Fachhochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Diplomarbeit

Zur Erlangung des akademischen Grades eines
Diplom-Ingenieur (FH)
über das Thema

Simulation of Advanced Large-Scale HPC Architectures

Eingereicht am Fachbereich 1 Ingenieurwissenschaften I
der Hochschule für Technik und Wirtschaft Berlin

Von: Frank Lauer, s0514918
1. Betreuer: Prof. Dr. Johann Schmidek
2. Betreuer: Prof. Dr. Dieter Kranzlmüller

Berlin, den April 17, 2010

Abstract:

The rapid development of massively parallel systems in the *High Performance Computing* (HPC) area requires good and efficient scalability of the applications. The next generation's design of supercomputer is today not certain in terms of what will be the computational resources, memory and I/O capabilities. However it is guaranteed that they become even more parallel due to developments such as multicore. Obtaining the optimal performance on these machines is not only a matter of hardware it is also an issue of programming design. Therefore co-development of hard- and software is needed. The question is: how to test algorithm's on machines which do not exist today.
To address the programming issues in terms of scalability and fault tolerance for the next generation, this project's aim is to design and develop a simulator based on *Parallel Discrete Event Simulation* (PDES) for testing arbitrary *Message Passing Interface* (MPI) based applications. Massive parallel environments with at least 10^7 virtual processes can be simulated. In comparison todays most denst supercomputer combine 10^5 cores togeter. The simulation it self is a transparent layer where the MPI applications run on top. These layer is designed to provide mechanism's for collecting metric data as well as test the applications behavior against failure. To do so a fault injection based on distibution models and in a directly manner can be done.

Acknowledgements

I would like to express my appriciation to my supervisors: Prof Vassil Alexandrov, Dr. Christian Engelmann, Prof. Dieter Kranzlmüller, and Prof. Dr. Johann Schmidek. Special thanks to Dr. Christian Engelmann for his time, patience, and understanding.

My father who did supported me during my study.

My colleagues at the ORNL, T.J. Naughton, Stephen L. Scott, Geoffroy Vallee, and my group leader Prof. Al Geist who where always helpfull.

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Diese Arbeit wurde bisher in gleicher oder ähnlicher Form an keiner anderen Prüfungsbehörde im Geltungsbereich der Bundesrepublik Deutschland vorgelegt und auch nicht veröffentlicht.

Lauer Frank, April 17, 2010

Contents

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 High Performance Computing (HPC) . . . 1
  - 1.1.2 Programming Of HPC . . . 5
  - 1.1.3 Discrete Event Simulation (DES) . . . 8
- 1.2 Project Proposal . . . 12

2 Related Work **15**

3 Problem Analysis And Specification **19**

4 Design **21**
- 4.1 Logical Process . . . 21
- 4.2 Virtual Machine . . . 22
  - 4.2.1 LP Runtime Environment . . . 23
  - 4.2.2 Virtual MPI . . . 24
  - 4.2.3 Integrating The Application Into The Simulator . . . 26
  - 4.2.4 Content Switch Logical Processes/Synchronisation . . . 27
- 4.3 Virtual Time/Global Virtual Time . . . 29
- 4.4 Message Queue . . . 33
- 4.5 Message Transport . . . 35
- 4.6 Overall Concept Of The Simulator . . . 36

5 Implementation **39**
- 5.1 Background Knowledge Function Call . . . 39
- 5.2 Virtual Machine . . . 43
  - 5.2.1 Content Switch Logical Processes . . . 43
  - 5.2.2 Virtual MPI . . . 51
  - 5.2.3 Virtual Time . . . 52
- 5.3 DES/PDES . . . 53
- 5.4 Message Queues . . . 54
- 5.5 Communication . . . 58

6 Testing **60**

6.1 Time Measurement . . . 60
6.2 Simulator . . . 62
6.2.1 Application: Heat Transfer . . . 62
6.2.2 Application: Numerical Quadrature . . . 69

7 Conclusions **73**
7.1 Future Prospects . . . 73
7.1.1 Overcome Restriction: MPI Calls Only In *main(...)* . . . 73
7.1.2 Implementing The Virtual Time . . . 74
7.1.3 Scheduling Policy . . . 75
7.1.4 Migration Of An LP To Another Node . . . 75
7.1.5 Enhanced MPI And System Call Instruction Set . . . 76
7.1.6 Optimistic PDES Approach . . . 76
7.1.7 Fault injection . . . 76
7.2 Known Issues . . . 77
7.2.1 Memory (Segmentation Fault) . . . 77
7.2.2 Out of resources . . . 77
7.2.3 Printf And Floating Point Values . . . 77

List of Acronyms **i**

Glossary **iii**

Appendices **ix**

A Test Programs **A**
A.1 Test pthread inital stack usage . . . A

B Project **C**
B.1 MV_main function call assembly code . . . C
B.2 assembly_code.h . . . C
B.3 datatypes.h . . . E
B.4 Function VM_synchronise_LP() . . . G

C **J**

User Manual **J**
C.1 Requirements . . . K
C.2 Installation . . . K
C.3 Usage . . . K
C.3.1 Simulation Restrictions . . . K
C.3.2 Supported MPI calls . . . K

C.3.3 Prepairing the application for simulation L
C.3.4 Compiling the simulator L
C.3.5 Executing the simulator L
C.4 Example: Ring message . . . . . M
C.5 Deploying the simulator on a new architecture . . . . . O

1 Introduction

1.1 Background

1.1.1 High Performance Computing (HPC)

HPC is a part of computer science in which applications need significant amounts of memory and processing power. Thus nowadays as well as in the beginning of supercomputing the values of the criteria that define, if an application is considered subject of the HPC field are changing over time. This is because since the first digital computers back in the 1940's the available computational resources that can be assigned to solve a single problem grow rapidly. Problems which in the 1940's had to be solved by HPC can today be solved by a simple calculator.

Thus, to get a better understanding of the term HPC it might be easier to define what a supercomputer is. Basically it can be said that a such a computer is predominant in terms of computational resources and memory. By that definition some kinds of supercomputers already existed even before the first digital computer had been built. Even today there are analogue machines that function as a kind of computer, the others are digital and hybrid ones. Specialised analogue data processing units could still be as fast as current leading digital computers or even faster. However they are only designed to solve one specific problem, whereas digital computers allow to solve a great range of problems, all problems that can be expressed in a mathematical way. As supercomputers are generally digital this thesis will deal only with those.

The first digital, programmable machine was built by Konrad Zuse in 1936. It had a 64 word memory, where each word contained a 22bit float value (8 bit exponent and 14 bit mantissa), a remarkable storage capacity of 64 floats or 1408 bit. An electrical engine provided a clock frequency of one Hertz. This and all following computer models are called supercomputers as they are predominant in their time due to their comparably large resources and high speed.

The Z1 and all its successors are designed to solve complex mathematical problems. Such equations however contain often a big part which demands floating point calculations. Therefore one way to measure the "Performance" of HPC is in *Floating point Operations Per Second* (flop/s). To determine the speed of a cur-

rent supercomputer, a specific benchmark test like LINPACK has to be performed. Introduced by Jack Dongarra, LINPACK measures how fast a N by N system of linear equations can be solved in flop/s. The Image 1.1 displays the enormous gain of speed from the early 1950's until today. The graph is taken from the HPC Asia 2009 Keynote Speech by Jack Dongarra [4].

It gives an overview of the basic architecture of the computer in the different

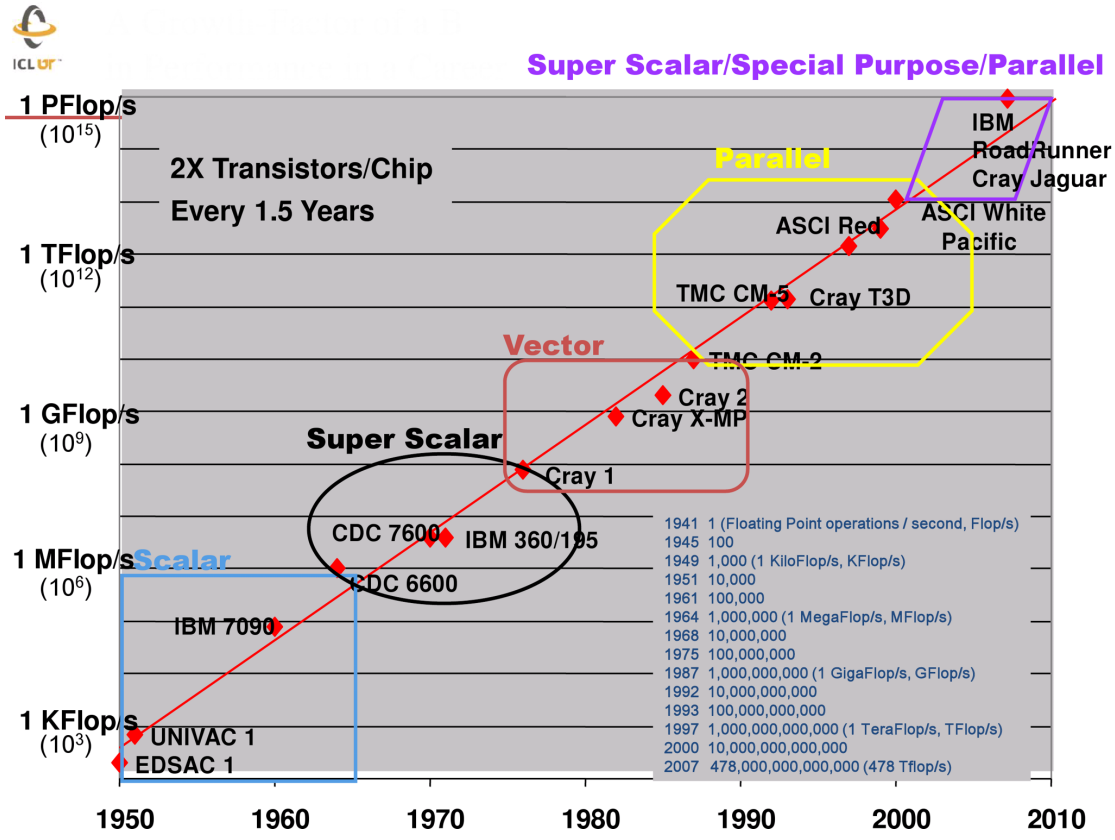

Figure 1.1: "Performance" history of HPC until today

epochs. Everything started with scalar computers like the Z1. They operate according to the von Neumann architecture [2] and complete a whole command cycle before they start with the next one. In the mid 1960's a new type of processor was invented. Its performance increase was not only due to increasing the operation clock frequency but also to highly super scalar command processing. This means that a command has not to finish all states of the von Neumann cycle before another one can be processed. This architectural innovation is today known as pipelining. This principle is nowadays being implemented in a very large range of

processors.

The vector approach was initiated to further accommodate the *Central Processing Unit* (CPU) design to the needs of common HPC applications. Hence the overhead by vector computation was reduced. The basic idea behind this is to optimise vector manipulations. Conventionally arithmetical operations on vectors have to run in a loop and only item at a time is modified. The new model reduces the amount of operations and piplines the memory accesses.

In the late 1980's exhausting the possible increase in main clock frequency as well as architectural gimmicks were not enough to fulfil the needs of the applications. To overcome the shortage of resources new supercomputers started to been built with parallel interconnected CPUs. Over time the systems became more and more dense with interconnected CPUs, but that also meant that the programming model for HPC applications had to be changed, see section 1.1.2 for further information.

In the last few years HPC has become a massive parallel specialised super scalar supercomputer area. From the beginning of parallel interconnected CPUs until today, the amount of interconnections has rapidly increased. Now six supercomputers, which are listed in the 2009 Novembers TOP500 [1], allow the user to utilise up to 10^5 cores at once. Cores are a combination of memory, at least one *Arithmetic logic unit* (ALU) and a connection to the *Input/Output* (I/O) interface. Several cores combined on one chip are called multi-core processor. This development was driven by various factors. Modern *Operating System* (OS) allow to run independent processes simultaneously, which mainly gains the advantage of being able to run these independent processes truly parallel. Furthermore increasing the clock frequency became physically more and more of a problem. The basic physics behind the used hardware require a miniaturisation of the elements and reduction of the distance between, when amplify the operation frequency. However increasing the frequency further also decreases the CPUs calculation fault tolerance and as well increases the probability of a total failure of the CPU. A problem which not only concerns the hardware but also the software in order to prevent application failure.

Reducing hardware failures has also high priority which results in the clock frequency almost remaining static at a level where failures are within a certain probability. All things considered the whole development is predictable applying Moore's law [26, 25], which says that every 18 months the computational speed will be doubled.

This may still be true but since applications run parallel on a broad scale effi-

[1] A list in which twice a year the public known supercomputers be ranked by there speed

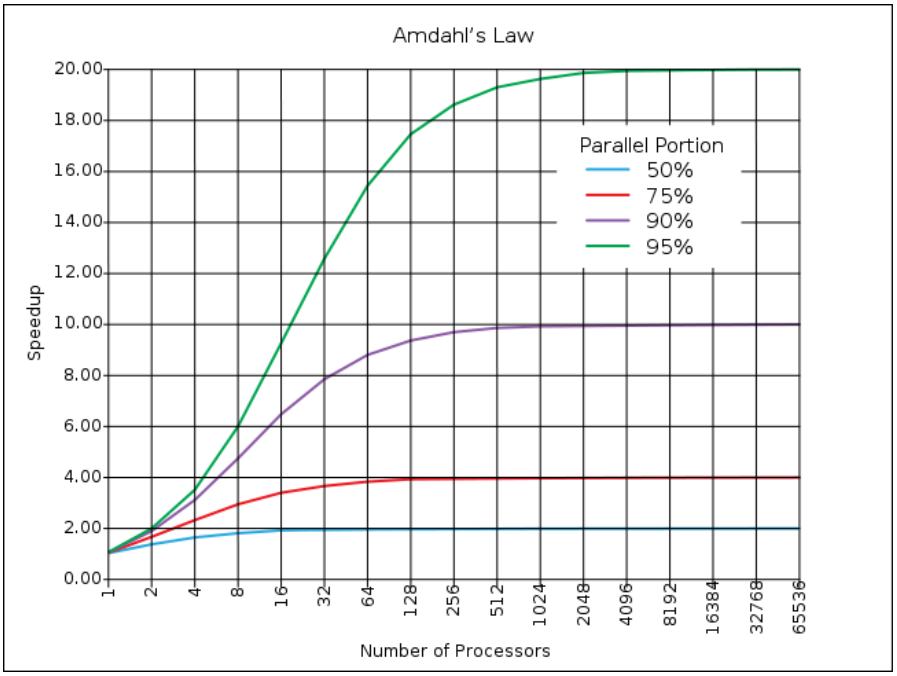

Figure 1.2: Amdahl's Law – Limits and Costs of Parallel Programming

ciency has to be considered too. One of Amdahl's laws [9, 2] is concerned with how much faster an application can run. The theoretical maximal speedup of a program which is being processed parallel, regardless of how many cores are interconnected, can be calculated by:

$$S = \frac{1}{1 - P} \tag{1.1}$$

Where P is the percentage of the code which can be parallelised and S is the gained speedup, which is measured in percent. The sequential execution time is when S equals 100% (1.0). The closer P comes to 100% (1.0) the higher the speedup, in theory it is infinite when P equals 100%. Nevertheless real world problems have always dependencies and parts which cannot be processed parallelised. Even more interesting is the possible speedup by a given amount of cores which compute in parallel. With another Amdahl equation we can account for that.

$$S = \frac{1}{C + \frac{P}{N}} \tag{1.2}$$

C stands for the portion of the program which has to be executed consecutive in percent and N for the amount of cores. The graphs in Figure 1.2 shows that

even if C is as small as 5% the speedup cannot be higher then 20. But more importantly it shows that far fewer cores are required to get the biggest part of the speedup, then to reach the maximum. Still with such restrictions it is useful to have machines with thousands of cores for various simulations, where the amount of computational data which can be parallelised is big enough.

As usually theory is far away from reality end even if an application could run twice as fast data dependencies between compute nodes can have a great slow down effect. Even though the interconnection network is a specialised one, with a very low latency and great bandwidth, communication can be a bottle neck. The upper graph of Figure 1.3 represents an overview of the time with the numbers of systems which have what type of network installed. In addition the graph on the bottom, which is produced by the project *Network Protocol Independent Performance Evaluator* (NetPIPE)[3] from the *Scalable Computing Laboratory* (SCL), shows the test results of a variation of these networks in comparison to bandwidth and with an indication of the latency. All latency values are only a couple of micro seconds but still this is more than a factor of 10^3 slower then a I/O operation on the main memory.

HPC as we know it today is a combination between vast computational resources and their interconnection.

1.1.2 Programming Of HPC

The data and instruction streams in HPC systems can be divided into four main categories, which is known as Flynn's taxonomy. Michael J. Flynn proposed in 1966 a specific classification of parallel computer architecture. Each architecture requires a programming model specialised in dealing with the available streams. The four categories in Flynn's taxonomy are the following[1]:

- *Single Instruction Single Data* (SISD) computers have one Control Unit that handles one algorithm using one source of data at a time. The computer tackles and processes each task one after the other, and so sometimes people use the word "sequential" to describe SISD computers. They are not capable of performing parallel processing on their own.

- *Multiple Instruction Single Data* (MISD) computers have multiple processors. Each processor uses a different algorithm but uses the same shared input data. MISD computers can analyse the same set of data using several different operations at the same time. The number of operations depends upon the number of processors. There are not many actual examples of MISD computers, partly because the problems an MISD computer can calculate are uncommon and specialised.

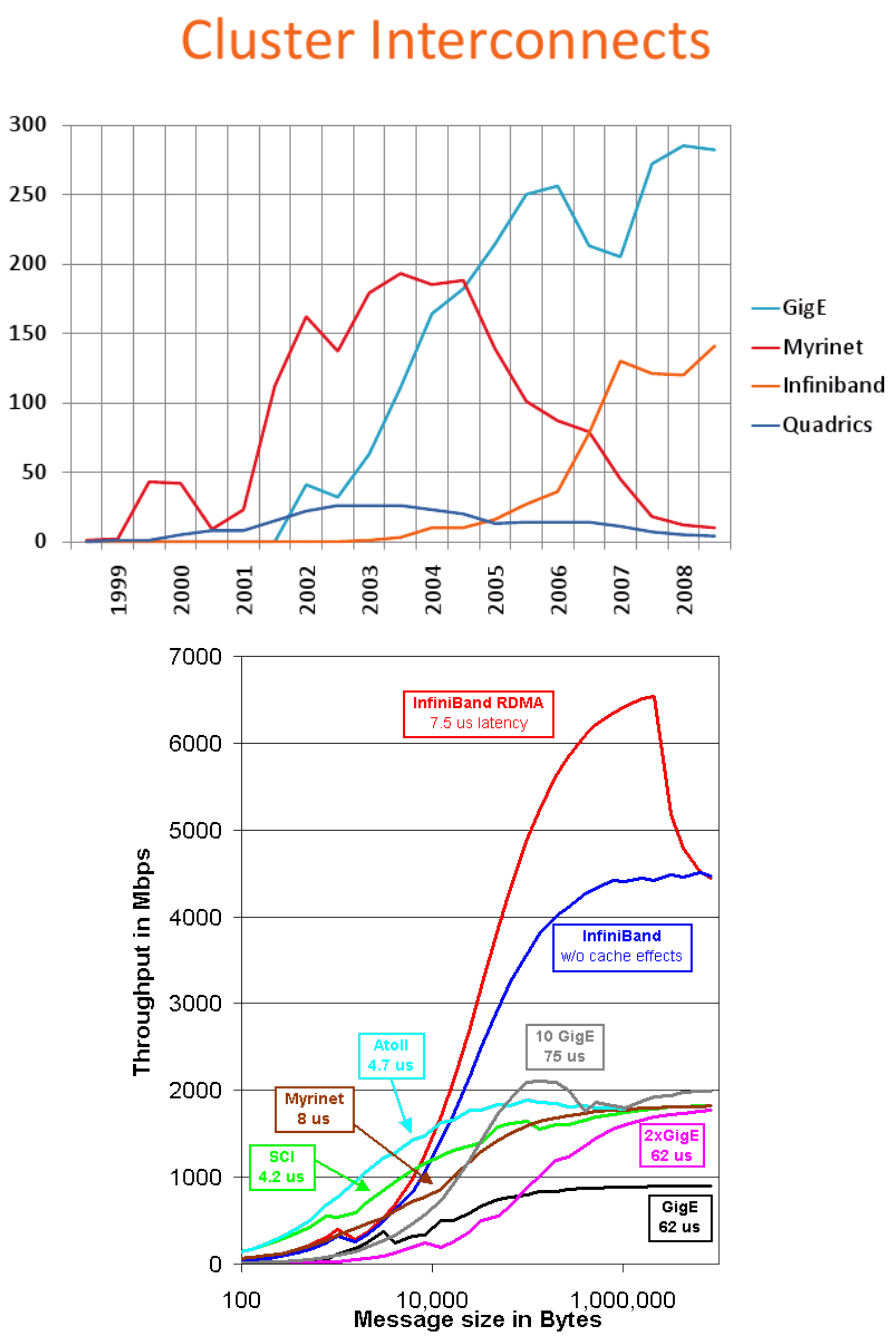

Figure 1.3: Cluster interconnections

- *Single Instruction Multiple Data* (SIMD) computers have several processors that follow the same set of instructions, but each processor inputs different data into those instructions. SIMD computers run different data through the same algorithm. This can be useful for analysing large chunks of data based on the same criteria. Many complex computational problems do not fit this model.

- *Multiple Instruction Multiple Data* (MIMD) computers have multiple processors, each capable of accepting its own instruction stream independently from the others. Each processor also pulls data from a separate data stream. An MIMD computer can execute several different processes at once. MIMD computers are more flexible than SIMD or MISD computers, but it is more difficult to create the complex algorithms that make these computers work. Single Program, Multiple Data (SPMD) systems are a subset of MIMDs. An SPMD computer is structured like an MIMD, but it runs the same set of instructions across all processors.

The face of HPC programming changed dramatically with the evolution of parallelised systems. It started with the interconnection of multiple SISD architectures and now with multi-core CPUs we rather have a network of MIMD computers. Of the thinkable way to program these machines, streams (pipes), shared memory, the predominant programming technique is MPI. MPI is a definition of an *Application Programming Interface* (API) that allows processes to communicate on the same [2] or over a connection to other nodes, by sending messages. There are several implementations Open MPI, MPICH, MPICH2, MVAPICH, TPO++ and Boost which provide an object oriented interface, to name only some of them. The idea behind this is that a problem will be divided into chunks which can be processed concurrently. Normally in sequential intervals the nodes have to be synchronised, which in terms of programming most of the time means that data has to be transmitted from one process to another. MPI can be regarded as a data copying protocol. In general a chunk of memory owned by one process is copied to another by sending a message. The MPI API hides the transport mechanism from the programmer.

A similar simple message passing API called *Parallel Virtual Machnine* (PVM) provides a comparable set of communication functionalities. Although MPI is predominant the PVM library is still in use. In practice it does not make a big difference which library is used for the programming model. Thanks to such standardisation in several APIs, programs nowadays are portable from one machine to another by recompiling them without major changes in the source code. Before

[2] If more MPI processes than cores run on one node, then this is called subscribing the node

such libraries were easily available the programs' source code had to be adjusted for use on other machines. Not only that MPI allows portability between different interconnection types but it also provides the opportunity to grant scalability for applications by design. cores.

1.1.3 Discrete Event Simulation (DES)

The projects aim is to simulate a transparent environment for executing MPI based applications. The simulation layer will be a PDES layer, which is a distributed *Discrete Event Simulation* (DES). DES is a powerful tool when it comes to investigating the behaviour of complex environments. It has become a way of life in all areas where size and complexity does not allow to process the analytical solution by conventional methods. Computers, vehicles, management, military, everywhere where entities of whatever kind interact in discrete intermittency with each other. Even though for applications which are not too complex a DES may be the right choice, when a specific behaviour is to be studied. Features of DES which allow a unique view into a process are:

- compress time or expand time
  Through implemented mechanisms the simulation time can be sped up or slowed down. This is useful when long runtime models have to be studied or if a specific time section is required in more detailed solution.

- restore system state
  Since this kind of simulation only changes the environment variables of events, the system state can be restored to any chosen time in the simulation where an event has occurred.

- stop and review
  Depending on the implementation the simulation can be stopped and certain parts can be reviewed. Such a review can be implemented by different approaches, but in the end additional memory or computation time is required.

- control sources of variation
  At every point of the simulation it can be suspended and the current data can be analysed or manipulated. By design it is comparably easy to implement an interface to access the model data. This gives the model's programmer the opportunity to debug the model itself. Furthermore it is a way to direct the model into a different path, or even to run different paths without having to rerun the complete simulation.

- Facilitates replication
  One key feature of the DES is that the identical start conditions produce identical results. This is a way to test the model for correctness.

First of all it is necessary do define in the system what an event is on the one hand and what an entity is on the other hand. This kind of simulation is based on the fact that an event occurs at a given point in time. For example if you switch on the light or press a button, a sensor gives a signal. Whereas the simulated entities perform activities such as moving from one point to another in a certain amount of time [7]. Such an entity is also called *Logical Process* (LP) since it is an independently running process, but it is only a simulated process. In other words we can say: "A discrete event simulation model assumes the system being simulated only changes state at discrete points in simulation time. The simulation model jumps from one state to another upon the occurrence of an event" [15, p. 31].

In an example model of a railroad system where one train is travelling from station A to station B the train represents the LP. To keep the example simple the train has only two states: one is idle and the other is moving. Now the state of an entity only can be changed by an event. Hence we need an event which in our case is the train starting at station A, whereby the train state switches to moving. When the train arrives at station B the entity state switches back to idle.

It is important for DES to distinguish between real time which is the time that passes in the real world and simulation time. The simulation time is a *Virtual Time* (VT) there for it can be faster or slower then the real time. Also events which occur in the simulator are not necessarily in the right order if aligned to the real time. In the train example both events can be inputted at once, but the train needs a certain time to travel from point A to point B. This can be factored into the VT. For example even if both events are inputted at the same VT the event of the arrival occurs at VT + duration. As you can see every event has to be associated with a defined point in VT.

Every implementation of DES relies on three key components. The first is the VT and getting the youngest VT of all LPs , it can be either *Global Virtual Time* (GVT) [40], *Continuously Monitored Global Virtual Time* (CMGVT) [12] or *Wide Virtual Time* (WVT) [36]. The second is a queue in which the events will be scheduled in the order of their occurrence. The last one are the state variables. Those variables store, after each processed event, the data of the simulated system and additional information like elapsed VT for the LP or its status in terms of simulation (running, suspended, terminated, etc.).

When observing such a simulation from an outside point of view one can see that

the simulation is nothing else then images of the simulated environment at certain points in VT, one image for each element. Every time such an image is created it will be backuped if a feature like review is required or if the synchronisation algorithm needs to be able to restore the system status to a previous point in VT. In the end the view consists of a chain of environment images for each LP. The knowledge of how these state variables are actually manipulated is not necessarily required for knowing the DES simulator code. Thus the actually simulated environment is separated from the DES which means that however complex the system may be the basic architecture of the simulation will not get any more complex.

The state of the art synchronisation mechanisms can be divided into two main categories:

Conservative: This is a comparably simple synchronisation mechanism. By design the VT s of all LP s have to be ahead of the next event's VT before the event can be processed. Now it is definitely no chance that one of the entities can generate a nother event which may have occurred earlier in VT. The disadvantage of this approach is that even if the right event is already generated, LPs may have to be delayed until the event is safe in that no other message can arrive with an earlier *Virtual Time Stamp* (VTS). Depending on the simulated environment it is possible that the waiting LP already knows from where the next event will come. For such systems a look-ahead algorithm can search the event queue for a matching event. Despite the fact that for some entities the scheduled VT of the event is in the future, it can be processed without risk of an causality error. This error is the result if one event which happens later in VT has been executed before an event which would have appeared earlier in the timeline of the same LP. The simulation flow might be improved through an look-ahead algorithm.

Optimistic: The main assumption here is that all events which are scheduled can be executed without causing a causality error [29]. However the basic design of DES cannot guarantee that the assumption is really true. In fact it is most probable that at some point a causality error will occur. In that case basically the simulation has to be stopped and a system roll back to a point preceding the error has to be performed. Thus depending on the implemented algorithm, all related LP s which receive an event after the error generated by the LP where the causality incident has taken place, somehow have to be restored to a time before the error. From there the events can be executed in the right order. To allow the system to be rolled back there has to be a mechanism which keeps a history of the events and also keeps a history of the models' state variables. In general this is the main weakness of this approach. The more complex a model is in regard to size and required memory for each LP, the higher the risk that the simulator has

to start swapping parts of the data or in the worst case an out of memory exception occurs. For example a simulation size of 1,000 LPs, where each LP needs 0.5 MBytes, the simulator uses 500 MBytes for merely resending the state variables alone. It does not appear to be that much for todays computers, but if we consider that in addition not only the history of events but also multiple backups of the system have to be stored, it becomes clear why memory is the main issue in all of this.

Using an optimistic approach, synchronisation will not block an LP, since events are processed independently from the GVT , but there is also potential for optimisation. In the most simple form of this approach, the hole environment will be rolled back in time if a causality error occurs. However there is only need for a roll back if an LP is related to the event that triggered the causality error. Several algorithms have been proposed which the this fact into consideration.

The Time Warp algorithm [15, 8, 39] actually only rolls back the LP which produces the violation. By checking the (risen) events, during the time range of the roll back, the receiving LPs will also be restored. As you can see this algorithm, which is in shape a tree beginning from the root, runs a wave of rollbacks through the simulation. In order to increase the performance, the essence of this algorithm, besides keeping the roll backs low, is of course in the first place to keep the wave as shallow as possible. The scheduling [32] of the events can have a significant influence on the amplitude of the wave. Which scheduling order, *Lowest Timestamp First* (LTF), *Lowest Local Timestamp First* (LLTF), *Grain Sensitive* (GS) [31] or others, is suited best strongly depends on the model.

Sometimes even if there was a causality violation the resulting events which are emitted by this LP might not change. Therefore the Lazy Cancellation Algorithm proposes that only those LP s are rolled back which have received faulty events. If the assumption that even wrong input data results in a true response is accurate then we can gain performance. However considering the delay of the roll back in the worst case the roll back could take more time then to restore the whole system in the first place.

Most optimistic implementations are a combination of different approaches to reduce the weaknesses of a single approach. Furthermore most non-specialised simulators provide a set of synchronisation algorithms. Which algorithm is suited best for the given simulation model can be determined by testing them.

PDES is the response to the growing availability of parallelised systems. On parallelised systems one simulation layer is distributed over multiple cores and some of the LPs are actually run simultaneously. Not only that we now can increase the

speed for one simulation run, we can also take advantage of the additional memory and other resources which most interconnected systems have.

1.2 Project Proposal

Given the latest development in the HPC area, systems are getting more an more complex regarding the amount of nodes and also the amount of cores per socket. With respect to HPC application scalability, fault tolerance and new programming models are more and more moved into the focus of attention. In 2009 the first supercomputers were able to run with a sustained performance of 10^{15} (peta) flop/s. At the end of this decade the first supercomputers will reach 10^{18} (exa) flop/s [6]. If the current trend continues these systems will have a huge amount of cores. But this is only one possible way future HPC systems could be designed. We can already assume today that the ratio between computation, communication and I/O capabilities will shift.

There is the quantum computer which does not work with bits in the classical

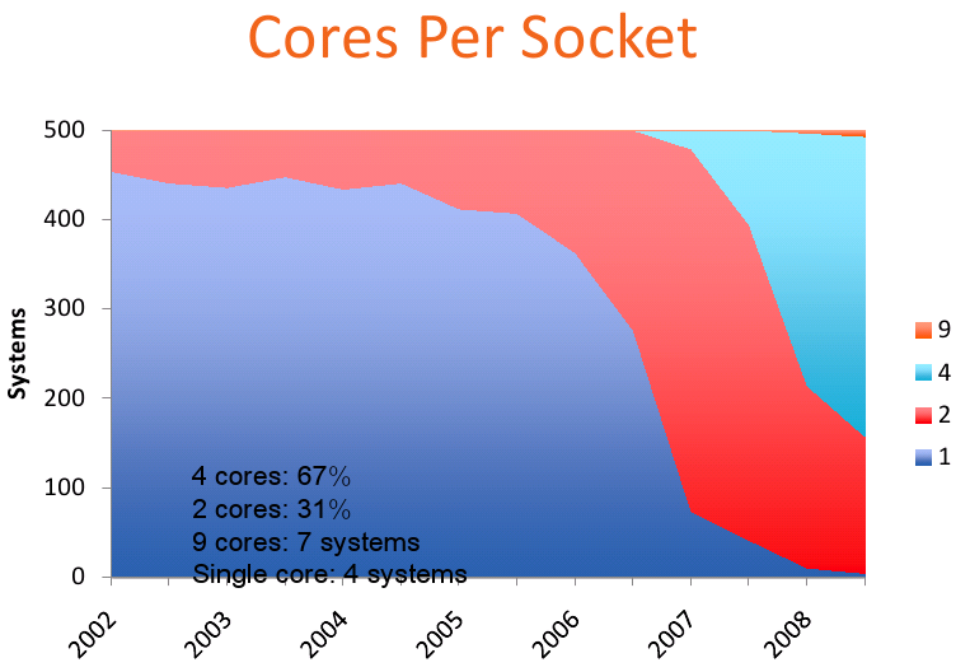

Figure 1.4: Cores per Socket [37]

sense. It manipulates quantum bits. These bits are able to be in one state with a

certain probability. The ALU is replaced by gates. Within those gates the quantum bits are transformed. One of the really big advantages of this procedure is that it does not matter how many bits pass such a gate at once. One good example is the breakdown of huge numbers: with only a few operations one should get all the prime numbers [24].

Computer science has for a long time been a multi-science discipline. Computing with cells is one topic in the biological computer science field. Those systems are called membrane systems. The cells internally have a hierarchical organisation which is defined by the membranes [14].

A more conventional research project is Paintable Computing [11]. It proposes small integrated circuits, which are equipped with memory and communicate via wireless transceivers as small as a grain of sand. Those circuits are uniformly distributed in a semi-viscous medium which could be used to paint supercomputers on surfaces for example a wall. Understanding the impact of an algorithm on new systems and pointing out its bottlenecks is the key for optimising future architectures. On the other hand knowing the bottleneck of applications is also crucial in order to use the available resources more efficiently. Therefore it has to be a co-development of all components.

HPC is open to all kinds of computational systems but in my opinion the near future probably belongs to some kind of combination of CPU and *Graphics Processing Unit* (GPU). Hybrid chips which combine CPU and GPU cores consisting of different sizes and providing different speeds. Thus the applications make use of the GPUs' specialised floating point vector computation and the CPUs can be used to handle the data preparation, and to do the integer compuatation and communication.

This project proposes to develop a simulator based on PDES which allows the user to simulate systems with thousands of cores. An MPI process in the simulation will be abstracted to an LP. Even though the behaviour of each LP will be like it was running in a self contained process, it is actually only an object of the simulation. All objects are processed in a fair share manner on n real CPUs. In such an environment the application's MPI messages will be transported by a simulated MPI layer. To account network delays correctly and remove the simulation's overhead the message delivery can be deferred by a calculated time of a modeled virtual network. The aim is to simulate an advanced large scale computer network environment with at least up to 10^7 virtual entities, for studying application behaviour on future architectures. Such a simulation requiers a lot of resources therefore the simulation will be scaleable over 10^3 nodes. The major points of interest are algorithm scalability, resource usage and fault behaviour.

Java Cellular Architecture Simulator (JCAS) and parts of the $\mu\pi$ simulator are available as a starting point. Basically the JCAS has to be rewritten from JAVA into C++ and the TCP/IP communication has to be replaced by MPI. This allows the deployment of the simulator on HPC systems where often no TCP/IP and JAVA support is available. The schematic layer concept is shown in Image 1.5. On the left side is JCAS ' current implementation and on the right hand side is the redesigned version with MPI and PDES.

The integrated set of DESs has to be replaced by PDES. Since the efficiency de-

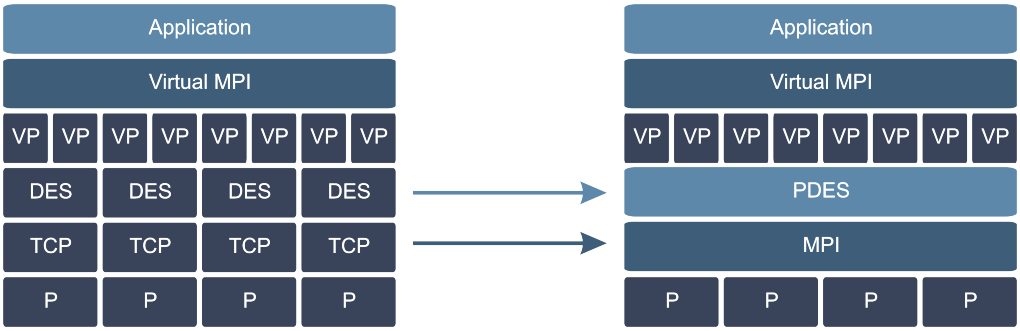

Figure 1.5: Technical layer design

pends on the combination of the application and the PDES synchronisation model, multiple sets of algorithms have to be implemented. First of all a conservative approach with a look-ahead mechanism, followed by an optimistic approach. After that an optimistic approach with a time-warp synchronisation will be implemented.

All synchronisations require a VT. The current implementation of JCAS so fare has no VT. So it has to be implemented in order to realise the PDES synchronisation models.

The current version of JCAS only supports fundamental virtual MPI functionality. This set of functions will be extended to be able to execute a wider range of applications in the simulator.

Investigation of fault tolerance of an algorithm is one of the main requirements with respect to the simulator. That is why the existing fault injection will be extended to inject faults based on failure distributions.

2 Related Work

The simulation of HPC systems for analysing purposes is growing in the past. There are several projects related in some kind with the proposal of this theses. Some of the closely related ones are:

- $\mu\pi$
- JCAS
- BIGSIM
- CHARM++
- AMPI
- MPI-SIM

$\mu\pi$: this simulator [27] is still under development by Kalyan S. Perumalla, Ph.D. from the Oak Ridge National Laboratory. $\mu\pi$ is a scalable, transparent system for experimenting with the execution of parallel programs on simulated computing platforms. The level of simulated detail can be varied for application behaviour as well as for machine characteristics. Unique features of $\mu\pi$ are repeatability of execution, scalability to millions of simulated (virtual) MPI ranks, scalability to hundreds of thousands of host (real) MPI ranks, portability of the system to a variety of host supercomputing platforms, and the ability to experiment with scientific applications whose source-code is available. In proof-of-concept experiments, $\mu\pi$ has been successfully exercised to spawn and sustain very large-scale executions of an MPI test program given in source code form. Low slowdowns are observed, due to its use of purely discrete event style of execution, and due to the scalability and efficiency of the underlying parallel discrete event simulation engine, μsik [28].

The $\mu\pi$ software is written in C/C++, as an application of the μsik PDES engine. Both $\mu\pi$ and μsik are portable to a large number of platforms. $\mu\pi$ has been tested on MPI-based platforms, including Linux, Mac OS X, Blue Gene/P, and Cray XT4/XT5. The MPI routines implemented, in C/C++ and FORTRAN, are:

- *MPI_Init(...)*

- *MPI_Finalize(...)*
- *MPI_Comm_rank(...)*
- *MPI_Comm_size(...)*
- *MPI_Barrier(...)*
- *MPI_Send(...)*
- *MPI_Recv(...)*
- *MPI_Isend(...)*
- *MPI_Irecv(...)*
- *MPI_Waitall(...)*
- *MPI_Wtime(...)*

Only the MPI_COMM_WORLD communication group is currently recognised.

JCAS: a simulator based on the cellular algorithms theory [13] developed by Christian Engelmann from the Oak Ridge National Laboratory 2002. The use of lightweight user-level processes allows the JCAS to scale a simulation system efficiently up to 10^6 LPs, on up to 10 real processors. It has been used for several years to explore algorithms scalability and fault tolerance in large-scaled systems. Due to the implementation language JAVA, the simulator can be deployed on various platforms. The communication between the compute nodes of a simulation is based on TCP/IP.

BIGSIM: was initiated by the BlueGene/C project to study programming issues in emulated future HPC systems [41]. It was developed by the University of Illinois. The parallel simulator BigSim is for performance predicting of machines with a very large number of processors. The simulator provides the ability to make performance predictions for machines such as Blue-Gene/L, based on actual execution of real applications. Based on the low level programming API provided by the emulator, several parallel programming languages are implemented on BigSim. They are MPI, CHARM++ [10] and Adaptive MPI. An online mode of the simulator is also useful in studying various performance issues in parallel applications, such as load balance and fault tolerance issues. Using the virtualisation of CHARM ++ [11], the BigSim emulator presents the execution environment of a petaflops class machine. CHARM ++ or AMPI[9] applications are compiled to run on the emulator just as though it were anyother architecture. BigNetSim currently includes

support for Torus, 3D-Mesh, Fat-Tree and Hypercube topologies. According to the article [38], the BigSim has been scaled over 128 processors by simulating a 8,192 node hyper cube back in 2005.

CHARM++: is an object-oriented portable parallel programming language [19] based on C++. Charm++ is an explicitly parallel language consisting of C++ with a few extensions. It provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven, thus helping one write programs that are latency tolerant. The language supports multiple inheritance, dynamic binding, overloading, strong typing, and reuse for parallel objects. Charm++ provides specific modes for sharing information between parallel objects. Extensive dynamic load balancing strategies are provided. It is based on the Charm parallel programming system, and its runtime system implementation reuses most of the runtime system for Charm.

Charm++ supports abstractions for special modes of information sharing, in addition to supporting communication via messages. It incorporates a message-driven scheduling strategy, which is essential for latency tolerance. It provides extensive support for dynamic load balancing, and prioritisation of messages. It supports a novel replicated type of object called a "branched chare" which has a sequential as well as parallel interface, and which can be used for efficiently programming data-parallel applications. Charm++ does not depend on an operating system provided threads package, hence avoids the corresponding overhead and non-portability. It is also one of the few systems that has been implemented on many commercial shared as well as large distributed memory machines.

AMPI: an MPI implementation and extension, that supports processor virtualisation [18]. AMPI implements virtual MPI processes (VPs), several of which may be mapped to a single physical processor. AMPI includes a powerful runtime support system that takes advantage of the degree of freedom afforded by allowing it to assign VPs onto processors. With this runtime system, AMPI supports such features as automatic adaptive overlapping of communication and computation, automatic load balancing, flexibility of running on arbitrary number of processors, and checkpoint/restart support. It also inherits communication optimisation from Charm++ framework. AMPI has been ported to a variety of supercomputing platforms, including Apple G5 Cluster, NCSA's IA-64 Cluster, Xeon Cluster, IBM SP System, PSC's Alpha Cluster and IBM Blue Gene. AMPI is an active research project. It already supports most MPI-2 features.

MPI-SIM: is a library for the execution driven parallel simulation of MPI programs [30]. MPI-SIM, built on top of MPI-LITE, can be used to predict the per-

formance of existing MPI programs as a function of architectural characteristics, including number of processors and message communication latencies. MPI-LITE is a portable library that supports multithreaded MPIThe simulation models can be executed sequentially or in parallel. Parallel executions of MPI-SIM models are synchronised using a set of asynchronous conservative protocols. MPI-SIM reduces synchronisation overheads by exploiting the communication characteristics of the program it simulates.

3 Problem Analysis And Specification

The JCAS is used as the basis for this project, because of its capability to simulate a large pool of entities 10^6 with the resources of only 10 computers. Even though the simulator is written in JAVA and does not allow to run unmodified code, the small resource requirements for simulating LP s distinguishes JCAS from all related projects and should represent a good foundation. One of the main issues of the current JCAS implementation are resource shortages for the simulated program. Many applications have a larger virtual memory need and require more computational resources then the current supported set of basic applications. Therefore the simulator itself has to be able to scale to at least 1,000 real processors to provide enough capability for each LP.

The fundamental problem in scaling the simulator is to keep simulation synchronisation overhead low in regard to network traffic and memory. Both components are strongly dependant on the used synchronisation model. All optimistic ways special to store the state in discrete intervals. Memory reserved by these state checkpoints cannot be released before the VTS of a stored state is in the past of the GVT. In a worst case scenario, this could lead to an out-of-memory error. It has to be investigated how it is possible to reduce the data and to swap out these checkpoints to a mass storage device.

Since the simulator shall be virtually scalable up to 10^7 processes, depending on the amount of messages the LPs are exchanging between each other, one of the usual known bottlenecks of HPC will kill the simulators performance. By exploring how to reduce virtual message data before sending it over the network, this effect can be cut down. After all, only compressing the data is not enough, because on the one hand this approach is only successful if the data can be compressed well and on the other hand compressing the data takes additional computational resources.

Avoidance of causality errors is essential to guarantee that the results are accurate and repeatable. All the PDES synchronisations rely on the VT. Coordinating the current GVT of all LPs with a conservative synchronisation could also lead to an error. For example, LP_x sends a message to LP_y and while the massage is still on

its way trough the network, the safe execute time is increased. A second message with a later VTS which is already in the event queue may be processed before the message on the network arrives in the queue. Appropriate security mechanisms have to be implemented to avoid such a scenario.

On the whole the minimum requirements are:

- Running unmodified MPI applications.
- Replacing the JAVA core of the JCAS by a C++ one.
- Replacing the TCP/IP communication by MPI.
- Implementing a conservative PDES.
- Implementing basic MPI capabilities.

Optional requirements are:

- Implementing the optimistic PDES synchronisation approach.
  1. A simple version which rolls the whole system back.
  2. A time warp algorithm.
- Extending virtual MPI capabilities.
- Implementing and extending fault injection from the JCAS.
- Adding fault injection based on failure distributions.
- Implementing resource usage analysing mechanisms.

4 Design

4.1 Logical Process

The first question is, how can an LP be implemented? An LP is basically a MPI process. For most applications we can strip this abstract definition down to a few elements. The core data components are:

- Status
  It is obvious that it is not possible to run several thousand LPs simultaneously, no matter which implementation is used. By assigning a status to the LP, it can be determined whether it is running, suspended, etc..
- VT
  The PDES mechanism synchronises the system with the VTs of all LPs. Here, the VT is simply a sum of the time an LP has actually used the CPU.
- Stack
  All local variables of a process or a thread are stored on the stack. In case of a simple application which does not dynamically allocate memory, the stack contains the current content of the LP, in regard to the advancement in the program flow.

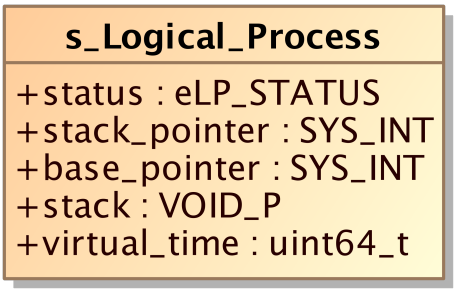

Figure 4.1: Concept of the LP struct

A struct combines these elements in one object, see image 4.1. The code which manipulates the data is not LP specific, it is only the advancement in the program flow which the stack will keep track of. In the end an LP is nothing more than a collection of data in the memory.

4.2 Virtual Machine

The virtual machine serves the purpose to create an environment for running a large pool of LPs and to enable the simulator to address each LP as an MPI rank. By design, an LP is reduced to being a data struct, which is manipulated by simulated code. The *virtual machine* (VM) has to provide specific environmental key features to guarantee a proper execution of the code.

- Run the code in a self-contained environment.
  The aim of the project is to simulate a large-scale HPC environment for MPI applications. All LPs' communicator handles involved in a simulation have to be implemented in a way that as far as the LPs are concerned, they run in a real MPI process. So the only communication between LPs consists of sending messages via the virtual MPI interface.

- LP content can be easily accessed for checkpointing.
  All optimistic PDES synchronisation mechanism architectures have no protection against executions which produce causality errors in the first place. But they do recover the whole system or affected sections after the detection of such an error. To enable the PDES to do a recovery, periodical checkpointing has to be performed. Each checkpoint of an LP represents a content dump at a specific virtual time. Through that the synchronisation can restore the content to a virtual time before the error occurred.

- A small memory foot print.
  Imagine the requirements to simulate at least 10^7 entities on up to 10^3 real nodes defined in the project proposal Section 1.2, that means one node simulates at least 10,000 (10^4) LPs. A comparably tiny memory demand of a few Kbytes lead to a huge memory usage. A 100 kbyte stack for each LP multiplied by 10^4 results in almost one Gbyte. With an optimistic PDES approach the memory requirements rise even more as the LPs have to be saved more often, therefore it is essential that the application has a small memory footprint.

4.2.1 LP Runtime Environment

Since an LP is a simulation of an MPI process the first thing that comes to mind is to implement it as an actual process. There are certainly advantages in this kind of implementation as there are no shared resources and the content switch is handled by the OS, to name only two. Running multiple processes on one node (subscribing a node) is supported by some MPI implementations, however, this does only make sense when only a small number of MPI ranks are spawned on one node. But the simulation has to be able to run no less then several hundred LPs on one node. Subscribing nodes on such a large scale is not intended by design. Anyway even if it might be possible, there are some major disadvantages of using processes. In terms of memory usage, every process requires at least one system page size for its structural data, plus the amount of memory for the text and data sections. These memory requirements are multiplied by the amount of LPs. And finally a process switch is comparably slow. Each switch generates a kernel trap, where the actual process content is swapped and replaced with another one. Considering the already mentioned downsides of real processes, the implementation as a process is not efficient and therefore this approach will not be pursued.

A thread implementation would be the next logical choice. Using a threading library like pthread, based on the *Portable Operating System Interface* (POSIX) definition, implements a thread as a light-weight user process. The name is chosen because a thread is literally implemented as a process. Some features like a user managed stack allow the programmer to influence the behaviour at runtime. Especially the user-managed stack could prove valuable for the project. Not only that a stack as small as 4kbyte can be used, but also the PDES synchronisation mechanisms get an easy access to allocating memory on the heap. Here is no need for special stack operations because the allocated memory can be addressed as a native datatype array. In a simple test program, which originally was written to verify the availability of the user-managed stack feature, an initial stack usage of $\approx$ 3kbyte could be calculated directly after the thread enters its starting routine, see test program A.1. This stack memory is used to store some of the threads' structure data. This allows for a very low memory footprint. Further a thread's content switch should be faster than the content switch of a process because shared resources like address space, signal handler, timer, etc. will not be changed invoking the switch. Sharing resources bears advantages as well as disadvantages. Depending on the implementation of the program being simulated sharing file descriptors and signal handlers can be a hindrance. Without certain mechanisms sharing them could change the program's behaviour. Wrapper functions would be necessary to prevent the simulator from altering the program's characteristics. Nevertheless, this is one possible implementation strategy.

By inspecting the implementation of a cell (LP) in the JCAS source code, a third approach can be found. Here a pool of cells runs in one single thread. The content switch itself is implemented and executed completely in user space [22]. This approach is not only basically an extension of using multiple threads but it also reduces the stress on the OS. Still from the LP's point of view everything runs in one real thread, but the content switch is speed up. So far it is the best course of implementation to keep the simulation's resource overhead low.

4.2.2 Virtual MPI

Considering the solution of the runtime environment analyses, running an LP in a thread does not allow the LP to be assigned to a real MPI rank. The current MPI definition determines only a process as a rank. To overcome this restriction a virtual MPI layer enables the simulator to address each LP with a virtual MPI rank. Keeping the datatype, a 32-bit integer defined by the MPI standard for a rank, allows to address 10^9 elements. This is 10^2 times more then the project was specified for. Each virtual rank will be assigned to a real MPI rank. The simulation size will be distributed in two parts. First, a sequential distribution of all virtual ranks to the real MPI ranks keeps adjacent LPs on the same node. The LPs are distributed this way because often communication occurs between neighbouring ranks and if they are on the same physical CPU, it is not absolutely necessary to communicate over the network. If the LPs have to be distributed unevenly, then the first n MPI ranks get one virtual rank more. For example, a simulation size of 100 LPs on 6 MPI ranks will be distributed as followed:

MPI rank	Virtual ranks		
	range	amount	
0	0 – 16	17	big chunk
1	17 – 33	17	big chunk
2	34 – 50	17	big chunk
3	51 – 67	17	big chunk
4	68 – 83	16	small chunk
5	84 – 99	16	small chunk
	Total	500	

In the second part, the relative virtual rank counts the assigned chunk from zero again. This chunk will now be distributed incrementally between the VMs. Distributing the 17 LPs of the MPI rank 2 among 3 VMs, looks like this:

VM	Relative virtual ranks						Virtual ranks					
0	0,	3,	6,	9,	12,	15	17,	20,	23,	26,	29,	32
1	1,	4,	7,	10,	13,	16	18,	21,	24,	27,	30,	33
2	2,	5,	8,	11,	14		19,	22,	25,	28,	31	

Finally, the VM keeps an array with the corresponding LPs. The incremental distribution is chosen because it is a fixed way to calculate the necessary routing information through the simulator. Neither lookup tables nor the simulation size affects the costs of calculation. The destination of a message is determined by three values.

1. An MPI rank
   The calculation requires stored information like chunk size, either big or small, the first virtual rank within the small chunk and the amount of big chunks.
   - virtual rank $<$ first virtual rank in small chunk (r1)
     $$= \frac{virtual\ rank}{big\ chunk\ size} \tag{4.1}$$
   - virtual rank $>=$ first virtual rank in small chunk (r1)
     $$= \frac{virtual\ rank - r1}{small\ chunk\ size} + amount\ big\ chunks \tag{4.2}$$
2. A VM index
   $$= relative\ virtual\ rank\ \%\ amount\ of\ VMs \tag{4.3}$$
3. An array index
   $$= \frac{relative\ virtual\ rank}{amount\ of\ VMs} \tag{4.4}$$

All MPI communication messages are wrapped into a simulator message. Generally, an additional header is attached to the message. Besides the necessary routing information, the header describes the nature of the message, see Image 4.2. A Tag allows to determine which function or event was responsible for the creation. Such a tag is called send, broadcast or barrier for example. The first version of the simulator will not simulate network delays. However, later versions will be able to address different kinds of topologies and component latencies. Therefore each message is stamped with two virtual time values, see Section 4.3 for the definition of virtual time. The first time value refers to when the message was created and the second is the calculated time which determines when the message should be available for the receiving LP. The user data of every MPI message gets stored within the wrapper message.

Intercepting MPI functions, defined in the class c_VIRTUAL_MPI, emulate the outside behaviour of the original MPI library. Internally, the functions perform a

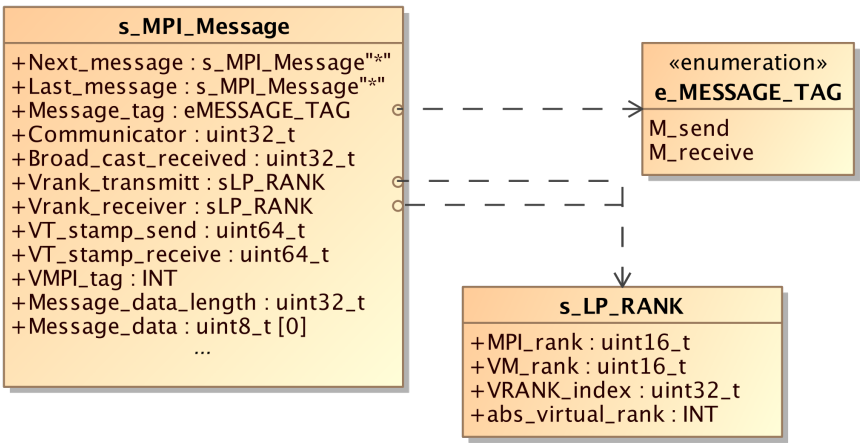

Figure 4.2: Concept of the virtual MPI message struct

number of operations: message wrapping, sending or receiving, message unwrapping. The functions are also used to trigger an LP content switch, in case the data is obtainable. For example when MPI_Recv() is called and the source has not sent the message yet.

4.2.3 Integrating The Application Into The Simulator

The first idea was simply to rename the main function, which would be done by an additional header file from the simulator. The file would provide a macro for the main function as well as for the MPI methods.

- Example: #define main(int argc, char *argv[]) main_app(int argc, char *argv[])

Further consideration brought up some weaknesses concerning this approach. One is the problem of global variables. Without an additional mechanism such variables would also be globally available to the simulator itself. Every LP would therefore access the same global variables. Unpredictable behaviour and wrong program output would very likely be the outcome.

Another problem is presented by the calls to the intercepted functions. The renamed main and its subfunction have no relation to the VM object in which the executed LP is located. Additionally, information has to be passed to the intercepting functions like a reference to the object. Complex programs with nested functions pose a problem as no assumption about the structure of the application can be made. So it cannot be determined where the reference actually has to be added.

Changing the main function into a member function of the *Virtual Machnine* (VM) object allows the LP to access it. All MPI calls can now be redirected by using the C characteristic, which links all MPI calls of a member function to the object's inherited MPI implementation first. The current design for this problem is still a preliminary one because it does not solve the global variable issue and only partly solves the object relation problem. Nested functions within the main routine will result in failure. As soon as an application calls a function which is no member of the object and this function does an MPI call, the real MPI library will be accessed. If this does not lead to the abortion of the simulation, the further course of the simulation is unpredictable. Hence there are two temporary restrictions with respect to the application being simulated.

1. No global variables are allowed
2. No MPI calls outside of the main function

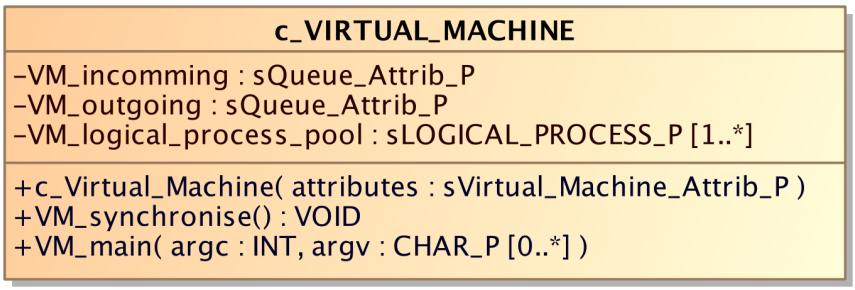

Figure 4.3: Concept of the Virtual Machine

4.2.4 Content Switch Logical Processes/Synchronisation

As designed in Section 4.2.1, each VM object is single-threaded. The function *VM_ synchronise()* allows in the first place to run a pool of LPs. From the LPs' point of view it seems as though it is placed as a self-contained process. The worker thread, a pthread with a usermanaged stack, will be started with a reference to the Virtual_Machine object that created it. The thread's *run_function()* calls the object's *start_ up()*. It is only used to call the *VM_ synchronise()* and could be skipped, but the synchronisation method should never be called from outside the object. The inline assembly used for the LP content switch is only intended to cope with the object's thread. By calling the synchronisation, the VM starts to cycle between two states.

1. Execute synchronised code
2. Execute LP code

Each time the state changes, it triggers a content switch. The swapping relies on the x86 architecture and the implementation of the C function call, which comprises a set of tasks.

- Store parameters on the stack or in registers
- Store jump back address on stack and call function
- Save used register

After entering the function, the stack is saved and replaced by the one reserved for the synchronisation tasks, see activity diagram 4.4. Before leaving the function, the data of another LP is restored. When the return statement is executed, the stored jump back address for this LP is used to brunch back to the position in the code when the LP called the synchronisation. As far as the LP is concerned, it only returned from a subfunction without having been swapped.

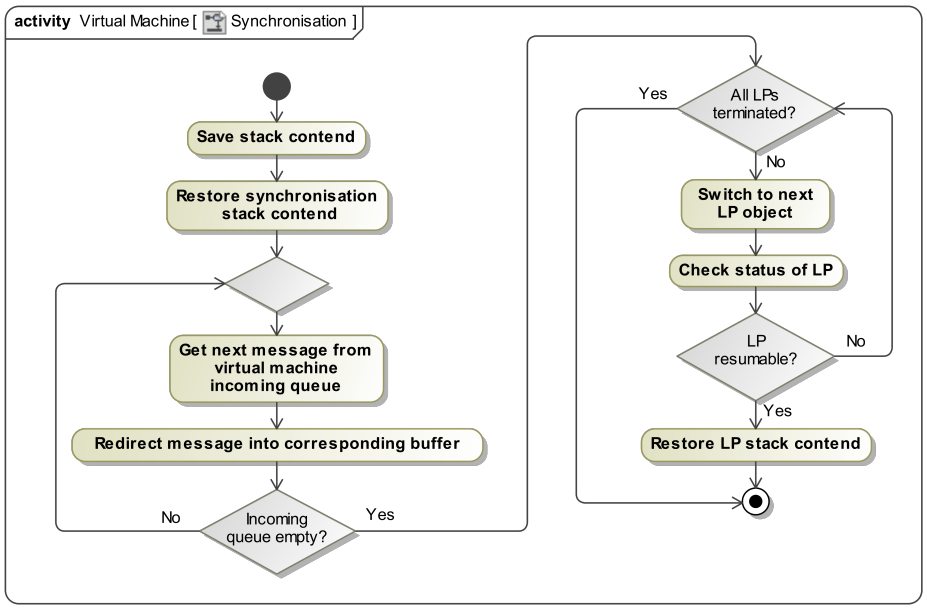

Figure 4.4: Activity synchronisation function

Jumping with the stack from one memory section to another can be done by

overwriting the *stackpointer* (SP) and *basepointer* (BP) registers. This is due to the fact that all stack operations are addressed relatively to the memory where they are pointing at. Unfortunately, no standard C or C++ way could be found to alter the register. So the current solution uses inline assembly.

- Intel

```
#define SET_STACK_POINTER(new_pointer)
                        __asm( "mov %0, %%esp": : "r"(new_pointer) : )
#define SET_BASE_POINTER(new_pointer)
                        __asm( "mov %0, %%ebp": : "r"(new_pointer) : )
#define GET_STACK_POINTER(store_pointer)
                        __asm( "mov %%esp, %0": "=r"(store_pointer) : )
#define GET_BASE_POINTER(store_pointer)
                        __asm( "mov %%ebp, %0": "=r"(store_pointer) : )
```

- AMD64

```
#define SET_STACK_POINTER(new_pointer)
                        __asm( "mov %0, %%rsp": : "r"(new_pointer) : )
#define SET_BASE_POINTER(new_pointer)
                        __asm( "mov %0, %%rbp": : "r"(new_pointer) : )
#define GET_STACK_POINTER(store_pointer)
                        __asm( "mov %%rsp, %0": "=r"(store_pointer) : )
#define GET_BASE_POINTER(store_pointer)
                        __asm( "mov %%rbp, %0": "=r"(store_pointer) : )
```

When porting the simulator to another architecture than Intel's x86 or AMD64 machine specific assembly code has to be added to the assembly_code.h header file.

4.3 Virtual Time/Global Virtual Time

Even though modern timers and libraries support resolutions within a nano second ($\eta s \widehat{=} 10^{-9} s$) range, they are too inaccurate for this purpose. Considering that today's CPUs operate with a clock frequency of n GHz (10^9Hz), n clock cycles represent a ηs. Thread timers usually do not use one of the hardware timers but are implemented as software timers. Synchronising software timers to ηs cannot be accurate since a refresh probably requires more then n cycles. The lowest eligible time base would be a μs (10^{-6}s). However with this time step, which represents thousands of clock cycles or thousands of operations respectively, the PDES synchronisation by virtual time can result in undetectable causality errors. To overcome this potential cause of errors, in a future survey an alternative solution like event-based logical clocks as in [10] has to be implemented.

Only the 64-bit C standard datatype allows to store an acceptable range of time in μs. Within this size, a range of up to 10^5 years can be represented. Using the next smaller datatype 32 bit would only allow a runtime of shortly over one hour. This counter will be updated whenever an LP executes synchronisation or virtual MPI code. The time span an LP utilises the CPU is calculated in two simple steps. First the thread's current CPU time, ascertainable by the function *clock_gettime(CLOCK_THREAD_CPUTIME_ID, struct timespec *tp)*, is stored directly after an LP's content is restored and before it is resumed. Then the thread executes the target application until a synchronisation section is entered again. Here the LP will be suspended like described in Section 4.2.4. By calculating the time difference between the stored and the now current CPU time, we get as results the LP's last run duration, which will be added to the LP's virtual time.

The design of the PDES's synchronisation mechanism is based on that VT. It is used to get the messages into a certain sequence and to ensure a causality error detection. The GVT of the system is the smallest VT value of the simulation. This time can be regarded as the safe time for event execution. All messages tagged with a younger virtual receive time cannot cause a causality error. Since all LPs have past this point in time, no LP generates a message at a point in time before the GVT anymore. An easy and very accurate way to get the GVT is a synchronised implementation. Here all LPs will be suspended, then the minimum VT will be determined and finally the simulation will be resumed. Suspending and resuming the simulation forces the system to waste some computational time at each reconciliation. When determining the GVT asynchronously, the determined value represents a time in the past of the system. Asynchronously means, that the simulation keeps running while updating the GVT. So the used data is only a snapshot of the system's past. The actual GVT of the continuously running simulation may already be ahead of the currently published one. Neither way we have extensive negative impact. The asynchronous approach is chosen because there is no major drawback for a conservative PDES and the memory consumption is only a little higher for the optimistic approach if the time span between published and actual GVT is small. The additional memory is required to store checkpoints which are collected during this time span.

Additionally to the wasted computational time the drawback inherent to the synchronous approach also raises a memory issue when the reconciliation interval is not suitable for the simulated application. At the beginning of the interval the determined time is the actual GVT of the system, but for the rest of the duration it also represents the point in time when the last update took place. To keep it accurate would require very short intervals and therefore a high computational

need for that task. On the other hand long intervals would have a higher memory need corresponding to the asynchronous approach.

Some asynchronous approaches propose to send a special message to all LPs. A corresponding response is sent back. By finding the smallest send time stamp of all messages, a possible past GVT is found. But the cost of this solution grows with the simulation size. In the PDES, not only the message has to be created, which is at a simulation size of 10^7 already a lot to handle but additionally all messages have to be sent over an expensive network connection. Also the LP itself has to be modified to participate correctly in the synchronisation process, which is contradictory to the requirement that the application is not to be adjusted to the simulation. Determining the GVT by generating such a huge traffic is too much stress for the system. Splitting the GVT replication into two steps allows to reduce the network traffic significantly. Instead of one single MPI node having to deal with finding the lowest VT among all simulated LPs, each MPI node finds the lowest VT within its assigned LPs as a first step. Afterwards this time called *Local Virtual Time* (LVT) is used to find the GVT. Thus each MPI node has to send only one value over the network instead of hundreds or thousands. The iterative search process is illustrated as a tree structure in image 4.5. The next higher level always represents a copy of the smallest element in the layer below. We have the root which is the GVT, the LVTs make up the branches and the VTs represent the leaves.

Appointing the LVT to one node can be done without any messages, even if more

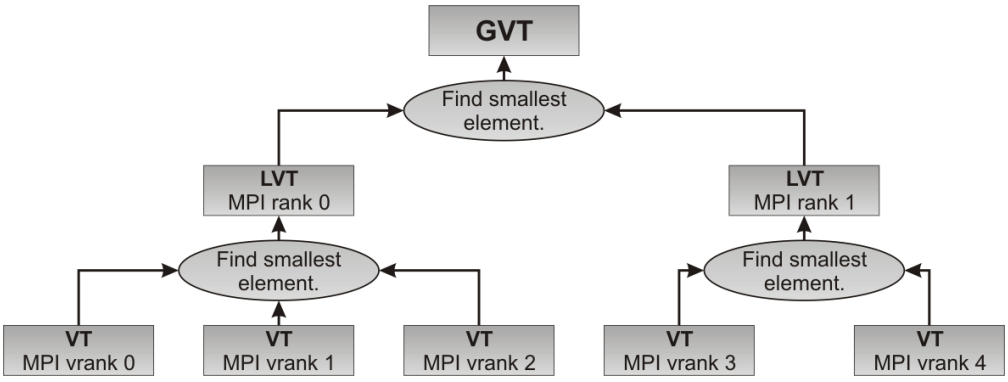

Figure 4.5: Dependency between GVT, LVT and VT

than one VM is running on it. A VM is nothing more than the thread of a process. With the address space being shared between the threads of a process, it is possible to do a VM-wide search for the LVT without creating additional communication

interfaces. Several implementations are being considered here. The general fact that a node only has to update the LVT at a change of the represented VT, is used here.

1. Reducing lists with all VTs.
   The idea was to generate a sorted list in which all VTs are stored. In case one LP's VT becomes greater than the next VT, it can be removed from the list and does not have to be checked any further, because there is at least one LP with a lower VT. Only changes on the head node, which always represent the LVT of the MPI node, initiate a search for the new LVT, by adding all removed VTs again. Afterwards, the new head node value is posted forward for a GVT synchronisation. The advantage here is that identifying a change in the LVT and finding the new LVT is simple. Also the time an LP needs to access its list elements can be reduced to a single dereferencing by providing a pointer to the list node in the LP's struct. The most CPU time consuming operation is to refresh the list. Each element has to search for the correct position $\Theta(n)$, where n is the amount of current nodes in the list, and has to be inserted $\Theta(1)$.

2. Sorted array of the VT
   All VTs of one MPI node are stored in a sorted array. As well as in the first idea a change of element zero of the array initiates an update of the LVT. An LP finds its own element by addressing the array with a stored index. Keeping the values always sorted allows to improve the complexity of the search to $\Theta(\log(n - x))$[1], when moving an element. To achieve this advantage every change of a VT generates additional costs for copying $\Theta(y)$ of the y elements between the VT's old and new position within the array. Also, the stored indices of the LPs belonging to the moved elements have to be updated. That could be avoided by storing a copy of the VT in the LP's object instead of the index. An LP does not know anymore on which index its virtual time is stored but it can search the array $\Theta(\log(n))$ for its last VT. We get rid of updating the LPs indices but now have to search the array. Neither way generates high costs.

3. Array of the LP's rank, sorted by VT
   To store the virtual rank instead of the VT would save half of the memory. The ranks are sorted in the array by the VT of the LPs. The memory usage is cut in half because a virtual rank is defined as datatype int. This datatype consists of 32 bits both on a 32-bit system and on a 64-bit system as opposed

[1] Cost for sorted array is $\Theta(\log(n))$, but since the VT can only advance in time the x lower x elements of the array can be ignored.

to the 64-bit datatype used for the VT. However sorting the array by the VT means that the virtual ranks are arranged stochastically in the array. So the search complexity is $\Theta(\log(n))$ to find an LPs index in the array. Additionally the repositioning creates costs of $\Theta(\log(n - x))$ for searching the new position and $\Theta(y)$ for moving elements.

4. Unsorted array of the VTs
   Here each LP addresses its VT by the relative virtual rank on the node. A copy of the LVT will be stored in an additional variable. If the time which is to be changed matches the LVT, the array is searched for the now youngest VT. Even though the search costs are $\Theta(n)$ since it is an unsorted array, the even more time-expensive writes for moving elements which are necessary in an approach with sorted arrays can be avoided.

Under the assumption that only a fraction of the VT changes effect the LVT the last approach should be the most effective.

Designed to utilise multi-core CPUs efficiently by running a set of VMs (worker threads) on one node, the array needs the appropriate thread-secure mechanisms. Therefore in the class "c_VIRTUAL_TIME_TABLE", see image 4.6, operations are protected by a pthread mutex against racing conditions. During normal operation the VT can only advance. By passing the last running duration of an LP to the function *VTT_update_virtual_time(...)* the time is added to the element. Allowing optimistic PDES approaches to rollback an LP a second function *VTT_set_virtual_time(...)* has to be able to set the absolute time.

The next iteration synchronises the LVTs to a GVT, which can be done the same way by setting up an array with all LVTs of the system. Every change to an LVT generates a message which is sent over the network to the root node. The root updates the time in the array. Detecting a change of the GVT follows the same rules applied to the LVT array. Considering that only the way of transmitting the values to the array is different using the class "c_VIRTUAL_TIME_TABLE" for both arrays is possible. Finally, by sending a broadcast message the root publishes the new GVT.

4.4 Message Queue

DES as well as PDES implementations have to arrange the events or, in this case, the messages in a consecutive temporal order. Naturally, the amount of messages varies constantly during a simulation. Using rigid structures like an array would mean either that the array has to be big enough for peak amounts or that it has to be resized once in a while. Adding a message would require to move data to create

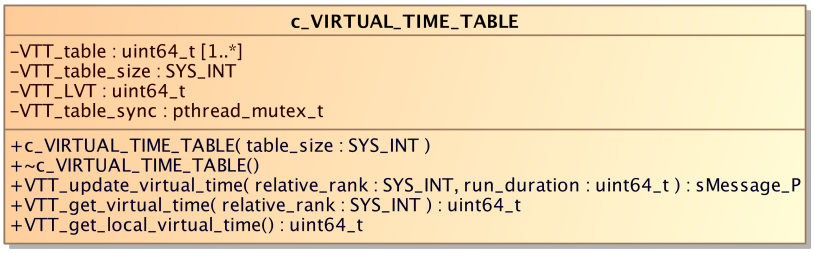

Figure 4.6: Concept of the Virtual Time Table

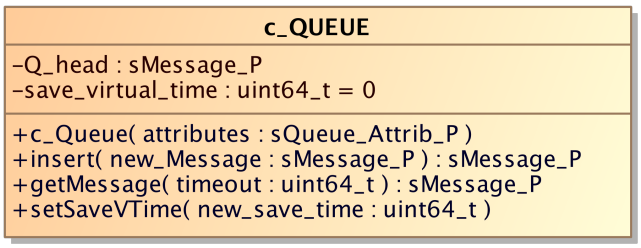

Figure 4.7: Concept of the Message Queue

a slot for a new element. In order to remove an element all following elements would have to be moved to close the gap. Using the array as a ring buffer would reduce the effort. This is mainly due to the fact that the head message will be primarily removed.This is done by increasing a single pointer without moving any data.

Dynamic data formats like a chained lists can adapt to the current message's appearance. A general benefit is the constant amount of operations for adding a new element. On the downside for lists, the search of the new element's position has a complexity of $\Theta(n)$ compared to $\Theta(\log(n))$ in a sorted array.

Whatever the storing structure used is, the queue is used likewise as an inter-process communication pipe to transport all messages from one code section or thread to another. Internally, it sorts all added elements by their receiving virtual time stamp. When the read function is called, the head element is returned. Considering the fundamentals of the conservative PDES here, a queue can be created which provides such a PDES synchronisation functionality. Based on the GVT, only messages which are save, with a receiving virtual time stamp which is lower or equal, will be returned. All other messages will be detained in the queue

until the advancement of the GVT releases them. With this approach, the queue only has a minimal additional overhead when removing an element. Instead of checking only for an empty queue, also an integer has to be compared. The get message function additionally allows to pass a timeout in ms. If the list is empty or only unsafe items are left, the method is blocking for the time specified. If a message becomes available during this time span, it will be returned. A zero is passed if blocking is unwanted and the constant WAIT_INFINITE to block infinitely.

The messages can be divided into two categories, point-to-point and collective. Group communication like a broadcast will not be replicated for each addressee. Instead, when a LP of a VM asks for one, a deep copy is returned. When a message-internal counter reaches one, the actual message is removed from the list.

The LPs run in fact not parallel, some can be more advanced in VT then others. In the worst case, a minority of LPs drop behind. This can stall other VM objects of the simulator. Only messages with timestamps younger or equal GVT will be delivered, to prevent causality errors from happening. In the worst case, all LPs of a virtual machine already wait for a message in the future of the GVT. Even if the right ones are already stored in the queue, they will not be released. To avoid such a scenario, a lookahead mechanism can deliver these messages if a combinational attribute footprint like source, mpi tag and communication type matches. Tagging the messages immediately when it is being inserted into the queue will reduce the search overhead. However, the list itself cannot identify a potential candidate since it has no information about the LP's status. Therefore, the VM could provide a check method which will be called before the message is added to the queue.

As the queue can be accessed by more the one thread, protection against racing conditions is necessary. Because of that all manipulating operations are enclosed by a mutex. An additional condition variable for the mutex is used to implement the timeout component.

4.5 Message Transport

A threaded communication class instance on each MPI rank binds the VM objects to the MPI_COWMM_WORLD communicator by forwarding the messages. Messages which address the same node are simply redirected from the outgoing queue to the corresponding incoming one of the VMs. Running the communication object in an own thread moves the real communication into the background. Which means that the actual exchange of MPI messages takes place asynchronously to the execution of the LPs. Theoretically the worker thread itself could also take

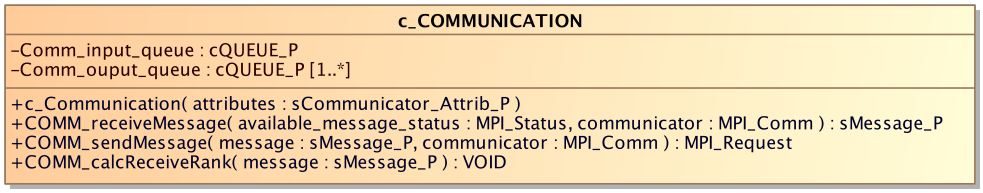

Figure 4.8: Concept of the Communication

care of that in a synchronisation section. However, long intervals between such sections could lead to an MPI buffer overflow. So using a threaded communication object should avoid such a scenario. Also current and future deployment systems are very likely equipped with multi-core CPUs. Increasing the priority of delivering messages by assigning one core to maintain the communication could improve the runtime behaviour of the simulation.

4.6 Overall Concept Of The Simulator

The concept of a process is schematically displayed in the image 4.9. Every process has at least three and up to n threads. The original thread of the process is idle and waits for the others to finish their work and return. One is spawned to handle the communication see Section 4.5 and one or more are created for the VM objects. Regarding a thread as a combination of resources, the VM threads could be seen as threads without a stack. Instead, allocated memory of the LPs will be used as such. Certainly a thread stack is also just a memory chunk, but in the sketch it shall highlight the way how a LP content switch is realised. When the simulator is enhanced by an optimistic approach, each LP needs images of its content from its past. This is already included into the draft.

However, a MPI environment has to be able to send messages. The process-internal message transport is shown in image 4.10. The separate components of the simulator are connected by message queues. Their use can be compared with a conventional pipe. Two or more objects get a reference to a message queue. At least one object primarily adds to the message queue and the others mainly receive from it. The current concept uses one queue to transport the messages to the communication object. This object collects all generated messages of the LPs and their VMs and arranges them in a chronological order. This optimises performance because the VT wise youngest messages are likely the next ones to be processed. By design, the queue see 4.4 returns them first. Therefore, the lower the VT, the higher the priority to be redirected or actually sent over the

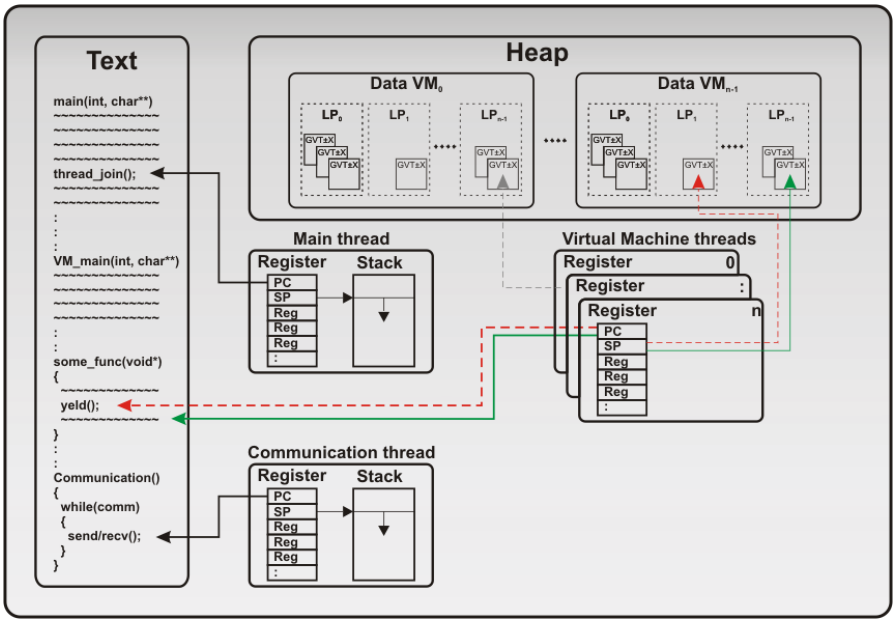

Figure 4.9: Simulator schematic draft

network. In the other direction, each VM is linked with an own queue. Basically, one would be enough, but then the queue would need to return the messages VM selective. Partitioning the messages to the VMs guarantees that each message from the queue belongs to a LP from the local pool. The sharing out is done by simply addressing an array of queues by the virtual rank's element "VM_rank".

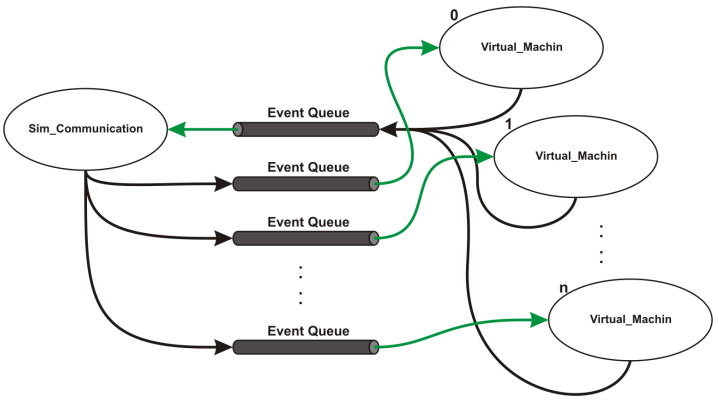

Figure 4.10: Process communication concept

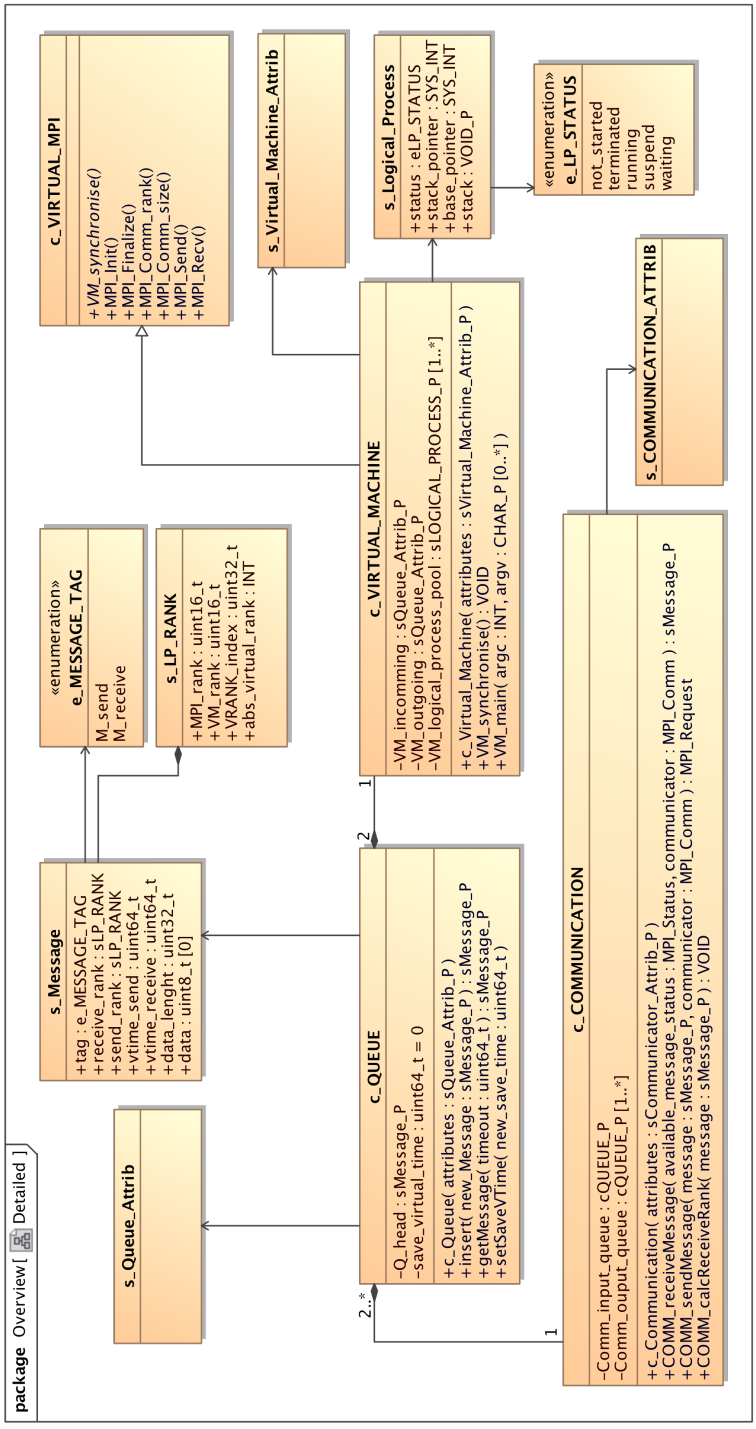

Figure 4.11: Detailed design overview

5 Implementation

This chapter describes in detail some of the implementation's key features. It gives an overview of how an LP is being executed. Also the supported MPI functionality and how the transport in the virtual MPI layer is realised are described. Additionally background information related to issues, which have arisen during the programming, is given.

5.1 Background Knowledge Function Call

A function call is not only a branch to a specific address in the program memory. It is a set of operations. Depending on the programming language, the programmer has to do some of this by writing the explicit code. When using high level programming languages, all these tasks are usually hidden by a simple function call like *foo();*. The compiler is then responsible for adding the required code, so that the statement is translated into a valid function call.

- Storing parameters.
- Store jump back address.
- Jump to address in code.
- Save used register.

As usual, there are many ways to achieve the desired goal. The programming language used, the compiler, compiler flags, etc., will all have their effects and therefore this analysis is mainly based on the mpic++ → *GNU Compiler Collection* (GCC) wrapper compiler using no flags. In this setup, a call creates a stack segment named *stackframe* (SF) storing the local data of the active function. In the majority of cases, such a frame is used in high level programming language to combine several sections [21].

- Callee parameters
- Return address to caller
- Saved registers

- Local variables

Depending on the implementation the separate sections may be handled in a multisegment approach, one stack storing only parameters, a second the remaining data.

The image 5.1 shows the stack some time after a process/thread was created. The coloured regions are SFs. These frames combine the previously mentioned sections for one function. In some cases, again depending on the various aspects, the stack frame [34] of the calling method may overlab the one of the callee. This is the case here. The addresses shown on the left side of the stack are added only to show that data is added to the bottom of the stack. A symbolic address representation of 0xXXXX will be used for further images related to the stack.

As in this project, an object-oriented programming language is used, the calling

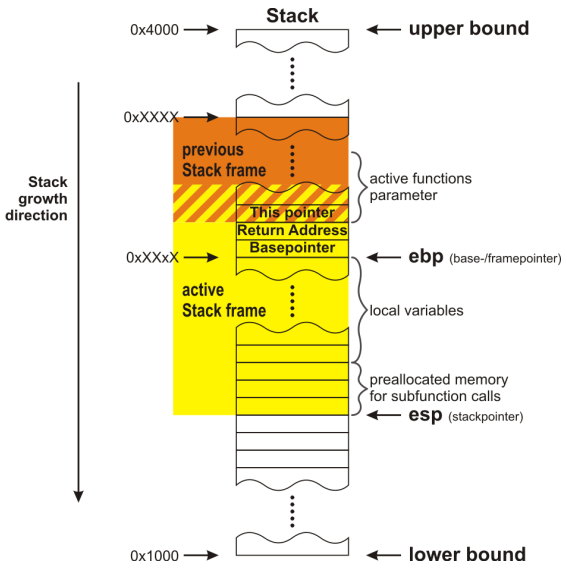

Figure 5.1: Call stack

of a member function will be examined. In particular of the *VM_main(...)* of a virtual machine instance. The this pointer serves as a mechanism to connect the function to the object. Accessing object attributes without a valid this pointer leads to unpredictable behaviour or a segmentation fault. This is why it is explicitly displayed here.

The image 5.2 shows the initial situation. On the left side, the program memory is listed with pseudo assembly code which is executed directly before and after a function call. The assembly code actually generated for the call is listed in the

appendix see B.1. The stack itself is shown in the the upper middle and in the lower middle there is the virtual machine object, which is located in the heap.

First the parameters are fetched. As usually done in C, this happens from the left

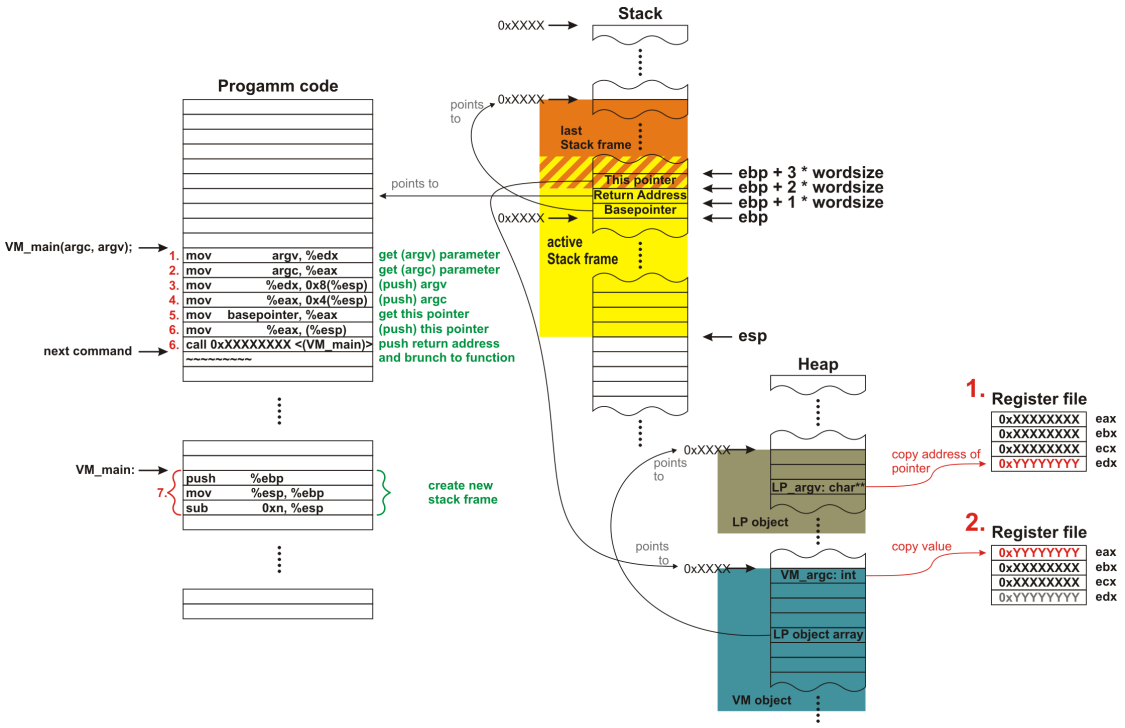

Figure 5.2: Call stack movements step one and two

to the right. Thus the last parameter is read from the memory first and stored in a register. Followed by the next to last parameter into another register and so on. Although it is now possible to pass these parameters by register, all parameters are actually passed by stack when using the mpic++ without optimisation option. This means the values have to be copied there. Although after entering the callee function it looks like the parameters are pushed [1] onto the stack, they are instead written relative to the SP, as shown in image 5.3. To allow such a write, the memory space has to be preallocated. To determine how much memory will have to be preallocated, the compiler parses the whole function for calls. The function which requires the largest amount of memory to store the parameters dictates the size of the allocated parameter section in the SF. This can been interpreted as "overlapping" into the next frame, since the parameters belong to the method

[1] Two main nodes of operation: either write data to stack then manipulate SP or manipulate SP then write data to stack

which is called.

Then an additional, not explicitly declared parameter is copied: the pointer to the virtual machine object. For this, see step five in image 5.4. Now, after all parameters are on the stack, the actual call can finally take place. It is a two step command. First the address of the next code line, the return address, is pushed, second the program branches to the address of a function which is called. Shown as step six in image 5.4.

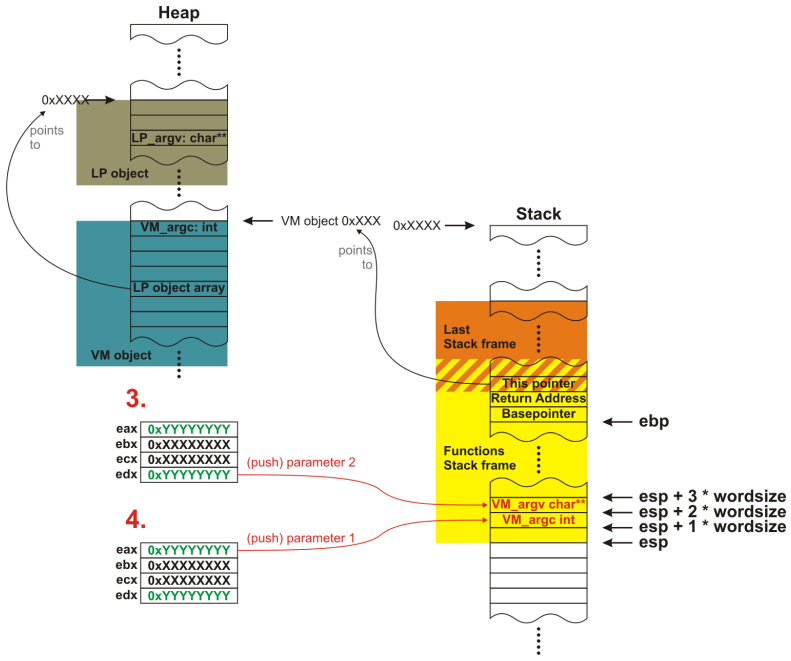

Figure 5.3: Call stack movements step three and four

As initial step, the function which is now active creates a new SF. Immediately after the function is entered, all registers which are currently in use, but at least the BP, will be pushed onto the stack. These saved values are automatically restored after the function's return statement. After that the current SP is copied into the BP register and becomes the new *framepointer* (FP). All local variables the active function will be addressed relative from this FP. To preallocate memory of the locals the SP is manipulated (subtract/add) by n bytes, which creates the frame. This already includes the maybe overlapping parameter bytes from to the subfunction calls, see image 5.5.

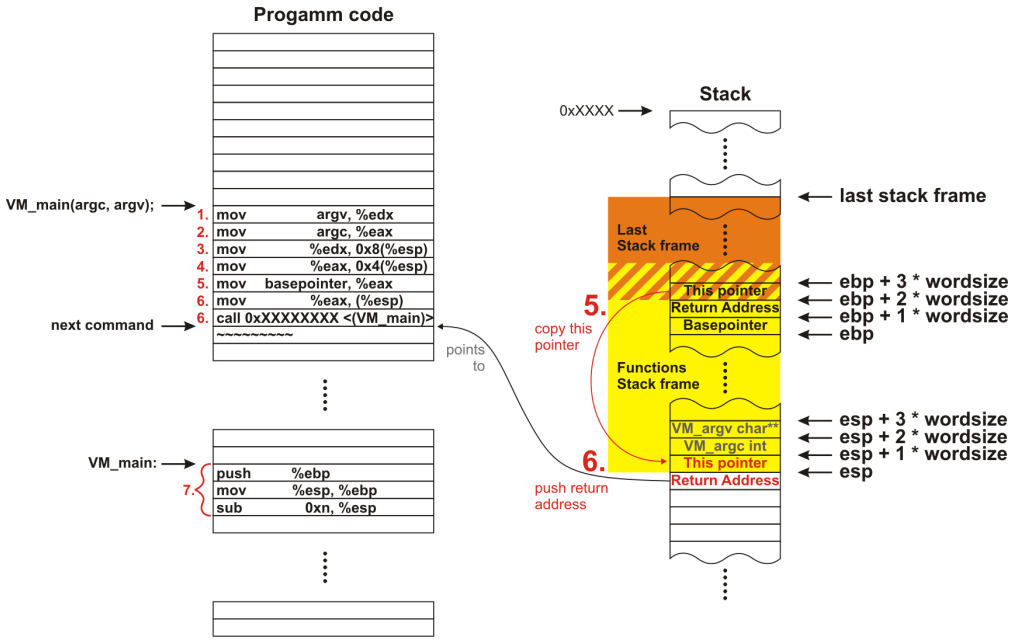

Figure 5.4: Call stack movements step five and six

5.2 Virtual Machine

5.2.1 Content Switch Logical Processes

The actual content switch is realised in the synchronisation function. This is the key function. It allows the VM to run multiple LPs. Depending on the amount of communication between the separate LPs, a large percentage of the total calls will be assigned to this function. Since the efficiency of the project is determined by the computation resources which the target application can use, it is important to keep the synchronisation function efficient.

Due to some complications during the implementation, two versions have been realised.

1. Copy the content.
   This approach is slightly simpler than the second one. Although the basic principle is the same, simply copying all of the content contains a smaller risk of programming errors. The concept is to store the BP and SP when first entering the function as point of reference. Thus whenever entering the method again all data between the current SP and the stored reference BP will be copied into a memory section assigned to the current active LP. As

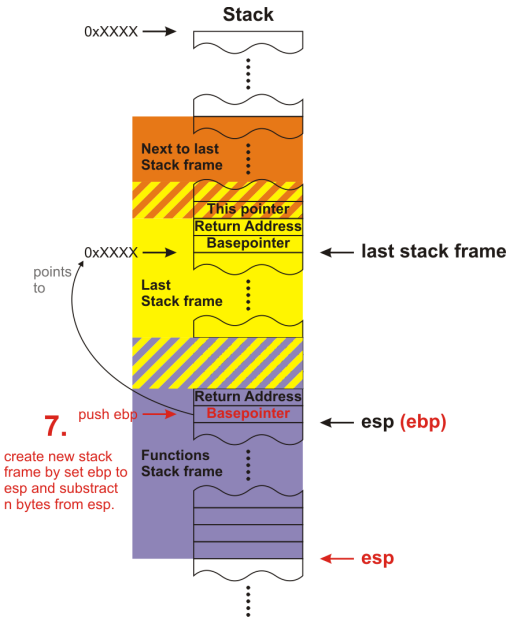

Figure 5.5: Call stack movements step seven

well as the BP and SP will be saved. Resuming an LP is done by copying the saved content back onto the stack, and restoring the saved BPand SP. Even though BP and SP have to be manipulated here, they are not designated to be set outside of the working thread's assigned stack boundary.

2. Move the BP and SP.
   By just manipulating the pointers, the actual stack content of the independent LP never has to be copied. As with the other approach the BP and the SP are saved as points of reference when first entering the function. But instead of copying the content, the pointers are manipulated in a way that each LP uses its own assigned chunk of memory as stack while it is actively running. When entering the synchronisation again the BP and the SP will be saved. To switch to another LPs content, the stored pointer values of this LP are restored. To do so it is necessary that the pointers are set outside the working thread's assigned stack boundary.

With both implementations, the synchronisation method serves two main purposes. The first is to redirect the messages from the incoming queue into the corresponding buffer queues. The second is the briefly described content switch between the LPs of an VM object.

Both version are displayed in the final activity diagram 5.6. The green and

turquoise sections highlight the difference between the two implementations. Green highlights the implementation, in which the stack content is simply copied, turquoise the one which only manipulates the pointers.

The redirection of the messages themselves is realised in a simple loop in which one message after another is fetched and added to the receiving rank's message buffer. In the case that it is no point-to-point communication, like a broadcast, the message will be added to a VM wide single queue. By separating point-to-point and broadcast communication, the messages addressed to multiple ranks can be stored as a single entry in the VM's global queue. Right now, only one MPI function is designed to be processed immediately after fetching a message from the queue. This is the barrier command, which releases all LPs waiting on that barrier by setting the LP's status back to running.

As proposed in sections 4.1, an LP itself is implemented as a struct, see image 5.7. An LP object array is created in which all LPs are stored. By adding an additional LP object, the synchronisation function can be implemented so that it is not necessary to check if the function is called for the first time when the referential BP and SP have to be saved. An object-based structure of the array is displayed in the image 5.8. As shown the memory used as a stack for the LPs is allocated as one chunk. The stack size can be determined by a program parameter whose value is multiplied by half of the system's page size. Allocating the memory as one chunk should avoid possible memory fragmentation and the chosen increment size is to guarantee a correct stack alignment.

After the creation of the thread, the index of the current active LP is set to the additional object in the array. When now entering the *VM_synchronise_LP()*, the first two assembly commands will save the BP and the SP into the additional object. Thus, the same code which saves the pointers of the LPs when entering, also stores the referential values. Even though, by using the additional object no check is necessary if the function is called for the first time. However, accessing the saved stack and base pointer causes additional overhead by addressing the object in the array first. Storing the values directly in a variable would be faster, but then a check would be required. With the approach which only manipulates the pointers, this simple mechanism basically saves the content of an LP. The other solution requires that additionally, the actual content is copied. When examining the code you can see that the copying is done in a loop, which copies byte per byte. Using a well tested standard library function like *memcpy(...)* instead would not only improve the code, it would also improve the efficiency. However, although this could be done when saving the content, it does not work when the data is restored. Such a function would of course perform a call, but the corresponding SF would be overwritten if the content to be copied extends the previous. Therefore

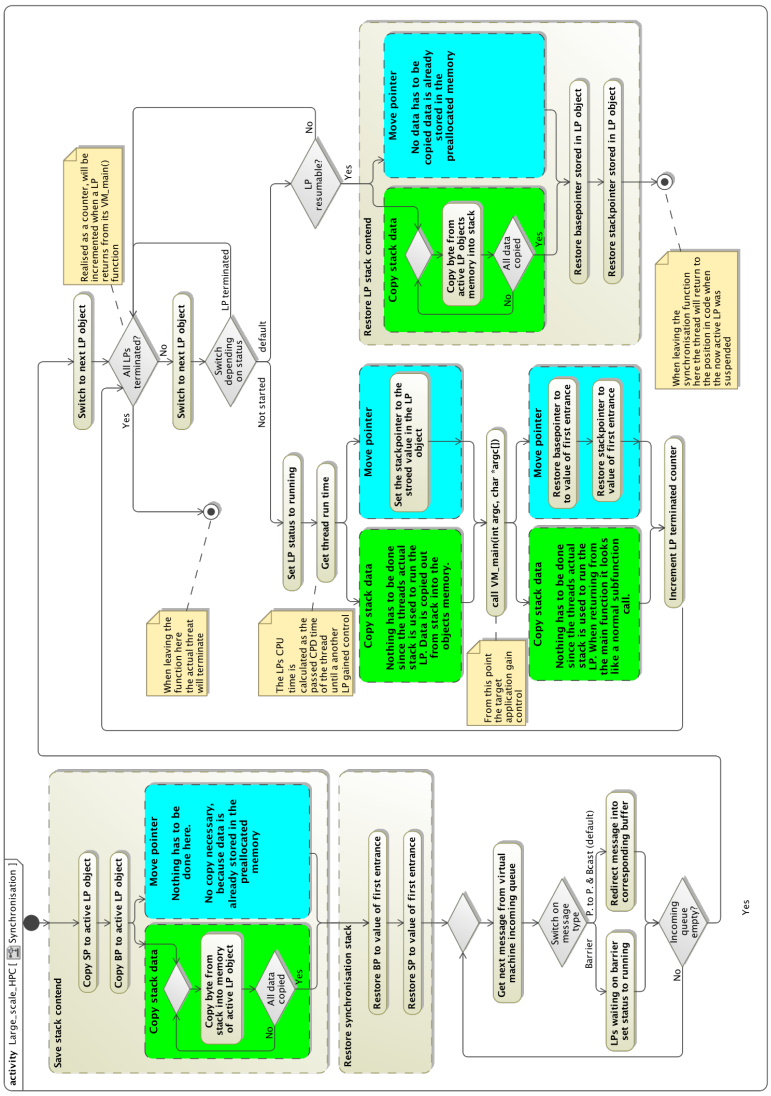

Figure 5.6: Activity diagram of implemented synchronisation function

the application will crash when the *memcpy(...)* executes it's return statement. Since it is not possible to use subfunctions to restore the content, both the saving as well as the restoring is performed by a simple loop.

In the following step, the reference call stack is restored by copying the pointers back into the registers. Through that the SF created when the function was called the first time is now active. Thereby from the view of the application flow it seems as if this is the initial call to the synchronisation function. From that point, each

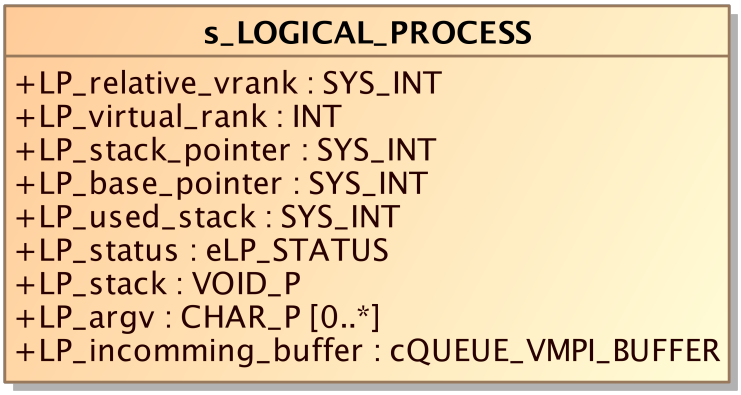

Figure 5.7: The logical process (LP) struct

LP can start the *VM_main(...)* function based on the same basis regarding the program flow.

The first approach needs no additional mechanism since it uses the original stack. All LPs can be started in such a way that the SF of the *VM_main(...)* is consecutively aligned with the one created when the *VM_synchronise_LP()* was called for the first time. When starting or resuming the next LP, the data written to the stack of the previously running LP can be considered as random data after being saved. The initial approach on the other hand does not write further SFs on the thread's original stack. Calling the main function is manipulated in a way that it does not arrange the frames in sequence but instead the *VM_main(...)* function's SF is placed in the LPs' allocated memory area.

This method starts at the same initial scenario described in section 5.1, except one additional assembly command before calling the main routine, highlighted red in the image 5.9. With this command, the SP is set n times the system's wordsize from the assigned memory boundary of the current active LP. The distance is required due to the mpic++ implementation of the call stack. Here, parameters are been written relative to the SP in the SFs overlapping section. If the offset to the boundary is to small, each call generates a segmentation fault, which was one issue during the implementation described later in this section. In case of any changes in the amount or types of the *VM_main(...)* parameters, this distance has to be adjusted. The implementation of a function call now creates a new stack frame at the position where the SP points to. After the synchronisation function has been called as often as there are LPs in the virtual machine, every LP has it's own *VM_main(...)* SF. These are all basically aligned behind the same last

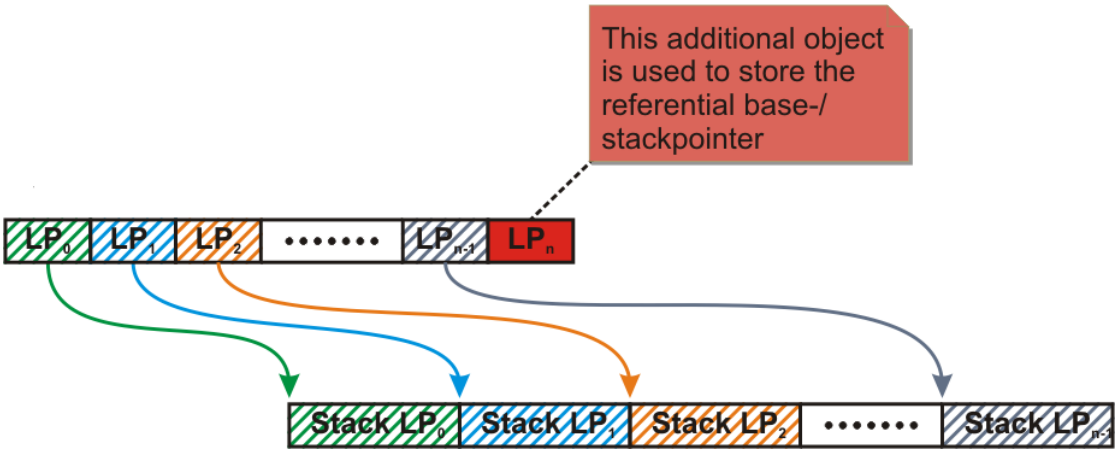

Figure 5.8: Array structure of the logical process (LP) array

active one, shown in image 5.10. When now a LP is resumed and the content is restored by copying the pointers, all further SFs are created independently and transparently to the other LPs in his own stack.

At this point during the development I was confronted with two issues. Both founded on a faulty knowledge about the real implementation of a function call.

The very first basic implementation of the content switch simply moved the SP of the executing thread from one LP's memory to another. The assumption was that the local variables are addressed relative from the SP. By setting the SP to an address in memory which points to a separate chunk of allocated memory for this LP, the content of the running thread should be switched.

The earliest test version based on this idea did run successfully. All LPs seemed to complete their main function and the program terminated afterwards without an error. All LPs had executed a simple 'hello world!' program. There was no communication nor synchronisation call at all during the execution of the main function, each LP terminated in a single subroutine call. Thus, this early test program can be seen simply as a loop which calls the main function multiple times. For a more real scenario the complexity of the main function had to be enhanced from a simple print statement to some loops which do calculations on dummy data. With a periodical call of the synchronisation method during the execution of the loop, the current LP is forced to suspend and transfer the control to the next LP. Thereby, multiple content switches are tested. To determine if a content switch was successful, a progress message is printed right before the synchronisation function is called. Despite the expectation that the LPs would cyclic print a

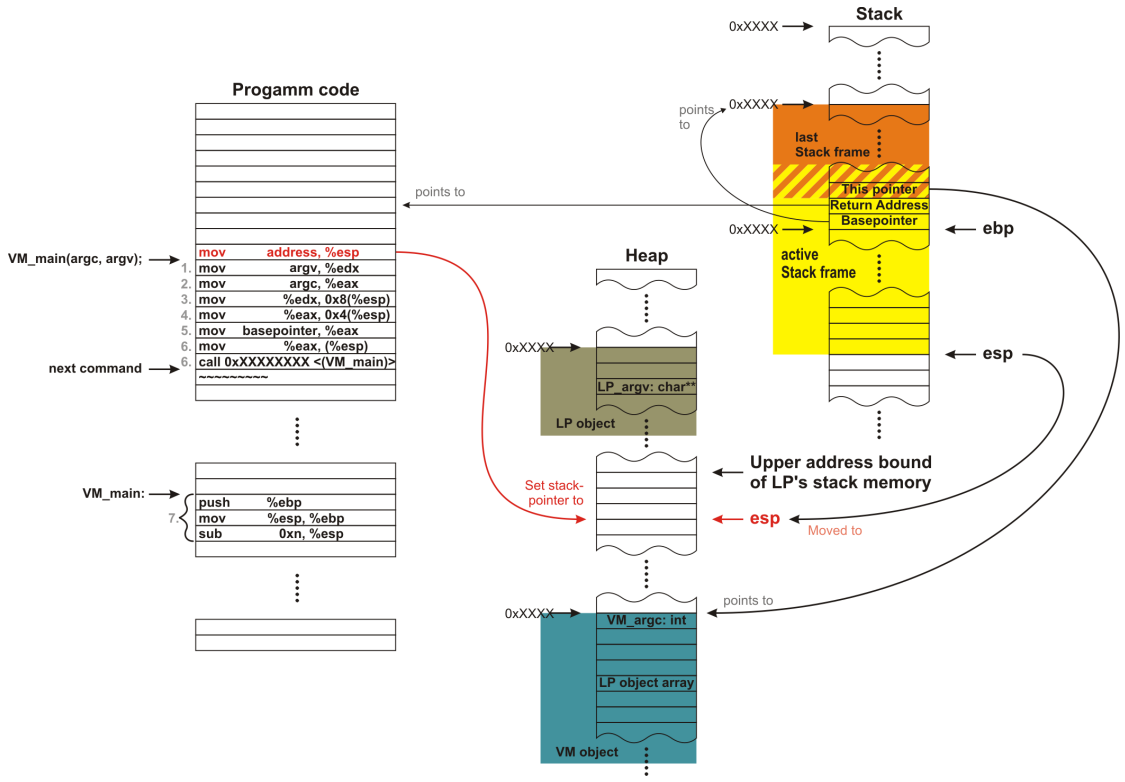

Figure 5.9: Start up initial situation

message, the prints indicated that only at the first cycle, until all LPs are started up, a content switch occurred. After starting the last LP it seemed that it never handed over the control until it finished. After that, it seemed like the next to last LP gained control and also did not release it until it finished. This pattern repeats itself until all LPs have terminated.

To determine the cause of this phenomenon, the application has been analysed using a debugger. At first everything seemed to be correct. Before the main function is called the SP has been moved to the desired position in the memory and after calling the main function, the local variables are located in the preallocated area. Further the control has been handed over to the LP next in line whenever the synchronisation method was called. In the control switch the corresponding SP has been copied into the SP register. After all LPs have been started and the content (SP) of the first has been restored again, but the LP still accessed the variables from the last started LP. Inspecting the assembly code produced of the main function, it was discovered that the variables are addressed relative from the BP instead of from the SP.

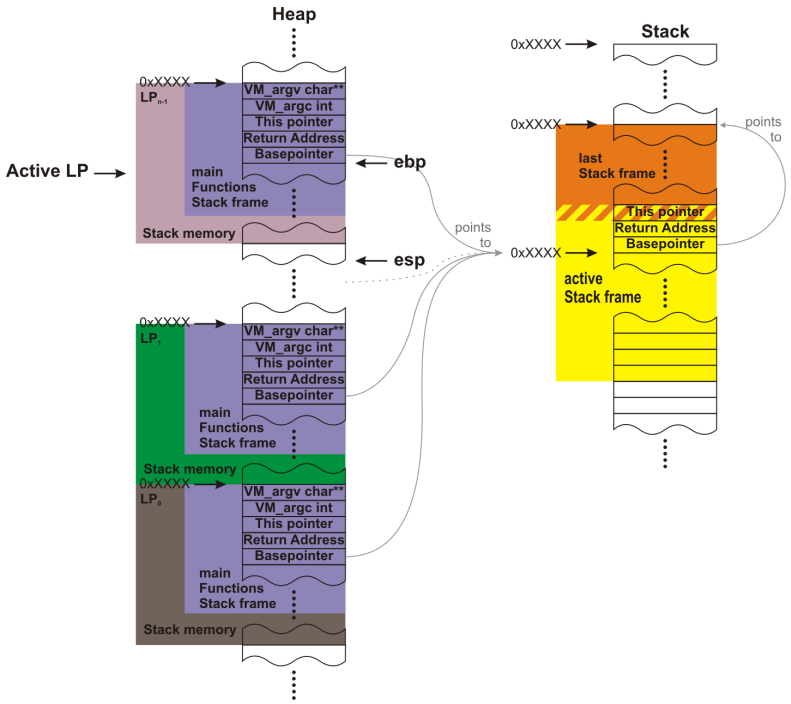

Figure 5.10: Stack after all LP s are started

This way, the phenomenon could be explained. Since the BP was never manipulated at the starting of all LPs, they did not really start independently, it was rather that each *VM_main(...)* function was indirectly a subcall of the previous one. Also the calls to the synchronisation did not switch the content, it only defined where in the memory the SF of the next call to the function will be created. A real switch to another content was only triggered when an LP hit the return statement of the main function. Until then, all LPs accessed the same content. However, a successful run of this test application was only possible because it is has such a basic main routine. If there were nested submethod calls and the synchronisation function would not be called symmetrically during the program flow, then it would very likely end in a segmentation fault.

The second issue was a segmentation fault. The curious about this behaviour was that the error took place at different points in the program flow. Additionally, the exception was always thrown from inside a *malloc(...)/free(...)* function, called from different positions in the code. The real source of the segmentation fault could finally be tracked down with a combination between the use of a debugger and the memory analyse tools valgrind/efence. The *VM_main(...)* call of the n.th LP violated the boundary of the allocated memory. More precisely,

the assembly section which passed the parameters did not push the data with the actual assembly command push, instead the data was written relative to the SP see 5.1. After all, the SP was already placed at a distance of a system word size to the boundary of the allocated memory. Certainly it was not enough to store the parameters. Originally, the reason for placing the pointer not directly at the boundary was that the mode of operation could not be determined without using inline assembly. To make the application highly portable, the inline assembly code should be used as little as possible. By reserving one word size as buffer, no writing outside the boundary of the allocated memory could be caused by the stack operating principle.

5.2.2 Virtual MPI

The still small amount of supported MPI methods already allows to programme basic parallel applications.

- MPI_Init()
  The init function is used to initialise some of the LPs variables. However, contrary to the real implementation it is not blocking. Assuming that all LPs will eventually call the function and the fact that the VT is not deployable yet, the runtime behaviour is changed to avoid a simulation delay at this point. For later versions, this method has to be extended by a barrier on the virtual communicator MPI_COMM_WORLD.

- MPI_Finalize()
  This function is right now a non blocking call, too. According to the produced result of an application, this does not matter. By not blocking, it simply terminates the LP. The structural program flow does not change because of this. Likewise to finalise a blocked mpi rank, the terminated LP does not participate at the communication any more. Still, this alters the runtime and has to be corrected in versions with VT measurements.

- MPI_Comm_rank() & MPI_Comm_size()
  Identical to their counterpart the virtual rank and the virtual communication size is determined and returned immediately.

- MPI_Isend()
  The MPI_Isend function has been implemented before the *MPI_Send(...)*, because the blocking send can be realised in the handshake sequence of its non blocking version and the blocking receive. As far as this function's purpose is concerned, it is to create an appropriate struct with the header information. The data is stored in a cross-platform manner appended to that.

As we can relinquish of a serialisation in homogeneous configurations a define switches between a memory copy and an MPI_Pack solution for adding the data. The define HOMOGENEOUS can be found in the file datatypes.h. Afterwards the new message is inserted into the outgoing queue. By copying the data the simulator may correct racing contition programming errors.

- MPI_Recv()
  The specified message will be searched in the LP's own buffer in an endless loop. If the matching one is found, the data from it is restored into the memory space passed as parameter. There is no verification between demanded, datatype size times amount elements, and the actual message data length. Trying to receive more data than present, will most likely result in a segmentation fault like in its real counterpart.

- MPI_Barrier()
  The barrier is not fully implemented yet. The principle was that every virtual rank sends a barrier notification to the root rank. After all communicator-associated virtual ranks are counted, a broadcast message allows them to resume.

5.2.3 Virtual Time

The design missed to address one potential source of causality errors during a GVT synchronisation. What happens with messages with smaller or equal VT stamps, which are still to be routed while an update is posted?

To ensure that there are no such messages, there is an interaction between the implementation characteristics of the queue and the communication class. This issue only arises if the simulator operates on the GVT. This implies the use of a sorted list. The sorting algorithm positions new elements before the first one with a greater receive VT. By adding the LVT change message after all critical ones, only items with a greater VT stamp can be following. To ensure that it is the last one, the virtual MPI send method's add the change message as last tasks. This works because of the sequential execution of the LPs. If the LVT changes, all LPs have already passed this time and therefore have placed their message in the queue.

Sorting the messages is the first step. The fact that the queue always disposes the head forces the communication to send the VT wise youngest messages first. As a transfer has to be concluded before the next can be commence, all messages have to be already at their destination when the GVT change is initiated. After all, the GVT is only the minimum of the LVTs, therefore no critical message relating to

the published GVT can be present on the network.

When implemented, the theoretical solution worked, unfortunately the time ascertainment for a thread is far too inaccurate. The design already proposed an increment of 1 μs for the VT. Using the function *clock_gettime(...)* with the constand CLOCK_THREAD_CPUTIME_ID there are variations up to several ms. A set of theoretical reasons could be responsible for the divergence. Modern CPUs equipped with multi-cores could be the first one. If there are slight variations in the operating frequencies of the separate cores, each one would have a different clock cycle time. Therefore, the same application would execute the same instructions in different durations. The same is true if using two separate unequally fast computers.

Furthermore, threads will be swapped for other tasks after a while. With a given accuracy of the OS-internal timer tick, which is in good system may a couple of μs, the determined runtime can only be a approximation. In the worst case, a thread will be resumed directly before or after a system tick and swapped just otherwise. Each swap can shift the stored runtime by almost a OS tick duration. Additional features like dynamic frequency scaling, also called CPU throttling, variate the clock cycle time even more. The extent of the variations is analysed in the section 6.1.

Due to the imprecision, the VT variates in dimensions, with undetectable causality errors possibly overlapping several thousand instructions. The current approach definitely has to be redesigned to be accurate enough for deployment.

5.3 DES/PDES

The current version of the project implements a conservative PDES synchronisation. Unfortunately, the general strategy may lead to a deadlock situation. If the virtual *MPI_Recv(...)* function cannot find the matching message yet, waiting for it does not advance the LP in VT. The design of the simulator assumed that a message neither would have arrived in the past, so that the virtual receive time stamp is lower then the VT of the LP, nor that the message would arrive some time after the receive function was called. In this case, the virtual receive time stamp determines the time the function will be blocked and this becomes then the new VT of the LP. As long as messages from the virtual future are delivered, this principle work. However, this violate against the rule of not releasing unsafe messages. Only when using a lookahead algorithm, this rule can be bypassed for messages which are certain not to generate a causality error. Speaking in terms of MPIthe MPI_ANY_SOURCE tag cannot be used as addressing attribute.

Assume that the LP which declares the GVT, waits for a future message of any source. Even though there would be a lookahead, the LP would keep waiting. However, since the waiting does not advance his VT, the GVT stucks. Thus, no further message will be delivered. In real systems, the messages will not be detained and all processes advance continuously in time. Letting all LPs progress in VT while waiting cannot be done in the simulator, because this could alter the sequence of events. Allowing just the LP which holds the GVT to increase its VT cannot produce an error. The question is how to determine the size of the increments. Too small and the system will be massively delayed, too great and a causality error my could occur again. The inaccuracy of counting the CPU utilisation intensifies the problem. Even a well-balanced application and an increment of 0.2ms ends in massive stalls.

Be that as it may, moving the perspective of the project in a slightly different angle we can get rid of the PDES and see the system as a "emulation" of a computer network in which all messages are delivering regardless to the GVT. The MPI self tasks mostly care of the synchronisation. Only in a scenario like load-balancing applications which operate with MPI_ANY_SOURCE could end in faulty results, if the network is seen as homogeneous. The results can be compared more with one which would be gained from a heterogeneous. This new aspect initiates a redesign of the messages queue to operate as a FIFO. No sorting is necessary here because the order is given by the MPI attributes.

5.4 Message Queues

The datatype list was chosen for the implementation of the queues. Arrays like lists have their disadvantages, a dynamic behaviour seems to be of more benefit here. The different message occurrence of every application will be supported best by this approach.

The functions for allocating and freeing memory are again a likely source of segmentation faults. The messages are one basic component. They are continuously created and destroyed in various code section. Centralised operations will remove this source of error. Therefore one function, adopted to the queue class, should be used to create messages. It ensures that the user defines the type and size of the message. Five different types can be generated right now. When passing a invalid value, an empty message is returned by default.

- MTYPE_EMPTY
  All values except the amount of databytes have to be set by the user.
- MTYPE_QUEUE_TAIL

Messages created with this tag are designed to indicate the end of a queue. The virtual receive time stamp is set to max value of $2^{64}\mu$s, which should be never reached by the application. In a sorted list, this keeps the message always at the end, because all other entries have to have a smaller value and therefore be in front. Additionally, this fact is used in the queue implementation. The get function has to keep all unsafe messages in the queue. With a value which is never reached, the tail node will be never removed from the list, without additional check.

- MTYPE_LVT_CHANGED
  This message is used to forward the new LVT.

- MTYPE_VM_FINALISED
  This control message is send with a higher priority by setting the virtual time stamp to zero. By doing that, it will the next message which will be send by the communication, in a sorted list.

- MTYPE_DISABLE_COMM
  Also a high priority control message. Designed to shutwon the communication in case of an interrupt or at the end of the program. Without the communication, the simulator is not able to run, and therefore it implies also the termination of the VM and so the abort of the application.

A second function creates a deep copy of the message struct and returns a pointer to the new object. Finally, a third function is used to delete a message. Before the memory will be freed, tests like if the message is still connected to a queue can be done here.

The queue itself is realised as a double chained list. For the current use, the double chained list only slightly simplifies the removing of an element. A good example would be the planned broadcast message. This message will be not replicated, instead each LP will get a reference to the same object, which will be retained in the list. When the last LPs acknowledges the receiving of the message, it can be removed from the list immediately because it has pointers to the previous and the next node. However, the main reason for the doubled chained list was a concept for a lookahead implementation. It proposes to create a second list which links nodes of already existing queue together, see image 5.11.

The test implementation uncovered some exposed weak points in the design. Messages of a node are already distributed amongst the running VMs. Narrowing down the amount of messages to only VM-specific messages eliminates redundant operations. Also it was envisaged that the top of the queue should always be executable, if it is in the past of the GVT. This is not true. For example, if one

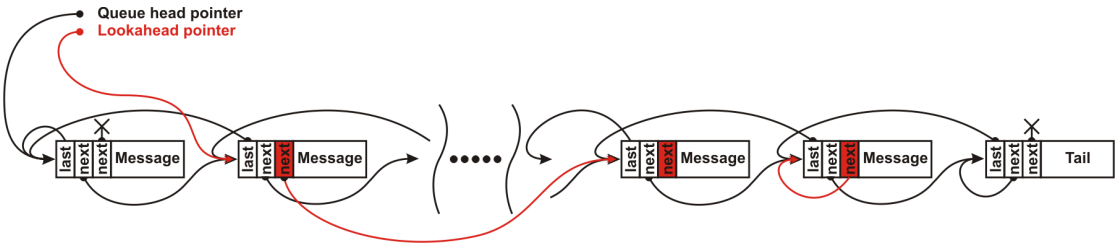

Figure 5.11: List with a second overlaying list

virtual rank is about to receive a sequence of messages from different sources, in a specific order. According to the concept until now, the queue only allows to extract the head message, the message with the youngest virtual time stamp. The chronological order of arrival does not have to match the desired order. If the head message is from source y instead of x, the next returned message cannot be processed. In fact, message after message has to be fetched until a matching one is found for either one of the LPs in the VM pool. Afterwards, all gathered elements have to be resubmitted to the queue.

Clearly this is an unacceptable implementation. Introducing the designed lookahead could improve this situation. With this implementation, if a head message is not executable, the lookahead points to one that is. In the short term this seems to keep the simulator running, but once a message is added to the list it cannot be marked as lookahead anymore. Consequently when a set of messages with the same LP as receptor is breed before the LP gets the turn to resume its work, only one is approved as lookahead candidate. Considering this, the implementation definitely needs a search ability.

When starting to search the queue, it would also making sense to continue the partitioning thoughts. Further separation on the LPs would simplify the delivery and improve the search time. Based on that, the original concept is expanded to a more MPI-like structure. Each LP is now equipped with a non-shared virtual MPI buffer. This buffer is a specialisation of the standard queue searchable for a union of MPI attributes. Apart from that, each VM gets a LP comprehensive MPI buffer for collective communication. Otherwise, a replication would be memory-intensive.

However, the new perspective that the PDES could be skipped in certain situations allowes to develop a second queue protocol. It simply forwards the messages in a *First In First Out* (FIFO) manner. The queue gets a pointer to the head and tail message. Appending a message can now be done directly, without passing

trough the whole list, since there is no need to comply with a certain order. This significantly lowers the queue overhead.

The current version of the project implements the queues as displayed in the image 5.12. All functionality of the sorted queue and the FIFO queue is defined in the abstract superclass c_QUEUE. A define controls the instantiation type of the object and therefore the mode of operation. The described MPI buffer queue is inherited from the FIFO queue type. It is possible to do so even with GVT, through the cascading sharing of the messages. If the GVT is active, the VM input queue will only release safe messages. To be save the message has to have a younger virtual time stamp then the GVT. Therefore all messages have to already be in the queue. So they are already sorted upon the GVT by their receiving virtual time stamp when extracting them from the queue. Adding them in that sequence to the FIFO based virtual MPI buffer they stay in the right order. Maintaining the time order can thus be guaranteed without a sorted queue with more overhead.

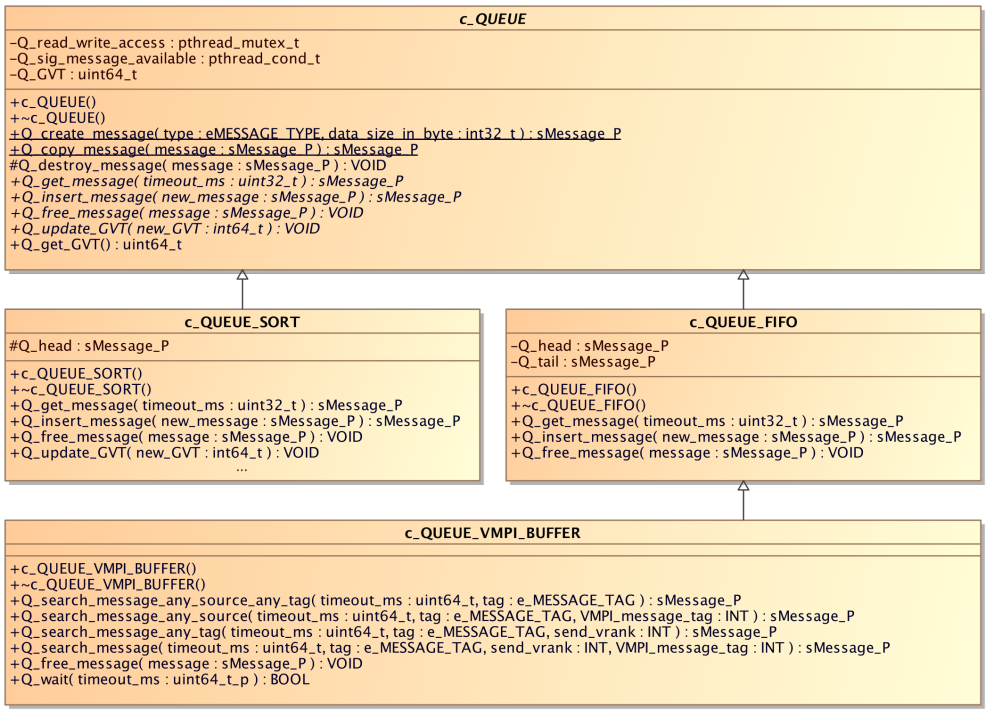

Figure 5.12: Implemented queue structure

5.5 Communication

The message flow correlates with the application to be simulated/emulated. It is uncertain which of the MPI ranks have to interact and when they have to do so. So the implementation needs to be flexible. Various communication structures are imaginable. To start with, the nodes could be connected as a ring. All messages will run in the cycle until they are at their destination, comparable to an endless object stream, where every MPI rank can add and remove items. Here every MPI rank has only two fixed interface partners. One as origin of the stream and the other as the one the stream has to be forwarded to. However, this simple concept comes with its downsides. Albeit it seems that the stream is on the network, most parts are actually stored in the MPI process' buffer. A high message exchange between the LPs could lead to a buffer overflow. Further messages may need more than one hop, which is network overhead. As the overhead increases with the growth of the simulators communication size, this concept will not be used for the implementation.

Also a star-like configuration, where all messages are being sent to one specific MPI rank from where they are distributed, would be conceivable. Regardless of the scale, a message has to do at most two hops. Depending on the implementation of the PDES synchronisation, having a centralised event queue could be beneficial. For example when updating a centralised queue in case of a rollback instead of coordinating the update on a distributed one. However, drawing the communication onto one node very likely becomes a major bottleneck that could be avoided by using other configurations.

Finally, a non-blocking all-to-all communication concept is used. The communication thread runs here in an endless loop. Each loop cycle can be used either to initiate a receive *MPI_Irecv(...)* or a send *MPI_Isend(...)*. The result of the function *MPI_Iprobe(...)*, checking if there are messages at hand, is used to branch between the two tasks. As long as the check is positive, the message will be fetched. By that criterion, receiving messages gets a higher priority than sending messages. This should theoretically reduce stress on the network. However, no further study has been done to verify this assumption.

Most of the data exchange consists of virtual MPI messages between the spread VM objects on the MPI ranks. There are still some control mechanisms like the LVT, GVT, etc., where no virtual MPI message header should be send. They will be handled directly and the additional information of the header is not necessary. In total the simulator operates with five different kinds of messages. Distinguishing between the different types is done by a message tag. All backed up tags are collected in an enumeration to increase the readability of the code and also to

reduce programming errors.

- MPI_VMPI_MESSAGE
  This tag indicates a virtual MPI message which is sent from one VM to another. After reconstructing the message, it is routed to one of the virtual machine input queues.

- MPI_CHANGE_LVT_SYNC
  To enable the root rank to keep track of all the LVTs and therefore the GVT of the system, the separate ranks only have to send their new LVT value. The associated source rank can be obtained from the MPI message itself.

- MPI_UPDATE_GVT
  The root rank sends a message with the new value to each node in case of a change. Unfortunately, this cannot be done by *MPI_ Bcast(...)*, because of the non-blocking strategy. Right now a message is sent in sequence to all. A more optimised variant would be to send it in a tree structure.

- MPI_RANK_FINALISED
  When a VM terminates, it sends a final message to inform the specific MPI rank. Each time the communication object receives such a message it counts the on the MPI rank running VMs down. After all VMs have been shut down, the system is informed by a MPI_RANK_FINALISED message that on the MPI rank no active LP will be processed anymore.

- MPI_SHUTDOWN_COMM
  On the basis of the MPI_RANK_FINALISED packages, the root rank monitors if there are still LPs active. Finally, after all nodes have reported their LP chunks' termination, the root nodes send the final message with the MPI_SHUTDOWN_COMM tag. Receiving this message, the communication objects break out of their loop and allow the simulator to exit.

Although the data is send with non-blocking methods, by waiting of the completion of the transmission with *MPI_ Test()* the messages are sent sequentially. Sending multiple messages simultaneously by using of the non-blocking MPI methods could improve the throughput, but this could also complicate the GVT synchronisation. So the first implementation is a more simple variant in which the messages are sent consecutively. Also this feature is used by the GVT synchronisation mechanism. However, the polling characteristic can potentially cost computational CPU time. Therefore, the thread is forced to suspend and sleep for the time span defined in sim_comm.h SUSPEND_TIME_US, if there are no incoming messages in the MPI buffer and if a send is not yet completed or if no messages are in the outgoing queue.

6 Testing

6.1 Time Measurement

An accuracy measurement of two time functions, the function *clock_gettime(...)* defined in time.h and the function *sg_get_cpu_stats_diff()* from the statgrab library, was initiated by the implementation of the VT. The GVT synchronisation mechanism which is based on VT leads to massive simulation stall. Variations in the VT accounting for the separate LPs are responsible for the high frequency of GVT synchronisations. An alternative approach to count the clock cycles by the function *clock()*, also provided by time.h, has been considered. However, the returned datatype is in general only 32 bit long. With a count range of 10^9 and modern CPUs where the clock frequency is already in the GHz range, an overflow will occur almost every second. Because of that, the use of this function was ruled out immediately.

In three test scenarios, the accuracy of the functions shall be tested in different

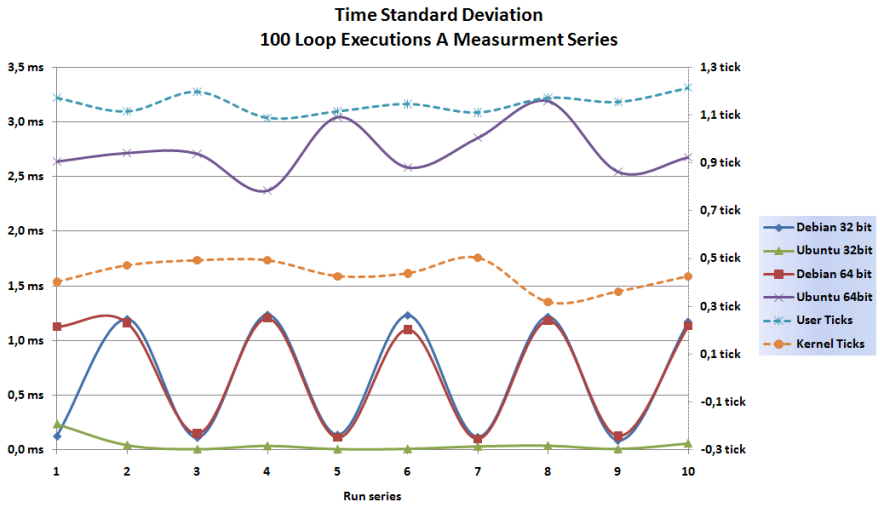

Figure 6.1: Experimental Standard Deviation 100 measurements a run

running environments. Regardless of the implementation, the test runs a loop, in

which dummy calculations are being done. This loop is the basis of the measurement. Measuring the execution time of the loop should theoretically always return the same duration.

The first scenario is a single process. The program collects a measurement series of

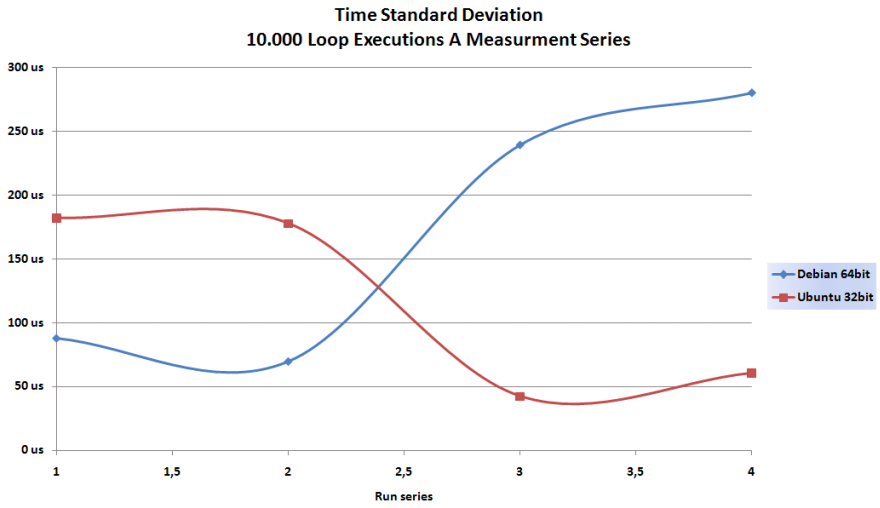

Figure 6.2: Experimental Standard Deviation 10 thousand measurements a run

multiple loop executions. Additional to the duration of the loop execution itself, the arithmetic mean of the durations, experimental standard deviation and the standard deviation of the mean is calculated. The second application is an MPI parallelised version of the first one. Multiple processes on separate compute nodes collect their data in the same manner. Afterwards, it will be gathered together and evaluated. Finally, the third version operates like the parallelised version but it uses threads instead of compute nodes.

Comparing the standard deviation results 6.1 of the single process runs shows that accuracy is related to the used OSThe test application was the only one running except for the basic OS tasks on the evaluated platform. So the periodical characteristics are caused by the system tasks. Best results are still far higher than the desired 1 μs. On the other side, the function *sg_get_cpu_stats_diff()* returns comparatively equal values for each measurement. Only one exemplary series of system and kernel ticks is shown in the same chart 6.1 as dotted lines. Noticeable is, that according to the gained values a system tick would be $\approx$5ms. Yet, if no more accurate solution can be found, counting in system ticks would be acceptable with regard to the variation of the measured values.

Repeating the test with a series performing 10.000 instead of 100 measurements 6.2 shows that the deviation converges to several μs. Anyhow, running the parallelised test 6.3 where one compute node does 10.000 measurements, has in all a deviation of a couple of ms. Finally the threaded version shows 6.4 also values up to a couple of ms. In conclusion, a VT clock has be at most of the order of several ms in order to avoid meanderings when counting the time in a wall clock manner.

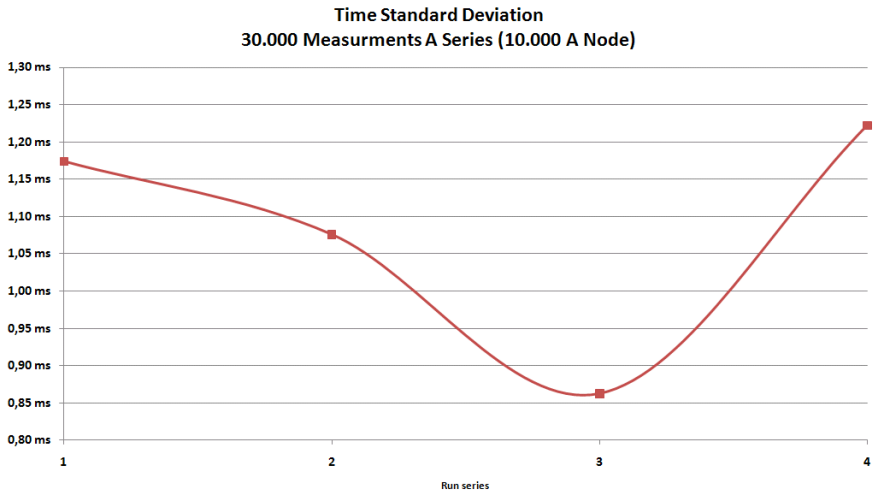

Figure 6.3: Experimental Standard Deviation 30 thousand measurements a run

6.2 Simulator

Two simple MPI applications have been written in order to prove the operability of the simulator. Either of them will be run with varying communication sizes on a small cluster. For the simulation ten nodes of the XTORC cluster are used. Each node is equally equipped with the following hardware:

The small amount of memory - compared to state-of-the-art systems - allows only a restricted pool of LPs on each node.

6.2.1 Application: Heat Transfer

The application simulates the heat distribution on a two-dimensional area over time. The size of the surface is specified as float values in the dimension X and Y. Each temporal iteration step forces a synchronisation between the nodes in the

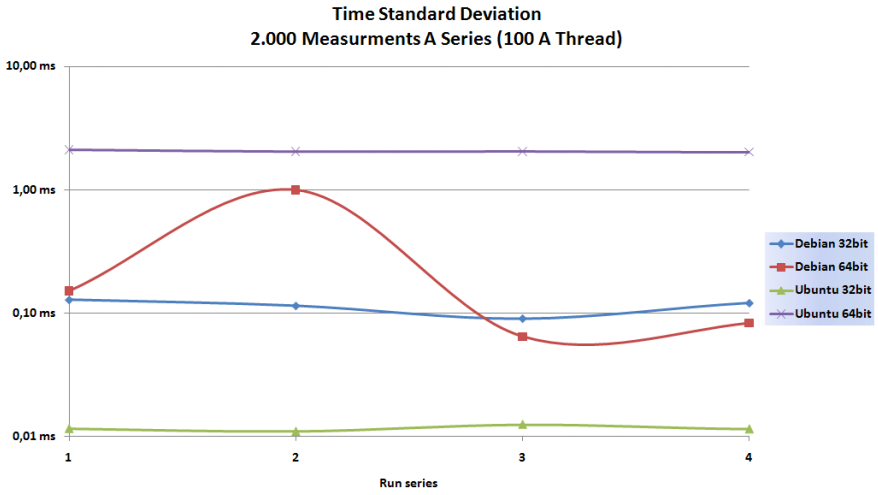

Figure 6.4: Experimental Standard Deviation 2 thousand measurements a run

Processor Type	Intel P4
Processor Architecture	32-bit
Cores per Processor	1
Clock Frequency	2.0GHz
Memory	0.75GB

Table 6.1: Test Platforms

communicator. The number of messages exchanged grows with the communication size and the number of iteration steps. Therefore this is a good application to get a stress test of the routing of the virtual MPI layer. Furthermore, the system does not only have to cope with a high amount of messages, but it is also forced to swap each LP content once in every iteration step.

Initially the area is distributed amongst the MPI ranks, see example with communication size of 100 nodes Image 6.6. A constant temperature source marked red is attached to each edge of the area. The temperature values have to be passed as parameters to the program. Except for these edge elements, the rest of the array is initialised to zero. The heat distribution is now calculated based on a simplified formula.

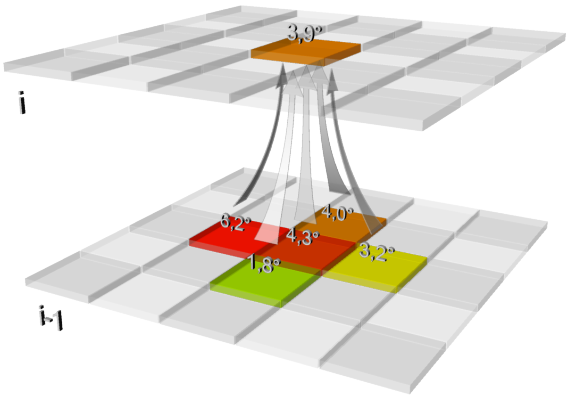

Figure 6.5: Heat Distribution

$$T_i = \frac{T_{i-1} + T_{i-1}^{[up]} + T_{i-1}^{[down]} + T_{i-1}^{[left]} + T_{i-1}^{[right]}}{5} \tag{6.1}$$

Where:

- T_i is the temperature of a point at the new time-step i;
- T_{i-1} is the temperature of the point at time-step i - 1;
- $T_{i-1}^{[up]}$, $T_{i-1}^{[down]}$, $T_{i-1}^{[left]}$, $T_{i-1}^{[right]}$ are the temperatures at the previous time-step, i - 1, of the upper, lower, left and right neighbouring points;

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Figure 6.6: Distribution area amongst ranks

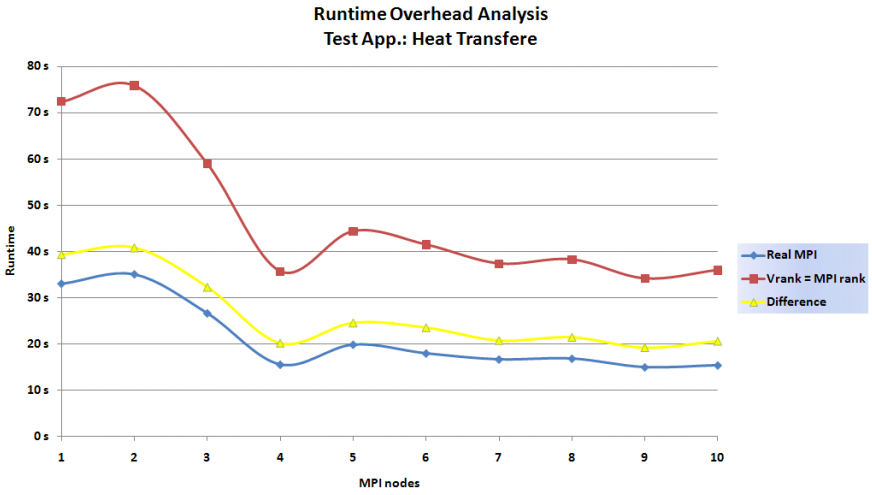

Figure 6.7: Runtime comparison where p = LP

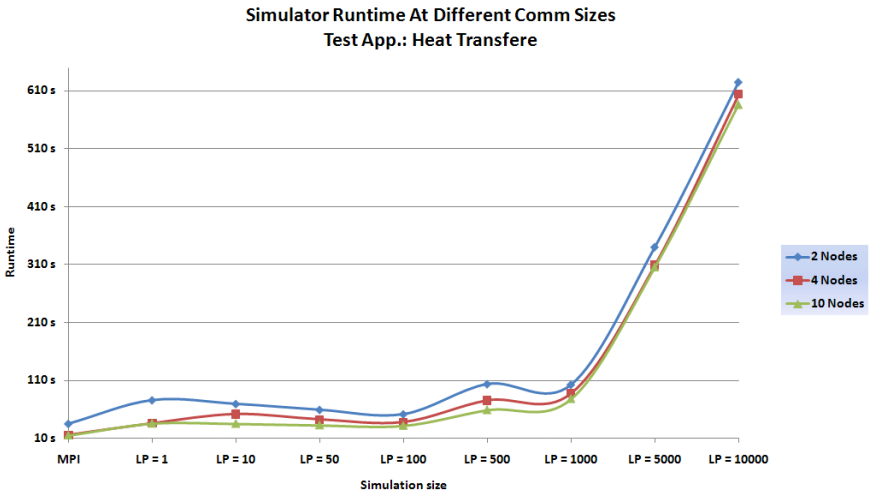

Figure 6.8: Virtual Scale Analysis

The output of the application is controlled by an optional filename parameter. If passed, the results will be written into a binary file. Otherwise the results will be printed on the screen, which is useful for small arrays of about 10 by 10 elements for a visual verification of the program output.

- Program usage:
  mpirun␣-np␣"p"␣./heat␣"it"␣"dim_x"␣"dim_y"␣"top"␣"right"␣"bottom"␣"left"␣[filename]

- p = Amount of MPI processes
- it = Calculation iterations
- dim_x = Array elements (float values) dimension x
- dim_y = Array elements (float values) dimension y
- top = Constant floating point value for arrays top row
- right = Constant floating point value for arrays right column
- bottom = Constant floating point value for arrays bottom row
- left = Constant floating point value for arrays left column
- filename = The filename in which the results will be written in binary form. Caution: File will be overwritten.

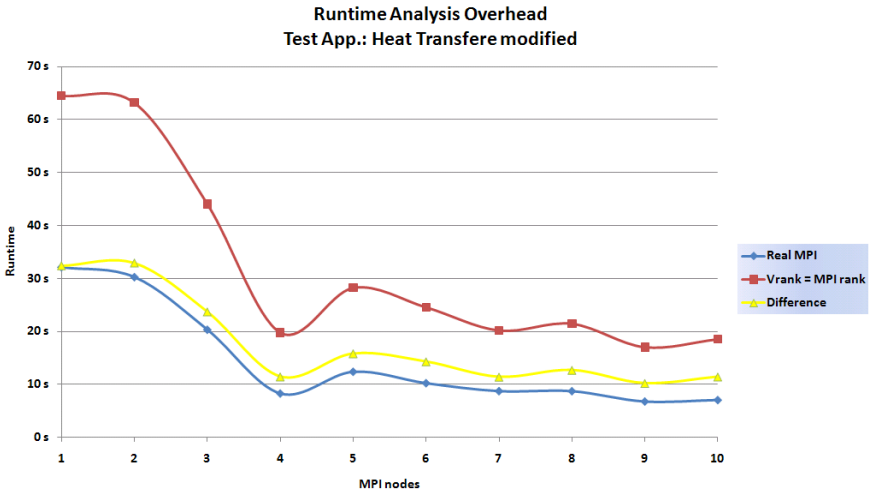

Figure 6.9: Runtime comparison where p = LP With Modified Application

The Image 6.7 shows a direct comparison between running the application in an MPI environment and running it hooked up to the simulator. To investigate the simulator's overhead at a small scale, each simulator node only hosts one LP. The yellow line represents the observed overhead. It stays at around 120% of the MPI run. Clearly the simulator is not meant to run that way. That is why the next series 6.8 analyses the runtime when increasing the LP pool on a fixed amount of nodes. Surprisingly, the runtime rapidly increases at a virtual communication size of 1,000 LPs. The reason for this behaviour is found in the

application that has been simulated. With the simulator supporting only a part of the MPI functionality, the results are gathered sequentially from each virtual rank. In addition a barrier was created, where all LPs wait for a message from the root node. As inefficient as this is already in MPI in combination with the stacked scheduling of the LPs, the run is delayed even more. As long as the next message is not in the root's virtual MPI buffer, LPs of the VM object are tried to be resumed. Considering a pool size of over 1000, resuming and searching the active LP's MPI buffer, requires significant computational time.

To confirm this analysis, the application has been modified. The gathering of the

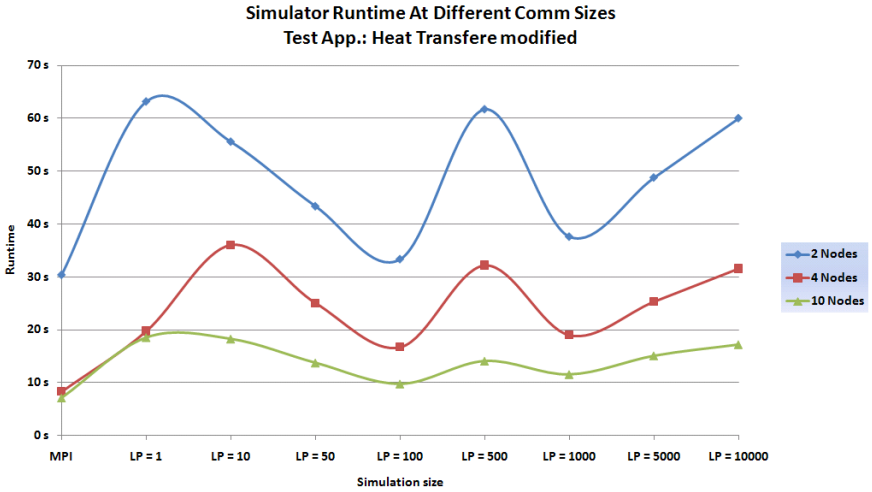

Figure 6.10: Virtual Scale Analysis With Modified Application

result data and the barrier have been removed. Though the communication after each iteration will still generate message flow, this time the small scale overhead test 6.9 shows an increasing discrepancy between the MPI and the simulation run, starting with about 100% growing to 130%. This is a more expected behaviour when all messages have to be routed through the virtual MPI layer.

The virtual scale test also presents a more anticipated result. The execution time stays more constant especially for a pool size of 1,000 LPs. Here the running time is drastically reduced. However, in both runs we get a substantial peak at a virtual communication size from about 500 nodes. Inspecting the real MPI run, it shows a slight peak when running on five nodes, too. Therefore, this should be a characteristic of the test application. This is proven with the second test application in the Section 6.2.2, where such a pool size shows no special characteristic.

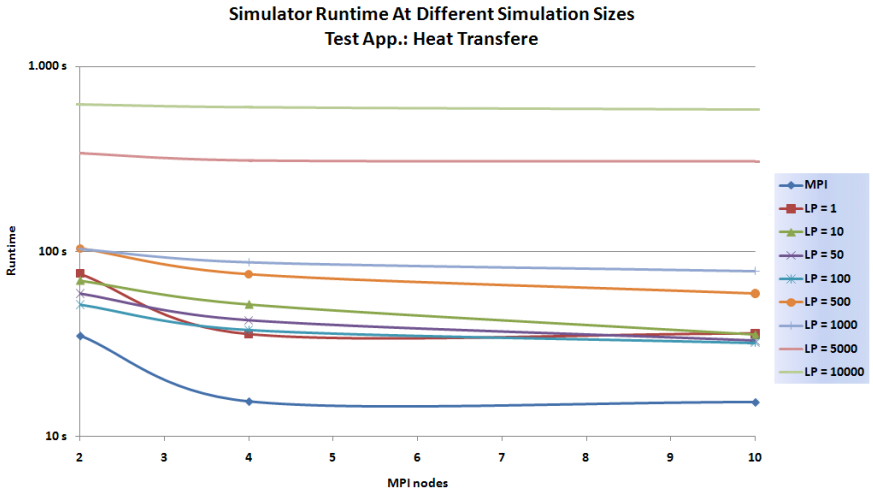

Figure 6.11: Efficiency Of Different Pool Sizes

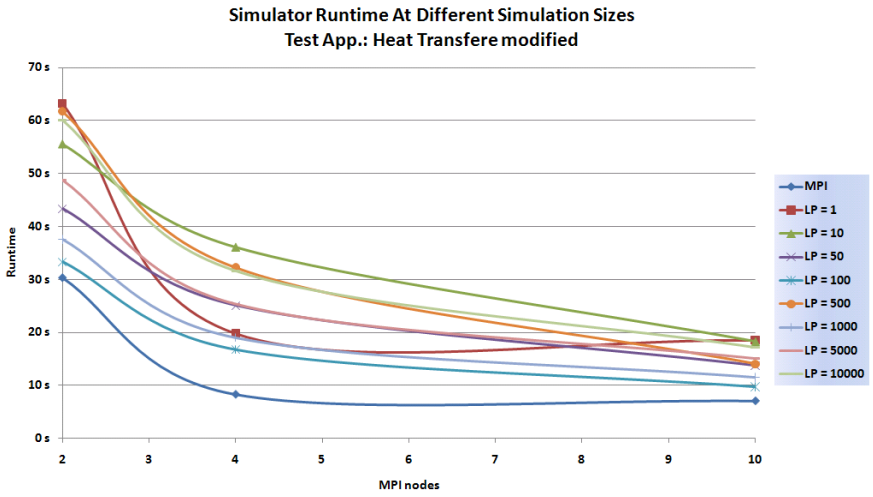

Figure 6.12: Efficiency Of Different Pool Sizes With Modified Application

Setting the results in another context shows that a higher virtual communication size, see Image 6.12 can be processed faster then a lower one. Emulating 50 or 100 LPs is more efficient, because the message size between the ranks is reduced. One feature of the simulator is that not all messages will be sent over the network. Only if the receiving virtual rank is located on a different MPI rank, the data is sent over the network. Otherwise the message is simple redirected to the corresponding

message queue. By reducing the message size, with the increased LP pool, less data is actually sent over the network. Without the collecting, in the modified version, this effect can be seen even more clearly in the chart 6.12.

After all, a valid application output has to be verified. Therefore, the binary output file from each run is compared with one generated by a real MPI execution. Since all files are identical to the reference, a correct program flow is assumed.

6.2.2 Application: Numerical Quadrature

The second application calculates the value of a definite integral. In order to solve the integral, we can divide it into n subintervals. These subintervals can be calculated - approximated as trapeziums - by the formula:

$$A_i = \frac{h_i \cdot (f(x_i) + f(x_{i+1}))}{2} \tag{6.2}$$

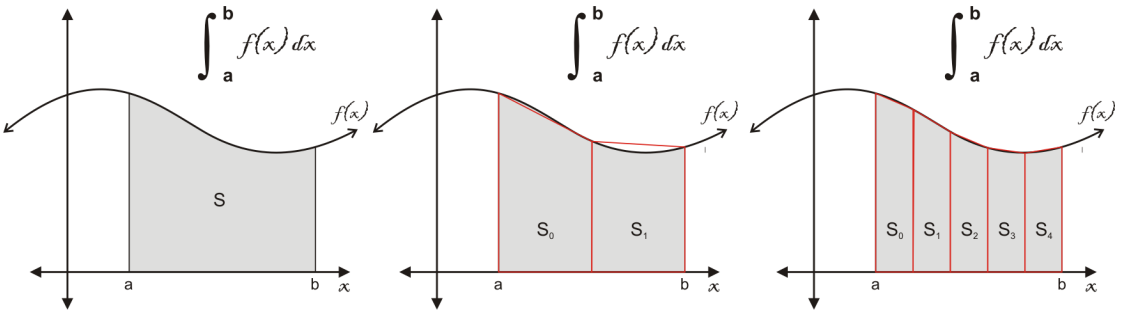

Figure 6.13: Integral

Summing up the values calculated for the separate subintervals gives an approximation for the solution of the original integral. The only communication in a parallelised version is the gathering of the subintervals. For this, each LP will be suspended and resumed while collecting the data. The simulator overhead should be low without frequent communication.

- Program usage:
  mpirun␣-np␣"p"␣./num_quad␣"input_file"␣"a"␣"b"␣"i"
- p = Amount of MPI processes
- input_file = This file contains the coefficient of the equation.
  File format: First line the number of coefficients as integer and in the second line

the coefficients, separated by blanks.
Example:
10
9␣4␣8␣8␣10␣2␣4␣8␣3␣6

- a = Integral lower bound
- b = Integral upper bound
- i = Subintervals on each MPI rank

The complexity of the problem of the program "numerical quadrature" is not related to the communication size. The defined subintervals are simply distributed amongst the compute nodes. Considering that the only communication and therefore the only content switch between LPs occurs when gathering the data, the simulator overhead should only increase slowly. However, this is only true up to a simulation size of about 5,000. Likewise to the heat distribution application, the results are gathered sequentially. As already known from Section 6.2.1, gathering the data in that way is even more inefficient in combination with the LP scheduling. This can be seen again in the chart 6.14.

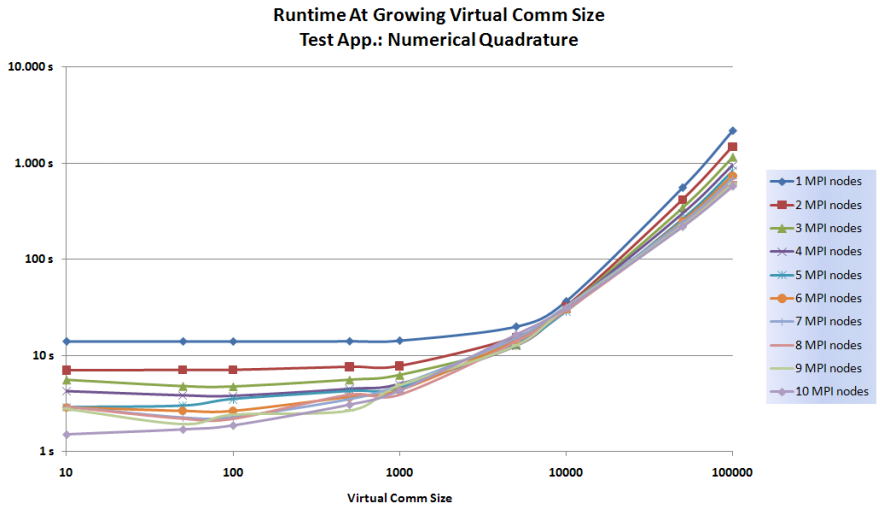

Figure 6.14: Runtime behaviour at growing virtual communication size.

By modifying the application in the same way as the heat transfer simulation, each LP can run with only a single content switch. With this modification, up to 10,000 nodes the runtime does not increase significantly while increasing the communication size. This can be seen in the Image 6.15. The complexity of the problem is

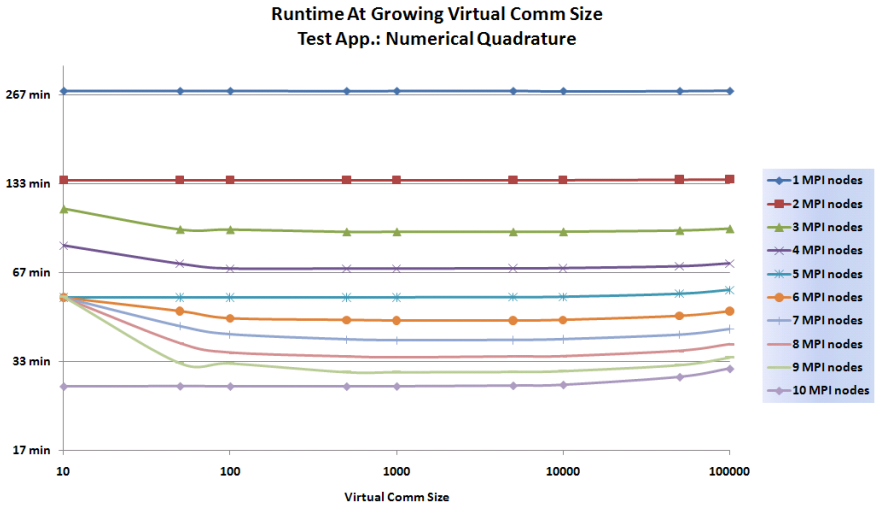

Figure 6.15: Runtime behavior at growing virtual communication size.

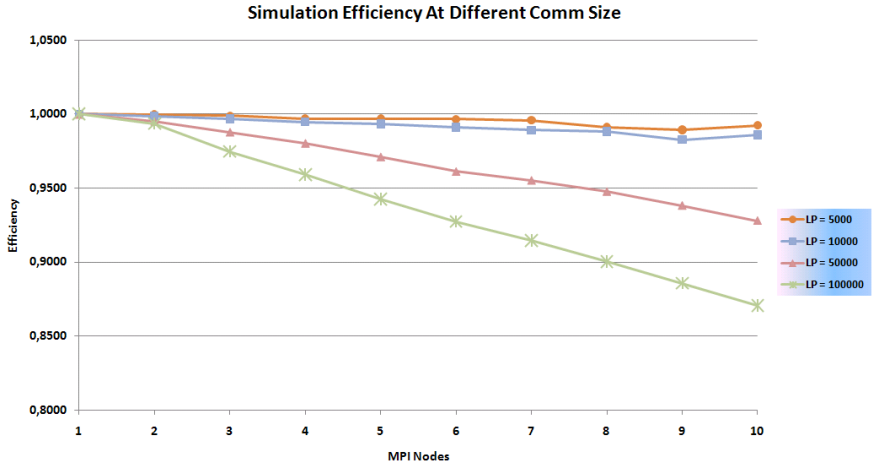

Figure 6.16: Efficiency when scaling the simulator.

increased in this test, which is why the runtime is already several minutes.

The very low memory footprint of the application already allows to simulate 100 thousand LPs on a single node in comparison to the five thousand of the heat distribution application. The simulator scaling efficiency chart 6.16 shows a constant drop down. At ten nodes, the efficiency compared to the run on one node has gone down to below 90% . Since the modified version does not communicate and

the GVT synchronisation is inactive, the parallelisation of the simulation should be almost 100% . The most likely source for the performance loss is the communication thread. In the design, the thread should suspend and transfer the CPU control while running idle. Because the first test runs of the heat distribution application have been slowed down by this feature, it has been disabed. Thus, the communication threads runs in a polling mode. In the end, the more MPI nodes are used, the more communication threads are wasting computational time. A proper suspend time which is adjusted to the application to be simulated has to be found, so that the message flow constraint between the separate MPI nodes stays low.

7 Conclusions

The final status of the project is a runable version, supporting the minimum requirements defined in Section 3. Although a conservative PDES approach is implemented, due to large variations between the measured values of the CPU utilisation of the separate LPs it is extremely inefficient. Therefore, from a practical point of view, the project does not meet the minimal requirements. The dimension of the discrepancy is analysed in the section 6.1. Furthermore, a suggestion to improve the efficiency or to use another approach is made in section 7.1.2.

When analysing MPI applications, there is only one case in which the PDES synchronisation is actually needed. When the MPI_ANY_SOURCE tag is used it can happen that a messages are processed in the wrong order. By defining the source this cannot happen, even when the MPI_ANY_TAG constant is used. This is because the messages are neither handled in a FIFO or a sorted queue. If a GVT is used, the sorting algorithm of the queue arranges all messages by their receive VT stamp. Thus the relative receiving time stamp calculated from the LPs VT has to advance, too. Therefore, sending messages sequentially from one LP to a specified other one cannot cause a causality error. In a FIFO approach it is basically the same: the mode of operation does not allow a message that is generated later to be sent earlier.

While the project does not meet the specifications practically, the current version could be seen as an emulator. It emulates a dynamic heterogeneous computer network. As long as the MPI_ANY_SOURCE is not used, the program flow stays accurate as desired. When an application which uses this tag is hooked up to the simulator, it is like deploying the application on a heterogeneous computer network. Therefore the execution flow may change from run to run.

7.1 Future Prospects

7.1.1 Overcome Restriction: MPI Calls Only In *main(...)*

The current implementation is still restricted in that MPI methods can only be called from the main function. New ways to integrate a more complex application into the virtual_machine class have to be investigated.

One could imagine to generate an object file of the application, which will be parsed for local function calls. By modifying them to act like a class member call, the application can keep a connection to an object. Furthermore, the addresses of the MPI library calls are replaced by the VM wrapper functions. Hence the calls should be redirected to the matching object. It is very important to be able to distinguish between local and library calls in order to make this concept work.

Another idea would be to enhance the VM class with an array of references to the VM objects. One additional static VM object is then created by each process. This static object is actually the only instance using the array. It contains references to all other instances. Now all functions to be intercepted will be simply renamed by a macro to call the member functions of the static object. When now entering the wrapper functions, the array will be searched for the matching reference. Therefore either the array holds reference and the running worker thread id combined or the VM object provides a getter function for its worker thread id. However, the thread itself can determine its id by using the function *pthread_ self()*. Overwriting this pointer with the one from the array should move the scope to the corresponding object. Further function flow will therefore only manipulate the correct variables. The switch itself is a single operation and searching the array could be reduced to $\Theta(\log(n))$ if the elements were sorted by the thread id. So this could be an efficient solution.

7.1.2 Implementing The Virtual Time

The test results from the section 6.1 show clearly, that counting the VT in a wall clock manner is not a feasible solution. However, when the VT strategy is retained, the system has to be prevented from massive stalling due to the resulting variation in the VT. One solution could be to keep track of the two lowest LVTs. The lowest is still referred to as the GVT. The second one is used to enhance the search for the LP whose VT is equal to the GVT. Theoretically, more than one LP could be at the GVT, anyhow with the way of recording the time it is very unlikely. Nevertheless this is a weak point of the strategy. Assuming that the LP is the only one, this one LP can be allowed to be looked up a second time, since it is the only one that can emit any events in the duration between the two times. Fetching messages from its future cannot cause a causality error. In case the result is still negative, the VT of the LP will be set to the second LVT + network delay + one. If network delay + one would not be added, we would have again two LPs on the GVT. In the worst case all LPs are blocked and the GVT increases in a step size of network delay plus one, which can again stall the simulator. However, it should improve the performance.

More suitable for the simulator could be a clock based on the events/messages

[33]. The enormous amount of memory needed is a major disadvantage of such an approach. The message header would have to be extended by an integer array as large as the size of the simulation. Using a 32-bit datatype would generate at a simulation size of 10^7 an additional message overhead of almost 40MBytes. Even applications with a small message flow would have a great need for additional memory. Despite this, a logical clock would be one way to know for sure the sequence of the message creation.

7.1.3 Scheduling Policy

Without a scheduling policy, the simulator may waste a great amount of computational time. A good example for this is the collecting algorithm of the test application heat distribution. As soon as the root node has to wait for a message, all other LPs on the node get their chance to resume, even though they are waiting for a message from the root. In that case a fair-share scheduling algorithm is inefficient. To privilege the LPs which have received a message last can improve the performance. Whatever strategy is used, it is as well as the PDES synchronisation, strongly dependent on the application to be simulated. Therefore it should be possible to switch between different algorithms.

7.1.4 Migration Of An LP To Another Node

The current mechanism to calculate the rank is based on a fixed distribution. The migration of an LP from one MPI rank to another is not provided. However, if requested this could be implemented with a small overhead for the system, as long as only a small part of the simulated LPs are moved. The transfer of the content can be done actually in a very simple way, since every content is stored in the LPs status variables. They can be serialised and sent as a message to another VM object. The more complex part is the address distribution. By design instead of using lookup tables the address is calculated. By that approach, the complexity of this operation stays fix while scaling the simulation size. Migrated LPs would be assigned new addresses although they should be addressable by their initial virtual mpi rank. Adding a lookup table for such LPs could be used to overwrite the calculated routing information. To do so migrated objects have to be appended to the initial object pool. Furthermore, the source object will be kept and marked as moved in the array in order to avoid an unnecessary update of the routing information of the following LPs. So the virtual rank will still be calculated, but if a migration entry exists it will overwrite the data and send the message to an additional LP in a VM object.

7.1.5 Enhanced MPI And System Call Instruction Set

The MPI instruction set - which is still rudimentary - is suitable for a proof of concept, but it can only support a narrow range of applications. The group communication would obtain the highest priority. Also, without wrapper functions for system calls, especially memory management, an optimistic PDES approach is impossible. Restoring an LP's stack content without restoring the allocated memory areas, can lead to wrong results, memory holes or segmentation faults. In the best case, it is only false results. However, if memory was allocated in during a rollback, resuming the application would again allocate the memory without freeing the previous one. A segmentation fault can also occur, even if memory was already freed before a rollback. Resuming the application generates at least an error when freeing the same address again.

In addition all I/O functionality has to be considered when wrapper functions are required to guarantee a correct application execution.

7.1.6 Optimistic PDES Approach

Optimistic PDES synchronisations have to store the status variables of the LPs periodically. These variables include current status information, the stack and all allocated memory sections. Therefore the simulator has to keep track which LP occupies memory where and of what size. After a defined number of events, all this data will be serialised and stored as a checkpoint. Furthermore, the bottleneck in memory that is often mentioned would require the simulator to swap these checkpoints to a mass storage device. A prediction which sorts the checkpoints according to their probability to be restored again, could be used to reduce read and write accesses to the mass storage. Depending on the implementation of the PDES, also a history of the processed messages older then the GVT has to be stored. Finally, the system needs a causality error detection.

7.1.7 Fault injection

Simulating applications correctly is only the first step of the project. Fault injection mechanisms shall allow to investigate the behaviour of the programs in case of errors. Therefore the simulator needs an interface to intercept the messages. Based on failure distributions the manipulation of messages should be possible, manually or at random.

7.2 Known Issues

7.2.1 Memory (Segmentation Fault)

Segmentation faults are still a big issue in the simulator. A known reason is that often the stack size assigned to the LPs is too small. The stack memory for all LPs is allocated in one sequential chunk. If a stack is too small, an LP can exceed into the space of the next one. In that case stack frames might get messed up. This leads at best to an invalid output, but more likely to a segmentation fault when unwinding a stack frame and returning to an invalid address in the program memory.

Without any knowledge of the application, a minimal size estimation cannot be done. Integrating a stack overflow protection would require that each stack is extended by at least one guard page of memory. This page has to be checked at every content switch. However, when an overflow is detected, the application may have already compromised other components of the simulator. A content switch is only done in an MPI wrapper function, but the stack usage between two such calls can be high. So in the end the LP could already use far more memory than it was assigned. Since all threads share one address space, it is possible that the allocated stack memory is placed behind a simulator object. Writing in that memory causes now no segmentation fault since the memory belongs to the process. Although it can obstruct a regular shutdown.

7.2.2 Out of resources

The current version of the simulator does not detect when the system runs out of memory. In such a case the first step is that the system starts to swap the memory. As could be observed in the tests, this often leads to a situation where a node does not respond anymore and needs to be restarted. The right stack size and the maximum amount of LPs have to be found by try-and-error.

Again at this point memory allocation and free functions can be used to at least detect when the simulation's memory usage reaches a certain level, therefore memory usage has to be monitored. If a certain threshold level is reached, the simulator should still be able to initiate a regular shutdown without blocking the node.

7.2.3 Printf And Floating Point Values

Applications hooked up to the simulator are not able to print floating point value at CPUs with a 64-bit architecture. The test simulations have revealed that printing the datatypes float or double on a 64-bit platform generates a segmentation

fault. Running the same code on a 32-bit architecture terminated successfully. The source of this error has not yet been determined, but it only concerns the code section of the to been simulated application, not the simulator. More specifically the error occurs only when one of the LPs' stacks is active.

Missaligned variables or the passing of an invalid pointer can be ruled out as the cause of the error. When casting the floating point value to an int datatype, it prints the expected integer part. Therefore the pointer has to be valid. Printing the addresses showed that the float variables are aligned correctly with a step size of four Bytes. More surprising is that it is even impossible to print a constant which is passed directly to the *printf(...)* function.

Somehow manipulating the stack and base pointers in combination with the operating principles of the *printf(...)* function leads to an error on 64-bit systems.

List of Acronyms

ALU: *Arithmetic logic unit*
API: *Application Programming Interface*

BP: *basepointer*

CMGVT: *Continuously Monitored Global Virtual Time*
CPU: *Central Processing Unit*

DES: *Discrete Event Simulation*

FIFO: *First In First Out*
flop/s: *Floating point Operations Per Second*
FP: *framepointer*

GCC: *GNU Compiler Collection*
GPU: *Graphics Processing Unit*
GS: *Grain Sensitive*
GVT: *Global Virtual Time*

HPC: *High Performance Computing*

I/O: *Input/Output*

JCAS: *Java Cellular Architecture Simulator*

LLTF: *Lowest Local Timestamp First*
LP: *Logical Process*
LTF: *Lowest Timestamp First*
LVT: *Local Virtual Time*

MIMD: *Multiple Instruction Multiple Data*
MISD: *Multiple Instruction Single Data*
MPI: *Message Passing Interface*

NetPIPE: *Network Protocol Independent Performance Evaluator*

OS: *Operating System*

PC: *Personal Computer*
PDES: *Parallel Discrete Event Simulation*
POSIX: *Portable Operating System Interface*
PVM: *Parallel Virtual Machnine*

SCL: *Scalable Computing Laboratory*
SF: *stackframe*
SIMD: *Single Instruction Multiple Data*
SISD: *Single Instruction Single Data*
SP: *stackpointer*

VM: *Virtual Machnine*
VM: *virtual machine*
VT: *Virtual Time*
VTS: *Virtual Time Stamp*

WVT: *Wide Virtual Time*

Glossary

$\mu\pi$	The name $\mu\pi$ is a Greek abbreviation for the English acronym MUPI for micro parallel performance investiagtor, [20, p. 3].
Bandwidth	The bandwidth defines the maximal amount of data in bit per seconds which can be transfered over a medium.
Core	A core is a combination of at least one ALU, memory and I/O interfaces. New models of processors combine several cores together.
Latency	The latency defines the time how long data takes to pass a component.
MPI	The message passing interface API allows a communication based on messages bewteen multiple computers.
Node	The definition of a node is depending on the system. Usually a node are the resources combined under one OS.
SPEEDUP	Defines how much faster a application runs.
State	State in term of a LP in this document referec to all data which belongs to a LP. This includes the stack, allocated memory in the heap and the simulators status variables.

Throttling	CPU throttling/processor throttling is the process when the CPU tries to avoid damage by overheating or reduces the power consumption. So if the temperature of the CPU exceeds some specified limits, the system will throttle down the CPU, allowing it to cool down and avoid damage. Also while a CPU utilisation is low the throttling to lower opperation frequency consumes less energy. This technique is used by many modern CPUs
TOP500	A list with the 500 most powerful computer systems, which is first assembled in 1993 and since maintained. Twice a jear computers ranked by their performance on the LINPACK Benchmark.

Bibliography

[1] How parallel processing works.
http://communication.howstuffworks.com/parallel-processing1.htm.

[2] John von neumann and von neumann architecture for computers (1945).
http://w3.salemstate.edu/ tevans/VonNeuma.htm.

[3] Netpipe.
http://www.scl.ameslab.gov/Projects/NetPIPE/NetPIPE.html.

[4] An overview of hpc and challenges for the future.
http://www.nchc.org.tw/en/research/index.php?RESEARCH_ID=9.

[5] Standard for information technology – portable operating system interface (posix). Electronic, December 2008. Base Specifications, Issue 7.

[6] Exascale expectations.
http://www.hpcwire.com/specialfeatures/sc09/features/Exascale-Expectations-70680617.html, 2009.

[7] Nael Abu-Ghazaleh. Optimized parallel discrete event simulation (pdes) for high performance computing (hpc) clusters. Technical report, AIR FORCE RESEARCH LABORATORY, August 2005.

[8] ACM International Conference on Computing Frontiers.
Scaling Time Warp-base Discrete Event Executiomn to 10^4 *Processors on a Blue Gene Supercomputer*, May 2007.

[9] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Conference 1967*, volume 30, pages 483–485, Atlantic City, NJ, USA, 1967.

[10] Koenraad Audenaert. Clock trees: Logical clocks for programs with nested parallelism. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 23(10):646–658, 1997.

[11] William Butera.
Programming a Paintable Computer.
PhD Thesis Proposal.

[12] Ewa Deelman and Boleslaw K. Szymanski. System knowledge acquisition in parallel discrete event simulation. In *1997 IEEE International Conference on Systems, Man and Cybernetics, Smc*, pages 2296–2301. Institute of Electrical & Electronics Enginee, 1997.

[13] Christian Engelmann. JCAS - IAA simulation efforts at Oak Ridge National Laboratory. Invited talk at the IAA Workshop on HPC Architectural Simulation (HPCAS), Boulder, CO, USA, September 1-2, 2009.

[14] Pierluigi Frisco. *Computing with Cells Advances in Membrane Computing*. Oxford University Press Inc., 2009.

[15] Prof. Richard M. Fujimoto. PARALLEL DISCRETE EVENT SIMULATION. *Commun. ACM*, 33(10):30–53, 1990.

[16] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[17] Helmut Herold. *Linux/Unix-Kurzreferenz*. Addison-Wesley, Martin-Kollar-Straße 10-12, D-819829 München/Germany, 3., aktualisierte auflage edition, November 2006.

[18] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[19] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Champaign, IL, USA, 1993.

[20] Ph.D. Kalyan S. Perumalla. *$\mu\pi$ A SCALABLE AND EFFICIENT PERFORMANCE INVESTIGATION SIMULATOR FOR PARALLEL APPLICATIONS*. Oak Ridge National Laboratory, draft edition, February 2009.

[21] Christian Lindig and Norman Ramsey. Declarative composition of stack frames. In *Proc. of the 14th International Conference on Compiler Construction, number 2985 in Lecture Notes in Computer Science*, pages 298–312. Springer, 2004.

[22] J. Liu, D. Nicol, B. Premore, and A. Poplawski. Performance prediction of a parallel simulator. pages 156 –164, 1999.

[23] Scott Mcmaster and Atif Memon. Call stack coverage for gui test-suite reduction. In *In Proceedings of the 17 th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006*, pages 6–10. IEEE Computer Society, 2006.

[24] Zdzislaw Meglicki. *Quantum Computing without Magic*. MIT Press, 55 Hayward Street, Cambridge, MA 02142, 2008.

[25] E. Mollick. Establishing moore's law. *Annals of the History of Computing, IEEE*, 28(3):62 –75, july-sept. 2006.

[26] G.E. Moore. Cramming more components onto integrated circuits.

[27] Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA.
$\mu\pi$: A Scalable and Transparent System for Simulating MPI Programs, March 2010.

[28] K.S. Perumalla. μsik - a micro-kernel for parallel/distributed simulation systems. 2005.

[29] T. Phan and R. Bagrodia. Optimistic simulation of parallel message-passing applications.

[30] S. Prakash and R.L. Bagrodia. Mpi-sim: Using parallel simulation to evaluate mpi programs. volume 1, pages 467 –474 vol.1, dec 1998.

[31] Francesco Quaglia and Vittorio Cortellessa. Grain sensitive event scheduling in time warp parallel discrete event simulation. In *PADS '00: Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 173–180, Washington, DC, USA, 2000. IEEE Computer Society.

[32] Francesco Quaglia and Vittorio Cortellessa. On the processor scheduling problem in time warp synchronization. *ACM Trans. Model. Comput. Simul.*, 12(3):143–175, 2002.

[33] Abhishek Rawat, Rohit Grover, and Sachin Maheshwari. Clock synchronization, 1998.

[34] Bjarne Stroustrup. An overview of the c++ programming language, 1998.

[35] Nianle Su, Hongtao Hou, Feng Yang, Qun Li, and Weiping Wang. Optimistic parallel discrete event simulation based on multi-core platform and its performance analysis.

[36] Summer Computer Simulation Conference.
WIDE VIRTUAL TIME WITH APPLICATION TO PARALLEL DISCRETE EVENT SIMULATION AND THE HLA RTI., July 1998.

[37] University of Tennessee, Oak Ridge National Laboratory, University of Manchester.
An Overview Of High Performance Computing And Challenges For The Future, March 2009.

[38] Winter Simulation Conference.
SCALING AN OPTIMISTIC PARALLEL SIMULATION OF LAGE-SCALE INTERCONNECTION NETWORKS, 2005.

[39] Winter Simulation Conference.
PARALLEL AND DISTRIBUTED SIMULATION: TRADITIONAL TECHNIQUES AND RECENT ADVANCES, 2006.

[40] Workshop on Principles of Advanced and Distributed Simulation (PADS).
Seven-O'Clock: A New Distributed GVT Algorithm Using Network Atomic Operations, November 2004.

[41] G. Zheng, Gunavardhan Kakulapati, and L.V. Kale. Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. page 78, april 2004.

Appendices

A Test Programs

A.1 Test pthread inital stack usage

Compiled by: g++ -Wall -o output input.c -l pthread

```
1  #include  <stdio.h>
2  #include  <stdlib.h>
3  #include  <unistd.h>
4  #include  <errno.h>
5  #include  <pthread.h>
6  #include  <sys/time.h>
7  #include  <limits.h>
8
9  #define PTHREAD_OK 0
10 #define STACK_SIZE (2 * PTHREAD_STACK_MIN)
11
12 void *thread_function(void * arg)
13 {
14   int dummyvar;
15
16   int a;
17   int b;
18
19   printf("The stacks lower bound is: \t\t%X\n",
20     (unsigned int)arg);
21   printf("The stacks upper bound is: \t\t%X\n",
22     (unsigned int)((int)arg + STACK_SIZE));
23
24   // Check stack growth direction
25   if (&a > &b)
26   {
27     printf("At down growing stack \t\t\t  %d = %X Bytes used\n",
28       (((int)arg + STACK_SIZE) - ((unsigned int)&dummyvar)),
29       (((int)arg + STACK_SIZE) - ((unsigned int)&dummyvar)));
30   }
31   else
32   {
33     printf("At down growing stack \t\t\t  %d = %X Bytes used\n",
34       (((unsigned int)&dummyvar) - (int)arg),
35       (((unsigned int)&dummyvar) - (int)arg));
36   }
37
38
39
40   fflush(stdout);
41   return NULL;
42 }
43
44 int main(int agrc, char *argv[])
45 {
46   int return_value;
```

```
47    void * mem = valloc(STACK_SIZE);
48    pthread_attr_t attrib;
49    pthread_t tid;
50
51    // Initialise a attribute object for the VN thread.
52    if ((return_value = pthread_attr_init(&attrib)) != PTHREAD_OK)
53    {
54      printf("Error while creating attribute\n");
55    }
56
57    // Set up the application managed stack into the user space
58    if ((return_value = pthread_attr_setstack(&attrib, mem, 2 * PTHREAD_STACK_MIN))
          != PTHREAD_OK)
59    {
60      printf("Error while set stack in attribute\n");
61    }
62
63    // Create thread
64    if ((return_value = pthread_create(&tid, &attrib, thread_function, mem)) !=
          PTHREAD_OK)
65    {
66      printf("Error while creating the logical nodes worker thread.\nError code: %d\
            n", return_value);
67      fflush(stdout);
68    }
69
70    // Wait for thread to be finised
71    if (pthread_join (tid, NULL) != PTHREAD_OK)
72    {
73      printf("Error while joining the worker thread");
74      fflush(stdout);
75    }
76
77    return 0;
78  }
```

B Project

B.1 MV_main function call assembly code

```
1  VM_main(VM_argc, VM_logical_processes[VM_active_LP].LP_argv);
2    mov    0x8(%ebp),%eax
3    mov    0xcc(%eax),%ecx
4    mov    0x8(%ebp),%eax
5    mov    0x40(%eax),%eax
6    lea    0x0(,%eax,4),%edx
7    mov    %edx,%eax
8    shl    $0x5,%eax
9    sub    %edx,%eax
10   lea    (%ecx,%eax,1),%eax
11   mov    0x1c(%eax),%edx
12   mov    0x8(%ebp),%eax
13   mov    (%eax),%eax
14   mov    %edx,0x8(%esp)
15   mov    %eax,0x4(%esp)
16   mov    0x8(%ebp),%eax
17   mov    %eax,(%esp)
18   call   0x8056a9c <c_VIRTUAL_MACHINE::VM_main(int, char**)>
19
20 INT cVIRTUAL_MACHINE::VM_main(INT argc, CHAR_P argv[])
21   push   %ebp
22   mov    %esp,%ebp
23   sub    $0x18,%esp
```

B.2 assembly_code.h

```
1  #ifndef __ASSEMBLY_CODE_H__
2  #define __ASSEMBLY_CODE_H__
3
4    /*! \file assembly_code.h
5     *  \brief All used inline assembly code is defined in this file
6     *
7     * The designed contend switch between LPs are rely on inline assembly. To
          switch from one contend to another a
8     * threads stack is moved to a spesific section in memory allocated for a
          logical process. When the simulator
9     * shall be deployed on a new platform the defines my has to be modified.
10    */
11
12   /*! \def   SET_STACK_POINTER
13    *  \brief Stores the current stackpointer
14    *
15    *  The macro stroes the current stackpointer in a system wordsize width
          variable. The address of the variable has to passed to the macro.
16    *  Additionally before saving the stackpointer all register will be pushed onto
          the stack. This is because the implemented version of a
```

```
17      *  LP switch. By default C/C++ does save the used register but since a diffrent
            LP can be executed while a LP is suspended C/C++ compiler
18      *  can not pedict which regisers have to be saved
19      */
20     /*! \def   SET_BASE_POINTER
21      *  \brief Stores the current basepointer
22      *
23      *  The macro stroes the current basepointer in a system wordsize width variable
            . The address of the variable has to passed to the macro.
24      */
25     /*! \def   GET_STACK_POINTER
26      *  \brief Restores the current stackpointer
27      *
28      *  The macro restores a saved stackpointer from a system wordsize width
            variable. The address of the variable has to passed to the macro.
29      *  This is the counterpart macro to SET_STACK_POINTER, which pops the stored
            register again.
30      */
31     /*! \def   GET_BASE_POINTER
32      *  \brief Restores the current basepoiner
33      *  The macro restores a saved basepointer from a system wordsize width variable
            . The address of the variable has to passed to the macro.
34      */
35     // The following define and undefine block is only for doxygen which requieres
            active define so that thay will be added to the documentation
36     #define SET_STACK_POINTER
37     #define SET_BASE_POINTER
38     #define GET_STACK_POINTER
39     #define GET_BASE_POINTER
40     #undef SET_STACK_POINTER
41     #undef SET_BASE_POINTER
42     #undef GET_STACK_POINTER
43     #undef GET_BASE_POINTER
44
45     #ifdef INTEL32
46       #define SET_STACK_POINTER(new_pointer)
47         __asm( "mov %0, %%esp":  : "r"(new_pointer) :  )
48       #define SET_BASE_POINTER(new_pointer)
49         __asm( "mov %0, %%ebp":  : "r"(new_pointer) :  )
50     #endif
51
52     #ifdef AMD64
53       #define SET_STACK_POINTER(new_pointer)
54         __asm( "mov %0, %%rsp":  : "r"(new_pointer) :  )
55       #define SET_BASE_POINTER(new_pointer)
56         __asm( "mov %0, %%rbp":  : "r"(new_pointer) :  )
57     #endif
58
59
60     #ifdef INTEL32
61       #define GET_STACK_POINTER(store_pointer)
62         __asm( "mov %%esp, %0": "=r"(store_pointer) :  )
63       #define GET_BASE_POINTER(store_pointer)
64         __asm( "mov %%ebp, %0": "=r"(store_pointer) :  )
65     #endif
66
67     #ifdef AMD64
68       #define GET_STACK_POINTER(store_pointer)
69         __asm( "mov %%rsp, %0": "=r"(store_pointer) :  )
70       #define GET_BASE_POINTER(store_pointer)
71         __asm( "mov %%rbp, %0": "=r"(store_pointer) :  )
72     #endif
```

```
73
74
75 #endif //__ASSEMBLY_CODE_H__
```

B.3 datatypes.h

```
1  #ifndef __DATATYPES_H__
2  #define __DATATYPES_H__
3
4    /*! \file datatypes.h
5     *  \brief  Definitions for datatypes and includes are done here.
6     *
7     *  All non standard datatypes used in the simulator are defined here. Also
          since all files have to include this one
8     *  all includes are done centralised here
9     */
10
11   #include  <stdint.h>
12   #include  <stdio.h>
13   #include  <stdlib.h>
14   #include  <unistd.h>
15   #include  <pthread.h>
16   #include  <limits.h>
17   #include  <string.h>
18   #include  <sys/time.h>
19   #include  <algorithm>
20   #include  <mpi.h>
21   #include  <sched.h>
22   #include  <sys/user.h>
23   #include  "defines.h"
24
25 using namespace std;
26
27   #define PTHREAD_OK 0
28
29   #define MPI_ROOT_RANK 0
30
31   #define TRUE  1
32   #define FALSE 0
33
34   typedef void                    VOID,
35                                   *VOID_P;
36
37   typedef int                     INT,
38                                   *INT_P;
39
40   typedef int                     BOOL,
41                                   *BOOL_P;
42
43   typedef char                    CHAR,
44                                   *CHAR_P;
45
46   typedef int8_t                  *int8_t_p;
47   typedef int16_t                 *int16_t_p;
48   typedef int32_t                 *int32_t_p;
49   typedef int64_t                 *int64_t_p;
50
51   typedef uint8_t                 *uint8_t_p;
52   typedef uint16_t                *uint16_t_p;
53   typedef uint32_t                *uint32_t_p;
```

```
54    typedef uint64_t                *uint64_t_p;
55
56    typedef struct timespec         timespec_t,
57                                    *timespec_t_p;
58    typedef struct timeval          timeval_t,
59                                    *timeval_t_p;
60
61    #if __WORDSIZE == 64
62      typedef int64_t               SYS_INT,              // System architecture
            dependent integer uses always full system bit with (here 64bit)
63                                    *SYS_INT_P;
64
65      #define MPI_BOOL      MPI_CHAR
66      #define MPI_INT8_T    MPI_CHAR
67      #define MPI_INT16_T   MPI_SHORT                     //According to the stdint.h,
             where the int16_t is defined, a short    is 16bit on 32bit arch as well
            as 64bit arch
68      #define MPI_INT32_T   MPI_INT                       //According to the stdint.h,
             where the int32_t is defined, a int      is 32bit on 32bit arch as well
            as 64bit arch
69      #define MPI_INT64_T   MPI_LONG                      //According to the stdint.h,
             where the int64_t is defined, a long int is 64bit on 64bit arch
70      #define MPI_UINT8_T   MPI_UNSIGNED_CHAR
71      #define MPI_UINT16_T  MPI_UNSIGNED_SHORT
72      #define MPI_UINT32_T  MPI_UNSIGNED_INT
73      #define MPI_UINT64_T  MPI_UNSIGNED_LONG
74      #define MPI_SYS_INT   MPI_LONG_LONG
75    #else
76      typedef int32_t               SYS_INT,              // System architecture
            dependent integer uses always full system bit with (here 32bit)
77                                    *SYS_INT_P;
78
79      #define MPI_BOOL      MPI_CHAR
80      #define MPI_INT8_T    MPI_CHAR
81      #define MPI_INT16_T   MPI_SHORT                     //According to the stdint.h,
             where the int16_t is defined, a short    is 16bit on 32bit arch as well
            as 64bit arch
82      #define MPI_INT32_T   MPI_INT                       //According to the stdint.h,
             where the int32_t is defined, a int      is 32bit on 32bit arch as well
            as 64bit arch
83      #define MPI_INT64_T   MPI_LONG_LONG                 //According to the stdint.h,
             where the int64_t is defined, a long lont int  is 64bit on 32bit arch
84      #define MPI_UINT8_T   MPI_UNSIGNED_CHAR
85      #define MPI_UINT16_T  MPI_UNSIGNED_SHORT
86      #define MPI_UINT32_T  MPI_UNSIGNED_INT
87      #define MPI_UINT64_T  MPI_UNSIGNED_LONG_LONG
88      #define MPI_SYS_INT   MPI_INT
89    #endif
90
91    /*! \struct s_LP_RANK
92    *  \brief  s_LP_RANK, sLP_RANK, sLP_RANK_P: This struct splits the absolute
            virtual rank into components.
93    *
94    * This struct splits the virtual rank into tree parts. Through that a message
            can easy directed
95    * to another logical process. The split seperates mpi rank, the index for the
            recieve queue (the
96    * corresponding virtual machine) and the index in the logical process array in
            the virtual machine
97    * object itselfe.
98    *
99    * Additional the absolute virtual rank is stored for may future application
```

```
100     *
101     * List of values:
102     *   - MPI_rank:          uint16_t
103     *   - VM_rank:           uint16_t
104     *   - VRANK_index:       uint32_t
105     *   - abs_virtual_rank: INT
106     */
107     typedef struct s_LP_RANK
108     {
109       uint16_t  MPI_rank;           /*!< The mpi rank on which this logical process
              is running on.                    */
110       uint16_t  VM_rank;            /*!< The virtual machine index in which the
              logical process is running on the mpi rank. */
111       uint32_t  VRANK_index;        /*!< The lp index where the logical process data
               is stored in the threads LP array.     */
112
113       INT       abs_virtual_rank;   /*!< The original absolute virtual rank
                                                 */
114
115     } sLP_RANK,
116       *sLP_RANK_P;
117
118    /*! \def   CREATE_LP_RANK_MPI_DATATYPE
119      *  \brief Creates a MPI datatype for the struct s_LP_RANK
120      *
121      *  \warning To allow a more easy modifikation and also to improve the code
              readability a MPI datatype for the struct s_LP_RANK
122      *  is created. Important this macro has to be modified when ever the struct is
              changed.
123      */
124
125     #define CREATE_LP_RANK_MPI_DATATYPE(MPI_Datatype_pointer)\
126   do\
127   {\
128     INT blocks = 3;\
129     INT block_count[]                = {     2      ,     1      ,     1    };\
130     MPI_Aint block_displacement[]    = {     0      ,     4      ,     8    };\
131     MPI_Datatype block_datatypes[]   = {MPI_INT16_T, MPI_INT32_T, MPI_INT};\
132     MPI_Type_create_struct(    blocks,\
133                                block_count,\
134                                block_displacement,\
135                                block_datatypes,\
136                                MPI_Datatype_pointer);\
137     MPI_Type_commit(MPI_Datatype_pointer);\
138   } while(FALSE)
139
140
141   #endif //__DATATYPES_H__
```

B.4 Function VM_synchronise_LP()

```
1   VOID cVIRTUAL_MACHINE::VM_synchronise_LP()
2   {
3     sMPI_MESSAGE_P next_message = NULL;
4     BOOL break_sync = FALSE;
5
6
7     GET_STACK_POINTER (VM_logical_processes[VM_active_LP].LP_stack_pointer);
8     GET_BASE_POINTER  (VM_logical_processes[VM_active_LP].LP_base_pointer);
9
```

```
10    // Go back to in stack to the pointer there this function hast been called in
          the first placed.
11    SET_BASE_POINTER  (VM_logical_processes[VM_LP_Thread_index].LP_base_pointer);
12    SET_STACK_POINTER (VM_logical_processes[VM_LP_Thread_index].LP_stack_pointer);
13
14    // Redirect all message into the buffer of the seperate LP(s) or into the one
          bcast buffer
15    do
16    {
17      next_message = VM_receive_queue->Q_get_message(0);
18
19      if (next_message != NULL)
20      {
21        switch(next_message->Message_tag)
22        {
23          case MT_MPI_BARRIER:
24            {
25              break;
26            }
27          default:
28            {
29              PRINT_DEBUG(HIGHEST_DEBUG_LEVEL,  "Debug@cVIRTUAL_MACHINE:
                    VM_synchronise_LP: VM received message from \n\tvrank '%d': \tmpi
                     rank '%d', \tvirtual machine '%d', \tLP index '%d' to \n\tvrank
                    '%d': \tmpi rank '%d', \tvirtual machine '%d', \tLP index '%d'!
                    Data size '%d'\n", (next_message->Vrank_transmitt).
                    abs_virtual_rank, (next_message->Vrank_transmitt).MPI_rank, (
                    next_message->Vrank_transmitt).VM_rank, (next_message->
                    Vrank_transmitt).VRANK_index, (next_message->Vrank_receiver).
                    abs_virtual_rank, (next_message->Vrank_receiver).MPI_rank, (
                    next_message->Vrank_receiver).VM_rank, (next_message->
                    Vrank_receiver).VRANK_index, next_message->Message_data_length);
30              fflush(stdout);
31              if (next_message->Broad_cast_received > 0)
32              {
33                // A message which is supposed to be received by multiple LP(s)
34                VM_incomming_buffer.Q_insert_message(next_message);
35              }
36              else
37              {
38                // Direct adressed mesage (Point to Point) so appent message to the
                      LPs incommin buffer
39                (VM_logical_processes[(next_message->Vrank_receiver).VRANK_index].
                      LP_incomming_buffer).Q_insert_message(next_message);
40              }
41
42              break;
43            }
44        }
45      }
46      else
47      {
48        break;
49      }
50    } while (TRUE);
51
52
53    // Seach for the next LP which can be resumed
54    //TODO: Check theoretical event that no LP can be resumed.
55    // Through VM_active_LP-- the current active LP is checked first if it can be
          resumed
56    //VM_active_LP--;
```

```
57    while((VM_finished_LP < VM_LP_on_VM) && (break_sync != TRUE))
58    {
59      VM_active_LP = (VM_active_LP + 1) % VM_LP_on_VM;
60      PRINT_DEBUG(HIGHEST_DEBUG_LEVEL, "Debug@cVIRTUAL_MACHINE:VM_synchronise_LP:
            Try to set index %lld as aktive vrank\n", (int64_t)VM_active_LP);
61      // Act depending on the LPs status. (start main, do noting, resume)
62      switch (VM_logical_processes[VM_active_LP].LP_status)
63      {
64        case LP_STATUS_FINISHED:        // Do nothing LP already finished
65            break;
66        case LP_STATUS_BARRIER:         // Do nothing LP is waiting on a barrier. If
              all
67            break;
68        case LP_STATUS_NOT_STARTED:     // Run the main function
69            PRINT_DEBUG(HIGHEST_DEBUG_LEVEL, "Debug@cVIRTUAL_MACHINE:
                VM_synchronise_LP: Virtual Machine %lld starts vrank %lld\n", (
                int64_t)VM_rank, (int64_t)VM_comm_handler[0].VC_offset_relativ_rank
                + ((VM_per_mpi_rank * VM_active_LP) + VM_rank));
70
71            // Set the status of the LP to running
72            VM_logical_processes[VM_active_LP].LP_status = LP_STATUS_RUN;
73
74            VM_CPU_time_thread = VM_get_thread_CPU_time();
75            SET_STACK_POINTER (VM_logical_processes[VM_active_LP].LP_stack_pointer);
76
77            //TODO: Pass the real parameters to the main function
78            VM_main(VM_argc, VM_logical_processes[VM_active_LP].LP_argv);
79
80            SET_BASE_POINTER  (VM_logical_processes[VM_LP_Thread_index].
                  LP_base_pointer);
81            SET_STACK_POINTER (VM_logical_processes[VM_LP_Thread_index].
                  LP_stack_pointer);
82
83            // Set the status of the LP to running
84            VM_logical_processes[VM_active_LP].LP_status = LP_STATUS_FINISHED;
85
86            PRINT_DEBUG(HIGHEST_DEBUG_LEVEL, "Debug@cVIRTUAL_MACHINE:
                VM_synchronise_LP: Virtual Machine %lld vrank %lld has finished the
                main function!\n", (int64_t)VM_rank, (int64_t)VM_comm_handler[0].
                VC_offset_relativ_rank + ((VM_per_mpi_rank * VM_active_LP) + VM_rank
                ));
87
88            VM_finished_LP++;
89
90            break;
91        default:                        // Resume this LP
92            // Restore matching base and stack pointer.
93            SET_BASE_POINTER  (VM_logical_processes[VM_active_LP].LP_base_pointer);
94            SET_STACK_POINTER (VM_logical_processes[VM_active_LP].LP_stack_pointer);
95            // To break out of the for loop around the switch the index is set equal
                  to VM_LP_on_VM
96
97            break_sync = TRUE;
98            break;
99      }
100   }
101
102   return;
103 }
```

C

User Manual

for the simulator of advanced large HPC architectures

J

C.1 Requirements

The simulator is deployable on all x86 or AMD64 systems. To port it onto another architecture see C.5

The simulator requires the following libraries:

- OpenMPI <=1.2
- boost <=1.35
- pthread

C.2 Installation

The simulator does not need to be installed. The project has only to be unpacked, compiled and linked with an MPI application.

- Unpack: tar -xf simulator.tar.gz
- Compile and Link: see C.3

C.3 Usage

C.3.1 Simulation Restrictions

Due to the simulators architecture it is currently only possible to do MPI library calls inside the *main(...)* function.

Also the simulator cannot deal with global variables in the MPI application.

C.3.2 Supported MPI calls

- int MPI_Init (int *argc, char **argv[])
- int MPI_Finalize()
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
- int MPI_Comm_size(MPI_Comm comm, int *size)
- int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

C.3.3 Prepairing the application for simulation

Since the simulator comes with its own *main(...)* function the applications will be renamed by a macro. The macro is defined in the simulator.h header file. Thus the main function will become a member function of the simulators virtual machine class. The header file has to be included into the file where the *main(...)* function is implemented.

- #include "simulator.h"

Now the object files of the application have to be generated.

C.3.4 Compiling the simulator

The generated object file path(s) and maybe required linking options have to be passed to the simulator's make command by the variable TARGET.

- make TARGET="list target application object files and linking options"

The object files and the linking options have to be separated by a whitespace. The simulator's binary output file is named "sim".

C.3.5 Executing the simulator

To run a simulation, the simulator needs three parameters.

1. Simulation size
   The amount of virtual MPI ranks. This has to be a positive integer which is greater then used real MPI rank(s) times virtual machine object(s) a MPI rank.

2. Virtual machines objects on each MPI instance
   The amount of virtual machine objects running on each MPI rank. Each virtual machine object has a worker thread of its own which executes a chunk of the virtual MPI ranks. This is designed to utilise multi-core chips.

3. Stack size for each virtual MPI rank
   The stack size is defined as a multiple of a half system page size. This parameter has to be chosen carefully. A to small stack results very likely in a segmentation fault. Furthermore the possible simulation size is directly related to the stack size. The amount of virtual MPI ranks that can be simulated is basically only restricted by the amount of available memory.

The current version does not keep record of resources usage especial memory. So the right balance between stack and simulation size has to be determined by a trial and error.

- Usage:
  mpirun -np Sim_size ./sim V_size VM_objects Stack_size app app_param_list

Where:

- Sim_size:
  is the amount of real MPI ranks the simulator is distributing
- V_size:
  is the amount of virtual MPI ranks to be simulated (For details see above).
- VM_objects:
  the amount of virtual machine objects on each MPI instance of the simulator (For details see above).
- Stack_size:
  the stack size assigned to each virtual MPI rank (For details see above).
- app:
  the name of the of the binary to be simulated. The provided string will be passed as argv[0] to the simulated application.
- app_param_list:
  the parameters which have to be passed to the application which is to be simulated.

C.4 Example: Ring message

This test application sends a ring message starting at a specific node passed as parameter. The simulator.h is already included in the test application. To compile it as an MPI application this include has to be commented.
Usage: mpirun -np x ./ring start_rank

Hooking the application up to the simulator:

1. Creating the test application's object file.
   $ cd /Inst_Path/simulator/Test_Applications/Ring_Message
   $ mpic++ -Wall -c ring.c

2. Compiling and linking the simulator with the application.
   $ cd /Inst_Path/simulator
   $ make clean $ make TARGET="./Test_Applications/Ring_Message/ring.o"

3. Executing the simulation
   $ mpirun -np 2 ./sim 10 2 2 ./ring 3

4. Program output:
   MPI rank '4' received message from rank '3'
   MPI rank '5' received message from rank '4'
   MPI rank '6' received message from rank '5'
   MPI rank '7' received message from rank '6'
   MPI rank '8' received message from rank '7'
   MPI rank '9' received message from rank '8'
   MPI rank '0' received message from rank '9'
   MPI rank '1' received message from rank '0'
   MPI rank '2' received message from rank '1'
   MPI rank '3' received message from rank '2'
   The program runtime was 488252376ns = 488252us = 488ms = 0s

Note: The printed runtime is the runtime of the simulator. This includes the simulator's start up and finalise tasks. Determining the application's runtime is not supported in the current version.

```
1  #include <mpi.h>
2  #include "./../../simulator.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <string.h>
7
8  int main(int argc, char *argv[])
9  {
10   int rank;
11   int comm_size;
12   int received = -1;
13
14   MPI_Status status;
15   MPI_Request request;
16
17   MPI_Init (&argc, &argv);
18   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
19   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
20
21   int start_rank = (argc < 2) ? 0 : (atoi(argv[1]) % comm_size);
22   start_rank = (start_rank < 0) ? 0 : start_rank;
23
24   if (rank == start_rank)
25   {
26     MPI_Isend(&rank, 1, MPI_INT, ((rank + 1) % comm_size), 0, MPI_COMM_WORLD, &
         request);
27     MPI_Recv(&received, 1, MPI_INT, ((rank - 1) % comm_size), 0, MPI_COMM_WORLD, &
         status);
```

```
28      printf("MPI rank '%d' received message from rank '%d'\n", rank, received);
29      fflush(stdout);
30    }
31    else
32    {
33      MPI_Recv(&received, 1, MPI_INT, ((rank - 1) % comm_size), 0, MPI_COMM_WORLD, &
            status);
34      printf("MPI rank '%d' received message from rank '%d'\n", rank, received);
35      fflush(stdout);
36      MPI_Isend(&rank, 1, MPI_INT, ((rank + 1) % comm_size), 0, MPI_COMM_WORLD, &
            request);
37    }
38
39    MPI_Finalize();
40    return 0;
41  }
```

C.5 Deploying the simulator on a new architecture

To make the simulator deployable on a new architecture the inline assembly code of the simulator has to be enhanced. Four inline assembly commands have to be created especially for the architecture. The four assembly commands are:

1. Push all CPU registers and save the stackpointer into a variable.
2. Save the basepointer into a variable.
3. Restore the stackpointer from a variable and pop all CPU registers.
4. Restore the basepointer from a variable.

See assembly_code.h for Intel's x86 and AMD64 versions.