



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Dynamic load balancing for Riemann
solvers with unsteady runtimes**

Raphael Schaller





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Dynamic load balancing for Riemann
solvers with unsteady runtimes**

**Dynamische Lastverteilung für
Riemann-Löser mit schwankenden
Laufzeiten**

Author:	Raphael Schaller
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Dipl.-Inf. Oliver Meister
Submission Date:	September 15, 2014



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Garching, September 12, 2014

Raphael Schaller

Abstract

In order to simulate tsunamis, the Chair of Scientific Computing of the TUM developed the *Shallow Water Equations Teaching Code*, which uses an augmented Riemann solver, amongst others. A characteristic of the augmented Riemann solver is its varying processing time, which depends on its input parameters like the water height or the bathymetry. Due to this variation, when the simulation is calculated on a multi-core or multi-processor system, the cells of the simulation domain can not just be divided into equal parts, each of which is assigned to a thread. Instead, a more sophisticated dynamic load balancing strategy has to be applied in order to achieve a good utilization of the processors and thus a low overall processing time of the simulation.

In this thesis, several of these load balancing approaches are introduced and compared to find out which one fits the augmented Riemann solver best.

The best load balancing strategy achieved a super linear speedup with 2 and 4 processors, an almost linear speedup with 8 processors and an about 20 percent better speedup with 16 processors than the no-load-balancing version.

Contents

Abstract	iii
1. Introduction	1
2. Introduction to the Shallow Water Equations Code	3
2.1. Principles of Shallow Water Equations	3
2.1.1. Solvers	3
2.1.2. Finite Volume algorithm	4
2.2. Architecture of the SWE Code	4
2.2.1. SWE_Block	4
2.2.2. SWE_Scenario	6
3. Approaches to Load Balancing	7
3.1. Approaches to Processing Time Prediction	7
3.1.1. Preceding Processing Time	9
3.1.2. Preceding Processing Times with Extrapolation	10
3.1.3. Classification	11
3.2. Approaches to Work-Allocation	13
3.2.1. List scheduling	14
3.2.2. Longest Processing Time	16
4. Implementation	17
4.1. Scenario	17
4.2. Application of OpenMP	17
4.3. Load Balancing Handling Wrapper	19
4.3.1. Preceding Processing Time	21
4.3.2. Preceding Processing Times with Extrapolation	21
4.3.3. Classification	21
4.3.4. List Scheduling	24
4.3.5. Longest Processing Time	24
4.3.6. Implicit Longest Processing Time	25

Contents

5. Comparison of Approaches	27
5.1. Comparison of Processing-Time-Prediction Approaches	27
5.2. Comparison of Work-Allocation Approaches	35
5.3. Best Combination	36
6. Conclusion	40
A. Scenario	41
B. Classification Array	42
C. Raw Data for Comparison	44
D. Bottleneck Values	47
Bibliography	48

1. Introduction

In 2004, one of the largest ever recorded earthquakes caused a devastating tsunami in the Indian Ocean which killed about 250,000 people [LLN08]. This is by no means the only tsunami which has happened recently, but nevertheless it shows the importance of research into tsunamis.

Nowadays powerful computer systems are capable of simulating tsunamis and thus giving aid to scientists trying to research them. The Chair of Scientific Computing of the Technische Universität München (TUM) is also exploring the techniques of tsunami simulation. One of the resulting projects is the *Shallow Water Equations Teaching Code* (SWE Code), which will be the subject of this thesis.

The SWE Code is primarily designed for teaching students parallel programming using for example OpenMP, MPI or CUDA, or combinations of these. It is therefore designed in a very flexible and modular way and furthermore features a lot of different methods and strategies for solving SWEs. However, this thesis focuses on *Wave Propagation* together with an *augmented Riemann* solver only.

One special characteristic about the augmented Riemann solver is that its processing time varies depending on its input parameters like the bathymetry or the water height. On multi-core/multi-processor systems each processor gets a chunk of the area and calculates the simulation there. Because of the variation of the processing time, some processors may be finished earlier than other ones, causing the former to wait and stay idle until the latter are finished. Thus, it is of great importance that every processor is assigned approximately the same workload.

The current version of the SWE Code features no load balancing at all. Depending on the simulation scenario, some processors may not be working to full capacity so no optimal scaling is accomplished. This thesis is dedicated to solving exactly this problem, i.e. to explore the best performing load balancing strategy for the SWE Code.

Figure 1.1 shows the scenario which will be used throughout this thesis. It is a one-dimensional dam break scenario with a block of water on the right which will turn into a wave flowing from the right to the left, where a shore is placed. Actually, it is two-dimensional, but the boundary conditions are always the same along the y -axis so it can be considered as one-dimensional. This is done to stretch the execution time of the simulation.

All test runs were performed on a dual socket octo core AMD Magny-Cours machine

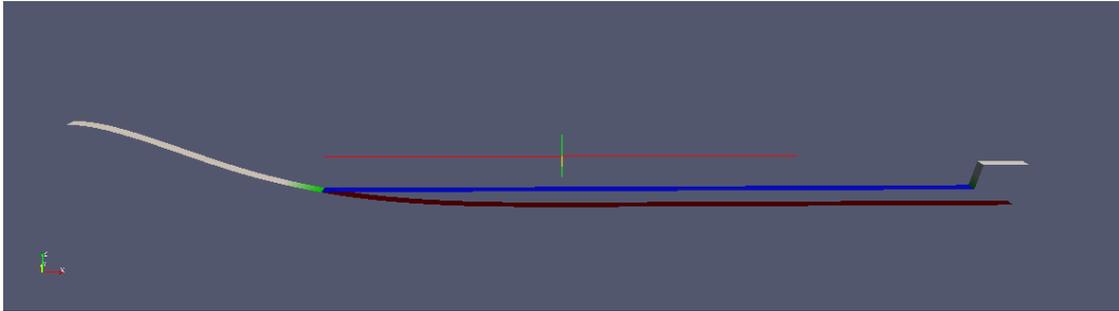


Figure 1.1.: ParaView plot of the boundary conditions of the simulation scenario which is used throughout this thesis. The brown area (amongst others) displays the bathymetry, the blue area (amongst others) displays the water height. The picture was generated using ParaView¹.

(called *MPP*) and a dual socket quad core Intel Nehalem machine (called *ICE*).

The best load balancing strategy achieved a super linear speedup with 2 and 4 processors, an almost linear speedup with 8 processors and an about 20 percent better speedup with 16 processors than the no-load-balancing version.

The thesis is structured as follows: Chapter 2 explains how SWEs work and briefly introduces the characteristics of the augmented Riemann solver. It also describes the architecture and class-design of the SWE Code.

In chapter 3, all load balancing approaches are characterized in a theoretical manner. Thus, no concrete implementations are shown.

These, however, can be found in chapter 4. The placement and usage of OpenMP is also illustrated there.

How well the aforementioned approaches work is compared in chapter 5. Furthermore, their advantages and drawbacks are pointed out.

Chapter 6 evaluates the accuracy and generality of the results and tries to give a recommendation as to which strategy to choose.

¹<http://www.paraview.org/>

2. Introduction to the Shallow Water Equations Code

In this chapter we will introduce the shallow water equations briefly and explain the architecture of the SWE Code. This chapter follows [BB12] closely.

2.1. Principles of Shallow Water Equations

The SWE Code uses a system of hyperbolic partial differential equations, which represent the two-dimensional SWEs:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t, x, y) \quad (2.1)$$

where h is the height of the water column, t the time, g the gravitational constant, and u and v are the velocities in x - and y -direction, respectively. $S(t, x, y)$ represents possible source terms, which is in case of the SWE Code only the bathymetry.

Equations (2.1) are derived from mass conservation and momentum conservation laws. This is shown in [LeV02] for one-dimensional SWEs.

2.1.1. Solvers

The code features several approximate Riemann solvers to solve equations eqs. (2.1): A fast and simple but inaccurate *Lax-Friedrichs solver*; a more complex and more accurate *f-Wave solver* which lacks inundation support; a very complex and expensive *augmented Riemann solver* supporting inundation; a *Hybrid solver* which combines the f-Wave and augmented Riemann solver.

As already stated, only the augmented Riemann solver is used in this thesis due to its varying processing times which greatly complicate load balancing.

2.1.2. Finite Volume algorithm

The computational domain is split into blocks, each represented by a Cartesian grid of cells. A vector $Q_{i,j} = [h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ is defined for each cell (i, j) .

The computation is based on the *Finite Volume* algorithm, so in each time step $t^n \rightarrow t^{n+1}$ for each grid cell, $Q_{i,j}^n$ is set to its new value

$$Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left(\mathcal{A}^+ \Delta Q_{i-1/2,j}^n + \mathcal{A}^- \Delta Q_{i+1/2,j}^n \right) - \frac{\Delta t}{\Delta y} \left(\mathcal{B}^+ \Delta Q_{i,j-1/2}^n + \mathcal{B}^- \Delta Q_{i,j+1/2}^n \right) \quad (2.2)$$

where $Q_{i,j}^n$ is $Q_{i,j}$ at time step t^n , Δt the length of the time step, and Δx and Δy stand for the size of the grid cell. $\mathcal{A}^\pm \Delta Q_{i\mp 1/2,j}^n$ and $\mathcal{B}^\pm \Delta Q_{i,j\mp 1/2}^n$ denote the solution of the Riemann problem located at the left, right, lower and upper edge of the (i, j) -grid-cell, respectively.

2.2. Architecture of the SWE Code

The most important classes of the SWE Code are `SWE_Block` and `SWE_Scenario`. The abstract class `SWE_Block` represents either the whole simulation domain or only a piece of it—more details can be found in section 2.2.1. The class `SWE_Scenario` specifies the boundary conditions and size of the scenario to simulate, further explained in section 2.2.2.

Additionally, there are several helper classes which handle input and output operations for example.

To start a simulation, first of all, an instance of a subclass of `SWE_Scenario` has to be constructed. Secondly, one or more instances of subclasses of `SWE_Block` need to be created. Finally, the simulation is controlled by calls to `SWE_Block` in a loop—this loop will be called the *main processing loop* in the following.

2.2.1. SWE_Block

The abstract class `SWE_Block` contains four 2-dimensional arrays, three of them to cover $Q_{i,j}$ defined in section 2.1.2, i.e. the water height h , the momentum hu and the momentum hv . Additionally, one holds the bathymetry b .

Each of the arrays is structured as shown in fig. 2.1. The inner (white) cells along with the *copy layer* (■ light grey cells) represent the actual simulation domain. The *ghost layer* (■ dark grey cells) is used to handle the boundaries of the block. For example this can be simulating a wall or, in case of multiple blocks, the connection between two blocks.

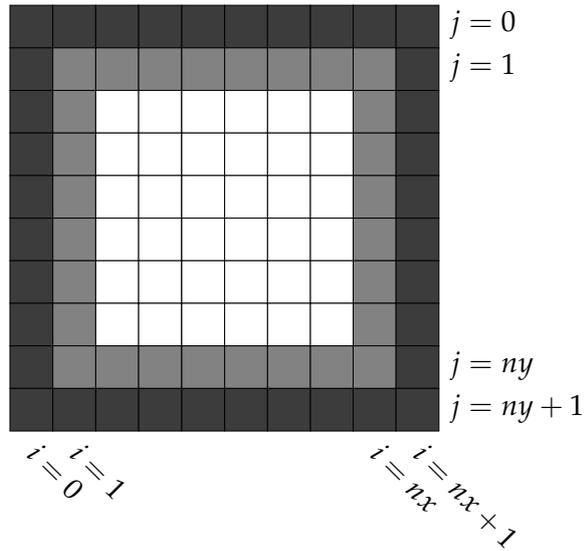


Figure 2.1.: Cell layout of SWE_Block. Cells filled with dark grey (■) mark the *ghost layer*, cells filled with light grey (■) mark the *copy layer*.

In the latter case, let us assume that there is a block A directly on the left of a block B , so the right-hand side of A has to be connected to the left-hand side of B . This is achieved by copying the content of the left copy layer of A to the right ghost layer of B and the content of the right copy layer of B to the left ghost layer of A at the beginning of each iteration of the main processing loop.

Furthermore, SWE_Block holds methods to control the simulation:

`setGhostLayer(...)` sets the ghost layer according to the boundary type of the block.

`computeNumericalFluxes(...)` is a pure virtual method which is supposed to execute the solver in order to obtain the solution of the Riemann problem at every edge of the grid cells.

`getMaxTimestep(...)` returns the maximum allowed time step size for the block depending on the values of h , hu and hv .

`updateUnknowns(...)` sets all of the aforementioned 2-dimensional arrays except b to their new value according to eq. (2.2).

The methods have to be called in the same order as they are mentioned above.

The class SWE_WavePropagationBlock is a concrete subclass of SWE_Block, implementing Wave Propagation. Despite the existence of several other subclasses, this is the only one used in this thesis.

2.2.2. SWE_Scenario

The class `SWE_Scenario` provides an interface and default implementations to allow the specification of a simulation scenario. This includes the time when the simulation ends (`endSimulation(...)`), the size of the simulation domain (`getBoundaryPos(...)`), the types of boundaries delimiting the domain (`getBoundaryType(...)`), the bathymetry (`getBathymetry(...)`), and the boundary conditions for water height velocities in x - as well as y -direction of the water (`getWaterHeight(...)`, `getVeloc_u(...)` and `getVeloc_v(...)`, respectively).

For this thesis a new scenario was implemented which is simpler by being 1-dimensional. Nevertheless, increased efforts were made to design it in a way that its simulation results in processor usage which is as unbalanced as possible. More details can be found in section 4.1.

3. Approaches to Load Balancing

In order to do load balancing properly, two steps are required. First of all, the processing time of each block is predicted, and secondly, each block is allocated to a processor; they will be called *processing time prediction* and *work-allocation* in the following and are discussed in sections 3.1 and 3.2, respectively.

3.1. Approaches to Processing Time Prediction

Since we have to calculate the load balancing *before* each iteration, we need to know the processing time of each block beforehand. There are several strategies to predict these which will be introduced in this section.

First we should define some variables: $\mathcal{B} = (B_1, \dots, B_N)$ is a chain of N blocks and each B_i again is a chain of C cells: $B_i = (c_{i_1}, \dots, c_{i_C})$. The p th cell of Block B_i is denoted as c_{i_p} . t^n is the current time step. The function $W : \mathcal{B} \rightarrow (0, \text{inf})$ maps each block to its processing time. Moreover, $W_i^n = W(B_i^n)$ is the processing time of a block B_i at time step t^n . $e_{i_{p,g}}^n = (c_{i_p}, c_{i_q}) \in \mathcal{E}_i$ is the edge between two adjacent cells c_{i_p} and c_{i_q} , while \mathcal{E}_i is the set of all edges of block B_i . Function $w : \mathcal{E}_i \rightarrow (0, \text{inf})$ maps each edge to its processing time. Furthermore, $w_{i_{p,q}}^n = w(e_{i_{p,q}}^n)$. \hat{W}_i^n and $\hat{w}_{i_{p,q}}^n$ are the *predicted* processing times of a block or edge, respectively. $h_{i_p}^n, (hu)_{i_p}^n, (hv)_{i_p}^n, b_{i_p}^n$ are the water height, momentum in x - and y -direction and bathymetry of cell c_{i_p} , respectively. Since we will not need to differ between the momentum in x - and y -direction in the following, we will use hx as replacement for hu and hv .

We say that

$$W_i^n \approx \sum_{e_{i_{p,q}} \in \mathcal{E}_i} w_{i_{p,q}}^n \quad (3.1)$$

and

$$\hat{W}_i^n = \sum_{e_{i_{p,q}} \in \mathcal{E}_i} \hat{w}_{i_{p,q}}^n. \quad (3.2)$$

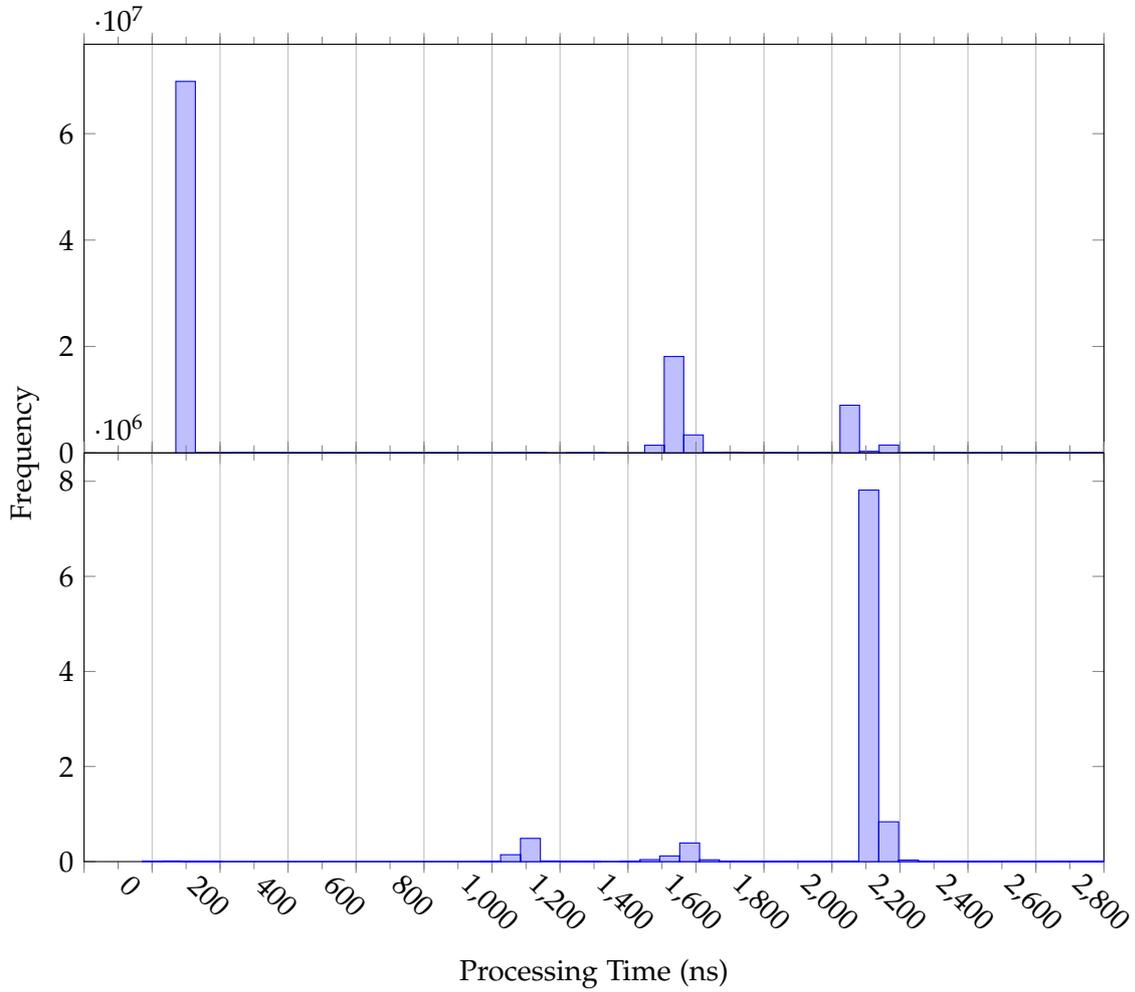


Figure 3.1.: Histograms of processing times of the augmented Riemann solver. The upper was generated by measuring the processing times during the simulation of the scenario. The lower was generated using random input parameters as described at the beginning of chapter 3. We can see that the processing times can be separated neatly into the categories $[0, 300)$, $[300, 1500)$, $[1500, 2000)$, $[2000, 2600)$.

In eq. (3.1) we use a \approx -sign, because the processing time of a block does not only include the processing times of all edges of its cells. It also includes processing times for method calls, loops, etc, which, however, will be neglected.

As already stated, the processing time of the augmented Riemann solver varies greatly depending on its input parameters. In order to get an idea of the possible outcomes, the processing times were measured in two different ways: Firstly, during the simulation of the scenario and secondly, by executing the solver with random input parameters.

For the purpose of reducing noise and measurement errors, the duration of *ten* consecutive executions of the solver with the same input parameters was measured.

The random input parameters were generated as follows: Since the solver is supposed to be executed for an edge between two cells, its input parameters are the water height, momentum and bathymetry, once for the first cell c_0 and once for the second cell c_1 , i.e. $h_0, (hx)_0, b_0$ and $h_1, (hx)_1, b_1$, respectively (the indices of the block and the time step can be neglected here). $(hx)_0, b_0, (hx)_1, b_1$ are picked randomly from $[-100, 100] \subset \mathbb{R}$. h_0, h_1 are treated in a special way in the SWE Code: When h_i is negative, h_i and $(hx)_i$ are set to 0. Thus, h_0, h_1 are chosen randomly from $[-10, 100] \subset \mathbb{R}$ and the same treatment is applied here.

The results of both approaches can be found in fig. 3.1. The upper histogram shows the distribution of the measured times during the simulation of the scenario, which resulted in about 10^8 measurements. The lower one was generated by executing the solver with 10^7 sets of input parameters which were generated as mentioned above.

The difference in the histograms is conspicuous, especially the absence of the bar at 200ns in the lower chart. It is, however, easy to explain: The solver can only be executed in the short time span, when $h_0 = h_1 = 0$, thus at a dry area. Since h_0 and h_1 are randomly chosen from $[-10, 100] \subset \mathbb{R}$, the chance of both being negative—they are set to 0 afterwards—is $(10/110)^2 \approx 0.83\%$. In the scenario, in contrast, there is a large dry area, so the solver will often be executed with $h_0 = h_1 = 0$.

The most simple processing time prediction approach is to set all prediction to the same value, say 1. So $\hat{W}_1^n = \hat{W}_2^n = \dots = \hat{W}_N^n = a$, where a is a constant. This is denoted as NONE.

In the following, three more complex approaches to predicting the processing time of a block will be introduced.

3.1.1. Preceding Processing Time

Under normal circumstances, the change of parameters (h, hx, b) of a cell within one time step should very small. This can be exploited by guessing that the processing time of the solver in time step t^n will be similar to the processing time in time step

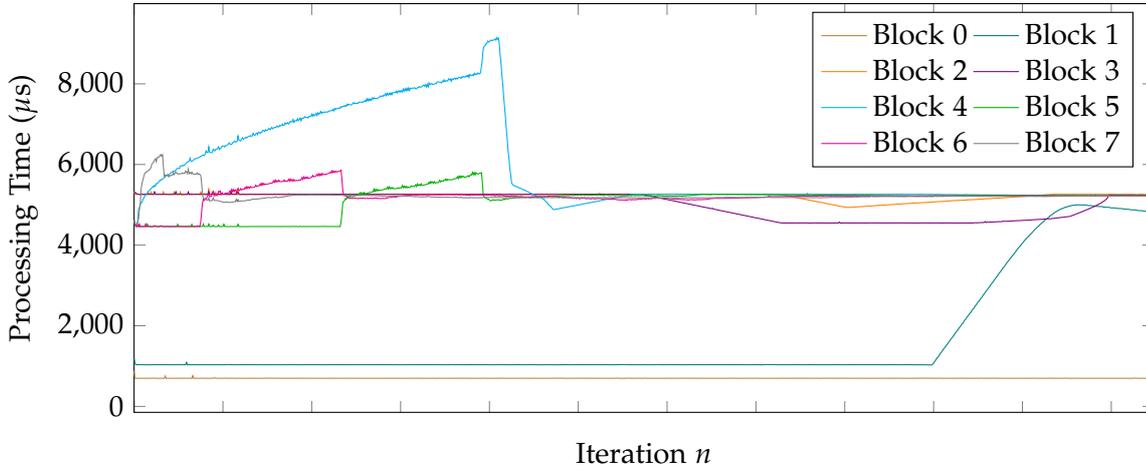


Figure 3.2.: This figure shows the course of the processing times of all eight blocks while simulating the scenario which was split into eight blocks. The blocks are numbered from left to right. We can see that the processing times do not change too much from time step to time step. This is confirmed by fig. 3.3.

t^{n-1} . Figures 3.2 and 3.3 confirm that the processing time of a block does only change slightly from time step to time step.

Using the *preceding processing time* (denoted as TIME) prediction algorithm, we just measure the processing time W_i^{n-1} of a block B_i in time step t^{n-1} and use exactly this value as prediction of the processing time \hat{W}_i^n , so $\hat{W}_i^n = W_i^{n-1}$. For the first time step, we set $W_i^0 = a, a > 0$ where a is a constant, equal for all blocks.

3.1.2. Preceding Processing Times with Extrapolation

Preceding processing times with extrapolation (denoted as TIME EXTRAPOLATION) works in a similar way. But instead of depending only on W_i^{n-1} , we employ a couple of preceding processing times—in this thesis up to five, so $\hat{W}_i^n = f(W_i^{n-1}, W_i^{n-2}, \dots, W_i^{n-5})$, where f is a function which extrapolates \hat{W}_i^n from its parameters.

Three different definitions of f were tested:

- Weighted average of three preceding times (denoted as AVG 3).

$$f(\dots) = 0.5W_i^{n-1} + 0.3W_i^{n-2} + 0.2W_i^{n-3} \quad (3.3)$$

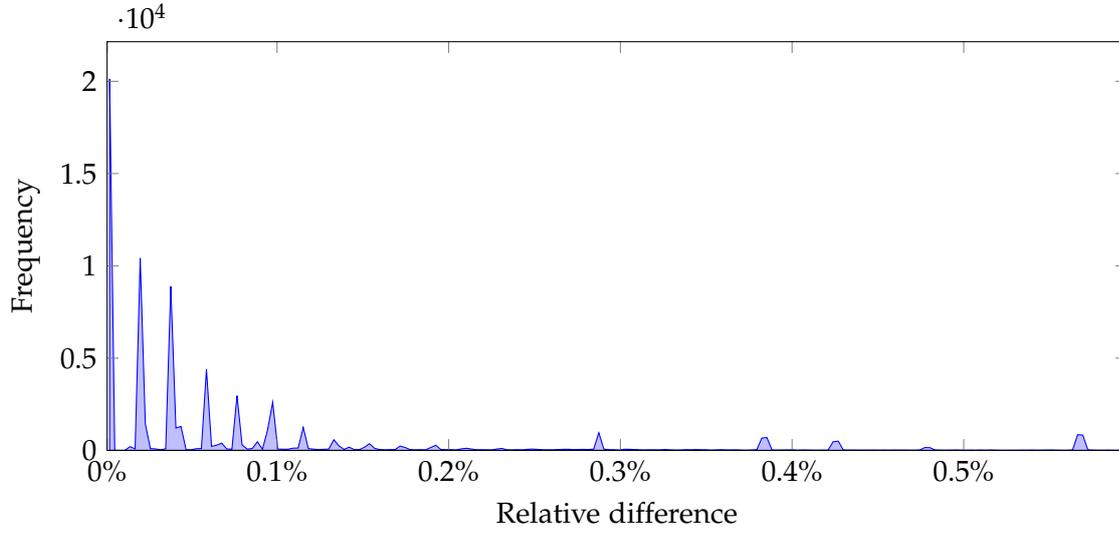


Figure 3.3.: This figure shows a histogram of the absolute values of the relative distances of two consecutive processing times of the same block while simulating the scenario. Thus, it displays $|W_i^n - W_i^{n+1}|/W_i^n$. As we can see, the changes are really small.

- Weighted average of five preceding times (denoted as AVG 5).

$$f(\dots) = 0.45W_i^{n-1} + 0.25W_i^{n-2} + 0.15W_i^{n-3} + 0.1W_i^{n-4} + 0.05W_i^{n-5} \quad (3.4)$$

- Linear extrapolation of five preceding times (denoted as LINEAR).

$$f(\dots) = \frac{2}{3} \left(W_i^{n-1} + W_i^{n-2} \right) - \frac{1}{3} \left(-W_i^{n-3} + W_i^{n-4} + W_i^{n-5} \right) \quad (3.5)$$

How well the approaches work will be discussed in chapter 5.

3.1.3. Classification

Acting on the principle of *classification* (denoted as CLASSIFICATION), we exploit the fact that the execution of the augmented Riemann solver only depends on its input parameters, i.e. it does not depend on any global state. Thus, we should be able to derive the predicted processing time $\hat{w}_{i,p,q}^n$ from the input parameters of the solver. We therefore look for a function g so that

$$\hat{w}_{i,p,q}^n = g \left(h_{i_p}^n, h_{i_q}^n, (hx)_{i_p}^n, (hx)_{i_q}^n, b_{i_p}^n, b_{i_q}^n \right). \quad (3.6)$$

3. Approaches to Load Balancing

To find the dependency between the parameters and the processing time, data was generated in the same way as was already stated at the beginning of this section, but in contrast, not only the processing time was saved, but a tuple

$$d^v = \left(h_{i_p}^v, h_{i_q}^v, (hx)_{i_p}^v, (hx)_{i_q}^v, b_{i_p}^v, b_{i_q}^v, w_{i_{p,q}}^v \right), \quad (3.7)$$

i.e. the input parameters together with the particular processing time were saved. In the following, d_u^v will be the u th element of the v th record of d . Further, a list of bit vectors b was defined with

$$\begin{aligned} b_0^v &= \begin{cases} 1 & \text{if } d_0^v = 0, \\ 0 & \text{else} \end{cases}, & b_1^v &= \begin{cases} 1 & \text{if } d_1^v = 0, \\ 0 & \text{else} \end{cases}, & b_2^v &= \begin{cases} 1 & \text{if } d_2^v \geq 0, \\ 0 & \text{else} \end{cases}, \\ b_3^v &= \begin{cases} 1 & \text{if } d_3^v \geq 0, \\ 0 & \text{else} \end{cases}, & b_4^v &= \begin{cases} 1 & \text{if } d_4^v \geq 0, \\ 0 & \text{else} \end{cases}, & b_5^v &= \begin{cases} 1 & \text{if } d_5^v \geq 0, \\ 0 & \text{else} \end{cases}. \end{aligned}$$

As we can see in fig. 3.1, the processing times can be separated neatly into four categories: $[0, 300)$, $[300, 1500)$, $[1500, 2000)$, $[2000, 2600)$ with centers 250, 1250, 1750, 2300, respectively. Thus, a list \tilde{w} was set according to the following:

$$\tilde{w}^v = \begin{cases} 250 & \text{if } d_6^v \in [0, 300) \\ 1250 & \text{if } d_6^v \in [300, 1500) \\ 1750 & \text{if } d_6^v \in [1500, 2000) \\ 2300 & \text{if } d_6^v \in [2000, 2600) \end{cases}$$

For example, let us say that the solver was executed with $h_{i_p}^v = 10, h_{i_q}^v = 0, (hx)_{i_p}^v = 5, (hx)_{i_q}^v = -30, b_{i_p}^v = -20, b_{i_q}^v = 30$ and the measured processing time was $\hat{w}_{i_{p,q}}^v = 1300ns$. Therefore, $d^v = (10, 0, 5, -30, -20, 30, 1300)$ and thus $b^v = (0, 1, 1, 0, 0, 1)$. Furthermore, $\tilde{w}^v = 1250$.

MATLAB's function `fitctree(x, y)` [MAT14] was then used to build a classification tree. x is supposed to be a matrix of floats, each row is a record and each column is a predictor— b was used here as x after casting it to float. y is supposed to be a vector of floats, each row of which is the outcome of the corresponding row of x . \tilde{w} was used as x after casting it to float. The resulting tree is shown in fig. 3.4.

To predict the processing time of a whole block B_i at time step t^n , we therefore classify each edge of this block according to fig. 3.4. We then combine eqs. (3.2) and (3.6):

$$\begin{aligned} \hat{W}_i^n &= \sum_{e_{i_{p,q}} \in \mathcal{E}_i} \hat{w}_{i_{p,q}}^n \\ &= \sum_{e_{i_{p,q}} \in \mathcal{E}_i} g \left(h_{i_p}^n, h_{i_q}^n, (hx)_{i_p}^n, (hx)_{i_q}^n, b_{i_p}^n, b_{i_q}^n \right), \end{aligned} \quad (3.8)$$

where g is the function which classifies an edge.

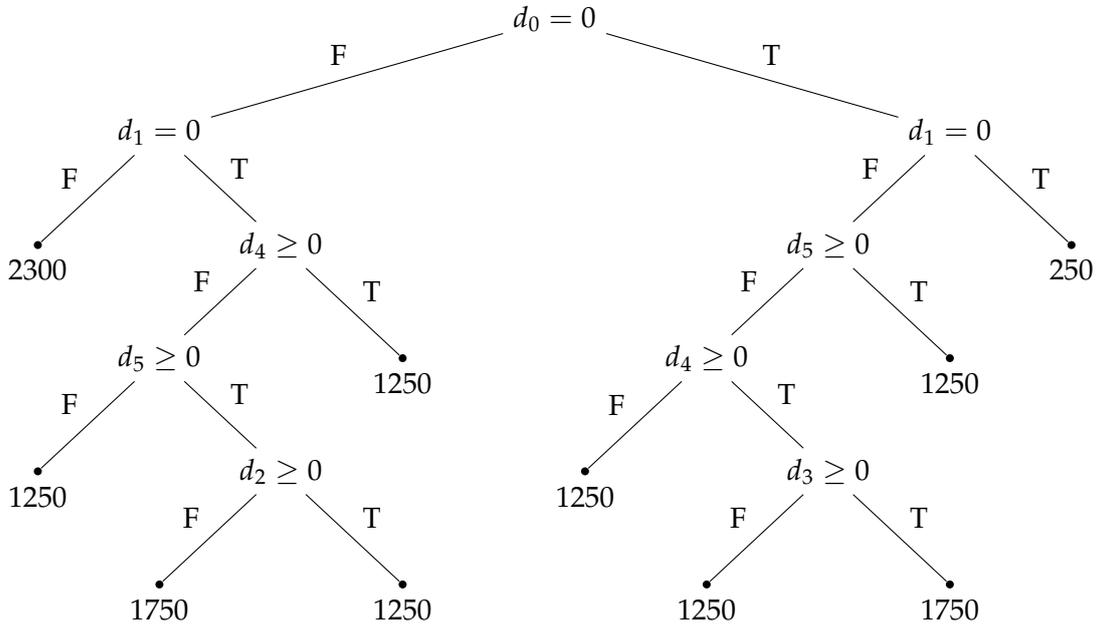


Figure 3.4.: Classification tree generated by `fitctree(...)` without any pruning.

3.2. Approaches to Work-Allocation

In the previous section we intended to predict the processing time \hat{W}_i^n of each block B_i in time step t^n . In this section we will try to allocate those blocks to a processor depending on \hat{W}_i^n , aiming at a usage of all processors which is as well balanced as possible.

The following can be found more detailed in [PA04], especially the content of section 3.2.1.

At first we should make some definitions: $\mathcal{B} = (B_1, B_2, \dots, B_N)$ is the chain of N blocks, $\mathcal{W} = (\hat{W}_0, \hat{W}_1, \dots, \hat{W}_N)$ are the associated weights, i.e. the predicted processing times. The chain \mathcal{B} shall be mapped onto the chain $\mathcal{P} = (P_1, P_2, \dots, P_K)$ of K processors. How the mapping is done in detail depends on the chosen work-allocation approach.

The quality of a mapping is determined by the *bottleneck value* β , which is defined as the load of the maximally loaded processors. Hence, the bottleneck value is to be minimized. The optimal bottleneck value is denoted as β^* .

Generally, we can distinguish between algorithms which preserve the order of blocks, so each processor is allocated a contiguous sub-list of \mathcal{B} , and those which can reorder the list of blocks. *List scheduling* (section 3.2.1) is of the former type, the *Longest Processing Time* algorithm is of the latter type (section 3.2.2).

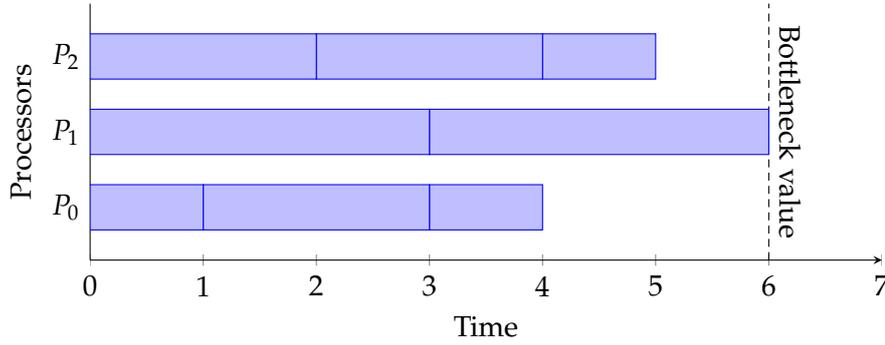


Figure 3.5.: Visualization of the *bottleneck value*: The blue bars are tasks to process. Processor P_1 finishes last, so the bottleneck value marks its finish line.

3.2.1. List scheduling

List scheduling restricts the aforementioned mapping to be contiguous, so every processor is allocated a contiguous subchain $\mathcal{B}_{i,j} = (B_i, B_{i+1}, \dots, B_j)$ of \mathcal{B} , as shown in fig. 3.6. The weight of this subchain is denoted as $W_{i,j} = \sum_{u=i}^j \hat{W}_u$, the total weight as $W_{tot} = W_{1,N}$. The mapping is described by $\Pi = (s_0, s_1, \dots, s_K)$ with $0 = s_0 \leq s_1 \leq s_2 \leq$

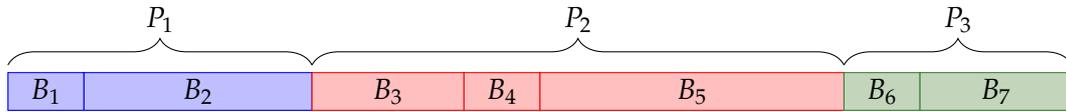


Figure 3.6.: List scheduling example.

$\dots \leq s_K = N$, where s_k is the number of the last block mapped to P_k , hence the number of the first block is $s_{k-1} + 1$. The load of processor P_k is denoted as $L_k = W_{s_{k-1}+1, s_k}$. The bottleneck value therefore is $\beta = \max_{1 \leq k \leq K} \{L_k\}$. Here, the optimal bottleneck value is $\beta^* = W_{tot}/K$.

The simplest method of mapping works like OpenMP's *static scheduling* [Ope11], in the following referred to as *no load balancing*: Every processor is allocated approximately the same number of blocks, so

$$\Pi = (0, \rho(N/K), \rho(2N/K), \dots, \rho((K-1)N/K), N), \quad (3.9)$$

where $\rho(x) = \lfloor x + 0.5 \rfloor$. This can be simulated using `NONE` together with list scheduling.

Though the 2- and 3-dimensional versions of list scheduling are NP-complete ([GM96] and [Nic94], respectively), an optimal solution of the 1-dimensional version can be found efficiently ([PA04]). Nevertheless, in this thesis we will stick to a simple approximate

3. Approaches to Load Balancing

algorithm presented in [MP97], since it can be calculated in parallel for the most part. We will call it *MP algorithm* in the following.

As the first step in this algorithm, we calculate the prefix sum of \mathcal{W} :

$$\bar{W}_i = \sum_{u=1}^i \hat{W}_u \quad (3.10)$$

Following that, we search for the largest index s_k where $\bar{W}_{s_k} \leq k\beta^*$. If $(\bar{W}_{s_k+1} - k\beta^*) \leq (k\beta^* - \bar{W}_{s_k})$, we increment s_k . This is visualized in fig. 3.7. Since \bar{W} is monotonically

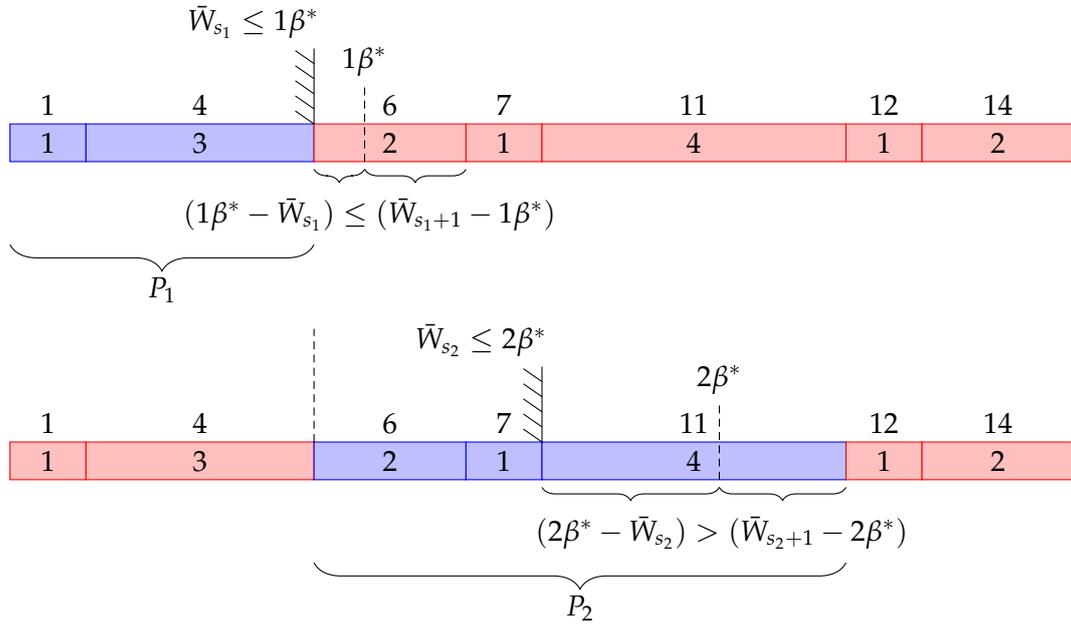


Figure 3.7.: Visualization of two steps of the MP algorithm. $K = 3$, $W_{tot} = 14$, $\beta^* = 14/3 \approx 4.67$. In the first (upper) step s_1 is *not* incremented, in the second (lower) step s_2 is incremented.

non-decreasing, the search for s_k can be performed using binary search. Thus, the algorithm takes $\mathcal{O}(N + K \log N)$ time, because we calculate the prefix sum ($\mathcal{O}(N)$) and perform $(K - 1)$ binary searches ($\mathcal{O}(\log N)$).

Whereas the prefix sum is difficult to parallelize—hence, we will only use a sequential version in this thesis, the $(K - 1)$ binary searches can be executed entirely in parallel. The complexity thus can be reduced to $\mathcal{O}(N + \log N)$ when executed on K processors.

As proved in [MP97], the MP algorithm is a 2-approximation and the upper bound of the bottleneck value is $\beta_{MP} < \beta^* + \hat{W}_{max}$, where $\hat{W}_{max} = \max_{\hat{W} \in \mathcal{W}} \hat{W}$.

3.2.2. Longest Processing Time

The *longest processing time* (LPT) algorithm was first described in [Gra69]. Here, in contrast to list scheduling, not only contiguous mappings are allowed. The mapping is denoted as a K -tuple of lists of indices of blocks

$$\Pi = \left((\pi_{1_1}, \pi_{1_2}, \dots), \overbrace{(\pi_{2_1}, \pi_{2_2}, \dots)}^{\Pi_2}, \dots, (\pi_{K_1}, \pi_{K_2}, \dots) \right). \quad (3.11)$$

Π_k is defined as the list of indices of all blocks assigned to processor P_k and

$$L_k = \sum_{\pi_{k_j} \in \Pi_k} \hat{W}_{\pi_{k_j}} \quad (3.12)$$

is the load of P_k .

The algorithm also uses a priority queue $\mathcal{Q} = (q_1, q_2, \dots, q_K)$ which contains the indices of the processors so that $L_{q_1} \leq L_{q_2} \leq \dots \leq L_{q_K}$. Thus, the index of the processor with the lowest load is placed at the front of \mathcal{Q} . Moreover, it requires the list of blocks \mathcal{B} to be sorted in descending order by their processing time, denoted as $\tilde{\mathcal{B}}$.

LPT works as follows: At the beginning, Π is a K -tuple of *empty* lists. We select the list Π_{q_1} of assigned work to the processor with the lowest load. Then we add the block with the longest processing time, i.e. the block at front of $\tilde{\mathcal{B}}$, to Π_{q_1} . Following that, we remove the block from $\tilde{\mathcal{B}}$. We repeat this until $\tilde{\mathcal{B}}$ is empty. The processors then execute the simulation at their assigned blocks, so P_k handles all blocks in Π_k .

None of these steps can be executed efficiently in parallel, hence, in this thesis, LPT will merely be implemented as a sequential algorithm. According to [Gra69], LPT is a $4/3 - 1/(3K)$ approximation. Since the priority queue is of size K and we have to insert N blocks, the complexity of the algorithm is $\mathcal{O}(N \log K + N \log N)$, where $N \log N$ is caused by the sorting (priority queue insert and remove operations both take $\mathcal{O}(\log n)$ time on a n -element priority queue and sort's running time is $\mathcal{O}(n \log n)$ on a n -element list [Cor+01]). Because N and K are positive this can be reduced to $\mathcal{O}(N \log(K + N))$.

4. Implementation

This chapter is all about concrete implementation. On the one hand, it explains how the scenario is designed (section 4.1) and how OpenMP is deployed in the existing code (section 4.2), on the other hand, it shows how the load balancing approaches listed in chapter 3 are implemented (section 4.3).

4.1. Scenario

The scenario shown in fig. 1.1 is designed in a way that it results in a processor usage which is as unbalanced as possible when executed with no load balancing. As we can deduce from the classification tree in fig. 3.4, an important factor that influences the processing time is, whether an area is dry or wet. The scenario therefore contains a large area which is dry at the beginning and becomes wet in the course of the simulation. Hence, the processing times of the blocks in this area is low at the beginning and becomes greater later on. This challenges the load balancing so we can verify its proper functioning.

The concrete parameters are listed in appendix A.

4.2. Application of OpenMP

As already stated in section 2.2, the simulation is for the most part executed and controlled within the *main processing loop*. It is basically constructed as shown in algorithm 4.1. Its important to note that each of the `ForEach` blocks have to be finished before the the next is allowed to start, thus it is not possible to execute two `ForEach` blocks in parallel.

How the OpenMP keywords are applied can be seen in algorithm 4.2. Within a simple OpenMP *parallel for loop* with static scheduling ([Ope11]), `setGhostLayer(...)` and `updateUnknowns(...)` are executed, since both methods do not vary their processing times, so each call to the same function requires approximately the same amount of time. Before the loop around `computeNumericalFluxes(...)`, the load balancing has to be calculated, so that every OpenMP thread is allocated a chunk of blocks. The load balancing calculation has to be done only once per iteration, hence it is surrounded

4. Implementation

```
while Simulation not finished do
|   foreach Block B do
|   |   B.setGhostLayer();
|   end
|   foreach Block B do
|   |   B.computeNumericalFluxes();
|   end
|    $m \leftarrow \max_{\text{all blocks } B} B.\text{getMaxTimestep}();$ 
|   foreach Block B do
|   |   B.updateUnknowns(m);
|   end
end
```

Algorithm 4.1: Construction of the main processing loop.

```
while Simulation not finished do
|   OMP parallel foreach Block B (static scheduling) do
|   |   B.setGhostLayer();
|   end
|   OMP parallel shared m
|   |   OMP single
|   |   |   Calculate load balancing, i.e. which blocks are assigned to which thread;
|   |   end
|   |   while Thread has blocks left do
|   |   |    $B \leftarrow$  next assigned block to this thread;
|   |   |   B.computeNumericalFluxes();
|   |   |   OMP critical
|   |   |   |    $m \leftarrow \max(m, B.\text{getMaxTimestep}());$ 
|   |   |   end
|   |   end
|   |   OMP barrier;
|   |   OMP parallel foreach Block B (static scheduling) do
|   |   |   B.updateUnknowns(m);
|   |   end
|   end
end
```

Algorithm 4.2: Construction of the main processing loop with load balancing and OpenMP.

by OpenMP's *single* keyword. Simply to place it outside of the parallel region is not possible since the number of threads is only known within a parallel region and is required for proper load balancing.

Within the `computeNumericalFluxes(...)`-loop, each thread handles all its assigned blocks. The maximum time step is also calculated there. Because the shared variable *m* is modified there, it is surrounded by the *critical* keyword. A *barrier* is placed between both loops to ensure that all threads finished this loop before they begin to execute `updateUnknowns`,

4.3. Load Balancing Handling Wrapper

In order to handle load balancing, a wrapper class `LBlockCollection` around `SWE_WavePropagationBlock` was written, basically containing a vector of all existing blocks. Further, it contains a method to add a new block (`addBlock(...)`) and one to initiate the calculation of the load balancing (`calculateLoadBalancing(...)`). Method `getBlock(k, i)` returns the *i*th block which is assigned to processor *k*. Operator `[i]` returns the *i*th block of the vector.

At construction of `LBlockCollection`, the desired processing time prediction strategy (section 3.1) is selected. One can choose between `NONE`, `TIME`, `TIME EXTRAPOLATION` and `CLASSIFICATION`. Within `LBlockCollection`, this is realized via *strategy pattern*: An abstract class `LBStrategy` declares the pure virtual methods `before_computeNumericalFluxes(...)` and `after_computeNumericalFluxes(...)`, which are to be called before and after the `computeNumericalFluxes(...)` call of every block, respectively. It also declares `getComparisonValue(...)` which returns the predicted processing time and enables one to compare those. The subclasses of `LBStrategy` are described in sections 4.3.1 to 4.3.3.

The individual blocks are not directly stored as `SWE_WavePropagationBlock` in the vector of `LBlockCollection`, but are again wrapped into a class `LBlock`. Method `computeNumericalFluxes(...)` of `LBlock` ensures that `before_computeNumericalFluxes(...)` and `after_computeNumericalFluxes(...)` are called before and after method `computeNumericalFluxes(...)` of `SWE_WavePropagationBlock`, respectively. It also forwards some important methods of `SWE_WavePropagationBlock`. In addition, it contains a `VtkWriter` to handle the output of this particular block.

The class design is shown in fig. 4.1.

The work-allocation strategy (section 3.2) is selected at compilation time via preprocessor macros.

4. Implementation

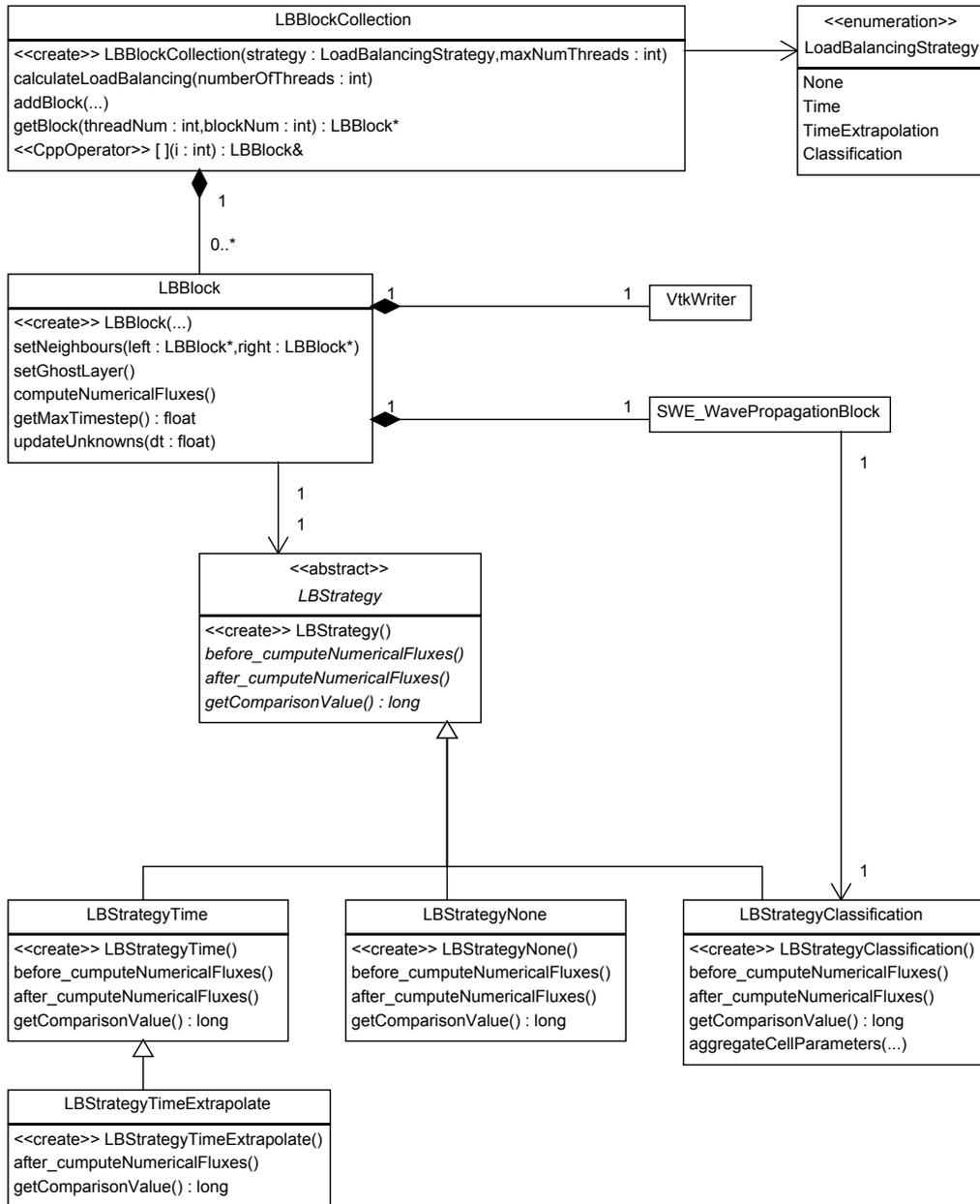


Figure 4.1.: Class diagram of the *load balancing handling wrapper*. The *strategy pattern* is used to render the selection of a specific load balancing strategy possible. All `LBStrategy...` classes therefore are subclasses of `LBStrategy`. Each `LBlock` contains one `SWE_WavePropagationBlock` and one `VtkWriter`. `LBlockCollection` bundles all `LBlock`s.

4.3.1. Preceding Processing Time

Preceding processing time prediction (section 3.1.1) is implemented in `LBStrategyTime`. It simply uses the POSIX function `clock_gettime(...)` in order to measure the elapsed time between the calls to `before_cumputeNumericalFluxes(...)` and `after_cumputeNumericalFluxes(...)`. `getComparisonValue(...)` just returns the measured processing time.

4.3.2. Preceding Processing Times with Extrapolation

Preceding processing times with extrapolation (section 3.1.2) is implemented in class `LBStrategyTimeExtrapolate` in a similar way to `TIME`. `before_cumputeNumericalFluxes(...)` saves the start time using `clock_gettime(...)`, the elapsed time is calculated by `after_cumputeNumericalFluxes(...)`. In addition, it stores the measured time in a circular buffer, holding up to five elements. In case there are too few values stored in the buffer (which happens at the beginning of the simulation), the measured time itself is used as processing time prediction. Otherwise, it is extrapolated using three to five past values. Three different extrapolation methods were tested which are listed in section 3.1.2.

4.3.3. Classification

`CLASSIFICATION` (section 3.1.3) is implemented in class `LBStrategyClassification`. In contrast to the other approaches, it does not only need data on block level but it needs data from each cell of a block. Thus, it features a method `aggregateCellParameters(float h0, float hu0, float b0, float h1, float hu1, float b1)` which is to be called within `updateUnknowns(...)` of class `SWE_WavePropagationBlock`, after eq. (2.2) was calculated. Within this method, the passed parameters are used to classify the edge, trying to predict the processing time by applying the decision tree in fig. 3.4. This was tested in three different ways which will be discussed in the following.

Hash Map

The *hash map* method is similar to the way the decision tree was generated in MATLAB. Since there are six decisions to make, the outcomes are all combinations of six bits, thus the numbers from 0 to 63. We therefore need an array which maps each of these 64 values to a number, representing the predicted processing time. This array is shown in appendix B.

The position within the array is calculated as follows:

4. Implementation

```
uint8_t pos = ((h0 == 0) << 0) | ((h1 == 0) << 1) | ((hu0 >= 0) << 2)
             | ((hu1 >= 0) << 3) | ((b0 >= 0) << 4) | ((b1 >= 0) << 5);
```

The advantage of the hash map is its branchless implementation, but it can result in poor cache behavior due to its size of 64 bytes.

Decision Tree

The *decision tree* approach means a sequence of nested if-clauses which replicate the classification tree in fig. 3.4. To speed up execution, some branches were merged. The code is displayed below, `classA`, `classB`, `classC` and `classD` denote the values 5, 25, 35 and 46, respectively.

```
if(h0 == 0)
    if(h1 == 0)
        acc += classA;
    else
        if((b1 >= 0) || (b0 < 0) || (hu1 < 0))
            acc += classB;
        else
            acc += classC;
else
    if(h1 == 0)
        if((b0 >= 0) || (b1 < 0) || (hu0 >= 0))
            acc += classB;
        else
            acc += classC;
    else
        acc += classD;
```

In contrast to the hash map method, this approach consists of a lot of branches, thus it may result in poor performance due to wrong branch prediction. The cache should not be a problem here, though.

Branchless Decision Tree

The *branchless decision tree* works in a similar way to the upper one but it completely avoids if-clauses:

```
const bool c0 = h0 == 0;
const bool c1 = h1 == 0;
const bool c2 = hu0 >= 0;
```

4. Implementation

```

const bool c3 = hu1 >= 0;
const bool c4 = b0 >= 0;
const bool c5 = b1 >= 0;

acc += classB;

acc += (((c0 && c1) << 31) >> 31) & (classA - classB);
acc += (((c0 && !c1 && (c3 && c4 && !c5))
        || (!c0 && c1 && !(c4 || !c5 || c2))) << 31) >> 31)
        & (classC - classB);
acc += ((!c0 || c1) << 31) >> 31) & (classD - classB);

```

It adds `classB` to the accumulator `acc` by default. Following that, all conditions are tested which would require another class to be added to `acc`. If so, the difference between `classB` and the actual class is added to `acc`.

This method is branchless and cache efficient, but a lot of calculation has to be done.

Comparison

The speed of all three approaches was compared by executing each of them 10^{10} times with random data. The random data were generated in the same way as described in section 3.1. All tests were executed on the Linux Cluster of the Leibniz Rechenzentrum, on the one hand on an Intel Nehalem-EP (ICE-cluster), on the other hand on an AMD Magny-Cours (MPP-cluster). The processing time are shown in table 4.1.

Run	ICE		MPP			
	1	2	1	2	3	4
Hash Map	627.28	627.01	735.01	802.82	803.00	736.20
Decision Tree	723.27	722.87	726.45	788.61	788.75	726.51
Branchless Decision Tree	626.82	626.78	726.47	794.32	794.19	726.44

Table 4.1.: Processing times in seconds of 10^{10} executions of the listed CLASSIFICATION methods on ICE and MPP. The best performing methods are marked.

On ICE, both hash map and branchless decision tree perform best. On MPP, a conspicuous behavior can be noticed: Though the exact same program was executed four times, the processing times vary greatly. However, the first and fourth run, and the second and third run show similar processing times. The reason for this could not be spotted, it may be a difference between some nodes of the cluster, a background task which was executed at the same time on the same node, or a difference between the

cores of the CPU. Nevertheless, the decision tree method is a good choice on MPP.

The methods are further compared in section 5.1 using “real” instead of randomly generated data.

4.3.4. List Scheduling

One characteristic of the *MP algorithm* (section 3.2.1) is that it is possible to execute it in parallel for the most part. As already mentioned at the beginning of this section, the method `calculateLoadBalancing(...)` is to be called to initiate the load balancing calculation. In this method only the prefix sum is calculated since it is processed sequentially. All the other calculation is done in `getBlock(...)` which is called in parallel.

Each thread—let’s consider thread number k —needs to know its first and its last assigned block. It could calculate—say—the position of its last assigned block and wait for thread $k - 1$ to finish the calculation of the position of its last block. Thread k could then infer the position of its own first block. This would require synchronization between the threads which again would slow down the execution.

To ensure independent threads, each thread calculates the position of its first as well as of its last block (called *limits* in the following). This is done in `getBlock(k, j)`, where k denotes the thread number and j the j th block that is assigned to this thread. Since this method is called every time a new block is requested from a thread, some sort of caching mechanism is required which is realized via nested `std::pairs`:

```
std::vector< std::pair< bool, std::pair< int, int > > > limits;
```

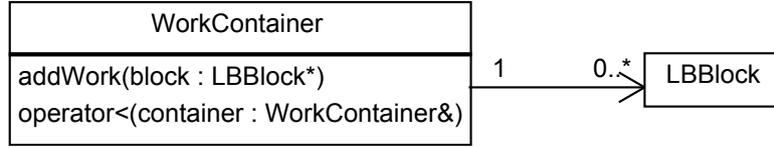
In `calculateLoadBalancing(...)`, for all i , `limits[i].first` is set to **false**. When `getBlock(...)` is called by thread k , it is checked whether `limits[k].first` is false. If so, the positions of the limits are calculated as described in section 3.2.1 and the position of the first block is stored in `limits[k].second.first` and the position of the block after the last one in stored in `limits[k].second.second`. Moreover, `limits[k].first` is set to true, indicating that the limits have already been calculated.

`getBlock(k, j)` then returns

$$\begin{cases} \text{limits}[k].\text{second}.first & \text{if } \text{limits}[k].\text{second}.first + j \\ & < \text{limits}[k].\text{second}.second, \\ \text{NULL} & \text{else} \end{cases}$$

4.3.5. Longest Processing Time

In contrast to list scheduling, the longest processing time algorithm cannot easily be parallelized, thus the load balancing has to be calculated in `calculateLoadBalancing(...)`.

Figure 4.2.: Class diagram of `WorkContainer`.

The mapping described in section 3.2.2 is realized by a vector of vectors:

```
std::vector< std::vector< LBlock * > > map
```

Moreover, a wrapper class `WorkContainer` (see fig. 4.2) adds additional functionality to each of the inner lists of `map`. It provides a method `addWork(LBlock *block)` to add a new block to the list and defines `operator<` to render sorting of the lists possible. It compares the sums of predicted processing times of two lists. It is important to note that `operator<` actually is the implementation of *greater*, because `std::priority_queue` uses `operator<` to sort its content in *descending* order but *ascending* order is required for our purpose (see section 3.2.2).

As the first step in the algorithm, `std::sort(...)` is used to sort the list of all blocks in *descending* order by their predicted processing time. After that, an instance of `std::priority_queue`, which uses aforementioned `operator<`, is filled with K `WorkContainers`. Finally, the ordered blocks are iterated and added one by one to the foremost element of the priority queue, i.e. the one containing the lowest sum of predicted processing times at this time.

`getBlock(k, j)` returns the j th element of the k th list in `map`, so

$$\begin{cases} \text{map}[k][j] & \text{if } j < \text{map}[k].\text{size}(), \\ \text{NULL} & \text{else} \end{cases}$$

which can be executed completely in parallel.

4.3.6. Implicit Longest Processing Time

The *implicit* longest processing time algorithm exploits the fact that the processors basically simulate the priority queue mentioned in the previous section: When a processor/thread k finishes its current block and therefore calls `getBlock(...)` to request a new block, k has to be the thread with the lowest total load at that time, i.e. the one which would be at the front of the priority queue.

In `calculateLoadBalancing(...)`, the blocks are sorted in descending order by their predicted processing time. Nothing else is done there. Each time a thread calls `getBlock(...)` to request a new block, `getBlock(...)` returns the foremost

4. Implementation

block which has not yet been assigned. In contrast to all other techniques, access to `getBlock(...)` needs to be synchronized using OpenMP's `critical` keyword, since all threads operate on the same list of blocks. Whereas the expensive calculation done in section 4.3.5 can be avoided, synchronization on the other hand may slow down the execution.

5. Comparison of Approaches

In this chapter we compare the processing time prediction and work-allocation approaches. We check how well they perform running on various numbers of processors, how well they scale and how much overhead is generated.

In the following the term *speedup* will be used frequently. The speedup S_K is defined as

$$S_K = \frac{T_1}{T_K} \quad (5.1)$$

where K is the number of processors, T_1 the processing time using only one processor and T_K the processing time using K processors. $S_K = K$ is called *linear speedup*, meaning that doubling the number of processors also doubles the speed. $S_K > K$ is denoted as *super linear speedup*, i.e. doubling the number of processors more than doubles the speed. [HP12]

As already mentioned in the introduction, the tests were executed on a dual socket octo core AMD Magny-Cours machine (called *MPP*) and a dual socket quad core Intel Nehalem machine (called *ICE*) of the Leibniz Rechenzentrum. The familiar scenario is used for all executions with a resolution of 6400×64 , so the domain is split into 6400 cells along the x -axis and 64 cells along the y -axis. Depending on the number of threads K , the domain is divided along the x -axis into $8K$ equally sized blocks.

The raw data of the following plots can be found in appendix C.

5.1. Comparison of Processing-Time-Prediction Approaches

Figures 5.1 and 5.2 show the speedups achieved in the tests on ICE and MPP respectively. Since we want to compare processing time prediction approaches in this section, the measurements are grouped by the work-allocation strategy. The blue bars represent the linear speedup which is our goal. Since all `TIME EXTRAPOLATION` strategy implementations achieve an almost equal speedup to `TIME` and all `CLASSIFICATION` implementations also perform with an almost equal speedup, they are merged using the average into one bar for `TIME` and one for `CLASSIFICATION` respectively.

In contrast, figs. 5.3 and 5.4 display the absolute processing time of the simulation. Here, the individual implementations of `TIME EXTRAPOLATION` and `CLASSIFICATION` show differences, thus they are not merged.

Only considering speedup, apart from implicit LPT, `TIME` performs best. However, using implicit LPT, `CLASSIFICATION` sometimes performs better. More precisely, on ICE, `TIME` is on average 4.05%, 0.41% and 0.43% faster than `CLASSIFICATION DECISION TREE` using list scheduling, LPT and implicit LPT respectively. On MPP, `TIME` is on average 10.51% and 0.21% faster and 0.03% slower than `CLASSIFICATION DECISION TREE` using list scheduling, LPT and implicit LPT, respectively. Since the speedup of a strategy is calculated on the basis of a single-threaded run, constant overhead is not visible in the chart. Constant overhead means that the overhead is always the same, independent of the number of processors. This can be seen at `CLASSIFICATION`: Looking at implicit LPT of fig. 5.1, `CLASSIFICATION` achieves a better speedup than `TIME` for 4 and 8 cores, but looking at fig. 5.3, the absolute processing time is either not better or even worse. This is caused by the significant overhead we can see in figs. 5.1 and 5.3 at the single-threaded executions using `CLASSIFICATION`. It results from the call to `aggregateCellParameters(...)` (see section 4.3.3) which has to be performed for every *cell* in the simulation domain, whereas all the other strategies only require a call of a method per *block*. The more cores are used, the more seldom `aggregateCellParameters(...)` is called by one core, so the overhead can be shared. `CLASSIFICATION` therefore *may* perform better than the other strategies when used on a system with more cores than 16.

Comparing the three `CLASSIFICATION` implementations, *decision tree* performs best on both systems which is a contradiction to the results of section 4.3.3. This is perhaps due to tests in section 4.3.3 performed with random data whereas these tests were performed using “real” data from the simulation.

None of the `TIME EXTRAPOLATION` implementations constantly outperform `TIME` (on average, they achieve a 0.045% lower absolute processing time on ICE and a 0.118% higher one on MPP), so it is not worth the trouble of using one of them.

How well the load is balanced can also be seen in figs. 5.5 and 5.6. These show how much greater the bottleneck value β is compared to the optimal bottleneck value β^* . In other words, they show the relative difference between the total processing time of one iteration divided by the number of threads and the processing time of the thread which finished last. For instance, if the plot shows 20%, the calculation of the last finished thread took 20% longer than the optimal bottleneck value. The plots confirm above’s observations. They also show that `CLASSIFICATION` sometimes performs really bad, for example in 5.6 for two cores with list scheduling, where the real bottleneck value is approximately 40% greater than the optimal one.

5. Comparison of Approaches

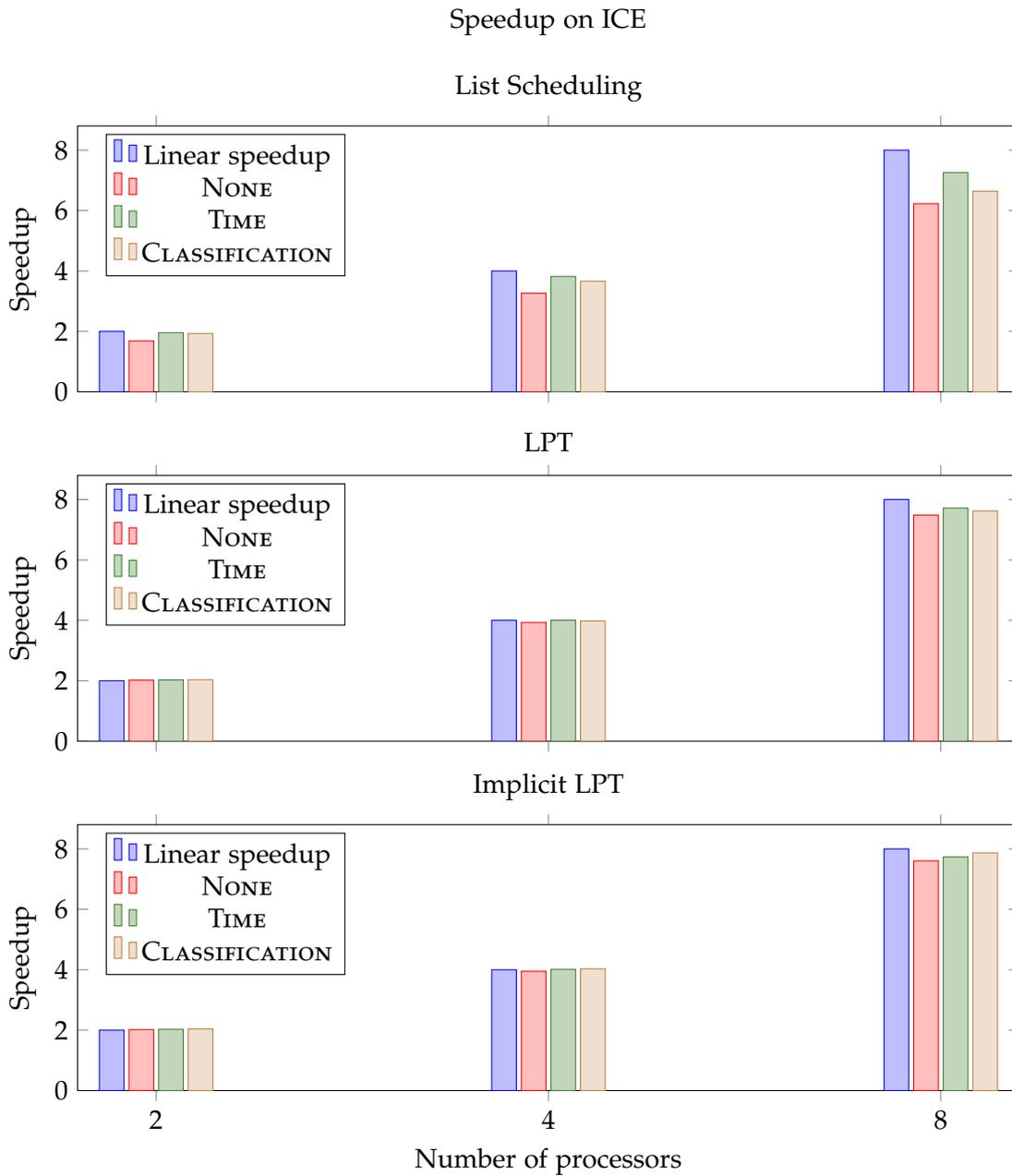


Figure 5.1.: Speedup comparison of processing time prediction approaches for 2, 4 and 8 cores on ICE for the entire simulation. The individual TIME (EXTRAPOLATION) and CLASSIFICATION strategies are merged since they show almost no difference.

5. Comparison of Approaches

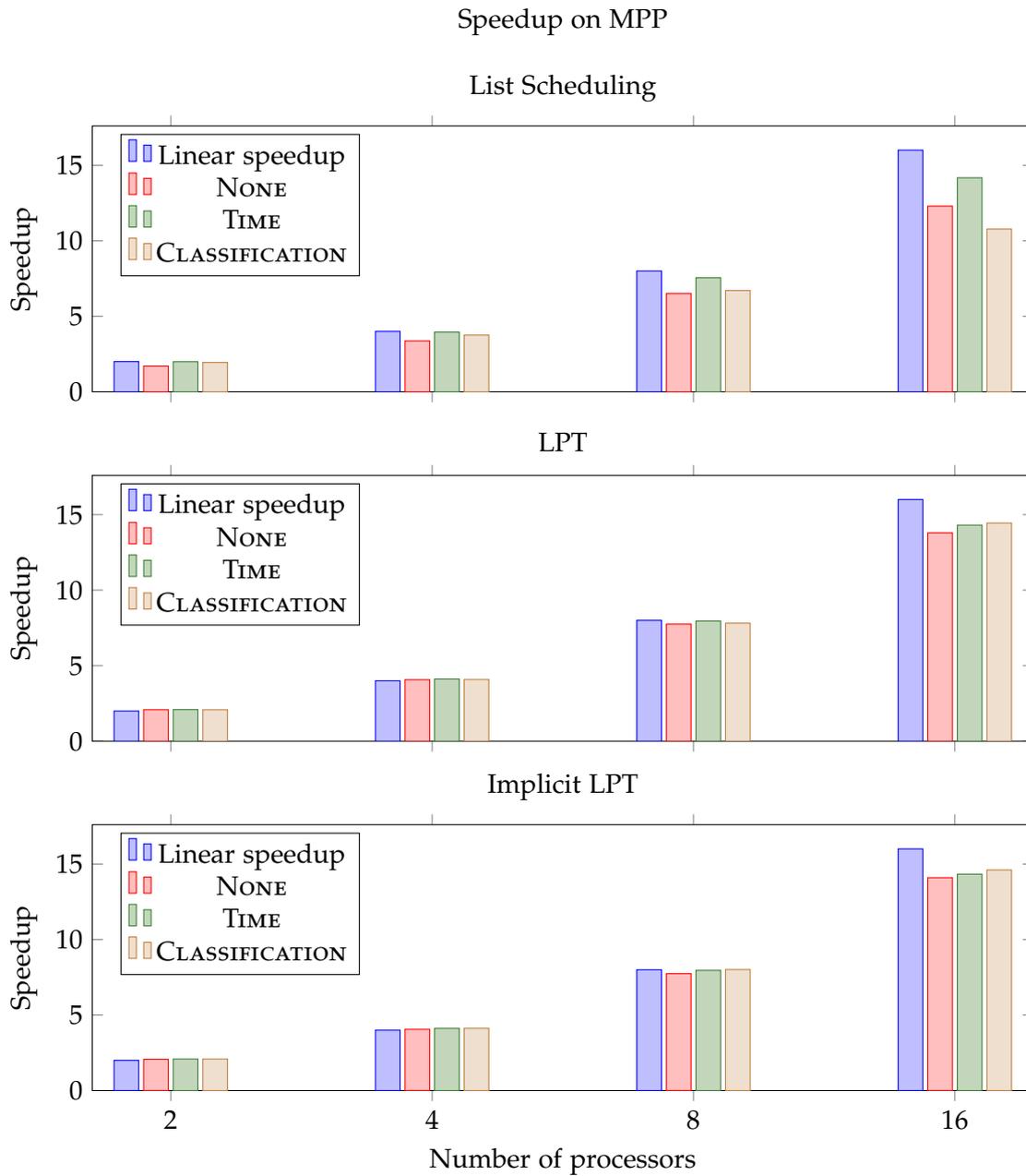


Figure 5.2.: Speedup comparison of processing time prediction approaches for 2, 4, 8 and 16 cores on MPP for the entire simulation. The individual `TIME` (`EXTRAPOLATION`) and `CLASSIFICATION` strategies are merged since they show almost no difference.

5. Comparison of Approaches

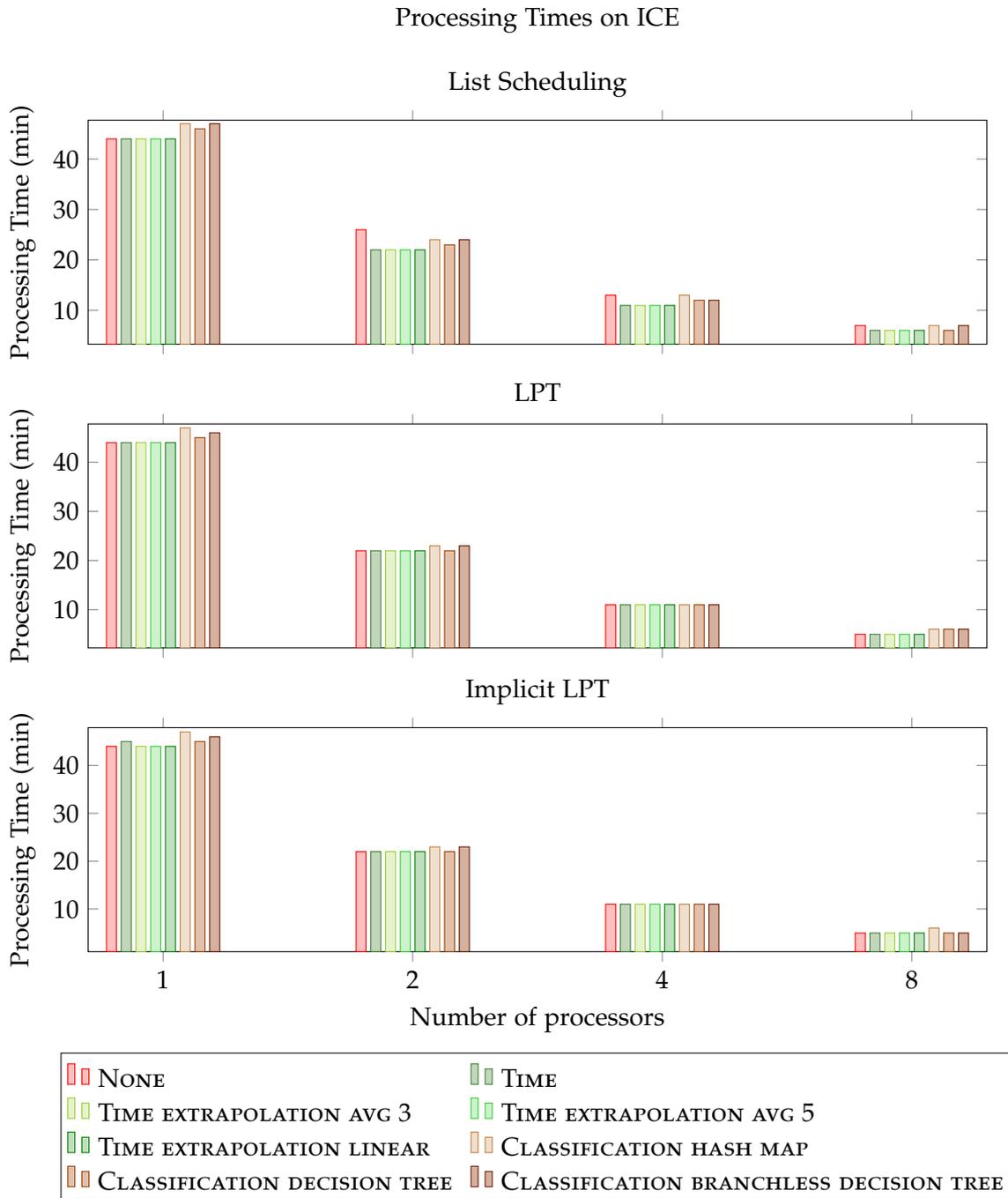


Figure 5.3.: Processing time comparison of processing time prediction approaches for 1, 2, 4 and 8 cores on ICE for the entire simulation.

5. Comparison of Approaches

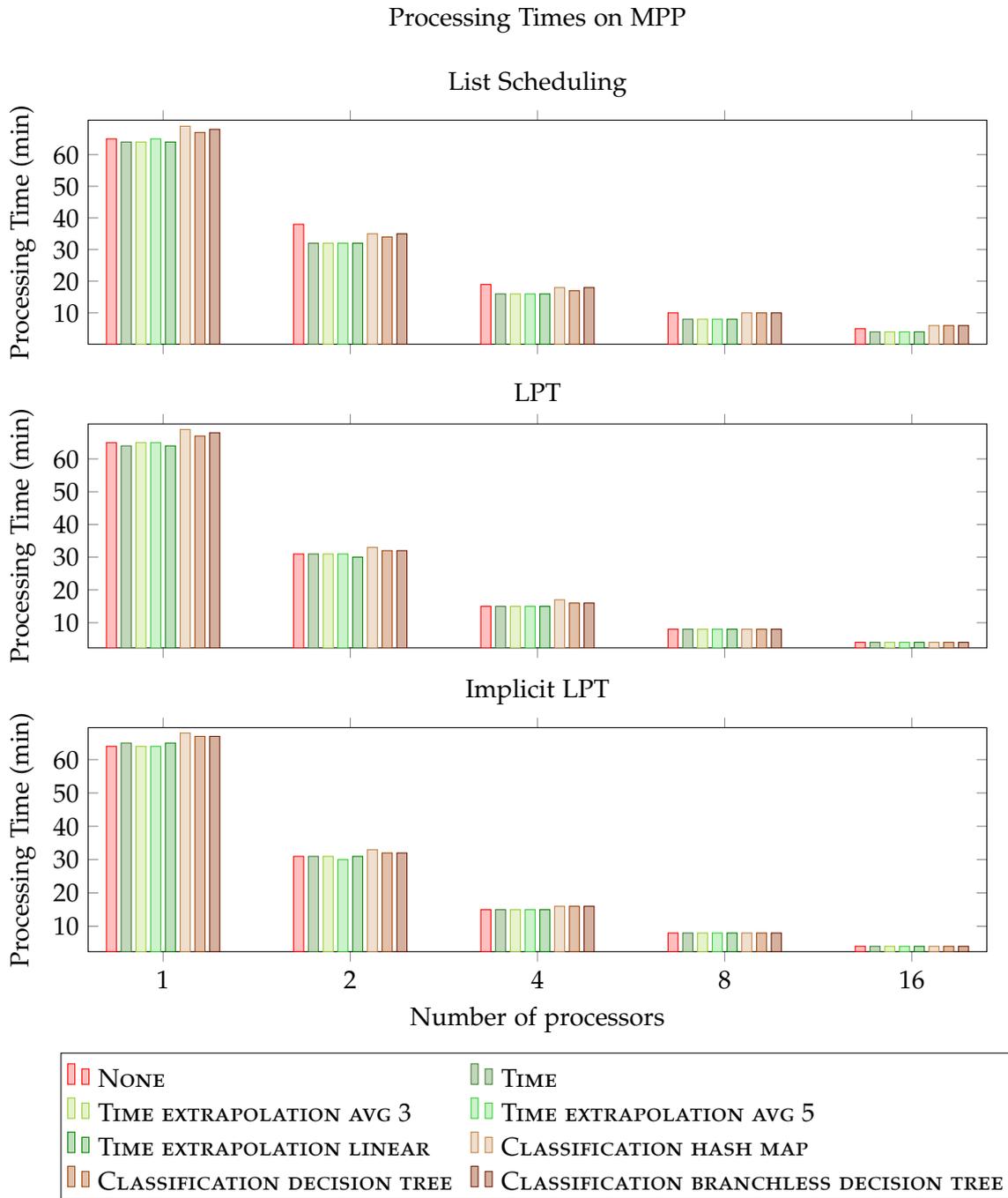


Figure 5.4.: Processing time comparison of processing time prediction approaches for 1, 2, 4, 8 and 16 cores on MPP for the entire simulation.

5. Comparison of Approaches

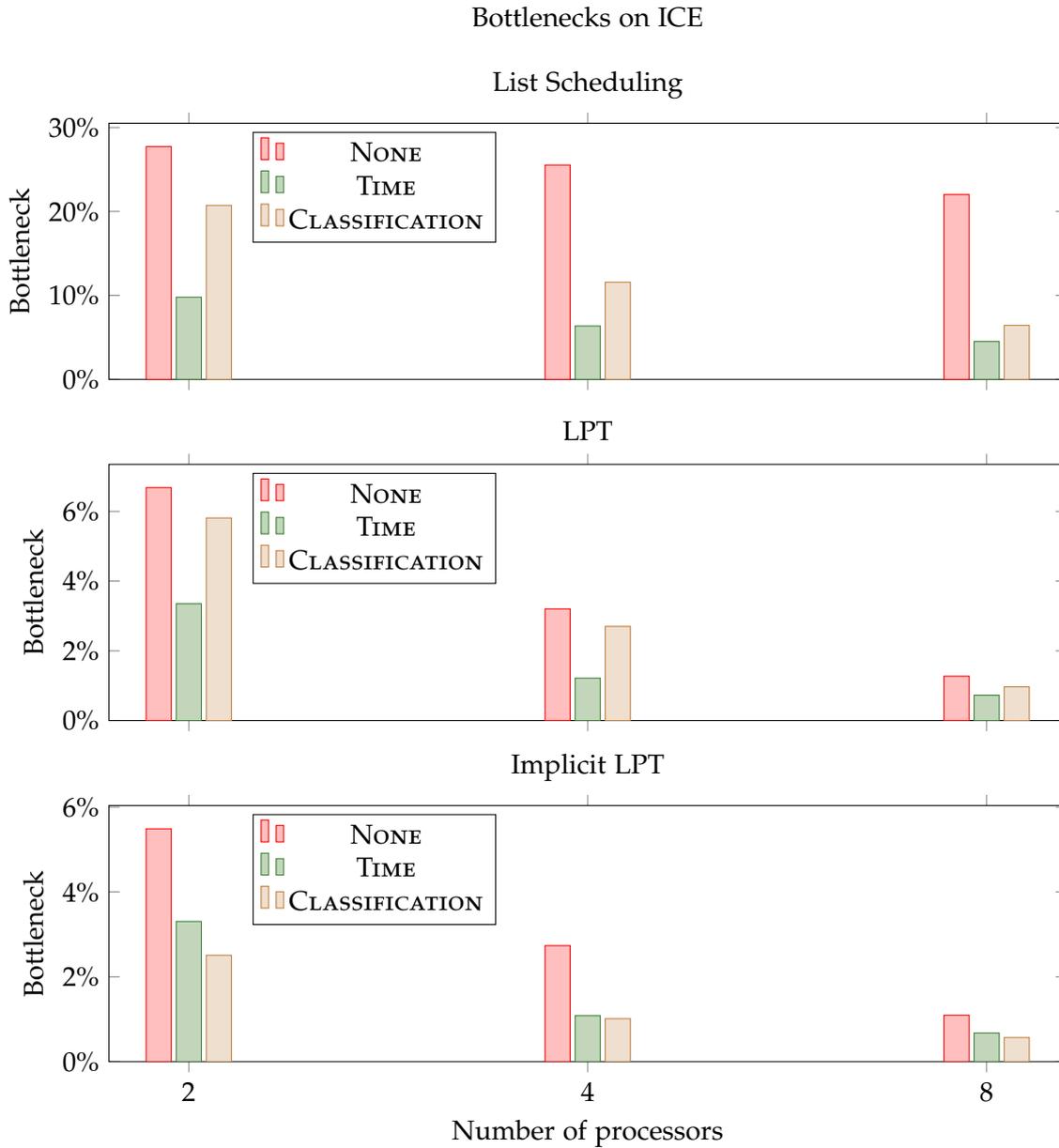


Figure 5.5.: This figure shows how much greater the bottleneck value β is compared to the optimal bottleneck value β^* . The displayed values are the average over the entire simulation executed on ICE. The individual TIME (EXTRAPOLATION) and CLASSIFICATION strategies are merged since they show almost no difference. The raw data can be found in appendix D.

5. Comparison of Approaches

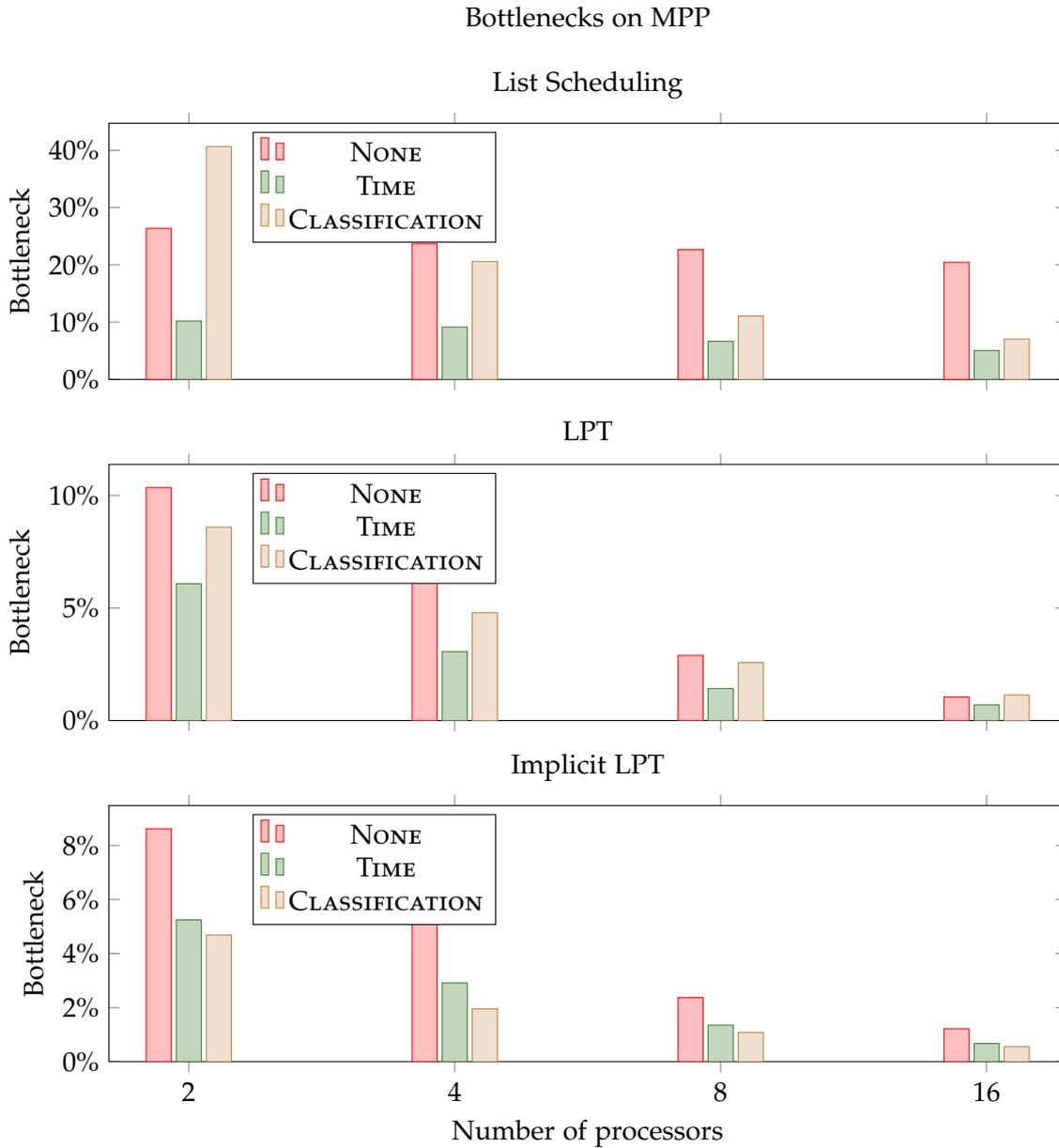


Figure 5.6.: This figure shows how much greater the bottleneck value β is compared to the optimal bottleneck value β^* . The displayed values are the average over the entire simulation executed on MPP. The individual TIME (EXTRAPOLATION) and CLASSIFICATION strategies are merged since they show almost no difference. The raw data can be found in appendix D.

5.2. Comparison of Work-Allocation Approaches

Figure 5.7 compares the speedup achieved with the work-allocation approaches. Again, the blue bar denotes linear speedup. Figure 5.8 displays their absolute processing times. Only `TIME` and `CLASSIFICATION DECISION TREE` are used here, since they were proven suitable in section 5.1.

Implicit LPT turns out to be the fastest in almost all situations concerning speedup as well as absolute processing times. It sometimes even achieves super linear speedup which may be caused by caching effects: The more cores (or processors, since we operate on dual-socket machines) are used, the bigger the cache is. So with fewer cores the data may not fit into the cache and (expensive) cache misses can be encountered, whereas with more cores the data may fit. [HP12]

LPT is generally slightly slower (on average approximately 0.52% higher absolute processing time and 0.62% lower speedup) than implicit LPT. Even if both algorithms are very similar, they differ in important points: First of all, implicit LPT only has to sort the blocks, LPT also has to calculate the actual allocation sequentially. Secondly, implicit LPT relies not only on the *predicted* processing times (which may be wrong), but it also includes the *actual* processing times in the calculation, since it requests a new block as soon as a core finishes its current block. LPT calculates the load balancing based only on predicted processing times. Thirdly, implicit LPT has to use one more OpenMP's critical region than LPT which can slow down the simulation. In our environment and when simulating our scenario, the first two points seem to outweigh the last most of the time. This may be different in other environments or when simulating different scenarios.

List scheduling usually performs worse than implicit LPT and LPT—on average approximately 7.46% higher absolute processing time and 6.56% lower speedup than implicit LPT. This is caused by the constraint which is imposed on list scheduling: The blocks can not be reordered. Nevertheless, there are advantages of list scheduling: For the most part, list scheduling can be calculated in parallel, therefore its complexity is $\mathcal{O}(N + \log N)$ (see section 3.2.1). LPT and implicit LPT, in contrast, have a complexity of $\mathcal{O}(N \log(K + N))$ (see section 3.2.2) and $\mathcal{O}(N \log N)$ (one sort of N blocks), respectively. Thus, when handling a large amount of cores and blocks, the cost of the load balancing calculation of implicit LPT and LPT may counteract the advantages of the better load balancing they achieve.

The aforementioned constraint can also be considered as a benefit: LPT and implicit LPT can freely assign blocks to arbitrary processors. This can be a disadvantage on NUMA architectures, when the assigned blocks are placed within the “wrong” memory and therefore cause slow memory accesses. Using list scheduling, most of the blocks stay at the same processor for a long time.

In summary, a clear winner is not discernible. LPT and implicit LPT perform almost equally well, and list scheduling may be the better choice on systems with many processors/cores and simulations with many blocks. Therefore, it depends on the simulation scenario and the machines on which the simulation is executed.

5.3. Best Combination

When we cover all possible combinations of processing time prediction and work-allocation strategies, tables 5.1 and 5.2 show the best combinations for a certain number of cores on a certain system, once comparing the absolute processing time, once comparing the speedup.

With respect to the absolute processing time (table 5.1), implicit LPT combined with a `TIME EXTRAPOLATION` strategy appears most often in the table, with respect to speedup, implicit LPT combined with a `CLASSIFICATION` strategy appears most often.

In section 5.1 we decided to stick with `TIME` instead of all the `TIME EXTRAPOLATION` strategies. Even if tables 5.1 and 5.2 indicate otherwise (the `TIME EXTRAPOLATION` strategies are often referred to), this decision is still reasonable: On ICE, `TIME` is only 0.50%, 0.14%, 0.01% and 0.003% slower (with respect to the absolute processing time) than the optimal processing time prediction approach with 1, 2, 4 and 8 cores, respectively. On MPP, it is only 0.54%, 0.05%, 0.03% and 1.46% slower with 2, 4, 8 and 16 cores, respectively.

In summary, implicit LPT combined with `TIME` is a good choice for a low absolute processing time, and implicit LPT combined with `classification decision tree` is a good choice for a high speedup. This is—as already mentioned—only valid for this thesis' scenario and when executed on ICE or MPP. Other scenarios and systems may behave different.

5. Comparison of Approaches

Speedup on ICE and MPP

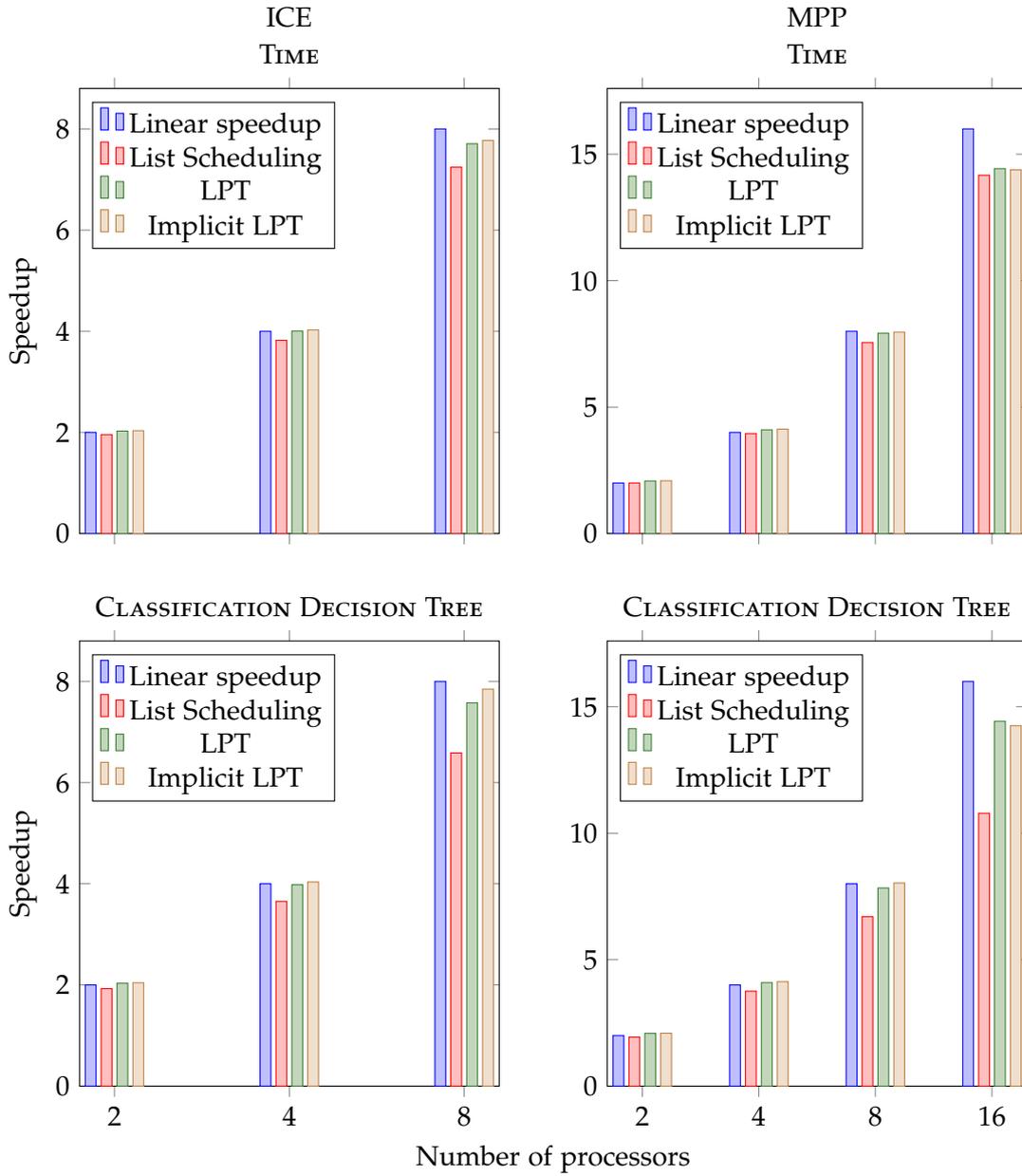


Figure 5.7.: Speedup comparison of work-allocation approaches on ICE and MPP for the entire simulation using TIME and CLASSIFICATION DECISION TREE strategies since in section 5.1 those were proven suitable.

5. Comparison of Approaches

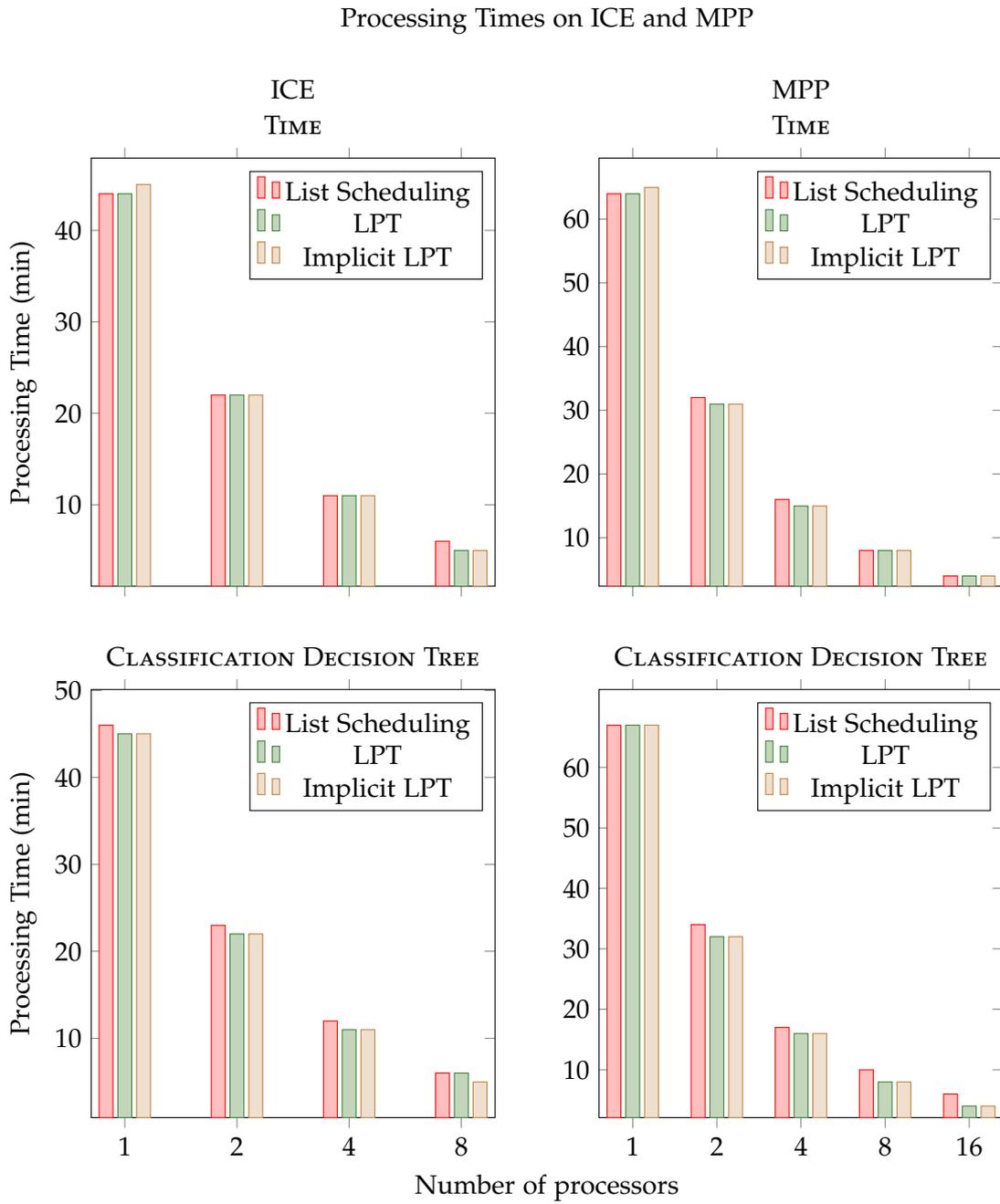


Figure 5.8.: Processing time comparison of work-allocation approaches on ICE and MPP for the entire simulation using TIME and CLASSIFICATION DECISION TREE strategies since in section 5.1 those were proven suitable.

5. Comparison of Approaches

K	ICE			MPP		
	Strategy	Time	Gain	Strategy	Time	Gain
1	LPT NONE	00:44:54.67	0.03%	ILPT NONE	01:04:25.75	1.22%
2	LPT TEA3	00:22:08.14	17.00%	ILPT TEA5	00:30:57.60	16.00%
4	ILPT TEL	00:11:11.53	18.74%	ILPT TEL	00:15:44.61	18.62%
8	ILPT TEL	00:05:47.87	19.61%	ILPT TEL	00:08:08.41	18.80%
16				ILPT TEA3	00:04:27.40	16.00%

Table 5.1.: The table shows the best performing combinations (with respect to the absolute processing time) of processing time prediction and work-allocation approaches when the simulation executed with a certain number of cores and either on ICE or MPP. *Time* denotes the absolute processing time and *Gain* says, how much a combination performs better than *no load balancing*. Abbreviations: *K*: Number of Cores; **ILPT**: Implicit LPT; **TEA3**: TIME EXTRAPOLATION AVG 3; **TEA5**: TIME EXTRAPOLATION AVG 5; **TEL**: TIME EXTRAPOLATION LINEAR.

K	ICE			MPP		
	Strategy	Speedup	Gain	Strategy	Time	Gain
2	ILPT CHM	2.04	26.60%	LPT TEA5	2.10	22.66%
4	ILPT CBDT	4.04	23.97%	ILPT CDT	4.13	22.52%
8	ILPT CDT	7.89	19.61%	ILPT CDT	8.03	23.43%
16				ILPT CHM	14.78	20.24%

Table 5.2.: The table shows the best performing combinations (with respect to speedup) of processing time prediction and work-allocation approaches when the simulation executed with a certain number of cores and either on ICE or MPP. *Time* denotes the absolute processing time and *Gain* says, how much a combination performs better than *no load balancing*. Abbreviations: *K*: Number of Cores; **ILPT**: Implicit LPT; **TEA5**: TIME EXTRAPOLATION AVG 5; **CHM**: CLASSIFICATION HASH MAP; **CDT**: CLASSIFICATION DECISION TREE; **CBDT**: CLASSIFICATION BRANCHLESS DECISION TREE.

6. Conclusion

In this thesis we compared several approaches to achieve load balancing which is as good as possible. We introduced a wrapper which handles the load balancing and makes it easy to add some other approaches. We defined three major types of processing time prediction approaches: `TIME`, `TIME EXTRAPOLATION` and `CLASSIFICATION`. `TIME EXTRAPOLATION` and `CLASSIFICATION` were further refined by different strategies or implementations. Additionally, we defined three types of work-allocation methods: List scheduling, LPT and implicit LPT. Finally, all combinations of processing time prediction and work-allocation approaches were examined.

Even if `TIME` combined with implicit LPT is the best choice for this thesis' simulation scenario executed on ICE or MPP with up to 16 cores, other combinations may be a better choice for other scenarios or on other systems. `CLASSIFICATION` comes with a high calculation overhead which may be reduced when executed on more than 16 cores. LPT and implicit LPT work well on few cores but may be expensive on systems with many more cores, where list scheduling may become a better choice.

In summary, further research on systems with more cores and processors is required. In this thesis only OpenMP was used, interaction with other interfaces like MPI also needs to be tested.

A. Scenario

The scenario used in this thesis is specified as follows:

- End time: 50
- Border positions:
 - Left: 0
 - Right: 1000
 - Top: 10
 - Bottom: 0
- Boundary types: walls
- Bathymetry:

$$b = \begin{cases} -20 + 100 \sin\left(\frac{\pi(600-x)^4}{2.592 \times 10^{11}}\right) & \text{if } x \in [0, 500), \\ -20 & \text{if } x \in [500, 1000] \end{cases}$$

- Water height:

$$h = \begin{cases} 20 - 100 \sin\left(\frac{\pi(600-x)^4}{2.592 \times 10^{11}}\right) & \text{if } x \in [0, 500), \\ 20 & \text{if } x \in [500, 950], \\ 50 & \text{if } x \in (950, 1000] \end{cases}$$

B. Classification Array

The classification array which is used in section 4.3.3, maps the conditions $h_0 = 0, h_1 = 0, (hx)_0 > 0, (hx)_1 > 0, b_0 > 0, b_1 > 0$ to a processing time prediction.

B. Classification Array

Pos	$b_1 \geq 0$	$b_0 \geq 0$	$hx_1 \geq 0$	$hx_0 \geq 0$	$h_1 \geq 0$	$h_0 \geq 0$	\hat{W}	Pos	$b_1 \geq 0$	$b_0 \geq 0$	$hx_1 \geq 0$	$hx_0 \geq 0$	$h_1 \geq 0$	$h_0 \geq 0$	\hat{W}
0	0	0	0	0	0	0	46	32	1	0	0	0	0	0	46
1	0	0	0	0	0	1	25	33	1	0	0	0	0	1	25
2	0	0	0	0	1	0	25	34	1	0	0	0	1	0	35
3	0	0	0	0	1	1	5	35	1	0	0	0	1	1	5
4	0	0	0	1	0	0	46	36	1	0	0	1	0	0	46
5	0	0	0	1	0	1	25	37	1	0	0	1	0	1	25
6	0	0	0	1	1	0	25	38	1	0	0	1	1	0	25
7	0	0	0	1	1	1	5	39	1	0	0	1	1	1	5
8	0	0	1	0	0	0	46	40	1	0	1	0	0	0	46
9	0	0	1	0	0	1	25	41	1	0	1	0	0	1	25
10	0	0	1	0	1	0	25	42	1	0	1	0	1	0	35
11	0	0	1	0	1	1	5	43	1	0	1	0	1	1	5
12	0	0	1	1	0	0	46	44	1	0	1	1	0	0	46
13	0	0	1	1	0	1	25	45	1	0	1	1	0	1	25
14	0	0	1	1	1	0	25	46	1	0	1	1	1	0	25
15	0	0	1	1	1	1	5	47	1	0	1	1	1	1	5
16	0	1	0	0	0	0	46	48	1	1	0	0	0	0	46
17	0	1	0	0	0	1	25	49	1	1	0	0	0	1	25
18	0	1	0	0	1	0	25	50	1	1	0	0	1	0	25
19	0	1	0	0	1	1	5	51	1	1	0	0	1	1	5
20	0	1	0	1	0	0	46	52	1	1	0	1	0	0	46
21	0	1	0	1	0	1	25	53	1	1	0	1	0	1	25
22	0	1	0	1	1	0	25	54	1	1	0	1	1	0	25
23	0	1	0	1	1	1	5	55	1	1	0	1	1	1	5
24	0	1	1	0	0	0	46	56	1	1	1	0	0	0	46
25	0	1	1	0	0	1	35	57	1	1	1	0	0	1	25
26	0	1	1	0	1	0	25	58	1	1	1	0	1	0	25
27	0	1	1	0	1	1	5	59	1	1	1	0	1	1	5
28	0	1	1	1	0	0	46	60	1	1	1	1	0	0	46
29	0	1	1	1	0	1	35	61	1	1	1	1	0	1	25
30	0	1	1	1	1	0	25	62	1	1	1	1	1	0	25
31	0	1	1	1	1	1	5	63	1	1	1	1	1	1	5

Table B.1.: Array which maps a combination of conditions to a processing time prediction.

C. Raw Data for Comparison

In the following tables the raw data are displayed which were used to to create figs. 5.1 to 5.4, 5.7 and 5.8.

Table C.1.: Measured processing times and calculated speedups of test runs on ICE.

This data were used to generate the charts in chapter 5.

Abbreviations: **K**: Number of Cores; **List**: List Scheduling; **ILPT**: Implicit LPT; **TEA3**: TIME EXTRAPOLATION AVG 3; **TEA5**: TIME EXTRAPOLATION AVG 5; **CHM**: CLASSIFICATION HASH MAP; **CDT**: CLASSIFICATION DECISION TREE; **CBDT**: CLASSIFICATION BRANCHLESS DECISION TREE.

	K	Processing Time			Speedup		
		List	LPT	ILPT	List	LPT	ILPT
NONE	8	00:07:12.75	00:05:59.93	00:05:54.66	6.23	7.49	7.60
	4	00:13:46.36	00:11:25.69	00:11:22.84	3.26	3.93	3.95
	2	00:26:40.34	00:22:14.12	00:22:16.11	1.68	2.02	2.02
	1	00:44:55.58	00:44:54.67	00:44:57.10	1.00	1.00	1.00
TIME	8	00:06:11.95	00:05:49.67	00:05:47.88	7.24	7.71	7.77
	4	00:11:45.04	00:11:12.99	00:11:11.61	3.82	4.00	4.03
	2	00:22:56.83	00:22:09.99	00:22:09.03	1.96	2.03	2.03
	1	00:44:54.30	00:44:54.95	00:45:03.55	1.00	1.00	1.00
TEA3	8	00:06:10.33	00:05:49.07	00:05:48.55	7.28	7.72	7.72
	4	00:11:45.59	00:11:13.49	00:11:12.49	3.82	4.00	4.00
	2	00:22:57.54	00:22:08.14	00:22:09.53	1.96	2.03	2.02
	1	00:44:54.44	00:44:54.44	00:44:50.05	1.00	1.00	1.00
TEA5	8	00:06:11.55	00:05:49.00	00:05:49.71	7.25	7.72	7.69
	4	00:11:46.37	00:11:12.84	00:11:12.06	3.82	4.01	4.00
	2	00:22:56.04	00:22:09.03	00:22:09.06	1.96	2.03	2.02
	1	00:44:54.88	00:44:54.92	00:44:50.20	1.00	1.00	1.00
TEL	8	00:06:10.96	00:05:49.12	00:05:47.87	7.26	7.72	7.75
	4	00:11:45.20	00:11:13.04	00:11:11.53	3.82	4.00	4.02
	2	00:22:56.54	00:22:09.68	00:22:08.20	1.96	2.03	2.03
	1	00:44:54.53	00:44:54.01	00:44:56.72	1.00	1.00	1.00

Continued on next page...

C. Raw Data for Comparison

Table C.1.: (Continued)

	K	Processing Time			Speedup		
		List	LPT	ILPT	List	LPT	ILPT
CHM	8	00:07:09.98	00:06:14.22	00:06:02.45	6.67	7.72	7.89
	4	00:13:01.06	00:11:58.95	00:11:52.02	3.67	4.00	4.01
	2	00:24:43.50	00:23:25.20	00:23:21.95	1.93	2.03	2.04
	1	00:47:47.89	00:47:45.00	00:47:38.43	1.00	1.00	1.00
CDT	8	00:06:59.29	00:06:04.27	00:05:51.53	6.59	7.58	7.85
	4	00:12:36.11	00:11:33.31	00:11:23.73	3.65	3.98	4.03
	2	00:23:53.82	00:22:37.32	00:22:29.52	1.93	2.03	2.04
	1	00:46:01.62	00:45:59.99	00:45:58.82	1.00	1.00	1.00
CBDT	8	00:07:03.64	00:06:08.80	00:05:57.52	6.66	7.63	7.87
	4	00:12:50.49	00:11:48.24	00:11:36.21	3.66	3.97	4.04
	2	00:24:21.03	00:23:03.84	00:23:04.07	1.93	2.03	2.03
	1	00:47:01.14	00:46:54.64	00:46:55.44	1.00	1.00	1.00

Table C.2.: Measured processing times and calculated speedups of test runs on MPP.

This data were used to generate the charts in chapter 5.

Abbreviations: **K**: Number of Cores; **List**: List Scheduling; **ILPT**: Implicit LPT; **TEA3**: TIME EXTRAPOLATION AVG 3; **TEA5**: TIME EXTRAPOLATION AVG 5; **CHM**: CLASSIFICATION HASH MAP; **CDT**: CLASSIFICATION DECISION TREE; **CBDT**: CLASSIFICATION BRANCHLESS DECISION TREE.

	K	Processing Time			Speedup		
		List	LPT	ILPT	List	LPT	ILPT
NONE	16	00:05:18.33	00:04:42.85	00:04:34.23	12.29	13.79	14.10
	8	00:10:01.48	00:08:23.22	00:08:19.20	6.51	7.75	7.74
	4	00:19:20.76	00:15:57.03	00:15:52.76	3.37	4.08	4.06
	2	00:38:09.00	00:31:13.45	00:31:07.88	1.71	2.08	2.07
	1	01:05:13.32	01:05:01.20	01:04:25.75	1.00	1.00	1.00
TIME	16	00:04:35.06	00:04:28.81	00:04:31.30	14.17	14.43	14.38
	8	00:08:35.97	00:08:09.49	00:08:09.76	7.55	7.92	7.97
	4	00:16:25.26	00:15:45.53	00:15:45.11	3.96	4.10	4.13
	2	00:32:27.67	00:31:04.60	00:31:07.60	2.00	2.08	2.09
	1	01:04:57.58	01:04:38.89	01:05:01.54	1.00	1.00	1.00
TEA3	16	00:04:34.89	00:04:31.70	00:04:27.40	14.13	14.39	14.53
	8	00:08:35.97	00:08:10.02	00:08:08.65	7.53	7.98	7.95

Continued on next page...

C. Raw Data for Comparison

Table C.2.: (Continued)							
K	Processing Time			Speedup			
	List	LPT	ILPT	List	LPT	ILPT	
	4	00:16:27.00	00:15:46.84	00:15:45.51	3.94	4.13	4.11
	2	00:32:37.36	00:31:08.14	00:31:05.43	1.98	2.09	2.08
	1	01:04:44.71	01:05:10.02	01:04:46.28	1.00	1.00	1.00
TEA5	16	00:04:35.33	00:04:35.00	00:04:32.51	14.17	14.19	14.26
	8	00:08:34.36	00:08:10.18	00:08:10.03	7.59	7.96	7.93
	4	00:16:24.31	00:15:46.77	00:15:46.21	3.96	4.12	4.11
	2	00:32:31.23	00:31:01.24	00:30:57.60	2.00	2.10	2.09
	1	01:05:01.98	01:05:03.03	01:04:47.26	1.00	1.00	1.00
TEL	16	00:04:33.12	00:04:33.82	00:04:35.95	14.21	14.21	14.14
	8	00:08:35.42	00:08:09.70	00:08:08.41	7.53	7.95	7.99
	4	00:16:23.19	00:15:46.63	00:15:44.61	3.95	4.11	4.13
	2	00:32:34.92	00:30:59.89	00:31:02.02	1.99	2.09	2.10
	1	01:04:41.74	01:04:52.01	01:05:01.16	1.00	1.00	1.00
CHM	16	00:06:25.95	00:04:47.11	00:04:39.65	10.78	14.42	14.78
	8	00:10:18.48	00:08:50.00	00:08:37.31	6.72	7.81	7.99
	4	00:18:23.00	00:17:00.09	00:16:45.16	3.77	4.06	4.11
	2	00:35:36.43	00:33:23.26	00:33:00.22	1.95	2.07	2.09
	1	01:09:18.61	01:09:00.74	01:08:53.73	1.00	1.00	1.00
CDT	16	00:06:13.85	00:04:40.68	00:04:43.04	10.79	14.43	14.25
	8	00:10:01.91	00:08:36.88	00:08:22.26	6.70	7.83	8.03
	4	00:17:53.80	00:16:29.19	00:16:16.48	3.76	4.09	4.13
	2	00:34:39.67	00:32:19.50	00:32:06.78	1.94	2.09	2.09
	1	01:07:13.69	01:07:29.26	01:07:13.50	1.00	1.00	1.00
CBDT	16	00:06:19.22	00:04:43.87	00:04:35.58	10.76	14.48	14.78
	8	00:10:09.74	00:08:47.39	00:08:27.87	6.69	7.79	8.02
	4	00:18:08.94	00:16:42.16	00:16:27.55	3.75	4.10	4.12
	2	00:35:03.64	00:32:50.01	00:32:30.44	1.94	2.09	2.09
	1	01:08:01.67	01:08:30.58	01:07:53.16	1.00	1.00	1.00

D. Bottleneck Values

	K	NONE	TIME	TEA3	TEA5	TEL	CHM	CDT	CBDT	System	
List	8	27.74	9.83	9.83	9.80	9.72	20.37	21.11	20.71	ICE	
	4	25.53	6.29	6.42	6.43	6.33	11.41	11.81	11.52		
	2	22.01	4.51	4.54	4.46	4.55	6.32	6.53	6.47		
LPT	8	6.68	3.34	3.35	3.37	3.34	5.71	5.93	5.79		
	4	3.20	1.18	1.20	1.24	1.25	2.67	2.68	2.76		
	2	1.27	0.72	0.73	0.71	0.74	0.98	0.98	0.94		
LPT	8	5.49	3.27	3.31	3.32	3.31	2.47	2.55	2.51		
	4	2.73	1.07	1.08	1.11	1.07	1.00	1.02	1.02		
	2	1.10	0.68	0.69	0.69	0.65	0.56	0.60	0.55		
List	16	26.36	10.12	10.12	9.99	10.52	40.97	40.51	40.49		MPP
	8	23.70	9.20	9.17	9.08	9.12	20.28	20.80	20.68		
	4	22.65	6.70	6.65	6.70	6.60	10.82	11.10	11.24		
	2	20.44	5.04	5.04	4.98	5.11	7.05	7.01	7.08		
LPT	16	10.35	5.89	6.16	5.72	6.54	8.83	8.48	8.46		
	8	6.33	3.07	3.04	3.06	3.08	4.75	4.81	4.82		
	4	2.89	1.42	1.42	1.41	1.44	2.55	2.60	2.57		
	2	1.05	0.71	0.68	0.70	0.69	1.15	1.11	1.16		
List	16	8.62	5.38	5.16	5.19	5.26	4.76	4.67	4.63		
	8	5.16	2.90	2.93	2.89	2.93	1.96	1.94	1.96		
	4	2.37	1.36	1.35	1.33	1.36	1.07	1.10	1.08		
	2	1.22	0.65	0.69	0.68	0.67	0.54	0.57	0.56		

Table D.1.: All values in *percent*. The table shows the relative difference between the actual bottleneck value β and the optimal bottleneck value β^* . These are the raw data of figs. 5.5 and 5.6

Bibliography

- [BB12] A. Breuer and M. Bader. “Teaching Parallel Programming Models on a Shallow-Water Code.” In: *Parallel and Distributed Computing, International Symposium on* (2012), pp. 301–308. DOI: <http://doi.ieeecomputersociety.org/10.1109/ISPDC.2012.48>.
- [Cor+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*. Vol. 2. MIT press Cambridge, 2001.
- [GM96] M. Grigni and F. Manne. “On the complexity of the generalized block distribution.” In: *Parallel Algorithms for Irregularly Structured Problems*. Springer, 1996, pp. 319–326.
- [Gra69] R. L. Graham. “Bounds on multiprocessing timing anomalies.” In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.
- [HP12] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.
- [LeV02] R. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.
- [LLN08] B. Levin, B. W. Levin, and M. Nosov. *Physics of tsunamis*. Springer, 2008.
- [MAT14] MATLAB. *Statistics Toolbox User’s Guide*. Version 9.0. The MathWorks, Inc., 2014.
- [MP97] S. Miguet and J.-M. Pierson. “Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing.” In: *High-Performance Computing and Networking*. Springer. 1997, pp. 550–564.
- [Nic94] D. M. Nicol. “Rectilinear partitioning of irregular data parallel computations.” In: *Journal of Parallel and Distributed Computing* 23.2 (1994), pp. 119–134.
- [Ope11] OpenMP. *Application Program Interface*. Version 3.1. OpenMP Architecture Review Board, 2011.
- [PA04] A. Pinar and C. Aykanat. “Fast optimal load balancing algorithms for 1D partitioning.” In: *Journal of Parallel and Distributed Computing* 64.8 (2004), pp. 974–996.