slowdown compared to the fastest available code, leading to an actual efficiency of \( \frac{1}{1.05} \cdot 50\% \approx 47.6\% \). This difference is small such that all values will be given “as is” and not as the discounted values.

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>NaSt2D Runtime</th>
<th>HPX Runtime</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>3360 × 3360</td>
<td>167.656</td>
<td>177.154</td>
<td>1.0567</td>
</tr>
<tr>
<td>6720 × 6720</td>
<td>654.498</td>
<td>678.735</td>
<td>1.037</td>
</tr>
<tr>
<td>13440 × 13440</td>
<td>2501.098</td>
<td>2712.663</td>
<td>1.0846</td>
</tr>
</tbody>
</table>

Table 4.1: Parallel overheads for the finite difference method
4.1 Preliminary considerations

(a) Time step $t = 2s$

(b) Time step $t = 7s$

(c) Time step $t = 25s$

(d) Time step $t = 25s$, velocity field

**Figure 4.2:** Visualization of a solution for the driven cavity problem on a $128 \times 128$ grid at different time steps for Reynolds Number $Re = 1000$
The Reynolds Number might influence the locality of the work of the solver and therefore the performance of the program. Preliminary strong scaling tests for the flow over a step in Section 4.3.1 with varying Reynolds Numbers were run. The runtimes for one time step with 1000 solver iterations for the flow over a step problem are compared in Figure 4.3. There was no significant difference in the overall scaling behavior of the application with some exceptions in the step from 16 to 32 processors. This effect is due to variance in network communication, since the experiment on 16 processors is fully local and the experiment on 32 processors involves 2 nodes with one bidirectional data exchange step between them.

Figure 4.3: Comparison of scaling properties for the flow over a step for different Reynolds Numbers

One property of the discretization that has to be considered when conducting these experiments is the number of iterations the solver needs to reach an acceptable accuracy. When starting with randomly initialized (or initialized with 0 as is the case here) values for the velocities the velocity field is not divergence free. In this case a large number of iterations are needed to solve the Poisson equation for the pressure (3.14) to the desired accuracy. On the contrary once the system is (nearly) divergence free only a small number of iterations are needed to smooth the error that now mostly consists of high frequencies. This effect has to be taken into account since it could influence the scaling properties of the application.

Figure 4.4 shows the result of three runs of the application for each grid size: In the first run the parameters were chosen such that 2 time steps are performed and the solver in each time step aborts after 1000 iterations. The second run has a similar setup with 5 time steps with 400
4.1 Preliminary considerations

solver iterations each. The third experiment executes 50 time steps with 40 iterations each. As in the experiment concerning the Reynolds Number only small differences in the efficiency and idle rates can be observed. The differences for 32 processors again come from networking.

These results suggest that it is reasonable to perform the strong scaling tests without taking the outlined effects into account.

![Comparison of scaling properties for the driven cavity problem for different numbers of time steps and solver iterations per time step](image)

**Figure 4.4:** Comparison of scaling properties for the driven cavity problem for different numbers of time steps and solver iterations per time step
4.1.1 Comparing HPX and MPI using the driven cavity problem

The driven cavity problem will be used for comparison due to limitations of the available MPI implementation. Comparisons with implementations for the three-dimensional case are not reasonable and fair due to the fact that extra cell layers in the $z$ coordinate are needed to be able to apply the difference quotients or stencils, which leads to skewed runtimes and scaling results.

The strong scaling tests were carried out using three different grid sizes while running for 10 time steps with 1000 solver iterations each. The MPI variant of the algorithm uses a blockwise Jacobi on the global level while employing the SOR method on the local grids. On the other hand the HPX counterpart uses a data dependency-based form of the SOR algorithm that is fully equivalent to the serial case (see Section 4.2). Figure 4.5 shows that the absolute runtimes of the MPI implementation are significantly smaller than the corresponding runtimes of the HPX version. This is due to the fact that the latter uses an approach based on flag fields to differentiate between boundary, obstacle and fluid cells while the former simply adjust the range of the local loop indices. This also means that no comparison can be made on complex geometries such as a porous medium in Section 4.3. In addition there are no data points for more than 256 processors for the MPI implementation since no reasonable output could be obtained.

The speedup and efficiency plots show that even though the version of the SOR algorithm defined through data dependencies and constraint-based synchronization might impose restrictions on the ability of the HPX runtime system to create work for each processor, a generally higher efficiency was achieved. The implemented HPX version is fully equivalent to the sequential SOR method. On the other hand the MPI variant uses a blockwise SOR method (using the SOR method locally without interdependencies between nodes). It exhibited an efficiency of approximately 20% for 556 processors and was outperformed by the task-based approach where 28% efficiency was observed in the same setting. This suggests that expanding the MPI version to use a flag field approach allowing for complex geometries should not yield a better performing code than the HPX version in Section 4.3.
4.2 Impact of data dependencies on scalability

As mentioned in Section 2.2 it is beneficial for the performance of an application to avoid global barriers and interdependencies of data which hinder concurrency. To assess this claim two solvers for the Poisson equation for the pressure (3.14), namely a blockwise Jacobi solver and the successive over-relaxation (SOR) method have been implemented. In terms of data dependencies these solvers are very different. In each iteration the Jacobi method only requires that the data of the previous iteration has been computed. The SOR method additionally depends on the data of the left and bottom neighbor (when using a lexicographical ordering from bottom left to top right) of the current iteration. As described in 3.3.3 this leads to a wave front approach when implementing the SOR solver using a constraint-based parallelization.

**Figure 4.5:** Comparison of HPX and MPI using the driven cavity problem
4.2.1 Comparing the blockwise Jacobi and SOR method

To evaluate the impact of these data dependencies on the scalability of the application, the driven cavity problem has been tackled using both the Jacobi and SOR solver. Note that this experiment cannot be used to and is not meant to make any assertions about which solver is better in terms of accuracy per parallel time step. This is due to the fact that the SOR method (as a variant of the Gauß-Seidel method) generally converges faster than a blockwise Jacobi solver. If the efficiency of the former was 25% and the latter had an efficiency of 50% it would not be accurate to conclude that the Jacobi solver is “twice as good” as the SOR method, since it might also converge half as fast, leading to a practical efficiency that is roughly similar (since double the iterations are needed to achieve the same accuracy). To be able to make conclusions of that nature one needs to define finer metrics that take exactly these effects into account. These experiments, although interesting and potentially insightful, are not part of this thesis. Here I am solely interested in the effects different data dependencies have on the performance of the application (to be able to compare a constraint-based approach in HPX to approaches based on global barriers such as a parallelization in MPI) and chose these solvers only for their different characteristics in that context.

4.2.1.1 Using the blockwise Jacobi solver

Figure 4.6 shows the results of a strong scaling experiment for the driven cavity problem on the Atacama cluster. For runs that involve at most 16 processors, the simulation can be run on a single node leading to good data locality and removing the need to exchange the values of boundary (ghost) cells with neighboring nodes. Consequently an efficiency of at least 80% for up to 8 processors was observed. For the smaller grids of sizes $3360 \times 3360$ and $6720 \times 6720$ the efficiency drops rapidly as the number of processors increases. One reason is that both grids are not large enough to allow for enough work for each processor when the work has to be distributed evenly across a large number of nodes. This prevents the runtime system from being able to hide the costs associated with the overhead of parallelization. In addition, the costs of communicating with neighboring nodes for the $6720 \times 6720$ grid is twice as high as for the smallest grid: For more than 64 processors the time spent on networking tasks starts to dominate the duration of the computational tasks. The average idle rates for the $6720 \times 6720$ grid reflect that. The idle rate quickly rises above 50% for 16 and above 60% for more than 32 processors. The graphs for the grid with $6720 \times 6720$ cells and $8 \times 8$ local blocks show that when more tasks (64) than threads available (16) are scheduled, this effect can be mitigated by allowing the runtime system to interlace computation and communication. Finally, for both grids the application stops scaling well for more than 256 processors. The biggest grid that fits into the memory of a single node was a grid of size $20160 \times 20160$. Since a grid of this size is large enough to allow different numbers of local blocks, three simulations were run. They exhibit good scaling up to 556 processors, stabilizing at an efficiency of approximately 45%.
It can also be seen that they differ slightly in the absolute runtime and idle rate. One reason is that with \(8 \times 8 = 64\) or \(16 \times 16 = 256\) local blocks the HPX thread scheduler can allow for (local) task stealing to occur (as described in Section 3.3). When there are \(4 \times 4 = 16\) local blocks only 16 HPX tasks will be spawned. Since 16 is also the number of threads on each node, each thread queue will be assigned exactly one task. Therefore after completing the task no stealing can take place and the thread must wait for the other threads to complete the computation. This is reflected in a higher average idle rate of around 40\% for 556 processors for the \(4 \times 4\) local block case versus an average idle rate of approximately 15\% for 556 and \(8 \times 8\) local blocks. This local load balancing phenomenon can also be observed for the grid of size \(6720 \times 6720\). The measurements furthermore show, that the application is memory bound: As the grid size and therefore the amount of work increases, the code can be efficiently run on more processors, with larger grids generally allowing for better efficiency. Consequently the amount of work (which is proportional to the number of fluid cells) greatly influences the scaling behaviour of the application.

Additionally this Figure shows how parallel overheads can dominate the computational work for example by scheduling too many fine-grained tasks. The graph for the \(13440 \times 13440\) grid with \(8 \times 8\) local partitions shows a significant drop in speedup and increase in idle rate when going from 400 to 576 processors. On the other hand the same grid with only \(4 \times 4\) local partitions does not exhibit the same behavior. This effect is due to the fact that each local partition in a \(8 \times 8\) division contains only \(1/4\) of the fluid cells the partitions in a \(4 \times 4\) division have. Moreover since it also contains 4 times as many local partitions the costs of creating and scheduling each task is too large compared to the reduction in runtime through parallel execution. More time is spent on managing the parallelization than the actual computation itself leading to worse usage of compute resources and an increase in overall runtime. Therefore a decrease in speedup and efficiency can be observed. This effect was also seen in Section 2.4.3.2.

Figure 4.7 shows the average idle rates (averaged over all local processors) as well as the arrangement of each of the \(36\) nodes (576 processors total) in time step 10 of the simulation on a \(20160 \times 20160\) grid. One can see the even distribution of work across all localities. Note that the nodes assigned to the top left and top right most partition of the domain have lower idle rates, due to the fact that the propagation of non-zero pressure values starts in these corners. This means that the solver has to do more work in these partitions.
Figure 4.6: Strong scaling experiment for the driven cavity problem using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster.
4.2 Impact of data dependencies on scalability

Figure 4.7: Heat map for the idle rates of the driven cavity problem on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver
4.2.1.2 Using the SOR solver

The same experiment as in Section 4.2.1.1 has been run with the SOR solver instead of the Jacobi method. The results of the strong scaling experiment can be found in Figure 4.8. The scaling behavior is similar to the experiment using the blockwise Jacobi solver: The more work is available (that is the bigger the grid) the longer the application scales. For the smaller grids (such as the $3360 \times 3360$ and to a smaller degree also for $6720 \times 6720$) the time spent on the parallelization (for example due to network communication and task overheads) starts to dominate the actual computational work. In the case of the smallest grid the application stops scaling after 64 processors. It then stabilizes at a speedup of 20 with an average idle rate of 40%. The effect can then be observed after 400 processors, indicated by the idle rate increasing significantly as well as a drop in the speedup.

Note that the $20160 \times 20160$ grid with $16 \times 16$ local partitions has a significantly better efficiency and lower idle rate than the other variants of that grid. This is due to the fact that the amount of cells allows for a division into many partitions (and therefore tasks) without restricting the amount of work per task. Hence the surface of the “wave front” of blocks that are in the same iteration of the SOR algorithm (that comes from the interdependency of the values to be computed on values previously computed in the same iteration) is divided into enough partitions to be able to fully utilize each processor. Therefore a balance between allowing enough work for each partition and scheduling as many tasks as possible must be found to mitigate the negative effects of data dependencies. On the other hand one can clearly notice the effect the data interdependencies during each iteration of the SOR have on the overall performance. The efficiency is only 30% for the biggest grid and 400 processors and even drops to 27% for 576 processors. This is in contrast to the Jacobi solver with an efficiency of around 45%. This supports the claim that too many data dependencies and global barriers impact the concurrency and therefore scalability.
4.2 Impact of data dependencies on scalability

Figure 4.8: Strong scaling experiment for the driven cavity problem using a SOR solver for a fixed amount of time steps and solver iterations on the Atacama cluster
Figure 4.9: Heat map for the idle rates of the driven cavity problem on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a SOR solver.
4.2.1.3 Comparison of both solvers

Figure 4.11 shows the ratio of the best runtimes out of 10 runs of each solver on every grid. The SOR method can be seen to be generally worse than the Jacobi method (as mentioned in Section 4.2.1 in the context of the correlation between efficiency and data dependencies). Note that for the combination of number of processors and grid sizes where the efficiency using the Jacobi solver is around 40% or more it performs better than the SOR method. For example for the \(6720 \times 6720\) grid and the best runtime (using the local decomposition into \(8 \times 8\) partitions) the efficiency starts to drop significantly after the increase from 64 to 144 processors (Figure 4.6). That is also where the runtime ratio in Figure 4.11 starts to drop and move towards being in favor of the SOR method. This effect can analogously be observed for all other grid sizes. This can be explained by the faster degradation of the scaling behavior of the application when using the Jacobi method: Once it stops scaling in a regular manner, the speedup (and thus also the efficiency) starts to drop significantly. The runtime plot in Figure 4.6 indeed shows that this is due to the runtime stagnating or even increasing when increasing the number of processors due to network communications and parallel overheads overshadowing computational work. On the other hand the speedup when employing the SOR method degrades less due to the data dependencies increasing the amount of work, since in addition to the data of the previous iteration the nodes also need to communicate some of the data of the current iteration.

The idle rates in Figure 4.10 support this claim. Due to the data dependencies hindering concurrency the SOR method generally does not utilize all available computing resources well and therefore exhibits a higher average idle rate compared to the blockwise Jacobi method.

![Figure 4.10: Average idle rates of the blockwise Jacobi and SOR method](image)
Figure 4.11: Ratio of the runtimes of the blockwise Jacobi and SOR method
4.2 Impact of data dependencies on scalability

4.2.2 Computing the residual - a global barrier

As described in Section 3.3.3, waiting for the calculation of the residual after each iteration and only continuing the solving process after the accuracy has been tested against the given tolerance imposes a global barrier on the flow of execution. This restricts the concurrency of the algorithm since some localities might have to wait for other localities to finish computing the residual (they might have more fluid cells, which is where the residual has to be computed), then wait for the partial residuals of each node to be sent to a central node which does a summation and tests for an acceptable precision before issuing the solver to continue to work.

To combat this, a cancellation token has been implemented that can retroactively be cancelled. This way the solver can continue solving independently of and parallel to computing the residuals, already schedule all iterations up to a given maximum number while periodically checking if the token has been cancelled. If it has been cancelled, all upcoming iteration can be aborted and the data potentially has to be rolled back to the iteration in which the desired accuracy has been achieved.

Figure 4.12 depicts a comparison of the conventional approach to the cancellation token. The results were obtained by taking the best runtime out of 10 runs for each combination of grid size, solver and cancellation token/waiting for the residual. For small grid sizes the global barrier approach is only around 5% slower. This is due to the lack of available work such that a gain in concurrency does not translate to a higher use of compute resources and therefore decrease in runtime. Similarly when using a small number of processors there is almost no advantage to be gained by using a cancellation token for most grid sizes.

When it comes to the biggest grid, waiting for the residual is consistently around 15% slower. The advantage that can be achieved through eliminating the global barrier could potentially be improved by queueing the tasks that compute the residuals with a lower priority than the tasks that perform the actual computation in each iteration. By default they all have the same priority and since they are enqueued in an order that mirrors a sequential execution they are also pulled from the queue in that order.
Note: $t_w$ is the runtime when waiting for the residual, $t_m$ is the runtime when using a cancellation token.

Figure 4.12: Ratio of the runtimes of the solvers when waiting for the residual to be computed to the runtime when using a cancellation token.
4.3 Flows in complex geometries

Complex geometries can be modelled with the use of obstacle cells as described in Section 3.2.1.3. For the values in these cells to be uniquely definable the discretization has to be fine enough. This means that only obstacle cells that neighbor one fluid cell or two fluid cells sharing a corner are allowed, where neighboring cells are cells that share an edge (not a corner). These rules were checked and accounted for during the generation of each grid.

Due to the results of the previous Section 4.2 only the Jacobi method will be used to perform the scaling tests.

4.3.1 Flow over a step

The simulation of the flow of a fluid over a step is another well studied problem. For the purpose of showcasing the task-stealing capabilities of the HPX runtime system a second rectangular obstacle is added on the opposite to the step. This can be seen in Figure 4.13a. The boundary conditions are set such that the top half of the left boundary has an inflow condition with a constant velocity and the right boundary has an outflow condition (see Section 3.2.1). The different grid sizes are chosen such that the domain has an aspect ratio of 4 to 1 and allows for even distribution of the cells to up to 49 nodes with 1 to 16 processors each. The obstacles are one fourth of the width of the domain and half the height of the domain. Their distance also is half the height of the domain. This leads to a setup where some of the processors will have a local grid that consists mostly or even completely of obstacle cells. Since no computation has to be performed on pure obstacle cells (that is obstacle cells that are only adjacent to other obstacle cells), these processors may be (mostly) idling. The parameters were chosen such that the obstacle surface fraction (ratio of obstacle cells touching fluid cells including boundary cells to total number of cells) is approximately 1% and the obstacle volume fraction (ratio of all obstacle cells to total number of cells) is around 25%; the Reynolds Number is 100. In Figure 4.14 the solution for various time steps is displayed. As in the classical example an eddie starts forming behind the step. In this case it is small due to the obstacle on the opposite side restricting the flow. One can also observe a second, larger eddie that forms after the top obstacle.
Figure 4.13: The domain for the flow over a step

Figure 4.14: Visualization of the solution for the flow over a step on a $512 \times 128$ grid at different time steps for Reynolds Number $Re = 100$
Figure 4.15 shows the results of a strong scaling experiment on the “Atacama” cluster for grids of different sizes with the described characteristics. One can, similar to the measurements in the driven cavity setting in Section 4.2.1.1, observe the memory boundedness of the problem: the scaling depends critically on the amount of work available for each processor. Note that the total number of cells of each grid equals the number of cells of the corresponding square grids in Figure 4.6. However, the obstacle volume fraction for the non-square grids is approximately $\frac{1}{4}$. This means that the amount of work available is reduced by around 25% since computations only need to be performed on fluid and surface obstacle cells. Note that the grids have the same total amount of cells as in the driven cavity problem. The general scaling behavior, that is that a bigger grid scales up to more processors with a higher efficiency, is also present with this geometry. On the other hand the algorithm is not as efficient compared to Section 4.2.1.1 since some nodes contain a large number of pure obstacle cells on which no work has to be performed. The efficiency achieved on the largest grid was 38%.

On nodes that only contain some of the obstacle cells the task stealing of the HPX runtime system was able to balance the local work load. This can be seen in Figure 4.16. It shows the distribution of the domain onto the nodes as well as the average idle rate for each node in time step 10. Localities that contain only fluid cells can be seen to have low idle rates comparable to the driven cavity problem. Contrastingly, nodes that only contain obstacle cells (such as the node with index (1, 2)) lack enough work to be able to fully utilize the available compute resources. The node with index (2, 2) is an example of a node that consists of roughly 50% obstacle cells. The heatmap shows that the local task stealing system was able to balance the load between the different processors resulting in an average idle rate of 47.96%. This is consistent with the results of the driven cavity problem depicted in Figure 4.6. With the grid of size $13440 \times 13440$ in that experiment, each node roughly has the same amount of work as node (2, 2) in the biggest grid of the flow over a step experiment. On the $13440 \times 13440$ grid an average idle rate per node of around 50% has been observed for 576 processors which is in accordance with an idle rate of 47.96% observed in Figure 4.16. Note that the heat map also shows a clear correlation between number of obstacle cells and idle rate. Nodes (3, 1), (2, 1), (3, 4) and (1, 2) have an obstacle cell ratio of approximately 0%, 50%, 75% and 100% and have idle rates of 14.42%, 54.47%, 63.56% and 78.68% respectively. Note that node (1, 1) has a slightly lower idle rate than expected since it lies on the root locality which performs the reduction of the partial residuals and corresponding accuracy check. This localized distribution of obstacle cells leads to a higher average idle rate compared to the driven cavity experiment.

One can also again observe that the overall efficiency of the application directly depends on the utilization of compute resources (indicated through idle rate).

These results confirm the memory boundedness (and consequently boundedness in the amount of work available) of the application that has been observed in Section 4.2.1. One possible remedy to some nodes not containing any work at all might be the `hpx::components::migrate` functionality.
Figure 4.15: Strong scaling experiment for the flow over a step using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster.
Figure 4.16: Heat map for the idle rates of the flow over a step on a $40320 \times 10080$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.
4.3.2 Flow with uniform obstacles

The third example expands on the problem of losing processing power due to nodes that contain only obstacle cells. On the left boundary the fluid was allowed to flow into the domain, whereas on the right boundary an outflow condition was enforced. The other boundaries have no-slip conditions (see Section 3.2.1). Square obstacles of uniform size were inserted into the domain in a regular grid pattern. Two different spacings of the obstacle cells have been considered which lead to the domain characteristics outlined in Figure 4.17c. The goal is to confirm that the scaling behaviour significantly depends on the work available. A regular pattern in the obstacle cells should allow for local load balancing through the task scheduler in contrast to the more localized obstacle cells of the flow over a step in Section 4.3.1.

![Layout with spaced out obstacles](image1)
![Layout with tightly grouped obstacles](image2)

<table>
<thead>
<tr>
<th>Grid</th>
<th>spaced out</th>
<th>tightly grouped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Obstacles</td>
<td>81 (9 × 9)</td>
<td>256 (16 × 16)</td>
</tr>
<tr>
<td>Obstacle Surface Fraction</td>
<td>0.25%</td>
<td>0.81%</td>
</tr>
<tr>
<td>Obstacle Volume Fraction</td>
<td>27.44%</td>
<td>82.94%</td>
</tr>
<tr>
<td>Reynolds Number</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Boundary Conditions</td>
<td>Left: Inflow</td>
<td>Right: Outflow</td>
</tr>
<tr>
<td></td>
<td>Bottom: No-Slip</td>
<td>Top: No-Slip</td>
</tr>
</tbody>
</table>

**Figure 4.17:** The domains with uniform obstacles
The visualization of the solution for the grid with spaced out obstacles can be found in Figure 4.18.

Figure 4.18: Visualization of the solution for the flow with uniform obstacles on a $630 \times 630$ grid at different time steps for Reynolds Number $Re = 1000$
Figure 4.19 shows the results for the domain where the distance between obstacles is exactly the side length of an obstacle. The application performs almost identically to the driven cavity strong scaling problem in Section 4.2.1.1. In contrast to the flow over a step (Section 4.3.1) where the obstacle cells are localized to only some of the nodes, here the cells are evenly distributed across all localities. This means that even though the amount of non-fluid cells is similar (25% for the flow over a step to 27.44% for the uniformly distributed obstacles) and therefore the total amount of work is the same, the runtime system can locally balance the load better if each node has a similar ratio of fluid cells to obstacle cells.

The average idle rate heat map in Figure 4.20 reveals that a homogeneous distribution of work has been achieved. Since the blocks of obstacle cells are not perfectly aligned with the local grids of each node, localities with a slightly higher obstacle volume fraction exhibit a higher average idle rate whereas nodes with a low amount of obstacle cells correspondingly have a lower average idle rate.

Moreover the results of the strong scaling experiment for the domain consisting of a regular pattern of obstacles with a distance of 10% the side length of an obstacle can be found in Figure 4.21. Here the amount of obstacle cells has tripled compared to the previous experiment. Consequently the application shows a significant drop in efficiency for more than 256 processors.

The average idle rate graph and the heat map in Figure 4.22 imply that the time spent on executing tasks was far outweighed by the time used for managing the parallelization. This leads to a high average idle rate across all localities and consequently a drastically reduced efficiency of the application.

These two experiments suggest that the task stealing facilities of the HPX runtime system work well, provided that enough tasks are scheduled. These tasks should neither be very small nor very large to allow the hiding of parallelization costs, in accordance to the results of the matrix vector multiplication in Section 2.4.3.3.
Figure 4.19: Strong scaling experiment for the flow through spaced out uniform obstacles using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster
Figure 4.20: Heat map for the idle rates of the flow through spaced out uniform obstacles on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.
Figure 4.21: Strong scaling experiment for the flow through tightly grouped uniform obstacles using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster.
Figure 4.22: Heat map for the idle rates of the flow through tightly grouped uniform obstacles on a 20160 × 20160 grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.
4.3 Flows in complex geometries

4.3.3 Flow through a porous medium

A more practical problem is the flow through a porous medium. In this example, spheres of varying sizes have been randomly placed in a square domain. This is also known as a “random close packing”. Additionally an inflow condition was defined on the left boundary and an outflow condition was set on the right boundary. The smallest grid of size $3360 \times 3360$ can be seen in Figure 4.23a. First the center of each circle was uniformly chosen from the domain. Then the radius was also uniformly chosen from the interval $[4, 800]$. This process was repeated until 300 obstacles were inserted into the domain while enforcing a minimum distance of 5 cells between each pair of obstacles. The bigger grids were obtained through repeated refinement of the initial version. Note that the obstacles are not actually spherical since the application can only deal with rectangular cells.

![Domain for the porous medium](image)

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle Surface Fraction</td>
<td>4.01%</td>
</tr>
<tr>
<td>Obstacle Volume Fraction</td>
<td>50.73%</td>
</tr>
<tr>
<td>Reynolds Number</td>
<td>100</td>
</tr>
<tr>
<td>Boundary Conditions</td>
<td>Left: Inflow</td>
</tr>
<tr>
<td></td>
<td>Right: Outflow</td>
</tr>
<tr>
<td></td>
<td>Bottom: No-Slip</td>
</tr>
<tr>
<td></td>
<td>Top: No-Slip</td>
</tr>
</tbody>
</table>

**Figure 4.23**: The domain for the porous medium
Only the solution on a part of the domain was visualized due to memory limitations. Figure 4.24 shows the behavior of the flow over various time steps.

Figure 4.24: Visualization of the solution for the porous medium on part of the domain at different time steps for Reynolds Number $Re = 1000$
Figure 4.25 shows the computed metrics for the strong scaling experiment for the porous medium. For more than 400 processors a drop in speedup can be observed for virtually all grid sizes. This is again due to the lack of work for each individual processor and task at that point and consistent with the results for the driven cavity problem 4.2.1.1. Since only 50% of the cells are fluid cells the grid of size 20160 × 20160 of the porous medium corresponds to the 13440 × 13440 grid of the driven cavity problem in terms of average work per task and processor. As can be seen the respective graphs show a similar scaling behavior with a drop in efficiency at around 400 processors and a peak speedup in the range of 150.

Note that on the grids that were partitioned into more local partitions an earlier drop off in speedup and efficiency can be observed. This is due to each partition consisting of fewer cells. Therefore each HPX task has a short execution time which was found to be detrimental to the performance of the application in Section 4.2.1 because of a lack of available work.

The lack of work being one of the causes for the reduced scalability of the problem is supported by the idle rate graph in Figure 4.25. For all but one grid and for 400 processors or more the average idle rate for each locality was measured to be 60% for the two largest grids and even 80% for the smaller grids. This indicates a significant underutilization of the computing power of the hardware. Additionally the drop in speedup for more than 400 processors is accompanied by a significant increase in the average idle rate of each locality.

Figure 4.26 shows the distribution of cells across the 16 nodes (for a total of 256 processors) and the respective average idle rate of each locality. Similar to the effect in the driven cavity problem the left half of the domain exhibits lower idle rates than its counterpart. This is due to non zero pressure values propagating from that side (since the left side has an inflow boundary condition and the values were captured in the tenth time step which corresponds to \( t = 0.0098s \)) and therefore requiring more CPU cycles in the solver. Figure 4.27 depicts the corresponding scenario for 36 nodes and a total of 576 processors in the tenth time step. Even though the left half of the domain still requires more work due to the inflow boundary condition one can no longer observe a significant difference in idle rates across the nodes since the number of fluid cells per nodes is large enough to enable the task scheduler to effectively use all processors.
4 Performance benchmarking

Figure 4.25: Strong scaling experiment for the porous medium using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster
Figure 4.26: Heat map for the idle rates of the flow through a porous medium on a $20160 \times 20160$ grid in the tenth time step on 16 nodes (256 processors total) using a blockwise Jacobi solver.
Figure 4.27: Heat map for the idle rates of the flow through a porous medium on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.
4.3 Flows in complex geometries

4.3.4 Flow through narrow channels

This example exaggerates the problems of the previous examples by simulating a flow through a narrow system of channels, such that only around 1% of the domain are fluid cells. Additionally, these fluid cells are spread out which leads to a large amount of nodes executing a very small amount of work or no work at all. Figure 4.29 shows the layout and characteristics of the domain.

The visualization of the inflow on the left hand side of the domain can be seen in Figure 4.28.

Figure 4.28: Visualization of the solution for the flow through narrow channels on part of the domain at different time steps for Reynolds Number $Re = 1000$
Figure 4.29: The domain with narrow channels
Figure 4.30 shows the scaling results on different grid sizes. The small grids have not been tested since the application barely scales for the $10080 \times 10080$ grid. Interestingly, despite the significant lack of work for each node the largest grid exhibits a good scaling behavior for up to 64 processors with an efficiency of around 45%. One can also see that the division into $4 \times 4$ local partitions performs the best, with the division into $6 \times 6$ and $8 \times 8$ partitions being less efficient. The reason for this the distinct lack of work to be performed if the computation is split into many tasks. Then the cost of managing the creation and scheduling the tasks offsets and even surpasses the advantage gained from a parallel execution. Note that a division into less than $4 \times 4$ would increase the time spent on each task. However this would not be beneficial since each node has 16 hardware threads. If less than 16 tasks are scheduled the hardware will be severely underutilized.

Providing enough work for each processor is difficult on this geometry. One reason is that due to the low amount of fluid cells that additionally are spread out across the entire domain large grids are needed to obtain a reasonable utilization of each nodes computing resources. An additional factor is the data structure that is used to store the individual cells. The HPX components that were used internally use a `std::vector` to store the cells in a row-major format, including all obstacle cells. In this particular scenario this is clearly wasteful in terms of memory, since approximately 98% of cells are not involved in any of the computations and therefore their values do not need to be stored. On the other hand a more complicated data structure that takes this into account might perform worse in terms of data access.

The individual idle rates for the largest grid and 64 processors in time step 10 can be seen in Figure 4.31. With 4 nodes the number of fluid cells can be seen to be large enough such that the runtime can keep the idle rates below 50%. On the other hand it is apparent from these idle rates that even for 4 nodes there are not enough fluid cells to generate the appropriate amount of work for 64 processors.

Figure 4.32 shows the distribution of cells onto the maximum number of 36 nodes (576 processors). It can be seen that almost all nodes idle for more than 85% of the time. This means that the available resources are drastically underutilized leading to the problematic scaling behavior in Figure 4.30 that is essentially dominated by the overheads of the parallelization for this number of processors. The outlier in terms of idle rate again comes from the fact that the partial residuals are reduced on one locality.
Figure 4.30: Strong scaling experiment for the narrow channels using a blockwise Jacobi solver for a fixed amount of time steps and solver iterations on the Atacama cluster
Figure 4.31: Heat map for the idle rates of the flow through narrow channels on a $20160 \times 20160$ grid in the tenth time step on 4 nodes (64 processors total) using a blockwise Jacobi solver.
Figure 4.32: Heat map for the idle rates of the flow through narrow channels on a $20160 \times 20160$ grid in the tenth time step on 36 nodes (576 processors total) using a blockwise Jacobi solver.
5 Conclusion and outlook

5.1 Conclusion

In this thesis I investigated the applicability of a task based programming model using fine-grained constraint driven synchronization for solving a discrete version of the Navier-Stokes equations for incompressible fluids on a large scale computing system. To this end an overview of the features of the HPX runtime system was given and showcased on a parallel matrix vector multiplication code that was compared to a MPI implementation.

A discretization of the Navier-Stokes equations using a finite difference approach on a staggered grid was then described and parallelized using the HPX framework. For this the algorithm was adjusted to be usable in a task based environment. Moreover I discussed some options to eliminate global barriers, for example barriers imposed by the solver.

The application was then tested using different solvers on a classical problem. The software exhibited the expected performance and scalability behavior. The various versions of the solvers showed that global barriers and data interdependencies hinder concurrency. Additionally it was shown that the granularity of the partitioning of the computational work and consequently the task size in this parallel environment greatly influences concurrency and therefore runtime.

Experiments on different classical domains of varying sizes with unique work load distributions have been conducted to evaluate the local load balancing capabilities of the HPX thread scheduler. In these scenarios the application showed good scaling results that were mostly only restricted by the memory limitations of the test environment.

5.2 Outlook

One might be able to relax the memory boundedness of the application by employing a data structure that forgoes the storage of pure obstacle cells while maintaining the ability to quickly and efficiently access cell values and corresponding neighboring cell values. Additionally leveraging the application to be usable in a three dimensional setup should lead to a substantial increase in work to be performed, because one gains an additional velocity component to be computed. Furthermore the application now needs to account for 26 instead of 8 neighbor
relations which increases the computational load. This would also allow for flows in very complex domains such as the 3D pore-scale flows described in [Yan+15].

The algorithm currently spends most of the time solving the Poisson equation for the pressure, which presents a major bottleneck for the overall runtime and scalability of the application. Implementing more sophisticated and efficient solvers that work well in a parallel setting would help alleviate this effect.

The use of higher-level parallelization facilities offered by HPX such as data distribution policies, executors with corresponding execution policies and parallel algorithms (see [Kai+15]) might also be beneficial to an efficient utilization of available compute resources and to overall runtime, as well as help implement a generic version with regards to portability and versatility of the algorithm.
6 Appendix

6.1 Code snippets

```c
// let m be the number of rows, n be the number of columns
// send local x values to other nodes
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
    x, local_n, MPI_DOUBLE, comm);

// do local computation
for (local_i = 0; local_i < local_m; local_i++) {
    local_rhs[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_rhs[local_i] += local_A[local_i*n+j]*x[j];
}
```

Listing 6.1: "Computing a matrix vector product using MPI"
Listing 6.2: "Barebone structure for HPX components representing a block of rows of a matrix"
void multiply(std::vector<double> const& A, block.data x,
   std::vector<double> rhs, boost::uint64_t block_rows,
   boost::uint64_t num_columns, boost::uint64_t block_id,
   boost::uint64_t local_blocks_begin)
{
    // calculate offsets from the id of the current block
    boost::uint64_t offsetA = (block_id - local_blocks_begin) * block_rows * num_columns;
    boost::uint64_t offsetR = (block_id - local_blocks_begin) * block_rows;

    for (boost::uint64_t i = 0; i < block_rows; ++i)
    {
        rhs[offsetR + i] = 0;
        for (boost::uint64_t j = 0; j < num_columns; ++j)
        {
            rhs[offsetR + i] += A[offsetA + i * num_columns + j] * x[j];
        }
    }
}

Listing 6.3: "HPX action representing the matrix vector multiplication"
auto range = boost::irange(local_blocks_begin, local_blocks.end);

// futures for the computation on each block
std::vector<hpx::future<void> > block_futures(num_blocks);

// get the data of x
// then asynchronously multiply for each block
hpx::future<void> f = x.get_data(0, num_columns).then(
    hpx::util::unwrapped(
        [&](block_data const& x_data)
        {
            for (auto block_id : range)
            {
                block_futures[b - local_blocks_begin] =
                    hpx::async(
                        &multiply,
                        boost::ref(A),
                        x_data,
                        boost::ref(rhs),
                        block_rows,
                        num_columns,
                        block_id,
                        local_blocks_begin
                    );
            }
        });

// wait for computations to finish
hpx::wait_all(block_futures, f);

Listing 6.4: "Creating the HPX dependency tree for the matrix vector multiplication for one block"
template<> void send_boundary<LEFT>(std::size_t step, std::size_t var, std::vector<hpx::shared_future<void> >& send_futures) {
    // when the local computations finish, send data to the left
    if (!send_futures.empty())
        hpx::when_all(send_futures).then(
            hpx::launch::async,
            hpx::util::bind(
                boost::ref(send_buffer_left_),
                boost::ref(data_[var]),
                step,
                var
            )
        );
}

template<> void receive_boundary<LEFT>(std::size_t step, std::size_t var, std::vector<hpx::shared_future<void> >& recv_futures) {
    // if there is a left neighbor, get the future corresponding to the
    // computation
    if (recv_buffer_left_[var].valid_)
        recv_futures[LEFT] =
            hpx::async(
                hpx::util::bind(
                    boost::ref(recv_buffer_left_[var]),
                    boost::ref(data_[var]),
                    step
                )
            );
}

template<> hpx::shared_future<void> partition_server::get_dependency<LEFT>(std::size_t idx_block, std::size_t idy_block, future_vector const& recv_futures, partition_data<hpx::shared_future<void> > const& calc_futures) {
    // nothing to do, if left most locality
    if (is_left_ && idx_block == 0)
        return hpx::make_ready_future();

    // if left most block, get future corresponding to neighbor locality
    else if (idx_block == 0)
        return recv_futures[LEFT];

    // else return future corresponding to left neighbor block
    return calc_futures(idx_block - 1, idy_block);
}

Listing 6.5: "Sending and receiving data to and from the neighboring locality to the left using HPX"
hpx::shared_future<void> calc_future =
  hpx::dataflow( // as soon as the futures at the bottom are ready
    hpx::util::unwrapped(
      hpx::util::bind(
        &stencils<STENCIL_SOR>::call, // execute the SOR method
        boost::ref(data_[P]), boost::ref(rhs_data_),
        boost::ref(fluid_cells_(nx_block, ny_block)),
        c.part1, c.part2, c.dx_sq, c.dy_sq,
        token // cancellation token
      )
    )
    // future that represents the setting of boundary/obstacle values
    , set_p_futures(nx_block, ny_block)
    // future that represents the computation of the right hand side
    , compute_rhs_futures(nx_block, ny_block)
    // future for the data to the left for current iteration
    , get_dependency<LEFT>(nx_block, ny_block,
      recv_futures_P[current], sor_cycle_futures[current])
    // future for the data at the bottom for current iteration
    , get_dependency<BOTTOM>(nx_block, ny_block,
      recv_futures_P[current], sor_cycle_futures[current])
    // future for the data to the right for last iteration
    , get_dependency<RIGHT>(nx_block, ny_block, recv_futures_P[last],
      sor_cycle_futures[last])
    // future for the data at the top for last iteration
    , get_dependency<TOP>(nx_block, ny_block, recv_futures_P[last],
      sor_cycle_futures[last])
  );

Listing 6.6: "Asynchronously executing SOR for a single block using HPX"
Bibliography


All links were last followed on July 29, 2016.