



## **PfiT - The User's Guide**

Documentation of the PfiT Reports and the raw metrics  
Best practices

January 14, 2020

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG)  
Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)  
KO 3394/14-1, OL 241/3-1, RE 1389/9-1, VO 1262/1-1, YA 191/10-1



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>User's Guide</b>	<b>7</b>
2.1	Explanation of the metrics of the reports . . . . .	7
2.1.1	Text report . . . . .	7
2.1.2	PDF report . . . . .	16
2.2	Sources of metrics . . . . .	16
2.3	Best practices for users . . . . .	22
2.3.1	Best practices . . . . .	22
2.3.2	General recommendations and checklists . . . . .	26
2.3.3	Using accelerators( OpenACC) . . . . .	36
2.3.4	OpenMP . . . . .	40
2.3.5	CUDA . . . . .	43
2.3.6	MPI . . . . .	56
2.3.7	C source 5 point stencil . . . . .	63
	Bibliography . . . . .	70
<b>A</b>	<b>Example PDF Report</b>	<b>73</b>



# Chapter 1

## Introduction

High-Performance-Computing (HPC) has become a standard research tool in many scientific disciplines. Research without at least supporting HPC calculations is becoming increasingly rare in the natural and engineering sciences. On top of that, new disciplines are discovering HPC as an asset to their research, for example in the areas of bioinformatics and social sciences. This means that more and more scientists without a deep understanding of the architecture and the functioning of such systems start using HPC resources. This knowledge gap is further enlarged as the complexity of HPC resources increases and gains significant importance in the field of performance engineering.

Most scientists that are new to HPC run their applications on local Tier 3 systems and are satisfied if their research problem can be solved on an available system in an acceptable time frame. The missing knowledge with respect to performance measurements will often lead to a lock-in, because they are not able to scale their calculations to a Tier 2 or Tier 1 compute resource. At the same time, Tier 3 compute centers typically lack sufficient human resources to work with each user individually on application performance. In order to increase awareness for performance issues and enable users to assess possible gains from performance improving measures, systematic, unified, and easily understandable information on performance parameters should be provided across all scientific communities. This especially pertains to the performance parameters of HPC jobs and the importance of performance engineering since HPC cluster are very expensive resources. This applies to the procurement of the hard- and software as well as for the service of the system (especially the power supply), in which e.g. the overall energy costs amount is located in a five year life cycle in the order of several hundred thousand euros for a typical Tier 3 system. To reduce these costs, jobs should have for example lower run time to save energy or less waiting time until starting the job for better utilization of the cluster. These goals can e.g. be reached, if the user gets insight into the runtime behaviour of its program for example via different kinds of reports, including additional automatic evaluation, interpretation and presentation of the performance metric values.

Although the usage of many performance measurement tools is mostly straightforward, the serious disadvantage is the missing explanation of the results in a clear and simple manner. Often the interpretation of generated reports requires expert knowledge – this makes the profiling for normal cluster users nearly useless since they usually don't have

the background to interpret the results. By automatically assembling all data provided by available tools into a single centrally organized framework it will be much easier for the user (and for administrators, too) to identify potential performance bottlenecks or pathological cases. In addition, with the help of an all-in-one monitoring, profiling and reporting tool, the user acceptance for code optimizations might be drastically increased, especially when the code tuning shows a considerable performance boost. Additionally, the interface may also help to overcome the gap of understanding and communication between experts and end users by incorporating all data into a shared documentation system.

In order to achieve these goals a monitoring, profiling and reporting tool set, partly based on existing solutions, was implemented in the scope of this project. This tool set will automatically collect performance metrics and present them to researchers in easy to understand summaries or as a timeline (in appropriate reports). The tool set is completed by a documentation and best practices information, detailing, as applicable, measures regarding further investigation of the problem, recommended changes to the job submission, and promising performance engineering targets.

This document presents the user's guide of the *PfiT* system. In this guide we will describe the *text report* as well as the *PDF report* and all metrics they present. Furthermore, the recommendations of the recommendation system will be presented as well as the criteria, when a recommendation should be made. Finally, best practices to create (highly) optimized programs are outlined.

# Chapter 2

## User's Guide

### 2.1 Explanation of the metrics of the reports

This section covers the description of the metrics of the *Text*- and *PDF* report. In the first subsection the metrics of the *Text* report will be discussed, and in the second subsection the metrics of the *PDF* report. In both subsections the metrics are divided into the following metric classes in the node view:

- Batch job summary,
- CPU (mean values),
- main memory (mean and maximum values),
- swap space (mean and maximum values),
- IO (work, NFS, scratch; mean values),
- network (infiniband, ethernet: mean values),
- GPU (if exists; mean values).

At the end of each *Text* report a job summary will be presented regarding selected metrics, for example, CPU mean, main memory high water mark, swap usage).

#### 2.1.1 Text report

The *Text* report should aid the user to identify pathological performance problems, for example, low CPU usage or swap activity. To serve this need various metric values are reported. Based on these values recommendations are automatically formulated to guide the user to improve the program. In this section the metrics of the *Text* report will be explained, in which we make the distinction between the *node metric view* (mean or maximum values over all timesteps of the job for every single node of the job) and the *summary view* (mean values over all job nodes and job timesteps). The calculation of these *Text* report metrics is described in subsection 2.2. The explanation of these metrics will be illustrated by a sample *Text* report of a job (condensed form) which ran on only one GPU node (see listing 2.1). In this small example all elements of the report will be explained.

## Batch Job Information

```

-----
User          *****
JobId         *****
Jobname       *****
Queue/Partition  gpu
Numer nodes   1
Nodelist      node323
Requested wall-clock time 00-11:59:00
Elapsed wall-clock time  00-11:59:25
Submit time    Sun Sep 29 19:17:59 CEST 2019
Start time     Sun Sep 29 19:17:59 CEST 2019
End time       Mon Sep 30 07:17:24 CEST 2019

```

## Per node utilization (mean values + memory high water mark (hwm))

		CPU			memory used		swap used
node	usage (in %)	idle (in %)	iowait (in %)	hwm GiB	mean GiB	mean GiB	
node323	198.38	1.62	0.00	2.08	2.06	0.000000	
1	198.38	1.62	0.00	2.08	2.06	0.000000	

## IO utilization (I) (mean values)

		/work			
		read		write	
node	data	ops	data	ops	
	MiB/s	1000/s	MiB/s	1000/s	
node323	0.03	0.00	0.47	0.00	
1	0.03	0.00	0.47	0.00	

## IO utilization (II) (mean values)

		/home		/scratch	
		read	write	read	write
node	data	data	data	data	
	MiB/s	MiB/s	MiB/s	MiB/s	
node323	0.00	0.09	0.00	0.00	
1	0.00	0.09	0.00	0.00	



```

+-----+-----+-----+-----+-----+

```

Network utilization (mean values)

```

+-----+-----+-----+-----+-----+
|           |           Infiniband           | | | |
|           | received |           sent           |
| node      | data    | ops    | data    | ops    |
|           | MiB/s   | 1000/s | MiB/s   | 1000/s |
+-----+-----+-----+-----+-----+
| node323   | 0.06    | 0.09   | 0.48    | 0.14   |
+-----+-----+-----+-----+-----+

```

```

+-----+-----+-----+-----+-----+

```

Per GPU utilization (mean values)

```

+-----+-----+-----+-----+-----+
|           | GPU  | GPU usage | GPU memory used |
| node      | no.  | (in %)   | GiB              |
+-----+-----+-----+-----+-----+
| node323   | 0    | 72.17    | 0.24             |
|           | 1    | 64.58    | 0.21             |
+-----+-----+-----+-----+-----+
| 1         | 2    | 136.75   | 0.44             |
+-----+-----+-----+-----+-----+

```

#### Recommendations

##### Max job memory utilization (Memory High Water Mark):

The programm used just a small portion of the host RAM memory (less than one quarter of the total RAM)! If possible use a partition with fewer memory **for** better utilization of the RAM

##### Mean job memory utilization:

The programm used just a small portion of the host RAM memory (less than one quarter of the total RAM)! If possible use a partition with fewer memory **for** better utilization of the RAM (i.e. **if** the high water mark fits into memory).

##### Elapsed wall clock time:

The used wall clock time of the job was larger than the requested wall clock time. As a consequence the job has not finished! Adjustment of the runtime is needed! Please try to caclulated the job runtime more precisely!

#### Job summary

```

-----
Elapsed wall clock time 100.1 % (00-11:59:25 of requested walltime)
Mean CPU usage         100.0 % (      16.0 of 16.0 cores)
Mean Hyperthread usage  98.4 % (      15.7 of 16.0 hyperthreads)

```

Max. main memory used	3.3 % (	2.1 of 62.8 GiB available memory)
Max. swap memory used	0.0 % (	0.0 of 1.9 GiB available swap)
Max GPU memory usage	2.1 % (	0.2 of 11.2 GiB available memory)

Listing 2.1: Example Text report

## Batch job

The following metrics are collected by the batch system (for example Slurm) before generating the report. These metrics should give a general overview of the job, for example JobID, jobname, start and end time of the job, number of nodes and nodelist of the job.

- User: The user identification string (e.g. user42),
- JobId, Jobname: These two metrics are the numerical and string job identifier.
- Queue/Partition: The well defined subset of nodes of the cluster, that the job should run on (or parts of it). In general, every queue has its own architecture characteristics, e.g. a partition with the example name *big* with 256 GiB in comparison to a partition *std* with 64 GiB. Another example is the partition *gpu* which contains nodes with additional graphic accelerator capability.
- Number nodes: Number of nodes of the cluster the job used (for example 32).
- Nodelist: The list of nodes of the cluster the job used. In Slurm an example list has the following appearance: node[134-136,213-241] that means the job allocated the nodes node134-node136 and node213-node241 (all in all 32 nodes). In this concrete example only node323 was allocated.
- Requested and elapsed wall clock time: The requested wallclock time of the job (maximum amount of user requested job run time) and the used runtime of the job (elapsed runtime) in the format dd-hh:mm:ss. In this example the user requested 11 hours and 59 minutes (00-11:59:00) and the elapsed walltime was 11 hours, 59 minutes and 26 seconds (00-11:59:26). The additional 26 seconds is batch system overhead, the program was finished exactly after 11 hours and 59 minutes.
- Submit, start and end time: Submit, start and end time of the job. In particular submit and start time of the job can differ significantly, when the job waits for a while in the batch queue. In this example submit and start time are identical (Sun Sep 29 19:17:59 CEST 2019).

## Node view

At the beginning of the presentation of the node view metrics the CPU, memory and swap metrics will be discussed. For convenience the respective part of the report will be copied at the beginning of the section.

Per node utilization (mean values + memory high water mark (hwm))			
+	+	+	+
	CPU	memory used	swap used

node	usage	idle	iowait	hwm	mean	mean
	(in %)	(in %)	(in %)	GiB	GiB	GiB
node323	198.38	1.62	0.00	2.08	2.06	0.000000
1	198.38	1.62	0.00	2.08	2.06	0.000000

Listing 2.2: Per node utilization CPU main memory and swap (mean values + memory high water mark (hwm))

### node

- *node*: Name of every node of the job. In this job only one node is used, but if a job uses more than one node, all nodes are listed one below another. In the last row the number of used nodes of the job will be presented. Analogously, in this row the sum of CPU usage, idle and iowait percentage over all nodes are listed below the appropriate item. This is valid for the memory and swap metrics, IO, network and GPU metrics, too.

### CPU

- *usage (in percent)*: CPU usage is the sum of the two raw metrics (*CPU*) *user* and (*CPU*) *system* (see 2.2). This metric is given in percent and 100% is the theoretically achievable time the CPU is in the user and system state over all cores of a node and over the job runtime (i.e. over all time measurement points of the job). The formula to compute the node percentage of the CPU usage is as follows:

$$\frac{cpu\_usage\_node}{100 \cdot n\_cores\_per\_node \cdot job\_runtime} = runtime\_percent\_of\_node. \quad (2.1)$$

This formula has the following items:

- *cpu\_usage\_node*: Sum of all ticks of all cores of the node processes being in user or system mode over the job runtime,
- *n\_cores\_per\_node*: Number of cores of the node,
- *job\_runtime*: Runtime of the job in seconds (more precisely: the difference between the first and the last measurement time step in the job; in general this value is smaller than the jobs runtime),
- *usage\_runtime\_percent\_of\_node*: The average runtime of all cores of the job on the node in user and system mode (in per cent).

If on one core two or more hyperthreads are located, the *cpu\_usage\_node* value often will be larger than 100%. The maximum value in the case of one core and one hyperthread is 100%, in the case of two hyperthreads 200%, in the case of four 400%, etc. A value below 50% will be marked as problematic (pathologic) because over one half of the node CPU time is wasted with other tasks than tasks in user or system state (i.e. idle or iowait time).

- *idle* (in percent): *CPU idle* denotes the accumulated value over all core ticks, when every single core was in the idle state. That means the idle process was running on the appropriate core(s), because no program was running on it. The formula to calculate the node idle time is analogous to the *cpu\_usage* case in which the summation is over the raw metric (*CPU*) *idle*.
- *iowait* (in per cent): This metric denotes the time waiting for IO. The formula is analogous to the above two formulas and sums over the raw metric *iowait*.

### Main Memory / memory used

- *High water mark* (*hwm*; in GiB): The maximum value of the used main memory of the node over all measurement points of the job (in the node view) and additionally over all nodes over all measurement points of the job (in the summary view). The formula to calculate the high water mark of node  $i$  ( $i \in \{node_1, \dots, node_n\}$ ) over all measurement steps  $t \in T = \{t_0, t_1, \dots, t_n\}$  in the job uses the raw metric *Resident Set Size* (*rss*) (see 2.2) and looks as follows:

$$hwm_{node_i} = \max_{t \in T} rss_{node_i}. \quad (2.2)$$

In the example job the *high water mark* is 2.08 / 64 GiB.

- *Mean memory value* (*mean*; in GiB): To compute the mean value of the used memory of a node over all measurement points of the job (in the node view) and additionally over all nodes and over all measurement points of the job (in the summary view) the raw metric *Resident Set Size* (*rss*) is used, too. The formula to calculate the mean memory value of the  $i$ th node over all measurement steps  $t \in T = \{t_0, t_1, \dots, t_n\}$  in the job has the following representation:

$$mean\_mem_{node_i} = \frac{1}{n+1} \cdot \left( \sum_{t \in T} rss_{node_i} \right). \quad (2.3)$$

*Mean memory* and the *memory high water mark* can differ significantly from each other because an application can use a very high amount of main memory over a short period and then falls back into low main memory usage. Consequently, the arithmetic mean over all time steps of the job of the node and the high water mark can/will differ largely. A short example shall exemplify this: With a job runtime of one hour and the constant memory usage of 10 GiB over the total runtime (except of 1 minute, where the main memory usage took the amount of 60 GiB), the memory mean value is about 11 GiB, a value far away from 60 GiB. In the example the job took 2.06 / 64 GiB mean memory, which is almost equal to the high water mark.

### Swap / swap used

- *High water mark swap* (*hwm*; in GiB (to be implemented)): The maximum value of the used swap of a single node over all measurement points  $t \in T = \{t_0, t_1, \dots, t_n\}$  of the job (in the node view) and additionally over all nodes of the job and over all measurement points of the job (in the job summary view). The raw metrics this report parameter is derived from are *SwapTotal* and *SwapFree* (*SwapUsed* =

$SwapTotal - SwapFree$ ; see p. 17) and the swap high water mark computes as follows:

$$hwm\_SwapUsed_{node_i} = \max_{t \in T} SwapUsed_{node_i}. \quad (2.4)$$

- *Mean swap value* (in GiB): The mean value of the used swap memory of one single node over all measurement points  $t \in T = \{t_0, t_1, \dots, t_n\}$  (in the node view) and additionally over all nodes of the job and over all of the jobs measurement points (in the summary view). Both values can differ significantly because an application can use a very high value over a short period (peak swap usage) and then falls back into low usage (see *swap used*). In the example no swap was used (0.0/2.0 GiB).

### IO (work, NFS, scratch)

IO utilization (I) (mean values)					
+-----+-----+-----+-----+-----+					
	/work				
	read		write		
node	data	ops	data	ops	
	MiB/s	1000/s	MiB/s	1000/s	
+-----+-----+-----+-----+-----+					
node323	0.03	0.00	0.47	0.00	
+-----+-----+-----+-----+-----+					
1	0.03	0.00	0.47	0.00	
+-----+-----+-----+-----+-----+					
IO utilization (II) (mean values)					
+-----+-----+-----+-----+-----+					
	/NFS		/scratch		
	read	write	read	write	
node	data	data	data	data	
	MiB/s	MiB/s	MiB/s	MiB/s	
+-----+-----+-----+-----+-----+					
node323	0.00	0.09	0.00	0.00	
+-----+-----+-----+-----+-----+					
1	0.00	0.09	0.00	0.00	
+-----+-----+-----+-----+-----+					

Listing 2.3: IO utilization

- read/write data (in MiB/s): In all three cases these metrics are reporting the bandwidth of the job per node and per second for the read or write operations. In the case of IO to or from the worksystem, the data will be handled over by for example *BeeGFS* or *Lustre* (over Infiniband), in the case of *home* via NFS and *scratch* is *localdata* (hard disk or SSD of the node the job runs on). The NFS value have to be treated with care, since for example the loading of the programs can also be accounted to NFS. This can lead to a high NFS read rate for short jobs, although no read operations are performed on *home*. In the example job there was just low IO traffic for all three topics.

- read/write operations (in 1000/s) (in this example case just for *work* / *BeeGFS*): These metrics document the number of operations the system needed to perform reading or writing the data and in this example no operations are reported.

## Network utilization (Infiniband)

Network utilization (mean values)					
+-----+-----+-----+-----+-----+					
		Infiniband			
		received		sent	
node	data	ops	data	ops	
	MiB/s	1000/s	MiB/s	1000/s	
+-----+-----+-----+-----+-----+					
node323	0.06	0.09	0.48	0.14	
+-----+-----+-----+-----+-----+					

Listing 2.4: Network utilization (Infiniband)

- read/write data (in MiB/s): These two metrics provide the bandwidth of the communication of the job over the network in the read and write case (mean values of every single node over all measurement time steps of the whole job period; currently infiniband only).
- read/write operations (in ops/s): The number of read and write operations denote the amount of operations to perform the network traffic (data and other administration data) (mean values per every single node over the whole job period).

## GPU utilization

Per GPU utilization (mean values)				
node	GPU no.	GPU usage (in %)	GPU memory used GiB	
node323	0	72.17	0.24	
	1	64.58	0.21	
1	2	136.75	0.44	

Listing 2.5: GPU utilization

- GPU No.: The local GPU number (from 0 to #GPUs-1 on the local graphics card). In this example two GPUs per node can be found.
- GPU usage: Denotes the GPU mean usage of one GPU over all measurement steps of the job.
- GPU memory used: Denotes the mean GPU memory usage of one GPU over all measurement steps of the job.

## Job summary

The example job summary is presented below:

Job summary			
-----			
Elapsed wall-clock time	100.1 %	(00-11:59:25	of requested walltime)
Mean CPU usage	100.0 %	(	16.0 of 16.0 cores)
Mean Hyperthread usage	98.4 %	(	15.7 of 16.0 hyperthreads)
Max. main memory used	3.3 %	(	2.1 of 62.8 GiB available memory)
Max. swap memory used	0.0 %	(	0.0 of 1.9 GiB available swap)
Mean GPU usage	68.4 %		
Max GPU memory usage	2.1 %	(	0.2 of 11.2 GiB available memory)

Listing 2.6: Job summary

- Elapsed wall-clock time (in percent): The elapsed wall-clock time of the job in percent of the total requested wall-clock time and in the format dd-hh:mm:ss (days-hours:minutes:seconds). In the example case the job has a runtime of 11 hours, 59 minutes and 26 seconds (100.1 % of the total requested walltime), in which the 26 seconds overhead is not program runtime but just batch system clean up, etc.
- Mean CPU usage (in percent): The average CPU usage of the job (summed over all job nodes and job time steps) in percent of the job runtime.
- Mean Hyperthread usage (in percent): All values larger than 100 % of the mean CPU usage are interpreted as hyperthread usage.
- Max. main memory used (in percent of the available main memory): The memory high water mark over all nodes and over all measurement steps:

$$hwm\_mem\_job = \max_i \{hwm_i\} \quad (i \in \{node_0, \dots, node_n\}). \quad (2.5)$$

- Max. swap memory used (in percent of the available swap space): The maximum used swap memory over all nodes and over all measurement time steps of the job.

$$hwm\_SwapUsed\_job = \max_i \{hwm\_SwapUsed_i\} \quad (i \in \{node_0, \dots, node_n\}). \quad (2.6)$$

- Mean GPU usage (in percent): The average GPU utilization of the job (summed over all job nodes and job time steps) in percent of the job runtime.
- Max GPU memory usage (in percent): The maximum GPU usage of the job (average over the sum of all job nodes and job time steps).

## Recommendations

To achieve a better job performance of the user's job, in the text report recommendations are automatically formulated on the basis of the selected metrics and their metric values. These recommendations and their criteria are discussed in subsection 2.3.1.

### 2.1.2 PDF report

The PDF reports task is to expand the output of the ASCII report in a graphical manner, for example, using bar charts and time series plots. Especially the time series plots yield further temporal insight into the job behaviour and extend the output of the text report with the time dimension. Thus, the user is able to see the chronological sequence of the behaviour of the job (per node).

The PDF report consists of three parts. In the first part a global job summary and the recommendations are displayed. In the second part, statistical information of the distributions on the nodes are given. The metric values of the text report are visualized by bar charts and box plots. Included are mean, maximum, minimum and standard deviation of values of the job for every metric. In the third part the time series plots of the behaviour of the jobs (per node) of selected metrics are shown.

At this place the pdf report will be explained, if the final appearance of this report is determined (01/2020).

Example using current version appears in Appendix A.

## 2.2 Sources of metrics

To present metric values in the various reports to the user and to formulate recommendations to achieve a better utilization of the cluster resources, raw metric values of the used cluster resources have to be collected from several sources. These sources were already presented in this section, whereas the relation to the metrics in the reports will be presented in section 2.1. The documentation of this relationship is necessary because some metrics of the reports are the result of the concatenation of the raw operating system metrics (for example by summation). A major source of the raw system metrics is the *procfs* pseudo-filesystem, while the counter values for network (currently only Infiniband), IO (work (currently only via *BeeGFS*)) and GPU are collected by the appropriate tools of the resources like *beegfs-ctl* (BeeGFS), *perfquery* (infiniband) and *nvidia-smi* (Nvidia GPU). In the following the sources of the raw system metrics will be presented and for every system metric its meaning, unit and their value interval.

The metric collector *Telegraf* collects via its plugins almost all metrics of the following sources, while *PfiTCollect* only collects those metrics, which are important for creating the reports (to save database storage).

### procfs

*procfs* (process filesystem) is a pseudo (virtual) filesystem, since all information in the corresponding files will be generated by the kernel on demand. In general it is mounted in */proc*. We have to distinguish between system and process data. System data is stored in files in */proc* (for example */proc/cpuinfo*) while process information can be found in */proc/pid* (for example */proc/1234/stat*, in which *pid=1234* is the process number). We will start the respective description of the *procfs* filesystem with the system information



files listed below. Notice: A reasonable collection rate is every 30 seconds, which has only a very slight impact on the runtime behaviour; see deliverable 2.2, Part I.

- */proc/cpuinfo* contains almost only static information about the CPU cores or hardware threads, such as, CPU family, CPU model and model name, stepping, recent frequency of every core or hardware thread (dynamic feature), L3 cache size, number of cores or hardware threads, instruction set architecture, cache alignment, address size and power management. This information can be used to summarize processor properties, as has been done in the reports. The collected metrics are *L3 cache size* and *cpu cores*.
- */proc/meminfo* holds static and dynamic information about the memory resources (i.e. main and virtual memory, swap space) and the for out purposes most important metrics are presented here. All values are given in KBytes.:
  - total usable (physical) main memory of the node minus a few reserved bits and the amount of the kernel binary (metric *MemTotal*),
  - the whole free (unused) main memory (*MemFree*); this denotes the amount of the main memory, which is not used at the moment by the application or the operating system,
  - active and inactive memory (*Active*, *Inactive*),
  - buffered and cached memory (temporary buffers for files of the file system; *Buffered*, *Cached*),
  - memory size of the page tables (*PageTables*),
  - pages used by the kernel stack (*KernelStack*),
  - memory which is used by FUSE by temporarily write back buffers (Filesystem in Userspace) (*WritebackTmp*),
  - total and free amount of the swap space (*SwapTotal*, *SwapFree*),
  - total and used amount of the vmalloc memory (*VmallocTotal*, *VmallocUsed*).

Most of these metrics are not directly available in the reports, but they are used to calculate the approximately used main memory and the swap space since both metrics have to be calculated and are not directly available. The metric *MemUsed* is calculated by summing up those parts of the memory, which are allocated by the operating system and not by the program and not by *MemFree*). After subtraction from *MemTotal*, *MemUsed* is calculated with the formula below.

$$\begin{aligned}
 MemUsed = MemTotal - (MemFree + Buffered + Cached + Slab \\
 + SwapCached + WritebackTmp + KernelStack \\
 + PageTables).
 \end{aligned}
 \tag{2.7}$$

*SwapUsed* is calculated in the following way:

$$SwapUsed = SwapTotal - SwapFree.
 \tag{2.8}$$

An interesting observation was made while running *PfiTCollect*, since the free swap space diminished und swap was used, although main memory usage was low. The swap activity was caused by kernel swapping (swapping out of kernel structures).

- */proc/stat* creates a CPU and system statistic. For every core and hardware thread of the node we get information (for example *user*, *system*, *idle*, *iowait*, *nice* (line *cpu(n)*)) which is also summed to deliver node wide information (line *cpu*):
  - Collecting the number of clock ticks (in *USER\_Hz*, which is on most systems 1/100 of a second) of every CPU core or hardware thread, when the CPU is for example in *user*, *system*, *iowait* or *idle* state (the metrics *user*, *nice*, *system*, *idle*, *iowait*, *irq*, *softirq* can be found in columns 2 until 8 of */proc/stat*) (valid for line *cpu* and *cpu(n)*). Furthermore, the metric values of virtual machines are listed in this file (time which is used for virtual machines; for example *steal*, *guest*, *guest\_nice*; column 9 until 11). The tick values have to be converted into seconds in the metric collector.
  - number of context switches (*ctxt*),
  - number of processes that are in the runnable state (*procs\_running*),
  - number of processes that are blocked and are waiting for completing IO (*procs\_blocked*).

The metrics, which are used currently in the text and PDF report are *CPU usage* = *user* + *system*, *CPU idle* = *idle* and *CPU iowait* = *iowait*. ??? Das scheint mir nicht ganz stimmig zu sein; wir bekommen CPU usage als Verhältnis in Prozent, während die anderen Werte CPU time (*user*, *system*, *idle* und *iowait*) in Sekunden ankommen. Wir bekommen alle 5 Werte vom Monitor bzw. Telegraf. Vielleicht ist es im PfiTCollect anders...bitte prüfen und Rückmeldung.

- The first three values of the file */proc/loadavg* denote the workload of the node in the last one, five and fifteen minutes. More precisely, they specify the average number of processes, which are in the runnable state or waiting for IO in the last one, five or fifteen minutes.
- */proc/uptime* contains the uptime of the system since the last reboot (first value) and the idle time of the system (second value; both in seconds)
- */proc/diskstats* (*local data*) The metric data about the local filesystem (scratch) are stored in this file. The following metrics will be collected (per disk and disk partition; accumulated since reboot (except *number of I/Os currently in progress*):
  - *number of reads completed* denotes the total number of completed reads,
  - *number of reads merged*: If two blocks should be read and are adjacent to each other, these two blocks are read in one task and *number of reads completed* will be incremented by one (not two, although two blocks should be read!). *number of reads merged* shows how often this action was carried out.
  - *number of sectors read* specifies the successful read of sectors since reboot.
  - *number of milliseconds spent reading* stores the accumulated value of milliseconds staying in read operations since reboot,

- *number of writes completed* (analogous to *number of reads completed*),
- *number of writes merged* (analogous to *number of reads merged*),
- *number of sectors written* (analogous to *number of sectors read*),
- *number of milliseconds spent writing* (analogous to *number of milliseconds spent reading*),
- *number of I/Os currently in progress* goes to zero, when IO operations terminate,
- *number of milliseconds spent doing I/Os* increases with the value of *number of I/Os currently in progress*,
- The *weighted number of milliseconds spent doing I/Os* field is incremented at each I/O start, I/O completion, I/O merge, or read of these stats by the number of I/Os in progress (field 9) times the number of milliseconds spent doing I/O since the last update of this field. This can provide an easy measure of both I/O completion time and the backlog that may be accumulating.

These metrics are collected, but there are additional metrics which can be found at <https://www.kernel.org/doc/Documentation/iostats.txt>.

- */proc/net/rpc/nfs* presents NFS traffic metric values. This file contains several lines, but only the one with the beginning *proc3* string is of interest for our needs. *proc3* means that in this line the NFS client statistics of NFS protocol version 3 are present. The only values of the *proc3* line, which are interesting in our case are reading and writing over NFS. These values are located in column 9 and 10 of the appropriate line and are measured in MiB since reboot. To collect data from this file, an NFS driver and filesystem have to be installed. Further information and metrics about this pseudo file can be found in <https://www.svennd.be/nfsd-stats-explained-procnetrpcnfsd/>.
- The */proc/pid/stat* file contains status information of the process PID, which information are used for example by the command *ps*. The following parameters (which are a selection of all parameters) are stored in this file and read by the collectors:
  - PID (process identifier) and PPID (parent process identifier) of the process,
  - process state (*state*),
  - number of minor and major page faults since process start (*minflt*, *cminflt*, *majflt*, *cmajflt*),
  - user, system and nice time (measured in clock ticks; *utime*, *stime*, *nice*),
  - number of threads spawned in this process (*num\_threads*),
  - start time of the process after booting the system (*starttime*),
  - size of the virtual memory of the process in Bytes (*vsiz*),
  - *resident set size* in pages (*rss*; this is just the number of pages which result from text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out),

- *processor ID* on which the process with PID was located during the last time slice.
- time for guest process on a virtual machine (*guest\_time*).

*PID* and *processname* are used as tag keys, the other metrics are field keys in *Telegraf*.

- */proc/pid/statm* delivers information about the memory usage (in pages):
  - size of the whole process virtual memory size (the same as *VmSize* in */proc/pid/status*),
  - resident set size (the same as *VmRss* in */proc/pid/status*; in KBytes),
  - amount of pages the code area as well as the data and stack area of the program uses (text, data).
- */proc/pid/status* provides much of the information of the files */proc/pid/stat* and */proc/pid/statm*. The advantage of the representation of the contents in this file is a better readability of the data for users. The following metric data were used:
  - name of the process or program name,
  - PID, PPID,
  - state of the process (i.e. running, sleeping, uninterruptible sleep, zombie, traced or stopped),
  - peak of the process virtual memory size (*VmPeak*; in KBytes),
  - size of the virtual memory of the process (*VmSize*; in KBytes),
  - *high water mark*, i.e. the peak RAM utilization the process used until now (*VmHWM*; in KBytes),
  - *resident set size* (*VmRSS*; in KBytes),
  - amount of memory of the data, stack and code area of the process (*VmData*, *VmStk*, *VmExe*; in KBytes),
  - total amount of used swap space (*VmSwap*; in KBytes),
  - number of threads of the process PID,
  - voluntary and involuntary context switches.

## IO (work)

To get the metrics of NFS and scratch traffic, we used the metrics of the files */proc/net/rpc/nfs* and */proc/diskstats*, see above. The data for the work filesystem has to be determined by a tool, like the *beegfs-ctl* (*clientstats*) tool, which measures the counter values of data and packet transmission by BeeGFS with no impact on the runtime behaviour. This concrete example is extendable, for instance, to Lustre.

## Network

The network metrics can be divided into infiniband and ethernet metrics. We will discuss the infiniband metrics first, which are collected by the *perfquery* tool with the *perfquery -xa* command (the ethernet metrics will be implemented in the near future). The flags *-x -a* (*= -xa*) show aggregated extended port counters rather than (basic) port counters for all ports. The following metrics are collected by *perfquery* (collection rate as above) and directly used in the report:

- *PortRcvData* and *PortXmitData* show the amount of data (in Bytes) received and sent by the port(s) (since reboot),
- *PortXmitPkts*, *PortRcvPkts* stores the amount of packets (in counts) received and sent by the port(s) (since reboot).

A few examples follow, which should illustrate the use of *perfquery*; they have been extracted from the manpage of *perfquery*.

- *perfquery* (reads local port performance counters),
- *perfquery -x 32 1* (reads extended performance counters from lid 32, port 1),
- *perfquery -a 32* (read performance counters from lid 32 and all ports),
- *perfquery -x -r 32 1* (reads extended performance counters and resets them; we don't need to reset the counters).

Remark: It must be mentioned, that the performance counters *PortXmitData* and *PortRcvData* have to be multiplied with four, since in *perfquery* the octets are divided by four.

These counters can also be found in the *sysfs* directory (for every file a counter)

- */sys/class/infiniband/mlx4\_0/ports/1/counters/port\_xmit\_data*
- */sys/class/infiniband/mlx4\_0/ports/1/counters/port\_rcv\_data*
- */sys/class/infiniband/mlx4\_0/ports/1/counters/port\_xmit\_packets*
- */sys/class/infiniband/mlx4\_0/ports/1/counters/port\_rcv\_packets*

Counting took place since booting, too.

## GPU (Nvidia)

GPU metrics of Nvidia cards can be retrieved with the *nvidia-smi* tool. The following request provides the metrics important for our purposes (collection rate as above; no impact on the runtime behaviour due to the call of *nvidia-smi* here, too) :

```
nvidia-smi --format=csv,noheader,nounits
            --query-gpu=index,gpu_name,memory.total,
            memory.used,memory.free,utilization.gpu,
            utilization.memory
```

The metrics are delivered in a csv format and have no units (see flag `-format`). This way of representing simplifies the parsing of the metric values. Those metrics have the following meaning:

- *index*: Denotes the local number of the GPU,
- *gpu\_name*: The official product name of the GPU (for example Tesla K40),
- *memory.total*: Total memory of one GPU,
- *memory.used*: Total used memory of one GPU (in MiB),
- *memory.free*: Total free memory of one GPU (in MiB),
- *utilization.gpu*: Percent of time over the past sample period during one or more kernels were executed on the GPU,
- *utilization.memory*: Percent of used memory over the past sample period.

We limit ourselves to Nvidia GPUs since the survey (see deliverable 1.2) showed, that no cards from other manufacturers are used. Further metrics can be gathered with *nvidia-smi* (see e.g. [https://www.microway.com/hpc-tech-tips/nvidia-smi\\_control-your-gpus/](https://www.microway.com/hpc-tech-tips/nvidia-smi_control-your-gpus/)).

## 2.3 Best practices for users

This section introduces hints and recommendations for users to gain a better job performance. It is divided into two main paragraphs. The description of the recommendations and when they will be thrown can be found in subsection 2.3.1. In the second paragraph checklists will be presented, which formulate advices regarding the development/compiling process and the preparation of the job run. The recommendation list as well as the checklists are in an initial state and will be extended permanently.

### 2.3.1 Best practices

Based on the subjects covered in the metrics documentation the user will be guided through steps to improve job performance. This list will be extended and refined until the end of the project.

Since in the text report as well as in the PDF report recommendations are formulated, those recommendations will be explained here and the conditions, when they will be thrown. The structure for every best practices topic is as follows:

- Header with a short problemdescription (Problem),
- condition, when this error occurs (Condition),
- description and explanation of the error (Description),
- hints to solve or to reduce the effect of this performance problem (Recommendation).

The explanation starts with the text report. **Important:** Do not only read the recommendations, please check the appropriate metrics and their values to check if the

recommendations are reasonable and to learn about the job and system behaviour.

The following problems can be discovered with this text report:

- Low mean CPU usage and high mean CPU idle and CPU iowait time (per node and per job),
- memory high water mark and mean memory (per node and per job),
- swapping/Paging issues (per node and per job),
- low GPU and GPU main memory utilization (per node and per job),
- high mean IO and network bandwidth usage (per node),
- problematic time limits of the job (to small requested or elapsed job wall-clock time),
- load imbalance.

#### Problem: Mean CPU usage below 50%

- Condition: Mean node CPU usage  $< 0.5$  and mean job CPU usage  $< 0.5$ .
- Description: *Mean node CPU usage* denotes the mean usage of all cores/hyper threads of the node, in which they are in the *user* or in the *system* state (see description of this derived metric in ??). That means, they are not waiting and they are doing something useful. If this average value for a node or for the job decreases below 50 %, the cpus on one node or of the job were waiting over the half of the job runtime (e.g. for IO).
- Recommendation: To gain a better runtime performance, it is crucial to increase this value by decreasing the idle time or IO waiting time. In many cases the *CPU usage* correlates with the IO or network traffic rate, i.e. in the course of doing IO the *CPU usage* reduces, while the IO rate is increasing. To verify this topic, please consult the (sub)tables CPU and IO in the text report and especially the combined CPU-IO diagram in the PDF report of the job. If in the latter an increasing IO rate and a decreasing CPU usage can be noticed this correlation give a hint towards this problem. The analogous situation can be seen and visualized for the network traffic rate and the *CPU usage*.

#### Problem: Mean CPU idle > 50%

- Condition: Mean node CPU idle  $> 0.5$  and mean job CPU idle  $> 0.5$ .
- Description: *Mean CPU idle* denotes the mean idle usage of the CPU, in which the CPU is in the *idle* state (see description of this derived metric in ??) That means, the CPU is waiting and is not doing something useful. If this average value for a node or for the job increases above 50%, the cpus on one node or of the job were waiting in the average over the half of the job runtime.



- Recommendation: Reduce the waiting time of the processes. In many cases the *CPU usage* correlates with the IO or network traffic rate, i.e. in the course of doing IO the *CPU usage* reduces (and *CPU idle* increases), while the IO rate is increasing. To verify this topic, please consult the (sub)tables CPU and IO in the text report and especially the combined CPU-IO diagram in the PDF report of the job. If in the latter an increasing IO rate and a increasing CPU usage can be noticed, possibly this problem arose. The analogue situation can be seen and visualized for the network traffic rate and *CPU idle*.

#### Problem: Mean CPU iowait > 10 %

- Condition: Mean node CPU iowait > 0.1 and mean job CPU iowait > 0.1.
- Description: This metric indicates the mean IO waiting time of the CPUs of the node or of the job (see description of this derived metric in ??).
- Recommendation: Reduce the waiting time for IO. Sometimes the *CPU iowait* correlates with the IO rate, i.e. in the course of doing IO the *CPU iowait* metric value increases, while the IO rate is increasing, too. To verify this topic, please consult the (sub)tables CPU and IO in the text report and especially the combined CPU-IO diagram in the PDF report of the job. If in the latter an increasing IO rate together with an increasing *CPU iowait* can be noticed, this correlation can be a hint for IO problems.

#### Problem: Maximum job memory utilization (Memory High Water Mark) below main memory of other queue

- Condition: mem\_hwm < mem\_hwm\_other\_queue
- Description: If the memory high water mark of the job is below the available memory of another queue it is recommended to run the job in another queue with fewer memory.
- Recommendation: Use queue with fewer available main memory.

#### Problem: Maximum job memory utilization (Memory High Water Mark) is larger than 95% of main memory

- Condition: mem\_hwm > 0.90.
- Description: If the memory high water mark is larger than 90% of the main memory, it is possible, that the job will begin to page/swap or that the OOM killer will kill the process of the job with this high value and as the consequence the job will be aborted.
- Recommendation: Try to reduce the memory workload of the program to reduce memory usage and to prevent paging/swapping. For example do not allocate memory for all arrays simultaneously but rather when they are needed. Another possible source of paging/swapping is, if there is a memory leak (see time series in PDF report). It is also recommended to get an evenly time distributed memory workload in contrast to strongly time varying memory usage.



**Problem: Swapping (mean swapping)**

- Condition:  $\text{mean\_swap} > 0.0$
- Description: A value of *mean\_swap* larger than 0.0 KiB could lead to severe paging/swapping, what means, that parts of a process memory will be written to disk. But this leads to performance problems, since writing and reading of those parts of the memory are very time consuming, because disks or disk/network are much slower than memory and memory busses. It have to be mentioned that this value shows the mean value of paging/swapping. As the consequence it is possible, that this value is relatively constant over the course of the job, which implies a relatively constant paging/swapping activity. Another possibility is, that only a few paging/swapping bursts cause this values. Please consult the column swap/hwm to get more insight inot this problem.
- Recommendation: Try to reduce the memory workload of the program to reduce memory usage and to prevent paging/swapping. For example do not allocate memory for all arrays simultaneously but rather when they are needed. Another possible source of paging/swapping is, if there is a memory leak (see time series in PDF report). It is also recommended to get an evenly time distributed memory workload in contrast to strongly time varying memory usage.

**Problem: Elapsed wall clock time of the job is below of 25 % of the requested wall-clock time of the job**

- Condition:  $\text{elapsed\_wall-clock time} < 0.25 \cdot \text{requested\_wall-clock time}$
- Description: The total runtime of the job (elapsed wall-clock time) took only 25 % of the user's requested job wall-clock time.
- Recommendation: If this job was not a test job or the job run was interrupted by a program error, we would advice you to adjust (to lower) the requested wall-clock runtime of the job. It is also possible, that no requested wall-clock time was given. In Slurm the maximum wall-clock time of the selected queue will be taken as the requested wall-clock time which leads eventually to longer queue waiting times.

**Problem: Elapsed wall-clock time greater than 100 %**

- Condition:  $\text{elapsed\_wall-clock time} \geq \text{requested\_wall-clock time}$
- Description: The total job runtime was greater than the requested wall-clock time. That could possibly lead to a problem if not all computations could be performed and data is lost.
- Recommendation: Please adjust the runtime if possible. If the used queue does not allow a higher runtime, change the queue if possible or ask your system administrator for more job runtime. A further method to circumvent this problem is to devide the whole problem into chunks of maximum wall-clock time which are independent of another or which allow to restart the job using the old data of the last run.

### Problem: No GPUs were used

- Condition:  $n\_gpus = 0$ .
- Description: The job was scheduled to a GPU queue, but no GPU was used.
- Recommendation: If no GPU will be used while job runtime, please use a different queue. This recommendation is obsolete if using GPU queues is allowed explicitly for non GPU jobs.

### Problem: Not all GPUs were used

- Condition:  $0 < n\_gpus < \max\_n\_gpus$ .
- Description: The job was scheduled to a GPU queue and used at least one GPU but not all GPUs.
- Recommendation: If possible use all GPUs of every node.

## 2.3.2 General recommendations and checklists

### Development process

At the beginning some notes, warnings and hints. The most important things in writing programs are<sup>1</sup>

- that the program does that, what you want,
- that the program is error free,
- that there are no security issues in the program,
- that the code is readable and commented,
- that the code is portable (often useful/needed),
- that the program is easily extendable.

Especially the first three items are very important and it is strongly recommended to write at the first stage a properly running program which is error free and has no security issues (in scientific computing this is a minor problem). After managing the first three steps successfully doing performance tuning can be done. But partly these goals can be solved in parallel. When designing the program the datastructures and algorithms can be chosen in such a way that inherent optimization can be done (use good designed and efficient algorithms and data structures). Programming development for optimization needs not only to consider the algorithms but also efficient data structures, too. It is important to mention, that sometimes optimization and portability are not compatible with each other every time.

Performance tuning takes place on three levels ([5]):

---

<sup>1</sup><http://www.wilkening-online.de/programmieren/c++-performance-optimierungen.html>

- system level,
- application level and
- microarchitecture level.

On the system level the program performance can be diminished by old drivers (for example network, GPU), old linux kernels, a wrong configured BIOS (NUMA), etc. This is the level of the system administrator and we expect, that the system is well configured. The application level (and partly the microarchitecture level) is the most interesting level for developers. On this level a speed up of two or three magnitudes can be managed by using

- effective mechanisms to overcome the memory wall,
- vectorization (with an ideal speed up-factor of 4 (double values, AVX/AVX2)),
- shared memory parallelism (with an ideal speed up-factor of the number of threads using OpenMP),
- distributed memory parallelism (with an ideal speed up factor of the number of launched MPI processes).

A speed up of one magnitude can be reached with tuning the microarchitecture level (effective pipelining, etc.).

In the following we will only present possible ways to optimize programs on the application level in a keyword character. In special cases we will link to appropriate documents, where the cases are explained in detail. At this place it is recommended to advise the two books [2] and [3], which are a valuable source of hints for performance engineering and parallelization.

- Use (highly optimized (parallel)) libraries for your tasks. Do not write your own code when you want for example to search, to sort and do matrix and vector operations. Do not reinvent the wheel! The elements of the libraries are in general highly optimized and comprehensively tested! The following list contains some libraries for numerical and statistical computing:
  - *GNU Scientific Library* (GSL) is a C/C++ library for numerical and statistical computing, which also includes sorting and searching<sup>2</sup>.
  - Boost-C++ libraries: A collection of C++ libraries<sup>3</sup>,
  - BLAS 1,2,3 (Basic Linear Algebra Subroutines) or OpenBLAS; (open) libraries for vector and matrix computations<sup>4</sup>,
  - Math Kernel Library (MKL): A highly optimized math and numerical library (Linear Algebra, FFT, PDEs, Deep Neural Network, etc.) for Intel CPUs by Intel (free download <sup>5</sup>).

---

<sup>2</sup><https://www.gnu.org/software/gsl/>

<sup>3</sup><https://www.boost.org/>

<sup>4</sup><https://www.openblas.net/>

<sup>5</sup><https://software.intel.com/en-us/mkl/choose-download/linux>

- Use efficient datastructures and algorithms ( $O(n \cdot \log(n))$  is better than  $O(n^2)!$ ). Selecting efficient algorithms and data structures is more important than optimizing a less efficient algorithm.
- Optimize only those parts of the code, which requires significant runtime or which are bottlenecks of the program. These parts of the program can often be detected with performance analysis tools. Some of them will be presented in the following:
  - Static code analysis: Static code analysis is performed while compile time. It checks the source code to find specified problems or errors. Many of those things can be detected by the compiler but static code analysis tools are doing further checks (for example guaranteeing coding standards, searching for possible memory leaks, buffer overflows). Some examples for static code analyzers are *Cppcheck*, *Splint/Lint* (C/C++), *Pylint*, *PyChecker* (Python).
  - *Valgrind*, which includes for example a memory checker to detect memory leaks (dynamical code analysis) or further profiling tools (see <sup>6</sup>).
  - *LIKWID* is a toolsuite to do performance analysis through profiling, for example *likwid-perfctr* (reads hardware performance counters) or *likwid-mpirun* (wrapper to start parallel applications).<sup>7</sup>
  - *Intel VTune Amplifier* is a commercial profiling tool by Intel with a graphical front end. *Intel VTune Amplifier* supports Software and hardware event sampling, memory debugging and profiling, thread profiling, etc.<sup>8</sup>
  - *gprof* is a free performance analysis tool (profiling), too, and the output is a flat profile (which contains the total execution time of every called function) and a call graph of the functions (with its residence time for every function (call)). A more detailed description can be found in [3], chapter 2.2.3.2 until 2.2.3.4.

There are other performance analysis tools which are listed in<sup>9</sup>.

- Try to use to work in the cache as much as possible (space and time dependency)! In [2], chapters 1.6 and 1.7, this important topic will be presented in detail.
- Try to parallelize the code on the level of instruction using CPU intrinsics (SSE/AVX).
- Try to parallelize the program by exploiting data parallelization. Therefore try to partition the problem in independent parts. After that analyze if it is favourable to use a shared or non shared memory approach or a hybrid approach (shared and non shared paradigm together). In the shared memory case parallelization will be done with the multithreading library OpenMP (or sometimes Pthreads). In the non shared case MPI in general will be used and in the hybrid approach both approaches (MPI with OpenMP). Additionally parallelization with the use of accelerators can be used.

---

<sup>6</sup><http://www.valgrind.org/>

<sup>7</sup><https://github.com/RRZE-HPC/likwid>

<sup>8</sup><https://software.intel.com/en-us/vtune>

<sup>9</sup>[https://en.wikipedia.org/wiki/List\\_of\\_performance\\_analysis\\_tools](https://en.wikipedia.org/wiki/List_of_performance_analysis_tools)

- Try to parallelize the code by using (summary of the above two points)
  - Instruction level parallelism (CPU intrinsics (SSE/ AVX)),
  - Shared memory parallelism (OpenMP, Pthreads),
  - Distributed memory parallelism (MPI),
  - accelerators/offloading (OpenACC, Cuda, OpenCL).

We will give for every kind of parallelization an introductory example in the subsections starting from subsection 2.3.3.

- Reduce IO since disks are much slower than CPU and memory and IO is very time consuming.
- Reduce MPI and communication actions to other processes, since (MPI) communication is often very time consuming.
- Since in HPC large loops over vectors or matrices take a considerable amount of time, we will have a closer look at potential bottlenecks in (large) loops:
  - Strength Reduction: Try to avoid expensive operations and replace them with an equivalent expression or operation which does not need as much cycles as the target expression (because the target expression/operations often require dozens of cycles to perform their computation in comparison to approximately 3-5 cycles for for example multiplication or addition):
    - \* replace  $\text{pow}(x, 2)$  or  $x * 2$  with  $x \cdot x$  (please have a look at [3], chapter 2.1.6.4),
    - \* floating point and integer division (an often occurring scenario is the division of an expression with a constant factor. It is advantageous to multiply the expression with the inverse of the divisor/denominator),
    - \* If one factor of the multiplication or division is a two potency, it is possible to replace the multiplication or division by a bit shift.
    - \* trigonometric and exponential functions (if possible create lookup tables instead of compute them every time).
  - Subexpression elimination: If an expression is often used in a loop store this expression value in a variable and use this variable instead of recalculating this expression every time. This saves computation time! It is possible that the compiler will do that for you (dependant of the optimization level) but it is recommended to do it manually because it is not sure that the compiler will do it automatically for you, since the compilers go only until a specified expression length (use scope variables). A more detailed and exemplified documentation is given in [3], chapter 2.1.6.6.
  - Loop-Invariant Code Motion: Try to reduce floating point operations. An useful example is, that calculations in a loop, which are not dependent of the loop variable (loop invariant expressions) should be calculated in front of the loop. This action is probably done by the compiler, but it could be useful to experiment. For further information see [3], chapter 2.1.6.7.

- If possible try to avoid calling a function in a large loop. Try to write a function which includes the loop because of the function overhead. Otherwise try to use function inlining or use macros, if the function only holds a few lines (inlining in C/C++ with the keyword *inline*). In function inlining the function call will be replaced with the function body of the called function (but the decision to do that is left to the compiler). This saves the overhead of the function call (for example creation of the parameters on the stack, saving return address, etc.) and gives the compiler more flexibility in reordering the code. The text of a macro will be always substituted. Please have a look at [3], chapter 2.3.2, for further information.
- Dead Code Removal: Try to write code without code, which will not be executed. Often the compiler will eliminate these code parts, but as said before, it is advantageous to take care of this aspect while programming and do not trust on the compiler too much. Furthermore, the compiler will possibly generate dead code while the optimization step and will eliminate this part automatically. Please have a look at [3], chapter 2.1.6.3..
- Try to avoid type casts (please see [3], chapter 2.3.5.1).
- Try to avoid (**if** or **switch/select**) conditions in loops (please see [3], chapter 2.3.3 and 2.3.4). If this is not possible, some recommendations:
  - \* If several conditions have to be evaluated, put the condition with the highest probability of occurrence in front of all, then the 2nd probable, etc.
  - \* If there are conditions in the **if** command with much calculations, then put that **if** branch to the end, where the most calculations have to be done (if the eventuation of all conditions are equally probable).
  - \* Use hashtables or arrays in case of **if** or **switch** in loops (C/C++; in Fortran **if** and **select**).
  - \* Minimize jumps in the code.
- Loop Unrolling: Try to unroll loops, that means for example

```
for ( int i = 0; i<n; ++i )
{

    A[ i ] = B[ i ] + C[ i ];

}
```

Listing 2.7: Loop unrolling (before unrolling)

will become to:

```
for( int i = 0; i < n; i+=4 )
{

    A[ i ] = B[ i ] * C[ i ];
    A[ i + 1 ] = B[ i + 1 ] * C[ i + 1 ];
    A[ i + 2 ] = B[ i + 2 ] * C[ i + 2 ];
    A[ i + 3 ] = B[ i + 3 ] * C[ i + 3 ];

}
```

```
}

```

Listing 2.8: Loop unrolling (after unrolling).

This action often can be executed by the compiler with a special flag (`-funroll-loops` for the GCC) or while using Interprocedural optimization or Profile Guided Optimization. But it is often worth trying to do this manually (don't trust the compiler too much), because it reduces the loop overhead (e.g. fewer increments, tests on boundaries and branches/jumps). It often also increases the amount of computation in the loop). Please see [3], chapter 2.4 for a deeper discussion on loop unrolling.

- Memory access patterns: Try to arrange your data in memory in that way, that access time is minimized and data can be accessed for example in a linear order. This is the fact in array (vectors). 2d arrays (allocated on the stack) are in memory stored in a linear fashion. When accessing matrices, it is important to know, that C/C++ stores matrices rowwise, while Fortran stores matrices columnwise. To save memory access time, iterated matrix access should take place row first in C/C++ and column first in Fortran. Furthermore in dynamical allocated arrays the rows are in general not stored in a contiguous way in the memory. Please have a look at [3], chapter 2.4.7, for a deeper discussion of memory access patterns.
- Optimize the code regarding the workload of memory (the lesser, the better and when possible equally distributed over time)!
- If possible use float instead of double if floating point calculations are executed (but only, if the results are precise enough). The goal is for example the reduction of the workload or a higher computation rate.
- Recursion is nice and elegant but due to performance reasons it is slow in contrast to iterative algorithms, which are doing the same thing.
- If possible array and object arguments of a function should be handed over by reference, not by value since the copying needs much more time than handing over the address of the object. Variables of basic datatypes can be handed over by value.
- Try to use mainly the stack since creating, administrating and deleting the heap storage is time consuming.
- Avoid NUMA issues!

### Compiler and compiling process

In this paragraph we will introduce compiler optimization flags to gain more program performance. But be careful: The presented optimization flags can have different performance effects on different problems and these flags are suggestions, because their use is very promising and lead in general to a better program performance regarding our example program (Jacobi iteration with a five point stencil (2.22)).



Furthermore the content of the compiler flags are taken (mostly in own words) from the manuals of the GCC and Intel compiler.

- Use at least the `On` flags ( $n \in \{0, 1, 2, 3, fast\}$ ), which have the following meaning for the Intel and GCC compilers :
  - \* `-O0`: (Almost) No optimization enabled. This is the default value for the GNU compiler when no optimization should be done and it is strongly recommended to compile with `-O0` when debugging or profiling should be performed (with `gdb` or `gprof`). Otherwise, in the case of debugging the debugger will for example show the error in codelines, where no error exists because of the optimization procedure (optimized out or permuted codelines, out of order execution).
  - \* `-O1`: The code will be optimized regarding execution time and code size. This optimization level optimizes for time but avoids aggressive optimization steps to keep the size of the object files and the binary within a limit or nearly constant (see <sup>10</sup> which flags are enabled for GCC, when selecting `-O1`).
  - \* `-O2`: More optimization in comparison to `O1` (including all optimization flags of `O1`): This is the default optimization flag for Intel compilers and the optimization regarding the runtime is still more intense than in `-O1` (see <sup>11</sup> which flags are enabled for GCC, when selecting `-O2`).
  - \* `-O3`: Even more optimization (including all optimization flags of `O1` and `O2`). This is the highest level which regards strict standard compliance. See <sup>12</sup> which flags are enabled for GCC, when selecting `-O3`.
  - \* `-Ofast`: Optimizing excessively for speed and disregards (with some additional flags) strict standard compliance (IEEE Floating Point Standard IEEE 754; see <sup>13</sup>).

In numerical computing it is recommended to use optimization flag `-O2` or `-O3`. But be careful: Sometimes the use of the above optimization flags will lead to wrong results (especially using `-O3` instead of `-O2`), so it is recommended to compile and run the job in the development stage with `-O0` and then with `-O1` and so on. If in all stages (including `-O3`) the results are nearly the same (within a predefined error), `-O3` is in many cases the best choice regarding the program performance. `-Ofast` can also be used but with much care because it disregards strict standard compliance. In our small benchmark scenario the runtime was approximately 10% below the `-O3` case and there was no significant error.

- Use further optimization flags of the appropriate compiler. The following flags are recommended to test with *GCC*:
  - \* `-march=cpu-type`: This flag generates assembly code for the given cpu-type using all features of this CPU. For example `-march=haswell` generates code for a machine with haswell CPU and optimizes it towards this

<sup>10</sup><https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

<sup>11</sup><https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

<sup>12</sup><https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>

<sup>13</sup><https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>



architecture and the instruction set architecture (ISA) of this CPU (or compatible ones) using all haswell features. Since in this example this optimization procedure is tailored towards the haswell architecture it is possible that the code will not run on other architectures than this (or compatible CPUs) or on older CPUs. An overview of the valid cpu-types can be found on <sup>14</sup>.

- \* **-mtune=cpu-type:** **-mtune** generates code which is optimized for the given ISA but the code will run on other CPUs. The following description in the GCC manual should explain that: „While picking a specific cpu-type schedules things appropriately for that particular chip, the compiler does not generate any code that cannot run on the default machine type unless you use a **-march=cpu-type** option. For example, if GCC is configured for i686-pc-linux-gnu then **-mtune=pentium4** generates code that is tuned for Pentium 4 but still runs on i686 machines.“. If **march** is enabled, **mtune** is enabled, too.
- \* **-msse, msse2, -mAVX, -mAVX2, etc.:** These flags enable the ISA extensions SSE, SSE2, AVX, AVX2, etc. If intrinsics are used in the code, the appropriate flag has to be set. With the flags **-mno-sse, mno-avx, etc.** these extensions will be disabled.
- \* **-mfma:** Enables Fused Multiply-Add, what means, that the compiler can process expressions like  $x \leftarrow x + y \cdot z$ .
- \* **-m64:** Generates code for the x86\_64 (64 Bit) architecture by setting int to 32 Bit and the integer type long and pointers to 64 Bit. There are options **-m32, -m16, etc.** with analogous functionality.<sup>15</sup>
- \* **-funroll-loops:** This flag enables loop unrolling in which the number of loops have to be known at compile time. This flag is implicitly enabled by activating the flags **fprofile\_gen / fprofile\_use** and **fprofile\_auto**. Otherwise it has to be explicitly set.
- \* **-ftree-vectorize:** Auto-vectorization of the code will be enabled. That means the compiler tries to vectorize parts of the code automatically in contrast to vectorize the code by hand by the user. This flag is automatically enabled with the **-O3** flag (GCC) and **-O2/-O3** flag (Intel).
- \* **-flto:** Applies *link time optimization (LTO)*. These optimizations will be applied at link time which allows GCC to optimize on a higher level because the whole program is considered for the optimization process. This is a kind of *interprocedural optimization (IPO)* which allows the elimination of dead code, reordering of the functions for better memory layout and locality, reducing duplications or inlining appropriate functions in loops<sup>16</sup>. In the common optimization techniques the optimization focus is only on single functions or blocks of codes.
- \* **-fwhole-program:** If the current compilation unit is the only program unit that has to be compiled, this flag can be applied. „All public functions and variables with the exception of main and those merged by attribute

<sup>14</sup>[https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86\\_002d64-Options.html](https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86_002d64-Options.html)

<sup>15</sup><https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html>

<sup>16</sup>Interprocedural Optimization (Wikipedia)

externally visible become static functions and in effect are optimized more aggressively by interprocedural optimizers“.<sup>17</sup> It is recommended to use *fwhole-program* and *ftto* not together.

- \* **-profile\_gen** und **-profile\_use**: Instruments the code and uses profiling to optimize the code (*Profile Guided Optimization (PGO)*). Using this flag generates in the first step a profiling report for the compiler (**prof\_gen**) in the first step. To process the report data the user has to run the instrumented version of the program a few times (to collect performance data/issues and to train the compiler; second step) and then to recompile the target file again with the **prof\_use** flag (without **-prof\_gen**). Running the PGO optimized program resulted in our scenario in significant speed ups (up to 15 - 20 percent) and is recommended to test.

But it should be mentioned again, that the the optimization success of these flags (or a combination of them) are often problem dependant. Optimization flags, which significantly speed up a program will eventually not speed up or even slow down another problem. The O-flags will in most cases optimize the program significantly, the rest of these flags are (sometimes very) promising optimization candidates.

In the case of the Intel compiler the following example flags are promising candidates for significant optimization of the code.

- \* **-On** ( $n \in \{0, 1, 2, 3, fast\}$ ) see the GNU compiler above.
- \* **-xHost**, **-xAVX**, **-xCORE-AVX2**: In the case of **xHost** the compiler generates code for that ISA which best fits the micro architecture of the target computer (see **march** flag for GCC). In the case of **-xAVX** and **-xCORE-AVX2** the compiler tries to generate code with application of the given ISA extension (in this case AVX and AVX2). These flags are analogous to **-march=native**, **-mavx**, **-mavx2** for the GCC.
- \* **-mtune=cpu-type**: The code will be optimized towards the selected cpu type (for example **cpu-type=broadwell**), but unlike **-xHost** extended instruction sets are not used what means, that the code is portable to other CPUs<sup>18</sup>.
- \* **-Ot**: All speed optimizations are enabled. See **-Ofast** for GCC.
- \* **-fast**: This flag optimizes the code for speed over the whole program<sup>19</sup>.
- \* **-vec**: This flag enables auto-vectorization (see **-ftree-vectorize** (GCC)).
- \* **-parallel**: Enables auto-parallelization.
- \* **-prof\_gen** / **prof\_use**: Using this flag generates in a first step a profiling report for the compiler (**-prof\_gen**). To process the report data the user has to run the instrumented version of the program a few times (to collect performance data/issues and to train the compiler) and then to recompile the target file again with the **-prof\_use** flag (without **-prof\_gen**).

<sup>17</sup>[https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86\\_002d64-Options.html](https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86_002d64-Options.html)

<sup>18</sup><https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-mtune-tune>

<sup>19</sup><https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-profile-guided-optimization-pgo-options>

Running the optimized program often yields a significant speed-up up to 15-20 %<sup>20</sup>.

- \* `-ipo` (interprocedural optimization): Setting this flag enables the Intel compiler to optimize across file borders what could lead to significant performance gain<sup>21</sup>.
- Try different compiler. In general the Intel compiler is the compiler, which generates faster code than other compilers, while the GCC is freely available. It's worth trying, which of the available compiler does the best job for your needs. This refers to different compilers as well as to different versions of a compiler. For example it is advisable to benchmark the program with more than one version of a compiler. In the *PECOH* project<sup>22</sup> it was for example seen, that especially the Finite Element Sea Ice-Ocean Model *FESOM2*<sup>23</sup> were more performant with the 2018 Intel compiler than with the 2019 one.
- OpenMP: To parallelize a program on the basis of the shared memory paradigm on multi core with threads, the OpenMP API (Open Multi-Processing) is very popular (an alternative is PThreads, but OpenMP is more convenient). To enable OpenMP the user have to include the appropriate header file `omp.h` (in C/C++), to include the code with the appropriate pragmas and OpenMP library functions and to compile the source file with `-fopenmp` (GCC) and `-openmp` (Intel).
- OpenACC: Using the GPGPU *OpenACC* is the equivalent to OpenMP and supports a heterogenous paradigm (that means the calculations are done on the CPU and the GPGPU). For further details please see 2.3.3.
- To get reports about the vectorization and optimization process the Intel compiler provides the flags `-vec-report` and `-qopt-report` which are reporting the successful and non successful optimization steps.

## Job run

- Try to reduce the queuing time of the job by
  - \* explicitly defining the requested wall-clock time, since otherwise the default maximum requested walltime of the queue will be used, which could lead to an unnecessary waiting time of the job;
  - \* explicitly adjusting the requested wall-clock time to the expected or measured wall-clock time.
- Using the right queue to gain for example enough job memory or GPU resources.
- Configure the batch job with the right parameters (queue parameter, job specific parameter).

<sup>20</sup><https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-profile-guided-optimization-pgo-options>

<sup>21</sup><https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo-options>

<sup>22</sup><https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start>

<sup>23</sup><https://fesom.de/models/fesom20/>

- Use process and/or thread pinning!
- Experiment with the speed-up rate, because a low speed up/bad scalability should result perhaps in fewer requested nodes.

### 2.3.3 Using accelerators( OpenACC)

In this subsection we will give an example on how to parallelize a sequential program by using CPU and GPGPU capabilities. But why using additional compute capabilities others than the CPU? There are several reasons:

- In recent days an increase of performance was reached by tuning the frequency, what means, a doubling of the frequency approximately doubles the performance. But free lunch is over because this kind of performance tuning leads from a defined frequency to very high power needs and a strong heat creation.
- The performance of the CPU grows faster, than the bandwidth of the memory bus. As a consequence the CPU has to wait longer for new data to compute.
- Problemsolving: Since single Core performance optimization with the traditional methods comes to an end, more cores on a die relaxed the problem. But with this kind of performance tuning parallelization entered the game and new programming paradigms has to applied. But since the CPU has its limitations, the graphic cards with its thousand of trivial streaming processors can do much more work in parallel, than the CPU and is potentially faster. This is the reason to use the cheap alternative offloading.

This subsection should be just an appetizer and covers very basic aspects of the parallelization using *OpenACC* which is a parallel programming paradigm for heterogeneous systems (CPU and GPGPU). It uses compiler pragmas (compiler directives), supporting functions by the runtime library and environment variables for parallelization. In contrast to *CUDA*, which is a low level extension of C/C++ and Fortran, it allows an incremental insertion of parallelization, which means, that the user can successively select regions of the serial code and try to parallelize them. These regions are in general loops, i.e. loop parallelization like in *OpenMP* will be done and the directive based approach is very similar to *OpenMP*. *OpenACC* was introduced by *Cray*, *CAPS*, *Nvidia* and *PGI* in 2012 for C/C++ and Fortran. The current version is 2.7 (2018), which is implemented in the *PGI* compiler (2.5 in the GNU compiler, version 9.2) .

In the following key aspects of *OpenACC* will be presented in listing 2.11. This code is an example implementation of the discretization of the instationary 2D heat conduction problem (an instationary linear partial differential equation of the order two with non vanishing right hand side; see the serial examplecode in section ??). This example was formulated using only linear arrays (vectors) instead of matrices, because of the problem, that the rows of dynamical allocated matrices are not necessarily neighboured in the memory which can lead to problems in accessing data. Without the `#pragma` regions one gets the serial version of the program. Basically the code is separated in parts which are calculated by the CPU and those, which are managed by the GPGPU (see Listing 2.11). If only one loop is executed in the GPGPU paragraph, then the surrounding braces can be omitted.

```

1 Sequential code (Code executed on the CPU)
2
3 #pragma acc parallel
4 {
5
6     #pragma acc parallel
7
8     #pragma acc loop
9         for (int i = 0; i < n; ++i )
10             (Code executed on the GPU)
11
12 }
13
14 Sequential code (Code executed on the CPU)

```

Listing 2.9: Five point stencil (OpenACC)

Outside the parallel region the code will be executed on the CPU. When creating a parallel region, the *OpenACC* compiler will create one or more gangs, which are executed in parallel und redundantly. A gang can execute for example a part of the for loop (that means the first gang will execute the first 100 loops, the second one the next 100 loops, etc.) and all gangs together executes the whole loop.

Now let's have a look at the example. The fully example can be downloaded from <https://profit-hpc.de/downloads/>. Until now (11/2019) only the PGI compiler supports the latest *OpenACC specification* (2.7), while the newest GNU compiler 9.2 supports the *OpenACC specification* 2.5, while the Intel compiler does not support *OpenACC*. In the following we use the PGI compiler 19.7. To compile the example below, the following line will manage that:

```
pgc++ -fast -Minfo=accel -ta=tesla:managed
```

The compilation flags are as follows:

- **-fast:** Optimizes the code as much as possible
- **-Minfo=accel:** Gives feedback about the parallelization procedure on the GPGPU with *OpenACC*. There are further other flags like **opt** (informs about code optimization) or **all** (feedback about all compilation steps).
- **-ta=tesla:managed:** Compiles for using the *OpenACC* code for the tesla architecture. In the case of **-ta=multicore** the code will be compiled to run on a multicore CPU by using threads. On using the additional managed flag instructs the compiler to build code for the GPGPU and the data movement will be managed automatically.

The compilation process generates in this case the following output:

```

1 231, Generating copyin(q[:N*N])
2     Generating copy(err,Aold[:N*N])

```

```

3      Generating copyin(D,dt,A[:N*N],dx)
4 237, Generating copyin(q[:N*N])
5      Generating copy(err)
6      Generating copyin(Aold[:N*N],D,dt,dx)
7      Accelerator kernel generated
8      Generating Tesla code
9      237, Generating reduction(max:err)
10     239, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
11     242, #pragma acc loop seq
12 242, Loop carried scalar dependence for err at line 250
13 256, Generating copyout(Aold[:N*N])
14     Generating copyin(A[:N*N])
15     Accelerator kernel generated
16     Generating Tesla code
17     259, #pragma acc loop gang /* blockIdx.x */
18     260, #pragma acc loop vector(128) /* threadIdx.x */
19 260, Loop is parallelizable

```

Listing 2.10: Five point stencil (OpenACC)

The line numbers at the left hand side are showing where the following text refers to. The most important message is, that the code has no errors and that it is parallelizable (line 260 respective line 19). In the lines 1 until 6 and 13 until 14 the data movement of the compiler for every loop is documented, in the lines 10 and 11, 17 and 18 the segmentation of the parallel loops is documented.

Now we will have a closer look to the jacobi kernel, which is the only region of the code, where parallelization was done. In the lines 18 until 49 and 52 until 55 of the listing two parallel regions are defined, which will be executed on the GPGPU. In the first parallel region (which consists of a combined *#pragma acc parallel loop*) the update of the values in every timestep in the nodes of the matrix/region will be done. Additionally a reduction will be executed, what means, that in every gang the value of the variable *err* is calculated (of the appropriate part of the loop) and at the end of the region the overall maximum will be calculated and stored globally in *err* (the other *err* values will be local to the appropriate gang). This method is called *reduction* because out of a set of values one value will be calculated with the help of an operator (reduced to one value). In our case we use the max operator as the reduction operator. Other operators are:

- +, \*
- max, min,
- &, | (bitwise and and or),
- &&, || (logical and and or).

Using the below example of code it can be demonstrated how the code can be iteratively parallelized. In the first step identify all regions, which are parallelizable, then, in the first parallelization step, parallelize the lines 18 until 49, compile the code and check, if there are no errors occurring. Further check if the optimization procedure provides the performance, you expected and then do the second parallelization step in the equal

manner. If the performance is not sufficient, repeat optimization and parallelization until the performance needs are met.

```

1  int JacobiSolver( int N, double dx )
2  {
3
4      int iterations = 0;
5
6      double err = tolerance;
7      double D = dt / ( dx * dx ) ;
8
9      //
10     // Start the iterative solver
11
12     // Iteration until the solver converges
13     while( (iterations < MaxIterations) && (err >= tolerance) )
14     {
15
16         err = tolerance;
17
18         #pragma acc parallel loop reduction( max: err)
19         copyin( Aold[ 0: N * N ], q[ 0: N * N ], D, dt, dx )
20         copy( err )
21         {
22
23             // Iterate over all nodes of the prescribed area/matrix
24             // (i counts to the y direction, j counts to the x direction)
25             for( int i = 1; i < N - 1; ++i )
26             {
27
28                 for( int j = 1; j < N - 1; ++j )
29                 {
30
31                     // Calculate the new values from the values of the 4 neighbours and the
32                     // (i,j) node of the previous time step
33                     A[ j + i * N ] = dt * q[ j + i * N ]
34                                     + Aold[ j + i * N ]
35                                     + D * ( Aold[ ( j + 1 ) + i * N ]
36                                             + Aold[ j + ( i + 1 ) * N ]
37                                             - 4.0 * Aold[ j + i * N ]
38                                             + Aold[ ( j - 1 ) + i * N ]
39                                             + Aold[ j + ( i - 1 ) * N ] );
40
41                     // Calculate the error of the iteration (in this case maximum error,
42                     // the euclidian error is also applicable.
43                     err = max(err, fabs(Aold[j + i * N] - A[j + i * N]));
44
45                 }
46
47             }
48
49         }
50

```



```

51 // Swap the arrays
52 #pragma acc parallel loop copyin(A[0: N * N]) copyout(Aold[0: N * N])
53     for( int i = 0; i < N; ++i )
54         for( int j = 0; j < N; ++j )
55             Aold[ j + i * N ] = A[ j + i * N ];
56
57
58     ++iterations;
59
60 }
61
62
63 if ( iterations < MaxIterations )
64     return iterations;
65 else
66     return MaxIterations;
67 %
68 }

```

Listing 2.11: Five point stencil (OpenACC)

Using the PGI C compiler we got an approximate speed up of eight in this example using the GPGPU (Nvidia Tesla K80). An interesting observation was, that using the multicore option the speed up was approximately twelve (node has 16 cores with two hyperthreads each). This was against our expectations, but the speed up value of eight was close to other benchmark on K80 GPGPUs of other users.

Some concluding remarks:

- Dynamical allocated memory have to declared in C/C++ together with the **restrict** keyword, to prevent preventing parallelization (pointer aliasing).
- No jumps out of or into the parallel region should be done (with break, return, goto, etc.)
- Never forget to submit the job to a queue qith GPGPU capabilities. Otherwise the job will fail.

An free introduction to *OpenACC* can be found on [7].

### 2.3.4 OpenMP

In this subsection the *OpenMP* version of listing 2.22 will be presented. *OpenMP* is a pragma based shared memory parallel programming paradigm. The difference to *OpenACC* is, that *OpenMP* acts on the CPU with threads and not on the GPU. The *OpenMP* API was presented in 1997 (Fortran) and 1998 (C/C++) in version 1.0. *OpenMP* was developed by several companies and institutions, for example AMD, IBM, Intel, Nvidia, NEC and more (the OpenMP Architecture Review Board (OpenMP ARB)). It is now (since 11/2018) in the version 5.0. Like *OpenACC* *OpenMP* supports loop parallelization with threads. *OpenMP* uses compiler pragmas, library functions and environment variables for parallelization and in contradiction to MPI *OpenMP* can be used for parallelization of programs on a multicore system, while MPI in general is used for internode



communication.

The following *OpenMP* listing parallelizes listing 2.22. We will introduce very basic *OpenMP* features to get an idea how to parallelize the serial code. For a deeper introduction to *OpenMP* please see for example [8] and [9]. As in the *OpenACC* case the code can be parallelized iteratively and consists of sequential and parallel parts (which are beginning with `#pragma omp parallel`). When the program enters the parallel part, a team of threads will be created. This team consists of a master thread and a number of additional threads which solve the problem in the specified region together (for example in a SPMD or a workshare manner). At the end of the parallel region, all threads (except the master thread) are going to sleep and will be woken up, when the code enters the next parallel region (which could be the same parallel region again). This is sketched in the listing 2.12.

To use *OpenMP*, the C/C++ programmer (we will restrict to C/C++ in this document) has to include the following line into the main source file (first line):

```
1  #include <omp.h>
2
3  Sequential code (Code executed on the CPU)
4
5  #pragma omp parallel
6  {
7
8      #pragma omp for
9      for (int i = 0; i < n; ++i)
10         (Code executed by team of threads on the CPU)
11
12 }
13
14 Sequential code (Code executed on the CPU)
```

Listing 2.12: Include OpenMP header in C/C++ and the main idea of OpenMP (sequential and parallel regions).

Furthermore a special flag has to be set for compiling an *OpenMP* program. We will present this flag for the Intel, GNU and PGI C compiler (every compiler with an example compilation line):

- Intel C compiler: `icc -openmp -o file file.c`
- GNU C compiler: `gcc -fopenmp -o file file.c`
- PGI C compiler: `pgcc -mp -o file file.c`

In our example case in line 14 of the listing the parallel region will be created with `#pragma omp parallel` and the library function `num_threads` creates as much threads as demanded (in our case the number of processors or hardware threads, which is implementation dependent of the compiler). In the next `#pragma` line, the following two for loops will be parallelized. At the moment we will only present the `#pragma omp for` part of the line. Without the `collapse` and `schedule` clause the outer loop will

be distributed over the threads (if possible evenly) to process them in parallel, so that every thread executes a part of the for loop. With the additional `schedule` clause, it is possible to control the kind of scheduling of the for loop over the team of threads. The following possibilities can be chosen.

- `schedule( static[,chunk] )`: Deal-out to every thread a chunk of initial or manually specified size of the for loop. For example a loop with 1000 iterations and a chunk size of 100 the 10 threads get for example 100 for loops and are working on them. This kind of scheduling is the default setting and has the fewest overhead, but a lack of this kind is the missing flexibility while scheduling.
- `schedule( dynamic[,chunk] )`: Deal-out to every thread a variable amount of loops. If a thread has finished its workload it grabs a new amount of loops (but this value is unpredictable). Since the logic of this case is complex, this kind of scheduling often has an overhead.
- `schedule( guided[,chunk] )`: Special case of the dynamical case to reduce scheduling.
- `schedule( auto )`: In this case the runtime can learn from further runs of the same loop.

In very often cases, a static scheduling is recommended because of the minor scheduling overhead (static scheduling is the default adjustment). Finally the `collapse` clause fuses those two for loops (which have to be perfectly nested loops, what means, that they come directly after each other) to one loop, to exploit further parallelism. The number in the clause denotes the number of loops, which should be collapsed. If this number equals for example to four, then four perfectly nested loops can be collapsed to one loop. The analogous situation can be seen in the next parallel loop and the parallel region ends at line 36. From line 37 until the end of the outer for loop the code will be processed again as a serial program and begins as a serial program at the head of the outer for loop, what means that the team of threads will be reduced to one thread again. If the code enters the parallel region again, the threads are activated again.

```

1  int JacobiSolver( int N, double dx )
2  {
3
4      double D = dt / ( dx * dx ) ;
5
6      int num_threads = omp_get_num_procs();
7
8      //
9      // Start the iterative solver
10
11     for ( int k = 0; k < MaxIterations; k++ )
12     {
13
14     #pragma omp parallel num_threads( num_threads )
15     {
16
17     #pragma omp for collapse( 2 ) schedule( static )
18         for( int i = 1; i < N - 1; ++i )

```

```

19     for( int j = 1; j < N - 1; ++j )
20         A[ j + i * N ] = dt * q[ j + i * N ]
21             + Aold[ j + i * N ]
22             + D * ( Aold[ ( j + 1 ) + i * N ]
23             + Aold[ j + ( i + 1 ) * N ]
24             - 4.0 * Aold[ j + i * N ]
25             + Aold[ ( j - 1 ) + i * N ]
26             + Aold[ j + ( i - 1 ) * N ] );
27
28 #pragma omp for collapse( 2 ) schedule( static )
29     for( int i = 0; i < N; ++i )
30         for( int j = 0; j < N; ++j )
31             Aold[ j + i * N ] = A[ j + i * N ];
32
33
34     }
35
36 }
37
38 return MaxIterations;
39
40 }
```

Listing 2.13: Five point stencil (OpenMP)

To gain better performance it is recommended to bind the threads while runtime to a CPU or SMT entity. This can be done for example on the command line with (BASH style):

```
export OMP_PROC_BIND=true
```

Otherwise it could happen, that the threads will change the CPU entities while runtime because of scheduling. This leads to runtime overhead and is decreasing performance. It is also possible to set thread binding via a library function. For more details please see<sup>24</sup>.

Finally the similarities of the initiating the parallelization with *OpenMP* and *OpenACC* are obvious, since both paradigms are using very similar pragmas and the same kind of parallelization method (directive based parallelization).

### 2.3.5 CUDA

In 2007 NVIDIA published the programming platform and programming model CUDA with a C/C++ language extension for computations on CPUs and NVIDIA GPUs. The sequential part of the CUDA program will be executed on the host (CPU) and special marked regions will be offloaded to be computed on the GPU (device). Offloading parts of a program to be computed on the NVIDIA GPU has the advantage, that the computation on the GPU is much more efficient, than on the CPU and speed up often will increase dramatically. CUDA was introduced as a C platform but nowadays it is

<sup>24</sup><https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf>

extended to Python, Fortran or Matlab.

Let's start looking at CUDA by examining some characteristics of the onboard GPU(s), for example the number of GPUs, the major and minor number of the GPU capabilities, the amount of global, shared and constant memory, the number of threads and blocks per direction, etc.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5
6  int main( int argc, char **argv )
7  {
8
9      cudaDeviceProp prop;
10
11     int count;
12     cudaGetDeviceCount( &count );
13
14     for ( int i = 0; i < count; ++i )
15     {
16
17         cudaGetDeviceProperties( &prop, i );
18
19         fprintf( stdout, "GPU number: %d\n", i );
20         fprintf( stdout, "GPU count: %d\n", count );
21         fprintf( stdout, "GPU name: %s\n", prop.name );
22         fprintf( stdout, "GPU major number: %d\n", prop.major );
23         fprintf( stdout, "GPU minor number: %d\n", prop.minor );
24         fprintf( stdout, "GPU total global memory: %zd (in GB)\n",
25             ( int ) ( prop.totalGlobalMem / 1024 / 1024 / 1024 ) );
26         fprintf( stdout, "GPU total constant memory (in KB): %d\n",
27             ( int ) ( prop.totalConstMem / 1024 ) );
28         fprintf( stdout, "GPU shared memory per block (in KB): %d\n",
29             ( int ) ( prop.sharedMemPerBlock / 1024 ) );
30         fprintf( stdout, "GPU maximum count of threads per block: %d\n",
31             ( int ) ( prop.maxThreadsPerBlock ) );
32         fprintf( stdout, "GPU maximum count of threads per dimension (x,y,z):
33             (%d, %d, %d)\n", prop.maxThreadsDim[ 0 ], prop.maxThreadsDim[ 1 ],
34             prop.maxThreadsDim[ 2 ] );
35         fprintf( stdout, "GPU maximum count of blocks per dimension (x,y,z):
36             (%d, %d, %d)\n", prop.maxGridSize[ 0 ], prop.maxGridSize[ 1 ],
37             prop.maxGridSize[ 2 ] );
38         fprintf( stdout, "GPU warp size: %d\n", prop.warpSize );
39         fprintf( stdout, "-----\n\n" );
40
41     }
42
43     return 0;
44
45 }
```

Listing 2.14: Listing to get the characteristics of the GPUs.

To compile this program the following requirements have to be fulfilled:

- A node/computer with a CUDA capable graphics card,
- a NVIDIA device driver,
- the NVIDIA CUDA toolkit and
- a C compiler and CUDA capable compiler (**nvcc** in our case) to compile the host and device code separately. The host part of the code will be compiled with the C compiler (for example GNU C Compiler) and the device code with the CUDA C compiler (**nvcc**) only with the call of **nvcc**. At the end of this step, the serial part and the CUDA part of the program will be bound to one program and executed.

To compile this program, use for example the following line (filename must end with **.cu**):

```
nvcc -arch=sm_35 -o getProperties getProperties.cu
```

in which **-arch=sm\_35** means, that we will use at least the compute capability 3.5 of the NVIDIA GPGPU. An example result of the program run looks as follows:

```

1 GPU number: 0
2 GPU count: 2
3 GPU name: Tesla K80
4 GPU major number: 3
5 GPU minor number: 7
6 GPU total global memory: 11 (in GB)
7 GPU total constant memory (in KB): 64
8 GPU shared memory per block (in KB): 48
9 GPU maximum count of threads per block: 1024
10 GPU maximum count of threads per dimension (x,y,z): (1024, 1024, 64)
11 GPU maximum count of blocks per dimension (x,y,z): (2147483647, 65535, 65535)
12 GPU warp size: 32
13
14 -----
15
16 GPU number: 1
17 GPU count: 2
18 GPU name: Tesla K80
19 GPU major number: 3
20 GPU minor number: 7
21 GPU total global memory: 11 (in GB)
22 GPU total constant memory (in KB): 64
23 GPU shared memory per block (in KB): 48
24 GPU maximum count of threads per block: 1024
25 GPU maximum count of threads per dimension (x,y,z): (1024, 1024, 64)
26 GPU maximum count of blocks per dimension (x,y,z): (2147483647, 65535, 65535)
27 GPU warp size: 32

```

Listing 2.15: List of the characteristics of the GPUs.

This cluster node has 2 NVIDIA Kepler K80 GPGPUs with total global memory of 11 GiB each. The compute capability number is 3.7. and describes the architecture of the GPGPU, for example the sizes of the global and shared memory, the number of registers, the features of the GPGPU. The number of threads is bounded to 1024 threads per block for the Kepler K80 GPGPU. The size of the constant memory is 48 KiB and the size of the shared memory is 48 KiB per block. The meaning of these characteristics will be explained later.

After getting a short overview over the hardware characteristics, we will start with a first example. In general a CUDA introduction will begin with the vector addition, the hello world example for CUDA. We will continue this tradition and in this example we will add two dynamically allocated vectors *a* and *b* on the GPU and store them in a third vector *c*. Listing 2.16 illustrates this by using CUDA.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  #define N 204748364
6  #define N_THREADS 16
7  #define OUTPUT
8
9  __global__ void vectoraddition( double *a, double *b, double *c )
10 {
11
12     unsigned long long int tid = threadIdx.x + blockIdx.x * blockDim.x;
13
14     if ( tid < N )
15     {
16
17         c[ tid ] = a[ tid ] + b[ tid ];
18         tid += blockDim.x * gridDim.x;
19
20     }
21
22 }
23
24
25 int main( int argc, char **argv )
26 {
27
28     double *a = ( double * ) malloc( N * sizeof ( double ) ) ;
29     double *b = ( double * ) malloc( N * sizeof ( double ) ) ;
30     double *c = ( double * ) malloc( N * sizeof ( double ) ) ;
31
32     double *d_a = NULL;
33     double *d_b = NULL;
34     double *d_c = NULL;
35
36     cudaMalloc( ( void** )&d_a, N * sizeof( double ) );
37     cudaMalloc( ( void** )&d_b, N * sizeof( double ) );
38     cudaMalloc( ( void** )&d_c, N * sizeof( double ) );

```

```

39
40 //
41 // Initialize the vectors a and b (vectors on host)
42 for( unsigned long long int i = 0; i < N; ++i )
43 {
44
45     a[ i ] = 1.0 * i;
46     b[ i ] = 1.0 * i;
47
48 }
49
50 cudaMemcpy( d_a, a, N * sizeof( double ), cudaMemcpyHostToDevice );
51 cudaMemcpy( d_b, b, N * sizeof( double ), cudaMemcpyHostToDevice );
52
53 vectoraddition<<< ( N + ( N_THREADS - 1 ) ) / N_THREADS, N_THREADS >>>
54     ( d_a, d_b, d_c );
55
56 cudaMemcpy( c, d_c, N * sizeof( double ), cudaMemcpyDeviceToHost );
57
58 //
59 // free allocated memory on device
60 cudaFree( d_a );
61 cudaFree( d_b );
62 cudaFree( d_c );
63
64 //
65 // free allocated memory on host
66 free( a );
67 free( b );
68 free( c );
69
70 return 0;
71
72 }

```

Listing 2.16: Listing of the vectoraddition in CUDA C.

This program is divided into 5 main parts:

- (Serial Code),
- allocating space for the vectors **a**, **b** and **c** in the host memory and initialization of *a* and *b* (line 28 - 30, 42 - 48),
- allocating space for the vectors **a**, **b** and **c** (resp. **d\_a**, **d\_b** and **d\_c**) on the device memory (line 32 - 38),
- copy the contents of the vectors **a** and **b** (host) to **d\_a** and **d\_b** (device),
- preparing and performing computations on the GPGPU (line 53 f. and 9 - 22),
- Copy the contents of the resultvector **d\_c** back to the host in vector **c** (line 56),
- Freeing allocated memory on device and host (see line 60 - 68),

- (Serial Code).

This is a often used approach in writing CUDA programs, that means executing the (serial) code on the CPU, allocating host and GPGPU memory, copying the contents of the memory of the host datastructures to the respective GPGPU datastructures, computing on the GPGPU and copying back the result. Now we will go more into detail by means of the above vectoraddition example. In the first step the serial code on the CPU will be executed. The next two steps are considering the memory allocation steps. In the first step the memory space for the vectors **a** and **b** will be dynamically allocated on the host memory (main memory) with `malloc` (line 28 - 30). In the second step dynamical allocating of memory on the device (GPU) will be executed, to store the vectors **a**, **b** and **c** in the GPU memory vectors **d\_a**, **d\_b** and **d\_c**. The additional allocation of memory on the GPGPU is necessary, since host and GPGPU memory have different address spaces and no automatical access to the data of the respective other memory is possible. If the host needs access to the GPGPU data (and vice versa), the data has to be copied from the device to the host (and vice versa). The vectors in the host memory (**a**, **b** and **c**) and in the device memory (**d\_a**, **d\_b** and **d\_c**) have the same number of elements and corresponds to each other in that way, that in the next step the data of vector **a** will be copied into the array **d\_a** and the data of vector **b** into the vector **d\_b** on the GPGPU by using the function `cudaMemcpy`. `cudaMemcpy` copies the contents of the vector **a** of the size `N * sizeof( double )` from the source **a** to the destination vector **d\_a** (the same for **b** and **d\_b**; see line 50 f.). The direction of copying is defined in the last argument of `cudaMemcpy`. In our case the direction of copying is given by `cudaMemcpyHostToDevice`, what means that the copyprocess takes place from **a** (**b**) to **d\_a** (**d\_b**). Finally it has to be mentioned, that the host vectors **a** and **b** were filled with values in line 42 until 48.

After presenting these four introductory steps, the actual computation process on the GPGPU can take place. This will be done with the call of the kernel(function) `matrixaddition`. A kernel is a part of the program (a function) which will run on the GPGPU. This kernel and its launch is the core of this listing and a kernel call (kernel launch) is a classical function call (in this case with the arguments `d_a`, `d_b` and `d_c` (see the round braces at the end of the function call)) with additional CUDA specific parameters (a defined number of threads and block of threads). In the following we will disclose the secret about the three `>` and `<`. The basic working unit of GPGPUs and of CUDA is the thread. Furthermore a predefined number of threads defines a block of threads and these block of threads constitutes a grid of blocks. This paradigm is called SIMT (single instruction, multiple threads), a subclass of SIMD (single instruction, multiple data). Every thread and block of threads gets its own unique identity number (`threadId` and `blockId`) to distinguish threads and blocks among themselves. We will illustrate this in figure 2.1 for the 1D case. In this example 32 threads can be found which are aggregated into four blocks of threads (each block consists of eight threads). `threadIdx.x` denotes the number of a thread in a threadblock, `blockIdx.x` the number of the block of threads. The global number of a thread can be computed as follows

```
unsigned long long int tid = threadIdx.x + blockIdx.x * blockDim.x
;
27 = 3 + 3 * 8
```



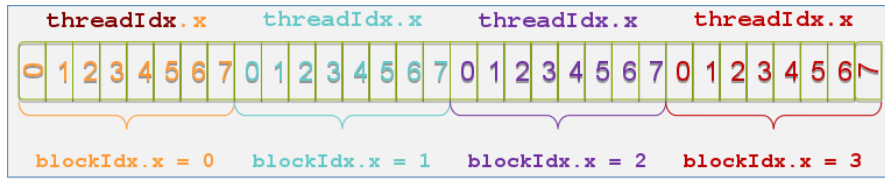


Figure 2.1: Exemplary presentation of the threads and block of threads.

in which `blockDim.x` denotes the size of a threadblock in x direction. In this toy example the size of a block in x direction is 8 threads. With these values in mind, the calculation of the global thread index can be done. If we want to compute the Index of thread 27, the formula will be concetized as in the listing above.

To summarize the results of this paragraph, we will list the central terms here:

- The thread is the basic working unit. Every thread gets its own thread id which is stored in `threadIdx.x` for the x direction.
- A number of threads is aggregated to a block of threads which x dimension is stored in `blockDim.x` and the block index is stored in `blockIdx.x`.
- The blocks again are aggregated to a grid, which dimension in x direction is stored in `gridDim.x`.

Away from the above toy example in the case of our *Kepler K80* GPGPU case these constants have the following values (see listing 2.15):

- In every block the maximum number of threads is 1024.
- In x direction 1024 threads can be used (`blockDim.x = 1024`),
- and the maximum count of blocks in the x direction of the grid is 2147483647 (`grdDim.x = 2147483647`).

After this theory paragraph we will explain the call of the kernel and the kernel in detail. In line 53 f. we will call the kernel with the definition of the layout of blocks and threads to solve the problem. In one dimension (especially in the case of the vectoraddition), the call could be of the following kind:

- `matrixaddition<< 1, 1 >>( d_a, d_b, d_c ),`
- `matrixaddition<< N, 1 >>( d_a, d_b, d_c ),`
- `matrixaddition<< 1, M >>( d_a, d_b, d_c ),`
- `matrixaddition<< N, M >>( d_a, d_b, d_c )`

Case 1 is equal to a single thread running on a CPU (but in this case on the GPGPU). To be more precisely the kernel will be launched in one block and with one thread. In the second case the kernel runs on  $N$  blocks and in each block only one thread will be used. The third case is vice versa, that means the kernel runs in one block and on  $M$  threads. In the last case the kernel will be executed on  $N$  blocks while in each block  $M$  threads are activated. Apart from these general examples in our case we start the kernel on  $N\_THREADS$  per block and with  $(N + (N\_THREADS - 1)) / N\_THREADS$  blocks. The correction of  $N$  in the nominator with the value of  $(N\_THREADS - 1)$  is caused by the fact, that the integerdivision will lead without correction to zero blocks (if  $N < N\_THREADS$ ) and thus no threads will be started or we will get too little blocks (exactly one block will be missing, unless  $N \bmod N\_THREADS = 0$ ).

The kernel will be executed by each thread and in the kernel `vectoraddition` the index of the thread will be calculated, which will do the computation for the element `tid`. If this index is smaller than  $N$  the calculation of the appropriate element of the two vectors will be done (line 17) and the index will be incremented (line 18). The kernel execution will end, if for all threads the condition is evaluated to false. This is a kind of implicit barrier and after finishing the kernel the code will be further executed in line 56. In this line the result of the vectoraddition will be copied back from the device memory (`d_c`) to the host array (`c`). After that operation it is possible to dump the vector to a file on disk or to send it to `stdout` (output on the screen).

After computing the matrix sum and copying the result back to the host, the allocated GPU memory can be freed with `cudaFree`, since it is not needed anymore (see lines 75 - 77).

At the end of this example we will consider the speed up of the CUDA vectoraddition. In the first case we will consider the speed up of the CUDA program with 1 block and one thread compared to the multiblock and multithread version. In the second example we will consider the Speed up of a single core and single threaded CPU program with the CUDA version with multiple blocks and threads.

Now we will slightly expand our example to 2D what means, that we will discuss matrix addition with CUDA by reference to listing 2.17.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5
6  #define N 2048
7  #define N_THREADS 32
8  #define N_BLOCKS 8
9  #define OUTPUT
10
11 __global__ void matrixaddition( double *a, double *b, double *c )
12 {
13
14     unsigned long int col_tid = threadIdx.x + blockIdx.x * blockDim.x;
```

```

15     unsigned long int row_tid = threadIdx.y + blockIdx.y * blockDim.y;
16
17     unsigned long int index = col_tid + row_tid * N;
18
19     while ( ( col_tid < N ) && ( row_tid < N ) )
20     {
21
22         c[ index ] = a[ index ] + b[ index ];
23
24         col_tid += blockDim.x * gridDim.x;
25         row_tid += blockDim.y * gridDim.y;
26
27     }
28
29 }
30
31
32 int main( int argc, char **argv )
33 {
34
35     double A[ N ][ N ];
36     double B[ N ][ N ];
37     double C[ N ][ N ];
38
39     double *d_a = NULL;
40     double *d_b = NULL;
41     double *d_c = NULL;
42
43     cudaMalloc( ( void** )&d_a, N * N * sizeof( double ) );
44     cudaMalloc( ( void** )&d_b, N * N * sizeof( double ) );
45     cudaMalloc( ( void** )&d_c, N * N * sizeof( double ) );
46
47     for( int i = 0; i < N; ++i )
48         for( int j = 0; j < N; ++j )
49         {
50
51             A[ i ][ j ] = ( double ) ( i + j );
52             B[ i ][ j ] = ( double ) ( i + j );
53
54         }
55
56     cudaMemcpy( d_a, A, N * N * sizeof( double ), cudaMemcpyHostToDevice );
57     cudaMemcpy( d_b, B, N * N * sizeof( double ), cudaMemcpyHostToDevice );
58
59
60     dim3 dimBlock( N_THREADS, N_THREADS );
61     dim3 dimGrid( ( N + ( N_BLOCKS - 1 ) ) / N_BLOCKS, ( N + ( N_BLOCKS - 1 ) ) /
        N_BLOCKS );
62     matrixaddition<<< dimGrid, dimBlock >>>( d_a, d_b, d_c );
63
64     cudaMemcpy( C, d_c, N * N * sizeof( double ), cudaMemcpyDeviceToHost );
65
66 #ifndef OUTPUT

```

```

67     for( int i = 0; i < N; ++i )
68     {
69         for( int j = 0; j < N; j++ )
70             fprintf( stdout, "%lf ", C[ i ][ j ] );
71             fprintf( stdout, "\n" );
72     }
73 #endif
74
75     cudaFree( d_a );
76     cudaFree( d_b );
77     cudaFree( d_c );
78
79     return 0;
80
81 }
```

Listing 2.17: Addition of two square matrices.

In this 2D example, we have to define the number of blocks and threads in x AND y direction. Line 60 defines the number of threads in x and y direction of every block. In this concrete example we defined 32 threads for each direction. The layout can be chosen differently (for example  $N\_THREADS = 16$ ), but for every selection it must be taken into account, that the maximum value of threads will not be exceeded. To define the number of thread blocks in line 61 the matrix size  $N$  will be divided by the number of chosen blocks. Since the integer division may result in zero, the correction with  $(N\_BLOCKS - 1)$  will circumvent this problem. After defining the number of blocks and threads in one block, the call of the kernel `matrixaddition` can be done with the defined counts of blocks and threads. The kerneldefinition of `matrixaddition` starts in line 11. The `__global__` qualifier at the beginning of the function head declares this function as a kernel function which will be executed on the GPGPU and is only callable from the host. In this kernel (with the Matrices A, B and C as arguments) the `matrixaddition` will be executed. In the lines 14 and 15 the row and column index of the unique thread is calculated and some new constants are introduced to control the work of the threads, which we will now explain:

- `threadIdx.x`, `threadIdx.y`: The index of a thread in a block.
- `blockDim.x`, `blockDim.y`: The dimension (number of threads) of a block in x and y direction.
- `blockIdx.x`, `blockIdx.y`: The blockindex in x and y direction.

In line 17 the actual index of the matrixelement to be computed will be calculated and the expression

$$unsignedlongintindex = col_{id} + row_{id} * N;$$

has the following meaning:

To process the result of the GPGPU computations (for example by storing the data in a file), the data has to be copied back from the device memory to the host (main) memory. This task will be done by another call of `cudaMalloc`.

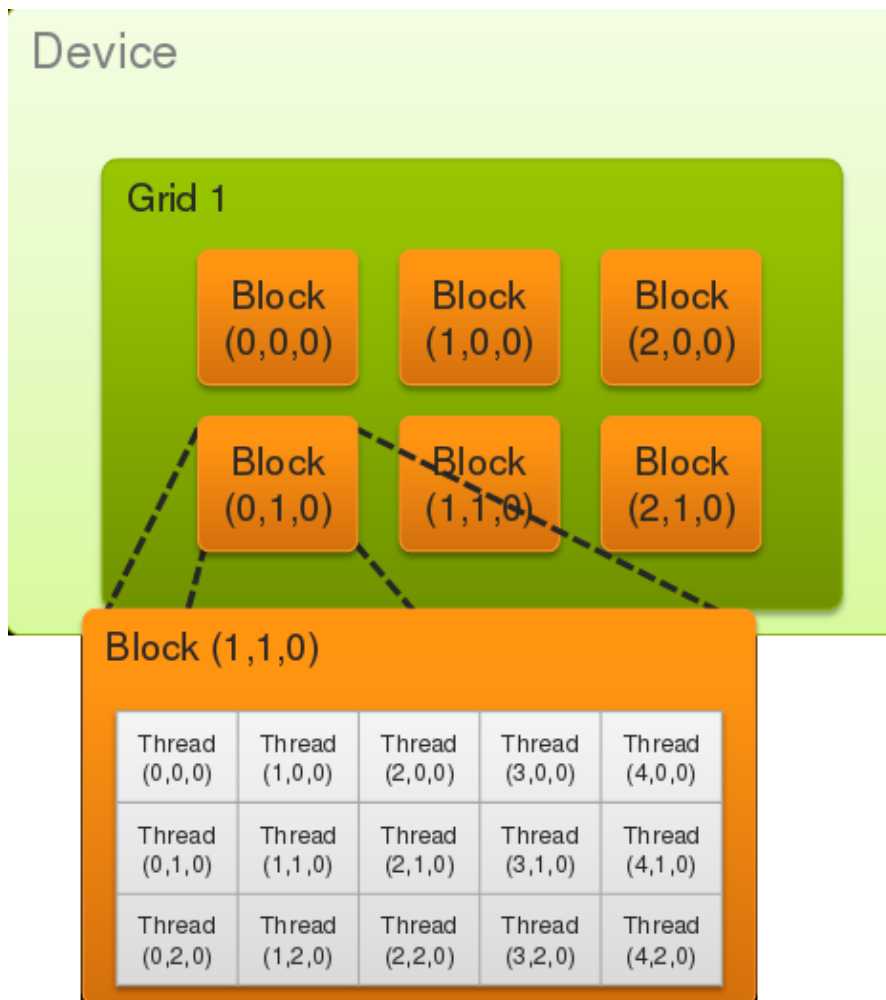


Figure 2.2: Exemplary presentation of the threads and block of threads in 2d.

After computing the matrix sum and copying the result back to the host, the allocated GPU memory can be freed with `cudaFree`, since it is not needed anymore (see lines 75 - 77).

At the end of this example we will consider the speed up of the CUDA vectoraddition. In the first case we will consider the speed up of the CUDA program with 1 block and one thread compared to the multiblock and multithread version. In the second example we will consider the Speed up of a single core and single threaded CPU program with the CUDA version with multiple blocks and threads.

The next two steps are considering the memory allocation steps. In the first step the memory space for the vectors **a** and **b** will be allocated on the host memory (main memory). In our case the allocation will be done in a static way, dynamical allocation is also possible. Furthermore we will treat the matrices on the host as 2D matrices, in which it is possible to flatten those datastructures to a vector of size  $N \times N$ . In the second step

dynamical allocating of memory on the device (GPU), to store the matrices A, B and C in `d_a`, `d_b` and `d_c`, will be done. This further allocation procedure has to be done, since host and GPU memory are different and CPU and GPGPU have no automatical access to the data of the other memory. To get access the data has to be copied from host to device or vice versa. The datastructures on the host (A, B and C) and the device (`d_a`, `d_b` and `d_c`) have the same number of elements and corresponds to each other in that way, that in the next step the data of A will be copied into the array `d_a` and the data of B into `d_b` by using the function `cudaMemcpy`. `cudaMemcpy` copies the contents of the size `N * N * sizeof( double )` from the source A to the destination `d_a`. The direction of copying is formulated in the last argument of `cudaMemcpy`. In our case the direction of copying is given by `cudaMemcpyHostToDevice`, what means that the copyprocess takes place from A (B) to `d_a` (`d_b`). Every thread gets its own unique identity number (ThreadID) to distinguish from each other threads. This paradigm is called SIMT (single instruction, multiple threads), a subclass of SIMD (single instruction, multiple data).

After these introductory three steps, the actual computation process on the GPGPU can take place. This will be done with the call of the kernel `matrixaddition`. A kernel is a part of the program (in our case a function) which will run on the GPGPU, especially allocated thread. This is the core of this listing and a kernel call is a classical function call (in this case with the arguments `d_a`, `d_b` and `d_c` (see the round braces at the end of the function call)) and additional CUDA specific parameters. In the following we will disclose the secret about the three `>>` and `<<`. The tasks CUDA will run on will be accomplished by hardware implemented threads. Today (01/2020) every GPGPU owns several thousands threads which can solve independantly a subproblem of the original problem. These threads are organized in blocks and these blocks are organized in a grid. We will illustrate this in figure ???. Every block consist of a number of threads and these are structured in each block in a 3D matrix like fashion. In our case this has the following consequences (see listing ???):

- In every block the maximum number of threads is 1024.
- In x direction 1024 threads can be used, in y direction too and in z direction only 64 threads are valid. If this example would be a vectoraddition, 1024 threads could be used (x direction) for a thread block which is in our case identical with the maximum number of threads in a block (see line *GPU maxThreadsPerBlock* in listing ???). If we want to partition a 2D area into blocks of threads, we generally need the threads of the y direction. But in this case we can only request for example 32 threads for the x and y direction since the maximum allowed numbers of threads of a block is 1024.

In the lines 60 and 61, we will define the layout of blocks and threads to solve the problem. In one dimension (especially in the case of the vectoraddition), the call could be of the following kind:

- `matrixaddition<< 1, 1 >>( d_a, d_b, d_c ),`
- `matrixaddition<< N, 1 >>( d_a, d_b, d_c ),`

- `matrixaddition<< 1, M >>( d_a, d_b, d_c ),`
- `matrixaddition<< N, M >>( d_a, d_b, d_c )`

Case 1 is equal to a single thread run on a CPU (but in this case on the GPGPU). To be more precisely the kernel will be launched on one block and one thread. In the second case the kernel runs on N blocks and in each block only one thread will be used. The third case is vice versa, that means the kernel runs in one block and on M threads. In the last case the kernel will be executed on N blocks while in each block M threads are activated.

In this 2D example, we have to define the number of blocks and threads in x and y direction. Line 60 defines the number of threads in x and y direction of every block. In this concrete example we defined 32 threads for each direction. The layout can be chosen differently (for example `N_THREADS = 16`), but for every selection it must be taken into account, that the maximum value of threads will not be exceeded. To define the number of thread blocks in line 61 the matrix size N will be divided by the number of chosen blocks. Since the integer division may result in zero, the correction with  $(N\_BLOCKS - 1)$  will circumvent this problem. After defining the number of blocks and threads in one block, the call of the kernel `matrixaddition` can be done with the defined counts of blocks and threads. The kerneldefinition of `matrixaddition` starts in line 11. The `__global__` qualifier at the beginning of the function head declares this function as a kernel function which will be executed on the GPGPU and is only callable from the host. In this kernel (with the Matrices A, B and C as arguments) the `matrixaddition` will be executed. In the lines 14 and 15 the row and column index of the unique thread is calculated and some new constants are introduced to control the work of the threads, which we will now explain:

- `threadIdx.x`, `threadIdx.y`: The index of a thread in a block.
- `blockDim.x`, `blockDim.y`: The dimension (number of threads) of a block in x and y direction.
- `blockIdx.x`, `blockIdx.y`: The blockindex in x and y direction.

In line 17 the actual index of the matrixelement to be computed will be calculated and the expression

$$unsignedlongintindex = col_{id} + row_{id} * N;$$

has the following meaning:

To process the result of the GPGPU computations (for example by storing the data in a file), the data has to be copied back from the device memory to the host (main) memory. This task will be done by another call of `cudaMalloc`.

After computing the matrix sum and copying the result back to the host, the allocated GPU memory can be freed with `cudaFree`, since it is not needed anymore (see lines 75 - 77).

At the end of this example we will consider the speed up of the CUDA vectoraddition. In the first case we will consider the speed up of the CUDA program with 1 block and one thread compared to the multiblock and multithread version. In the second example we will consider the Speed up of a single core and single threaded CPU program with the CUDA version with multiple blocks and threads.

### 2.3.6 MPI

In contrast to OpenMP, MPI is based on the distributed parallel paradigm. That means,

We will begin with a look at the vectoraddition

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mpi.h"
5
6  #define NR_ELEMENTS 1000000
7
8  int main( int argc, char **argv )
9  {
10
11     double a[ NR_ELEMENTS ], b[ NR_ELEMENTS ], c[ NR_ELEMENTS ];
12
13
14     MPI_Init( &argc, &argv );
15
16     int rank = 0;
17     int size = 0;
18
19     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
20     MPI_Comm_size( MPI_COMM_WORLD, &size );
21
22     double a_recv[ NR_ELEMENTS / size ];
23     double b_recv[ NR_ELEMENTS / size ];
24     double c_recv[ NR_ELEMENTS / size ];
25
26     if ( rank == 0 )
27         for ( int i = 0; i < NR_ELEMENTS; ++i )
28         {
29
30             a[ i ] = ( double ) i; //( rand() % 1000 );
31             b[ i ] = ( double ) i; //( rand() % 1000 );
32
33         }
34
35     MPI_Scatter( a, NR_ELEMENTS / size, MPI_DOUBLE, a_recv, NR_ELEMENTS / size,
36                MPI_DOUBLE, 0, MPI_COMM_WORLD );
37     MPI_Scatter( b, NR_ELEMENTS / size, MPI_DOUBLE, b_recv, NR_ELEMENTS / size,
38                MPI_DOUBLE, 0, MPI_COMM_WORLD );
39
40     for( int i = 0; i < NR_ELEMENTS / size; ++i )

```



```

39     c_recv[ i ] = a_recv[ i ] + b_recv[ i ];
40
41     MPI_Gather( c_recv, NR_ELEMENTS / size, MPI_DOUBLE, c, NR_ELEMENTS / size,
42               MPI_DOUBLE, 0, MPI_COMM_WORLD );
43
44     if ( rank == 0 )
45     {
46         for( int i = 0; i < NR_ELEMENTS; ++i )
47             fprintf( stdout, "%lf ", c[ i ] );
48
49             fprintf( stdout, "\n" );
50
51     }
52
53     MPI_Finalize( );
54
55     return 0;
56
57 }
```

Listing 2.18: Five point stencil (Serial version)

Now the vectoradditionexample will be slightly extended to the matrixaddition.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mpi.h"
5
6  #define NR_ELEMENTS 10000
7
8  int main( int argc, char **argv )
9  {
10
11     double A[ NR_ELEMENTS ][ NR_ELEMENTS ];
12     double B[ NR_ELEMENTS ][ NR_ELEMENTS ];
13     double C[ NR_ELEMENTS ][ NR_ELEMENTS ];
14
15
16     MPI_Init( &argc, &argv );
17
18     int rank = 0;
19     int size = 0;
20
21     double start, end;
22
23     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
24     MPI_Comm_size( MPI_COMM_WORLD, &size );
25
26     double A_recv[ NR_ELEMENTS / size ][ NR_ELEMENTS ];
27     double B_recv[ NR_ELEMENTS / size ][ NR_ELEMENTS ];
28     double C_recv[ NR_ELEMENTS / size ][ NR_ELEMENTS ];
29
```

```

30     start = MPI_Wtime();
31
32     if ( rank == 0 )
33     {
34
35         for ( int i = 0; i < NR_ELEMENTS; ++i )
36             for ( int j = 0; j < NR_ELEMENTS; ++j )
37             {
38
39                 A[ i ][ j ] = ( double ) ( i + j ); //( rand() % 1000 );
40                 B[ i ][ j ] = ( double ) ( i + j ); //( rand() % 1000 );
41                 C[ i ][ j ] = 0.0;
42
43             }
44
45     }
46
47     MPI_Scatter( A, ( NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, A_recv, (
48         NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, 0, MPI_COMM_WORLD );
49     MPI_Scatter( B, ( NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, B_recv, (
50         NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, 0, MPI_COMM_WORLD );
51
52     for( int i = 0; i < NR_ELEMENTS / size; ++i )
53         for ( int j = 0; j < NR_ELEMENTS; ++j )
54             C_recv[ i ][ j ] = A_recv[ i ][ j ] + B_recv[ i ][ j ];
55
56     MPI_Gather( C_recv, ( NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, C, (
57         NR_ELEMENTS / size ) * NR_ELEMENTS, MPI_DOUBLE, 0, MPI_COMM_WORLD );
58
59     if ( ( rank == 0 ) && ( NR_ELEMENTS < 10 ) )
60     {
61
62         for( int i = 0; i < NR_ELEMENTS; ++i )
63         {
64
65             for( int j = 0; j < NR_ELEMENTS; ++j )
66                 fprintf( stdout, "%lf ", C[ i ][ j ] );
67
68             fprintf( stdout, "\n" );
69
70         }
71
72     }
73
74     end = MPI_Wtime();
75
76     fprintf( stdout, "Runtime with %d processes: %lf!\n", rank, end - start );
77
78     MPI_Finalize( );
79
80     return 0;

```

80 }

---

Listing 2.19: Five point stencil (Serial version)

After treating two introductory examples to exemplify the usage of MPI, one further step to the Jacobi problem will be done. This is the matrixmultiplication in two different versions. In the first version the multiplication will be done with the blocking point to point operations send and receive (MPI\_Send and MPI\_Recv).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mpi.h"
5
6  #define NR_ELEMENTS 4
7
8  double frob_norm = 0.0, frob_norm_part = 0.0;
9
10 int main( int argc, char **argv )
11 {
12
13     double start = 0.0, end = 0.0;
14     double B[ NR_ELEMENTS ][ NR_ELEMENTS ];
15
16     int rank = 0, size = 0;
17     int err = 0;
18
19     MPI_Init( &argc, &argv );
20
21
22     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
23     MPI_Comm_size( MPI_COMM_WORLD, &size );
24
25     if ( size - 1 > NR_ELEMENTS )
26         MPI_Abort( MPI_COMM_WORLD, err );
27
28     double A_recv[ NR_ELEMENTS / ( size - 1 ) ][ NR_ELEMENTS ];
29     double C_recv[ NR_ELEMENTS / ( size - 1 ) ][ NR_ELEMENTS ];
30
31
32     if ( rank == 0 )
33     {
34
35         start = MPI_Wtime();
36
37         double A[ NR_ELEMENTS ][ NR_ELEMENTS ];
38         double C[ NR_ELEMENTS ][ NR_ELEMENTS ];
39
40         for ( int i = 0; i < NR_ELEMENTS; ++i )
41             for ( int j = 0; j < NR_ELEMENTS; ++j )
42             {
43
44                 A[ i ][ j ] = ( double ) ( i + j );

```

```

45         B[ i ][ j ] = ( double ) ( i + j );
46         C[ i ][ j ] = 0.0;
47
48     }
49
50     for( int i = 1; i < size; ++i )
51         MPI_Send( A[ ( i - 1 ) * ( NR_ELEMENTS / ( size - 1 ) ) ], NR_ELEMENTS * (
NR_ELEMENTS / ( size - 1 ) ), MPI_DOUBLE, i, 100, MPI_COMM_WORLD );
52
53     for( int i = 1; i < size; ++i )
54         MPI_Send( B, NR_ELEMENTS * NR_ELEMENTS, MPI_DOUBLE, i, 101, MPI_COMM_WORLD
);
55
56     for( int i = 1; i < size; ++i )
57         MPI_Recv( C[ ( i - 1 ) * ( NR_ELEMENTS / ( size - 1 ) ) ], NR_ELEMENTS * (
NR_ELEMENTS / ( size - 1 ) ), MPI_DOUBLE, i, 102, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
58
59     end = MPI_Wtime();
60     fprintf( stdout, "Runtime with %d processes and no output: %lf!\n", size,
end - start );
61
62     //
63     // Output of result to file
64
65     FILE *fp = fopen( "output.txt", "w" );
66
67     fprintf( stdout, "rank = %d, sum = %lf\n", rank, frob_norm );
68     fprintf( fp, "%lf\n", frob_norm );
69     for( int i = 0; i < NR_ELEMENTS; ++i )
70     {
71
72         for( int j = 0; j < NR_ELEMENTS; ++j )
73             fprintf( fp, "%lf ", C[ i ][ j ] );
74
75         fprintf( fp, "\n" );
76
77     }
78
79 }
80
81
82 if ( rank > 0 )
83 {
84
85     MPI_Recv( A_recv, NR_ELEMENTS * ( NR_ELEMENTS / ( size - 1 ) ), MPI_DOUBLE,
0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
86     MPI_Recv( B, NR_ELEMENTS * NR_ELEMENTS, MPI_DOUBLE, 0, 101, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
87
88     for( int i = 0; i < NR_ELEMENTS / ( size - 1 ); ++i )
89         for ( int j = 0; j < NR_ELEMENTS; ++j )
90             {

```

```

91
92     C_recv[ i ][ j ] = 0.0;
93     for ( int k = 0; k < NR_ELEMENTS; ++k )
94     {
95
96         C_recv[ i ][ j ] = C_recv[ i ][ j ] + A_recv[ i ][ k ] * B[ k ][ j
97     ];
98         frob_norm_part = frob_norm_part + C_recv[ i ][ j ];
99     }
100
101 }
102
103
104     MPI_Send( C_recv, NR_ELEMENTS * ( NR_ELEMENTS / ( size - 1 ) ), MPI_DOUBLE,
105     0, 102, MPI_COMM_WORLD );
106
107     fprintf( stdout, "rank = %d, partial sum = %lf\n", frob_norm_part, rank );
108     MPI_Reduce( &frob_norm_part, &frob_norm, 1, MPI_DOUBLE, MPI_SUM, 0,
109     MPI_COMM_WORLD );
110
111 }
112
113 MPI_Finalize( );
114
115 return 0;
116
117 }
```

Listing 2.20: Five point stencil (Serial version)

The following is a larger extension of the simple matrix multiplication.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mpi.h"
5
6  #define NR_ELEMENTS 4
7
8  int main( int argc, char **argv )
9  {
10
11     double A[ NR_ELEMENTS ][ NR_ELEMENTS ];
12     double B[ NR_ELEMENTS ][ NR_ELEMENTS ];
13     double C[ NR_ELEMENTS ][ NR_ELEMENTS ];
14
15
16     MPI_Init( &argc, &argv );
17
18     int rank = 0, size = 0;
19     int err = 0;
20
21     double start = 0.0, end = 0.0;
```

```

22     double frob_norm = 0.0, frob_norm_part = 0.0;
23
24     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
25     MPI_Comm_size( MPI_COMM_WORLD, &size );
26
27     MPI_Datatype rowtype;
28
29     if ( size > NR_ELEMENTS )
30         MPI_Abort( MPI_COMM_WORLD, err );
31
32     double A_recv[ NR_ELEMENTS / size ][ NR_ELEMENTS ];
33     double C_recv[ NR_ELEMENTS / size ][ NR_ELEMENTS ];
34
35     start = MPI_Wtime();
36
37     if ( rank == 0 )
38     {
39
40         for ( int i = 0; i < NR_ELEMENTS; ++i )
41             for ( int j = 0; j < NR_ELEMENTS; ++j )
42             {
43
44                 A[ i ][ j ] = ( double ) ( i + j ); //( rand() % 1000 );
45                 B[ i ][ j ] = ( double ) ( i + j ); //( rand() % 1000 );
46                 C[ i ][ j ] = 0.0;
47
48             }
49
50     }
51
52     MPI_Type_contiguous( NR_ELEMENTS * ( NR_ELEMENTS / size ), MPI_DOUBLE, &rowtype
53         );
54     MPI_Type_commit( &rowtype );
55
56     MPI_Scatter( A, 1, rowtype, A_recv, 1, rowtype, 0, MPI_COMM_WORLD );
57     MPI_Bcast( B, NR_ELEMENTS * NR_ELEMENTS, MPI_DOUBLE, 0, MPI_COMM_WORLD );
58
59
60     for ( int i = 0; i < NR_ELEMENTS / size; ++i )
61         for ( int j = 0; j < NR_ELEMENTS; ++j )
62             C_recv[ i ][ j ] = 0.0;
63
64     for( int i = 0; i < NR_ELEMENTS / size; ++i )
65         for ( int j = 0; j < NR_ELEMENTS; ++j )
66             for ( int k = 0; k < NR_ELEMENTS; ++k )
67             {
68
69                 C_recv[ i ][ j ] = C_recv[ i ][ j ] + A_recv[ i ][ k ] * B[ k ][ j ];
70                 frob_norm_part = frob_norm_part + C_recv[ i ][ j ];
71
72             }
73

```

```

74
75     MPI_Reduce( &frob_norm_part, &frob_norm, 1, MPI_DOUBLE, MPI_SUM, 0,
76               MPI_COMM_WORLD );
77
78     MPI_Gather( C_recv, 1, rowtype, C, 1, rowtype, 0, MPI_COMM_WORLD );
79
80     end = MPI_Wtime();
81
82     fprintf( stdout, "Runtime with %d processes: %lf!\n", size, end - start );
83
84     if ( rank == 0 )
85     {
86         FILE *fp = fopen( "output.txt", "w" );
87         fprintf( fp, "%lf\n", frob_norm );
88         for( int i = 0; i < NR_ELEMENTS; ++i )
89         {
90             for( int j = 0; j < NR_ELEMENTS; ++j )
91                 fprintf( fp, "%lf ", C[ i ][ j ] );
92
93             fprintf( fp, "\n" );
94         }
95     }
96
97     fprintf( stdout, "\n" );
98
99     MPI_Type_free( &rowtype );
100
101
102     MPI_Finalize( );
103
104     return 0;
105 }

```

Listing 2.21: Five point stencil (Serial version)

### 2.3.7 C source 5 point stencil

The following code is the complete serial implementation of the instationary heat conduction equation which was diskretized with the Finite Difference Method (equidistant discretization in both directions). This code was originally taken from [6] and was strongly modified for our needs (stack arrays to dynamic linear arrays (with appropriate initialization procedures), initialization with explicit values out of file, store results into file to visualize the data, etc.).

```

1  #include <cmath>
2  #include <fstream>
3  #include <iostream>
4  #include <iomanip>
5  #include <string>

```

```
6  #include <stdlib.h>
7
8  using namespace std;
9
10
11 //
12 // Define global variables
13
14 int Npoints = 0;
15 int MaxIterations = 0;
16
17 double x_begin = 0.0, x_end = 0.0;
18 double dx = 0.0;
19 double dt = 0.0;
20 double tolerance = 0.0;
21 double start = 0.0, end = 0.0;
22
23 string path_init_file;
24 string path_output_file;
25
26 double *restrict Aold;
27 double *restrict A;
28 double *restrict q;
29
30
31 //
32 // Declare functions
33
34 void read_init_file( );
35 double* create_and_init_matrix_Aold( int );
36 double* create_and_init_matrix_A( int );
37 double* create_and_init_matrix_q( int, const double, const double );
38 int JacobiSolver( int, double );
39 double evaluate_solution( double );
40 void output_solution( );
41
42
43 int main( int argc, char *argv[] )
44 {
45
46 //
47 // Read the init file with parameters
48 read_init_file( );
49
50 Npoints = ( ( x_end - x_begin ) / dx ) + 1;
51 dt = 0.25 * dx * dx;
52
53 const double M_PI_squared = -2.0 * M_PI * M_PI;
54 const double M_PI_dx = M_PI * dx;
55
56
57 //
58 // Create and init matrices A, Aold, q
```



```

59  double *Aold = create_and_init_matrix_Aold( Npoints );
60  double *A = create_and_init_matrix_A( Npoints );
61  double *q = create_and_init_matrix_q( Npoints, M_PI_squared, M_PI_dx );
62
63
64  //
65  // Solve linear equations, evaluate solution and output of the data
66
67  start = clock();
68  cout << setprecision( 7 ) << setiosflags( ios::scientific );
69  int itcount = JacobiSolver( Npoints, dx );
70  end = clock();
71  cout << "Time to solution (JacobiSolver): " << ( end - start ) / CLOCKS_PER_SEC
    << endl;
72
73  start = clock();
74  double result = evaluate_solution( M_PI_dx );
75  end = clock();
76  cout << "Time to solution (evaluate_solution): " << ( end - start ) /
    CLOCKS_PER_SEC << endl;
77  cout << "Jacobi: Mean l2 error between approximated and exact solution is " <<
    result / Npoints << " in " << itcount << " iterations" << endl;
78
79  start = clock();
80  output_solution( );
81  end = clock();
82  cout << "Time for output: " << ( end - start ) / CLOCKS_PER_SEC << endl;
83
84
85  //
86  // Deallocate memory
87
88  delete( Aold );
89  delete( A );
90  delete( q );
91
92
93  return 0;
94
95 }
96
97
98 //
99 // Read the init file
100 void read_init_file( )
101 {
102
103     ifstream file_init;
104     string filename = path_output_file + "init.dat";
105
106     file_init.open( filename.c_str(), ios::in );
107     if ( file_init == NULL )
108     {

```

```

109
110     cerr << "No init file was generated!Exiting ... \n!" << endl;
111
112     exit( 1 );
113
114 }
115
116 file_init >> x_begin;
117 file_init >> x_end;
118 file_init >> dx;
119 file_init >> MaxIterations;
120 file_init >> tolerance;
121 file_init >> path_init_file;
122 file_init >> path_output_file;
123
124 file_init.close( );
125
126 return;
127
128 }
129
130 //
131 // Create and initilaize the matrices A, Aold and q (q is source term)
132 double *create_and_init_matrix_Aold( int N )
133 {
134
135     Aold = new double[ N * N ];
136
137     //
138     // Initialize all elements to zero
139     for ( int i = 0; i < N * N; ++i )
140         Aold[ i ] = 0.0;
141
142
143     //
144     // Initial conditions -- all zeroes
145     for( int i = 1; i < N - 1; ++i )
146         for( int j = 1; j < N - 1; ++j )
147             Aold[ j + i * N ] = 1.0;
148
149
150     //
151     // Boundary Conditions -- all zeroes
152     for( int i = 0; i < N; i++ )
153         for ( int j = 0; j < N; ++j )
154             {
155
156                 Aold[ j ] = 0.0;
157                 Aold[ j + N * ( N - 1 ) ] = 0.0;
158
159                 Aold[ i * N ] = 0.0;
160                 Aold[ ( N - 1 ) + N * i ] = 0.0;
161

```

```

162     }
163
164     return Aold;
165
166 }
167
168 double *create_and_init_matrix_A( int N )
169 {
170
171     A = new double[ N * N ];
172
173     //
174     // Initialize all elements to zero
175     for ( int i = 0; i < N * N; ++i )
176         A[ i ] = 0.0;
177
178     //
179     // Initial conditions -- all zeroes
180     for( int i = 1; i < N - 1; ++i )
181         for( int j = 1; j < N - 1 ; ++j )
182             A[ j + i * N ] = 1.0;
183
184     //
185     // Boundary Conditions -- all zeroes
186     for( int i = 0; i < N; i++ )
187         for ( int j = 0; j < N; ++j )
188             {
189
190                 A[ j ] = 0.0;
191                 A[ j + N * ( N - 1 ) ] = 0.0;
192
193                 A[ i * N ] = 0.0;
194                 A[ ( N - 1 ) + N * i ] = 0.0;
195
196             }
197
198     return A;
199
200 }
201
202 double *create_and_init_matrix_q( int N, const double M_PI_squared, const double
    M_PI_dx )
203 {
204
205     q = new double[ N * N ];
206
207     //
208     // setting up an additional source term
209     for( int i = 0; i < N; ++i )
210         for( int j = 0; j < N; ++j )
211             q[ j + i * N ] = M_PI_squared * sin( M_PI_dx * i ) * sin( M_PI_dx * j );
212
213     return q;

```

```

214
215 }
216
217
218
219 //
220 // Function for solving the linear equations with the iterative Jacobi solver
221 // (implement the 5 point stencil)
222
223 int JacobiSolver( int N, double dx )
224 {
225
226     double D = dt / ( dx * dx ) ;
227
228     //
229     // Start the iterative solver
230     for ( int k = 0; k < MaxIterations; k++ )
231     {
232
233         for( int i = 1; i < N - 1; ++i )
234             for( int j = 1; j < N - 1; ++j )
235                 A[ j + i * N ] = dt * q[ j + i * N ]
236                     + Aold[ j + i * N ]
237                     + D * ( Aold[ ( j + 1 ) + i * N ]
238                         + Aold[ j + ( i + 1 ) * N ]
239                         - 4.0 * Aold[ j + i * N ]
240                         + Aold[ ( j - 1 ) + i * N ]
241                         + Aold[ j + ( i - 1 ) * N ] );
242
243         for( int i = 0; i < N; ++i )
244             for( int j = 0; j < N; ++j )
245                 Aold[ j + i * N ] = A[ j + i * N ];
246
247     }
248
249     return MaxIterations;
250
251 }
252
253
254 //
255 // Function for evaluating the approximated solution ("comparison" between
256 // approximated and exact solution)
257 double evaluate_solution( double M_PI_dx )
258 {
259
260     double ExactSolution = 0.0;
261     double sum = 0.0;
262
263
264     for( int i = 0; i < Npoints; i++ )
265     {
266

```

```

267     for( int j = 0; j < Npoints; j++ )
268     {
269
270         ExactSolution = -sin( M_PI_dx * i ) * sin( M_PI_dx * j );
271         sum += fabs( Aold[ j + i * Npoints ] - ExactSolution );
272
273     }
274
275 }
276
277 return sqrt( sum ) ;
278
279 }
280
281
282 //
283 // Output of the resulting matrix to file (for visualization)
284 void output_solution( )
285 {
286
287     ofstream file_out;
288     string filename = path_output_file + "test.mat";
289
290     file_out.open( filename.c_str(), ios::out );
291     if ( file_out == NULL )
292     {
293
294         cerr << "No output file was generated! Exiting ... !\n" << endl;
295         exit( 1 );
296
297     }
298
299     file_out << setprecision( 5 ) << setiosflags( ios::scientific );
300     for ( int i = 0; i < Npoints; ++i )
301     {
302
303         for ( int j = 0; j < Npoints; ++j )
304             file_out << Aold[ j + i * Npoints ] << " ";
305
306         file_out << endl;
307
308     }
309
310     file_out.close( );
311
312     return;
313
314 }

```

Listing 2.22: Five point stencil (Serial version)

The resulting results will be stored into a file, which content can be visualized for example with *matlab* or *octave*. With the following lines of *matlab/octave* code the visualization

can be done.

```
Z =load('test.mat', '-ascii')
x = linspace(1, 251, 251);
y = linspace(1, 251, 251);
[X, Y] = meshgrid(x, y);
surf(X, Y, Z);
```

Listing 2.23: Code for plotting the data of the Jacobi iteration

# Bibliography

- [1] Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2010.
- [2] Eijkhout, V.: [Introduction to High-Performance Scientific Computing](#).
- [3] Severance, C.; Dowd, K: [High Performance Computing](#).
- [4] Püschel, M: [How to write fast numerical code](#).
- [5] Supalov, A.; Semin, A.; Klemm, M.; Dahnken, C.: [Optimizing HPC Applications with Intel® Cluster Tools](#), Apress Open, 2014.
- [6] Hjorth-Jensen, M.: [Computational Physics, Lecture Notes Fall 2015 \(August 2015\)](#), University of Oslo
- [7] [OpenACC Resources](#)
- [8] [OpenMP books](#)
- [9] [OpenMP tutorials](#)
- [10] Gropp, W.; Lusk, E.; Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, third edition, The MIT press, 2014.
- [11] MacDonald, N. et al: Writing Message Passing Parallel Programs in MPI, Version 1.8.2, Edinburgh Parallel Computing Centre (EPCC), University of Edinburgh, www source: [https://www.researchgate.net/publication/239179288\\_Writing\\_Message\\_Passing\\_Parallel\\_Programs\\_with\\_MPI/link/00b495286755ac55f1000000/-download](https://www.researchgate.net/publication/239179288_Writing_Message_Passing_Parallel_Programs_with_MPI/link/00b495286755ac55f1000000/-download) (last access date: 2019-12-29).





Appendix A

Example PDF Report

# ProfiT-HPC Report

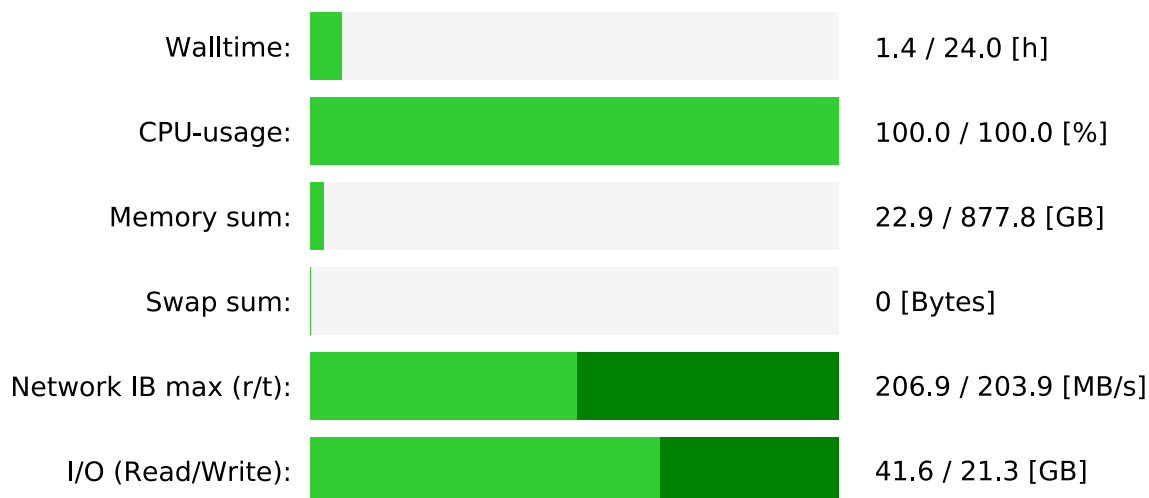
## Job Overview:

Job-ID : 1352076  
User name : random.user  
Queue : medium-fmz  
Number of nodes : 13  
Requested cores : 40  
Requested time : 24.00 h  
Used time : 1.42 h  
Time of job start : 04/10/2019 08:47:24  
Time of completion : 04/10/2019 10:12:20

## Node Information:

CPU model:  
Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz  
Memory per node: 68 GB  
Sockets per node: 2  
Cores per socket: 10  
Threads per core: 1  
Node list:  
gwdd[027,032-033,040-041,043-047,063,067,]  
gwdd[070]

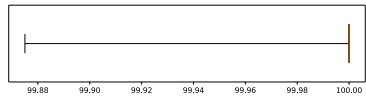
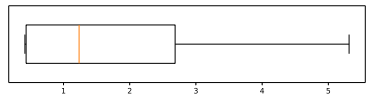
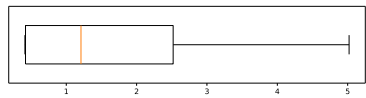
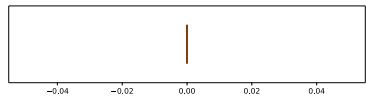
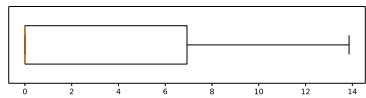
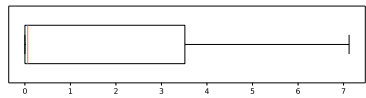
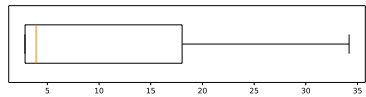
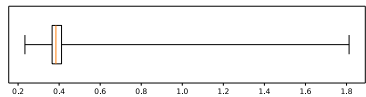
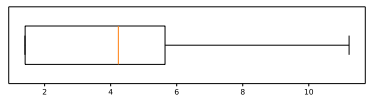
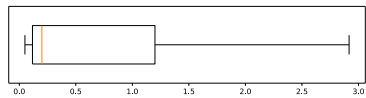
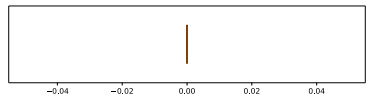
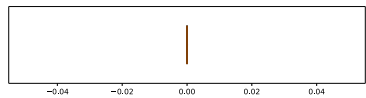
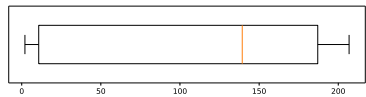
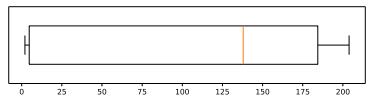
## Global Summary of Resource Usage



## Recommendations:

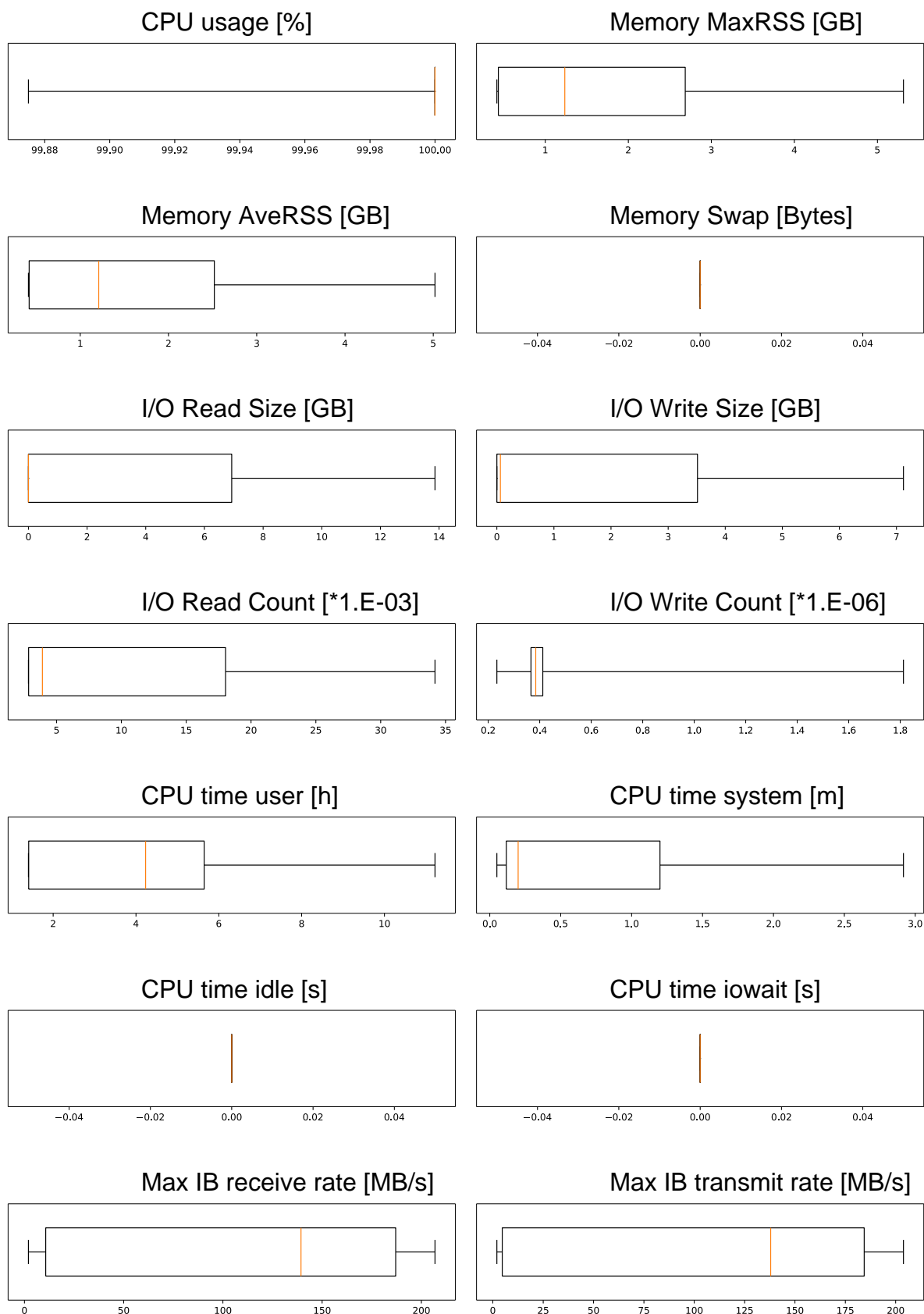
No problems detected!  
Good work!

## Node Distributions

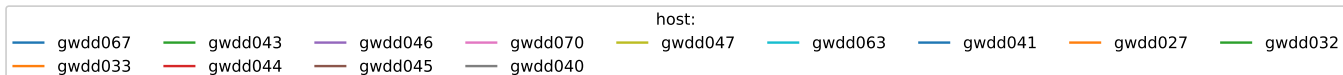
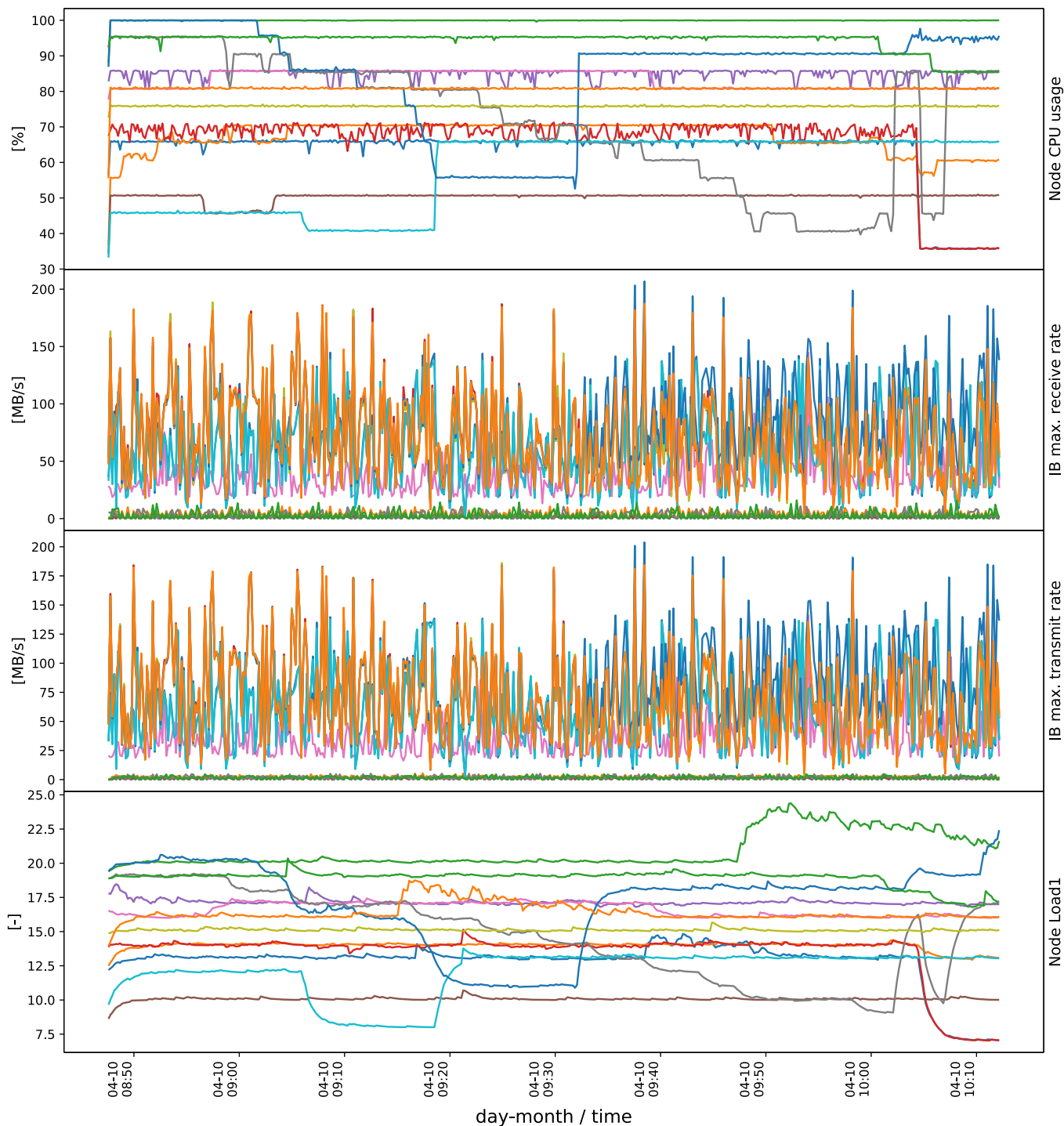
Metric	Units	Min.	Max.	StdDev.	Ave.	Boxplot
CPU usage	[%]	99.88	100.00	0.03	99.99	
Memory MaxRSS	[GB]	0.42	5.31	1.58	1.76	
Memory AveRSS	[GB]	0.41	5.02	1.47	1.67	
Memory Swap	[Bytes]	0.00	0.00	0.00	0.00	
I/O Read Size	[GB]	0.00	13.86	4.74	3.20	
I/O Write Size	[GB]	0.00	7.12	2.41	1.64	
I/O Read Count	[*1.E-03]	2.83	34.19	10.46	10.09	
I/O Write Count	[*1.E-06]	0.23	1.81	0.40	0.49	
CPU time user	[h]	1.40	11.22	3.27	4.33	
CPU time system	[m]	0.05	2.92	0.84	0.66	
CPU time idle	[s]	0.00	0.00	0.00	0.00	
CPU time iowait	[s]	0.00	0.00	0.00	0.00	
Max IB receive rate	[MB/s]	2.01	206.85	87.03	95.90	
Max IB transmit rate	[MB/s]	2.01	203.86	87.68	92.51	

## Node Distributions

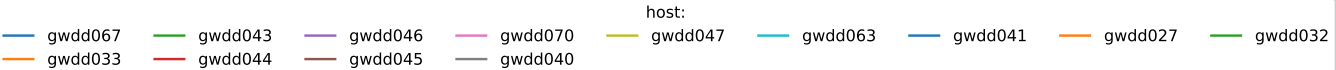
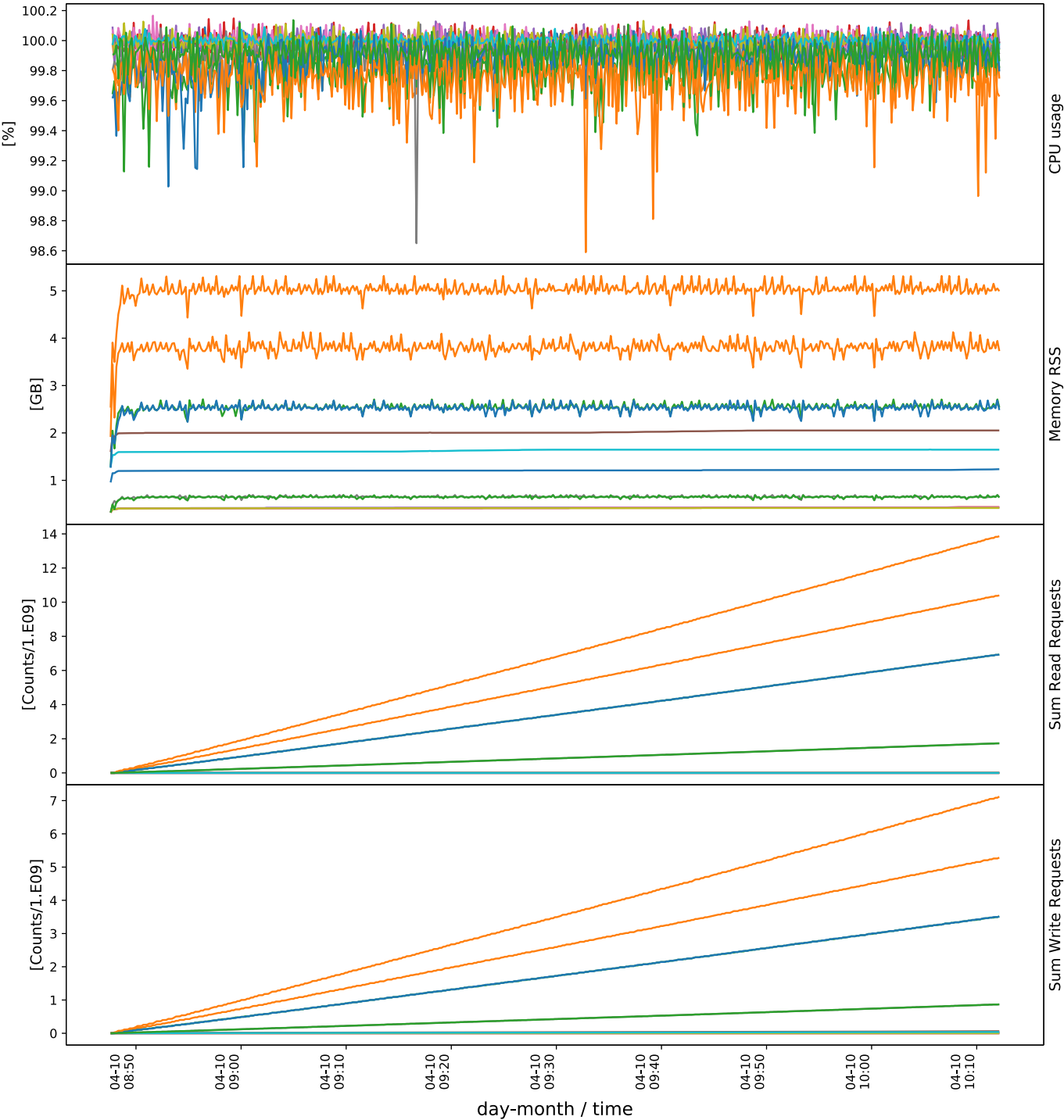
(Box plots: minimum, average, maximum, 50% boundaries)



## Timeseries Plots



Timeseries Plots



## Key

### Global Summary Definitions:

The global summary bar chart gives information about how resources are being used or how far they are from a maximum value. Each bar represents the comparison of two values, which are listed to the right of the diagram.

Walltime: The time duration between time of job start until time of job completion. It is also referred to as used time. This value is compared in the bar chart with the requested time.

CPU usage: Average of the CPU usages (see glossary) of all job's processes on allocated nodes. It is compared in the bar chart to the value 100% or to the highest value, if larger than 100%.

Memory sum: Sum of the maximum memories (RAM) used by the job on allocated nodes. It is compared in the graph to the sum of the total RAM memory of the allocated nodes.

Swap sum: Measure of how much swap space on a disk had to be used because of full physical RAM memory.

Network: Compares maximum node traffic (Infiniband), the maximum amount received and the maximum amount transmitted per node.

I/O (Input/Output): Compares the amount of read input to the amount of write output.

### Node Distribution Definitions:

Bar graphs containing node distribution information. Overall values are computed for each node. Then the minimum, maximum, average and standard deviation of these values are computed.

CPU usage: Time average of the sum of CPU usages (see glossary) of all job's processes on each node divided by the number of cores on that node.

Memory MaxRSS: Maximum of the sum of the RSS (see glossary) values of each process of the job. It refers to the maximum amount of physical memory held by a processes for the job.

Memory AveRSS: Average of the sum of the RSS (see glossary) values of each process of the job.

Memory Swap: Indicates how much disk space was needed for execution of any processes of the job.

I/O Read or Write Size: Amount of input read or output written by job's processes on the node.

I/O Read or Write Count: Count of number of read or write requests of job's processes on the node.

Max IB receive rate: maximum (high water mark) of infiniband receiving rate of a node.

Max IB transfer rate: maximum (high water mark) of infiniband transmitting rate of a node.

CPU time: -user Sum of amount of time CPU was executing user instructions for the job's processes. -system Sum of amount of time CPU was executing system instructions for the job's processes. -idle Sum of amount of time CPU was idle while executing job's processes. -iowait Sum of amount of time CPU was waiting for I/O for the job's processes.

### Timeseries:

Node CPU usage: Sum of CPU usages of all cores on each allocated node divided by the number of cores allocated on that node for the job.

Node Load1: Average values of load of the last minute of the node; see also glossary.

CPU usage: Average of CPU usages of all job's processes on the node.

Memory RSS: Sum of the job's process RSS (see glossary) on the node.

Sum Read Counts: Sum of the job's read requests on the node.

Sum Write Counts: Sum of the job's Write requests on the node.

IB max. receive rate: Maximum infiniband receiving rate during job on the node.

IB max. transfer rate: Maximum infiniband transmitting rate during job on the node.

### Glossary:

CPU: Central Processing Unit

CPU usage: Percentage of time for a given time increment, for which a CPU was utilized by a process or vice versa

Load1: Load average for last 1 minute; a measure of how processor cores are being used in counts, where 1 refers to full use of one core

RAM: Random Access Memory; physical memory

RSS: The Resident Set Size; the amount of physical memory (RAM) held by a process

Swap: How much disk space was utilized during process execution instead of RAM