Technische Universität
Kaiserslautern

Out-of-Order Execution of Data-Flow Process Networks for Streaming Applications

Dimitri Blatner

December 27, 2011

Bachelor Thesis

Supervisors:
Prof. Dr. Klaus Schneider
Daniel Baudisch

Embedded Systems Group
Department of Computer Science
Technical University of Kaiserslautern

Declaration of Authorship

I hereby declare that I have written this thesis entitled "Out-of-Order Execution of Data-Flow Process Networks for Streaming Applications" independently and resources used in this work are stated entirely. Passages quoted literally or analogous from other works or the internet – including tables, maps and figures – are emphasized with the specification of the source.

Kaiserslautern – December 27, 2011

(Signature)

Dimitri Blatner

Acknowledgement

Thanks to all the inspiring people at the Embedded Systems group I am working with, especially to my supervisors Prof. Dr. Klaus Schneider and Daniel Baudisch, who gave me priceless knowledge and support in linguistic, logical, and amicable manners.

Special thanks to my friends and family around me for standing me by all the time, giving me ideas and sharing wonderful and unforgettable time with me.

Abstract

This thesis is about out-of-order execution and dynamic scheduling of data-flow process networks for a task-based parallelization of streaming applications, known from dynamic processors at the instruction level. Like in modern processors, streaming applications typically have the problem of having some dependencies between subsequent inputs, e.g. audio and video decoders. That is why they can not be effectively parallelized, since the inputs have an impact on the application state. With independent inputs, only dependencies within the application have to be considered. The goal is to provide a synthesis tool to translate synchronous guarded actions to MPI applications for MPI computer clusters, a starting point for further improvements.

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Related Work . . . 11
1.3 Structure and outline . . . 14

2 Fundamentals **15**
2.1 Synchronous Languages . . . 15
2.2 Synchronous Guarded Actions . . . 17
2.3 Data-Flow Process Networks . . . 18
2.4 Out-of-Order Execution . . . 19
2.5 Message Passing Interface . . . 21

3 OOO Execution of DPN **25**
3.1 Components . . . 30
3.1.1 Task Functions . . . 31
3.1.2 Dispatcher . . . 32
3.1.3 Worker . . . 34
3.1.4 In- and Outputs . . . 35

4 Benchmarks **37**

5 Conclusion **39**

6 Future Work **41**

List of Figures **42**

Bibliography **45**

Chapter 1

Introduction

1.1 Motivation

Automatic parallelization of software is an important effort in computer science to increase the computational performance of computer systems. Parallel computer architectures with multi/many-core processors or grid computing environments define a standard in computer systems and become more and more an ubiquitous topic in embedded systems. It is important not only to use, but actually utilize them efficiently to achieve a better performance. This has also a great impact on energy efficiency, which today is a very important and widespread topic.

Some approaches work with tasks, mapping dependent or independent tasks of the application to threads. Unfortunately, thread synchronization is an expensive operation and slows down the overall speed as well as limited parallelism does due to data dependencies. To make threads affordable, each one has to have a minimal amount of instructions that sometimes can not be easily achieved. Furthermore, the system's latency time grows linear to the number of threads, so threads will be excluded from this thesis. Other approaches rely on compiler construction and improve parallelism with techniques like software pipelining, where the body of a loop in an application is split into stages. Again others introduce new communication and computation models (e.g. message passing, shared memory models in distributed systems). The focus of this thesis is on compiler construction, compiling applications using the model of computation (MoC) of data-flow process networks (DPN).

When it comes to heterogeneous clusters, scheduling the computational work statically can barely utilize the cluster to full performance, because each compute node of the cluster can have different cache behaviors and timings.

This dynamic behavior can invalidate assumptions used by static scheduling, even if detailed knowledge about the used hardware and timings is available, which is rarely the case. Also the computation itself can vary depending on given input values, where static scheduling causes waiting scenarios and is therefore inefficient. In those cases dynamic scheduling and out-of-order execution can achieve more throughput and performance. To increase parallelism right from the beginning, the synchronous programming paradigm was chosen in this thesis, that works implicitly parallel.

The approach in this thesis applies the dynamic scheduling and out-of-order execution techniques, as known from processor design, to improve work balancing and throughput. These techniques were first introduced for the System/360 mainframe computer of IBM and are also implemented in almost all personal computers since the Intel Pentium Pro or the AMD K6 processors[1]. Since there is currently no implementation available that combines these techniques and works out of the box, it is a point of interest if this approach can really improve the performance of future parallel systems, especially with respect to the communication overhead of dynamic scheduling. Streaming applications are suitable for this approach as they deal with a great number of sequential inputs – usually not independent from the output – and therefore are predestinated for parallelization.

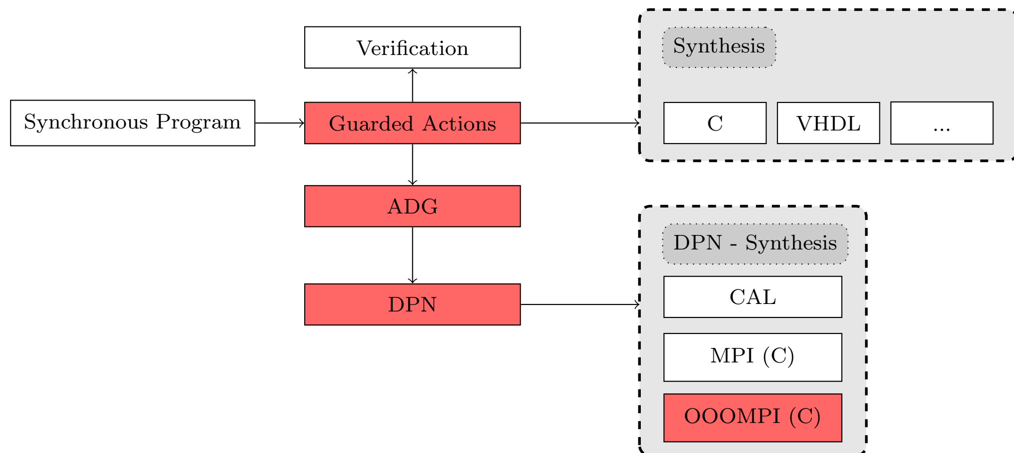

Figure 1.1: Synthesis flow for the generation of MPI code from a synchronous system.

The main contribution is the implementation of a synthesis tool for an

[1]See "Out-of-order execution", *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/63mEPWDgq` (accessed Dec 8, 2011)

automatic translation of DPNs into applications for heterogeneous computer clusters. The target programming language is C using the Message Passing Interface (MPI), more specifically the MPICH2 implementation. Figure 1.1 shows the starting point of the DPN, namely synchronous Guarded Actions (GA) and a directly abbreviated Action Dependency Graph (ADG). They are generated from the synchronous language Quartz and packed in the intermediate format Averest Interchange Format (AIF).

In general, DPNs can represent any kind of application, but are usually unbalanced with respect to the computational effort of their nodes. Moreover the computational effort of a single node may vary for different inputs.

One of the main advantages of MPI is that it can be used in heterogeneous computer clusters, i.e. networks of computers differing in hardware, performance, etc., and is not limited to shared-memory multiprocessor (SMP) or chip multiprocessor (CMP) systems. The MPI code is portable which means the code can be run in different environments, from single core processors up to high performance computing (HPC) clusters.

1.2 Related Work

This section will give a short overview on existing research and industry standards in the field of task-level parallelism based on compiler techniques. While there is much effort in parallelizing an existing sequential application at present, it is rarely the case that todays parallel architectures are considered in detail. Primarily scientific applications accomplish high utilization rates and throughput on HPC clusters and have to be written manually. New techniques have already been and have to be developed and there are currently some promising approaches available. A more detailed discussion of popular models with examples according to SMP architectures is available at [1].

To preserve the overview despite the variety of available task-based approaches for exploitation of performance of parallel systems and limit the scope of this thesis, StarSs was chosen to be presented in this thesis among others ([2], [3]).

OpenMP [4, 5, 1] – is currently the leading programming model in industry for SMP architectures and was a joint development by hardware manufacturers and software engineers, headed by Hewlett-Packard, IBM, Intel, and Oracle. The standard supports the widely used languages C, C++ and Fortran and extends them by compiler directives. OpenMP is designed to be highly portable and to keep parallel programming as well as parallelizing

existing sequential applications simple, since it was designed by many different contributors. It is based on a fork-join concept which means wherever code inside a thread shall run in parallel, this thread is forked into multiple sub-threads, which are joined at the end of their parallel execution. At the beginning and the end of the execution stands one main thread. Since version 3.0, the concept of tasks is supported explicitly, where a task is a block of code that can be executed immediately or deffered until later. The main thread generates the previously specified tasks, queues them dynamically in a task queue and dispatches them among multiple worker threads if a parallel execution is possible. That makes the parallel fork-join based programming model of OpenMP more flexible. OpenMP is explicitly designed for shared-memory architectures, therefore it is a main disadvantage when it comes to distributed systems. Hybrid approaches of OpenMP with MPI or cluster usage of OpenMP [6] is currently a big research topic, but it is still challenging due to limited support from MPI (see [7], [8]) and is not the topic of this thesis, since here only a distributed architecture is considered.

StarSs [9, 10, 11, 12, 13, 14, 2, 15, 16, 17, 18] – stands for Star superscalar and is the name for a family of node-level task-based programming model written in C and Fortran, all targeting the ease of programming. It was developed at the Barcelona Supercomputing Center[2] among others. Parts in a sequential application, that shall run concurrently, are defined as tasks. Like OpenMP, StarSs based applications express tasks and dependencies between tasks with compiler directives. Detected tasks are scheduled among the available resources on a specific architecture. The following programming models are instances or extensions of StarSs and each of them addresses a particular type of architectures:

SmpSs [13, 14] focusses on the ease of programming for SMP (shared-memory multiprocessor) systems and code portability. It is based on CellSs [16] that is one of the first instances of StarSs for the Cell/Broadband Engine processors from IBM. SMPSs is designed for exploiting parallelism of sequential applications on shared-memory architectures and therefore is not suitable for clusters. For clusters, SMPSs was combined with MPI [11].

GpuSs [10] addresses platforms with one general-purpose processor and multiple graphic processor units (GPUs).

OmpSs [12] as well as ClusterSs [9] are designed for clusters of SMPs, where OmpSs combines OpenMP with StarSs for heterogeneous SMP clusters whereas ClusterSs utilizes a homogeneous cluster with a global address space memory.

[2] See `http://www.bsc.es` for more details.

CompSs [18] – a new version of GridSs – targets the ease of development of applications exploiting their inherent parallelism for heterogeneous clusters/grid computing environments.

TaskSs [17] extends StarSs by an out-of-order, task analyzing and scheduling pipeline on chip multiprocessor (CMP) platforms. It discovers task dependencies dynamically from tasks, that are generated by a sequential thread. Task informations and readiness of operands are stored in multiple task reservation stations (TRS), that communicate with each other. Object renaming tables (ORT) map operands to their computing producer and their latest version. For every ORT, there is one object versioning table (OVT) that tracks live operand versions. It is similar to Tomasulo's algorithm (see Section 2.4), that is also used in the approach of this thesis.

Intel Threading Building Blocks [19, 1] (TBB) is a part of Intel's Parallel Building Blocks (PBB) family of parallel-programming models. It provides template libraries in the programming language C++ for multi/many-core environments available for Windows, Linux and Mac OS. TBB implements tasks as lightweight C++ code blocks, which can be (de-)allocated much faster than threads (due to special processor instructions and usage of task queues). For every processor core, one worker thread is instantiated, which spawns tasks during its runtime. TBB puts this tasks at a task queue of the corresponding worker thread. If a worker thread runs out of tasks, it can steal tasks from other worker threads.

The known technique Grand Central Dispatch[3] (GCD) from Apple uses a similar mechanism, providing data constructs to specify parallel code blocks. Those blocks are placed on various system- or user-queues to describe concurrency. The operating system manages a thread pool and schedules the blocks of the different queues to the threads of the thread pool.

OoOJava [20] extends the programming language Java by a single annotation (*rblock*, "reorderable block") in sequential applications on multi/many-core platforms. Data dependencies among *rblocks* are discovered statically and the *rblocks* are scheduled statically among the processors. The *rblocks* can be nested and there is an implicit top-level *rblock* for the main method. Every parent *rblock* is responsible for its child *rblocks* and manages their dependencies as well as its own dependencies.

[3] See "Concurrency Programming Guide" at `http://www.webcitation.org/64Ezl8RhS` (accessed Dec 27, 2011), and "Grand Central Dispatch (GCD) Reference" at `http://www.webcitation.org/64Ezy4oHv` (accessed Dec 27, 2011).

1.3 Structure and outline

This thesis is organized as follows:

Chapter 2: *Fundamentals* introduces MPI, Synchronous Languages, GA, DPN and out-of-order execution.

Chapter 3: *OOO Execution of DPN* presents the approach in more detail, combining the previously introduced foundations. Furthermore, the synthesis tool with details on its implementation is explained.

Chapter 4: *Benchmarks* shows a performance analysis and comparison between the implementation from this thesis and an implementation that does not use dynamic scheduling and out-of-order execution.

Chapter 5: *Conclusion* summarizes the results of the benchmarks and knowledge acquired from this work.

Chapter 6: *Future Work* shows ideas for optimizations, how to improve the presented implementation and therefore increase performance.

Chapter 2

Fundamentals

2.1 Synchronous Languages

Besides the common programming paradigms (imperative, functional or object-oriented programming) a paradigm called *Synchronous Programming* appeared more than 20 years ago [21]. It was developed to describe and program parallel reactive systems. Popular examples are SIGNAL [22], ESTEREL [23, 24], LUSTRE [25, 26] and QUARTZ [27, 28, 29, 30, 31, 32]. Moreover, SIGNAL and Esterel were the first, that were used for systems with safety-critical constraints, or other reactive systems with hard real-time constraints.

This thesis concentrates on – but is not limited to – the imperative synchronous language QUARTZ developed by the Embedded Systems research group at our university as a part of Averest[1], as it has the same basic principles as all of the above languages. The syntax of Quartz is similar to Esterel and therefore different to the *declarative* style of Signal or Lustre [23, 32]. The synchronous model of computation (MoC) and the corresponding Quartz syntax are explained at a suitable abstraction level.

Synchronous applications (or synchronous systems) in one of the above languages work in cycles (macro steps), where finitely many statements (micro steps) are grouped to a macro step. A macro steps correspond to one logical time unit, whereas a micro step is executed in infinitely short or zero time [29].

The code structure is as follows: the code is separated in blocks executed at a specific point in time (macro steps) and each block consists of instructions (the micro steps) and is delimited by a separator. The sequential order of the code blocks is also the order of their execution in time. Because the unfolded code in each code block is executed in parallel, one of

[1]For more details, see `http://www.averest.org` .

the main advantages of synchronous languages is implicit parallelism. The internal state of the application is defined through the values of all occurring variables and the currently observed macro step and is always deterministic. That means that each variable has exactly one value in each macro step. In every macro step the synchronous system reads all inputs, computes all outputs with respect to the internal state and updates the internal state for the next macro step. A drawback of this model is the possibility of cyclic dependencies [21, 28, 29].

In analogy one can consider a digital circuit with inputs and outputs and a clock timer, the behavior per clock tick is represented by the code between two `pause;` statements. In fact, synchronous programs can be effectively transformed into hardware circuits, since the synchronous MoC ensures perfect synchrony, determinism [21, 32] and precise timings, that are also needed by streaming applications. Hence, synchronous programming is a very powerful approach for modeling and verification of embedded systems. However, these advantages are not for free: *schizophrenia* and *causality problems* have to be solved [33, 34, 29, 32] by a compiler. The example from Figure 2.1 describes the behavior of a digital inverter.

```
1 module Example(nat ?i, int !o) {
2     loop{
3         if(i != 0)
4             next(o) = i*(-1);
5         else
6             next(o) = 0;
7         pause;
8     }
9 }
```

Figure 2.1: Quartz code example.

The system gets an input value i (indicated by "`?`"), where i is a natural number, and returns an output value o (indicated by "`!`"), where o is an integer value. It runs an infinite loop with a conditional statement in each macro step (the "`pause;`" statement is the delimiter). The "`next`" statement means that the result of the output variable is available in the next macro step although it is computed in the current macro step (more in Section 2.2). If the input is zero, the output will be zero, otherwise the output is inverted by multiplying -1 to the input.

The next section presents an intermediate format for describing synchronous systems.

2.2 Synchronous Guarded Actions

Synchronous Guarded Actions (GA) are a formalism for describing concurrent systems and have already been used as an intermediate language during the compilation process of synchronous programs. They are closely related to Guarded Commands of E.W. Dijkstra [35]. A GA consists of a *guard* – a conditional statement γ – and an *action* – an operation α, that is executed if the guard holds, where α can be an *immediate* assignment ($\gamma \Rightarrow \alpha : x = \tau$) or a *delayed* assignment ($\gamma \Rightarrow \alpha : next(x) = \tau$) [32]. Actions are executed immediately if the guard holds, that means within the same macro step. In case of delayed actions the result of the computation is available in following macro step. Note that self referencing assignments in form of $next(x) = x+1$ are allowed in GAs, but $x = x+1$ leads to a conflict, because the value of x has to be uniquely determined.

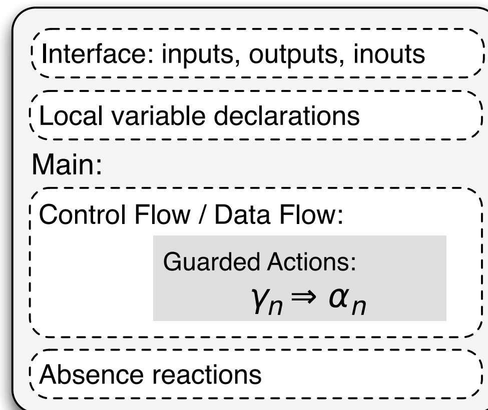

Figure 2.2: Basic structure of a synchronous system described with guarded actions.

The interface specifies variables (inputs, outputs, inouts) for connectivity to the environment and are either *events*[2] or *memorized*[3]. Local variables are only visible to the system. The main part contains a set of guarded actions, that describe the systems behavior, with the meaning "`if (`γ`) holds, execute` α". γ is the guard and α the immediate or delayed action. Absence

[2]Variable is set to a default value (usually 0) in the particular macro step, if it is not set to another value or read from the environment.

[3]Variable retains its value the particular macro step, if it is not set to another value or read from the environment.

reactions take place, e.g. when an interface variable have no assignment in a particular macrostep, setting the variable either to a default value (for event variables) or the value of the previous macro step (for memorized variables).

With this representation the control and data flow of synchronous programs can already be described as a set of GAs [32]. Usually control and data flow are separated due to their purpose, but in practice both can be treated the same way. GAs are much more comfortable to synthesize into other languages or formats since no loops or other more complex control flow expressions exist anymore. For use in this thesis the set of GAs is encapsulated in an XML format called the AIF (Averest Interchange Format). An action dependency graph can be directly abbreviated from the GAs, as shown in Figure 1.1, moving towards DPNs.

2.3 Data-Flow Process Networks

Data-Flow Process Networks [36] is the name of a MoC and is a special case of Kahn Process Networks [36, 37, 38]. DPNs are also related to Petri nets[4] [39].

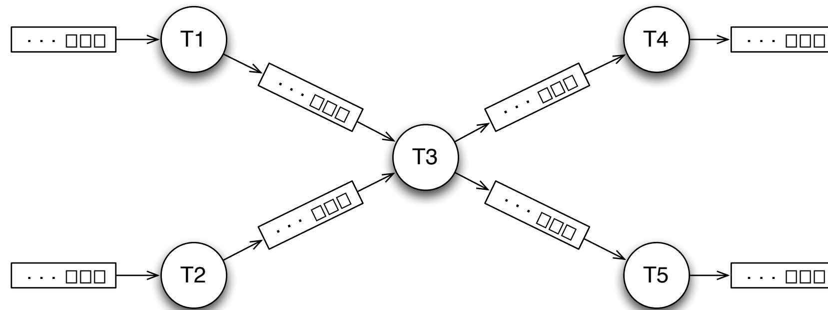

Figure 2.3: Simple example of a DPN. *T1* to *T5* are processes connected via unidirectional and unbounded FIFO queues. The arrows signalize the data flow direction.

A network consists of one or more processes, where the processes interact via unidirected and unbounded FIFO (first in, first out) queues, also called (infinite) sequences or more specifically streams (without going into detail of the formal semantics). Additionally a process is "a mapping from one or more

[4]Introductional references for petri nets are available at `http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/`, alternatively at `http://www.webcitation.org/63WP71Ox3` .

input sequences to one or more output sequences" [36]. The process behavior is defined by firing rules and is described as a function on streams. Every element or token of an input/output stream is consumed/produced once by a process, but a process can also consume/produce more than one token or even no token (depending on the firing rules). Considering the example in Figure 2.3, the processes are the nodes *T1* to *T5* with the FIFO queues in between. Writes to output queues take place immediately and are therefore *non-blocking*. Reads from input queues are *blocking*, that means a process is halted until enough tokens are in the input buffers it attempts to read. If all required input tokens are available, the process is automatically fired. Hence, the computations of the processes are performed independently from each other. However, it is also possible to let a scheduler assemble and nest the execution of processes, instead of having blocking reads and non-blocking writes [36].

A process depends on other processes, if at least one of its input streams is an output stream of another process. In this case the depending process waits until its input streams were filled by the other processes. This implies an asynchronous behavior of the entire DPN. There is no need for a global control-flow, since it is implicitly given by the data dependencies of the processes [36]. The execution order is as follows: first T1 and T2 can fire (depending on their input queues), then T3 and finally T4 and T5.

Without going into details, DPNs can be either deterministic or non-deterministic. Hence, a DPN is a comprehensive MoC to describe single-, multithreaded as well as distributed systems and applications for parallelism purposes.

A DPN is *synchronous* if all nodes are fired in a fixed periodic schedule, where every node can fire more than once in this schedule [36, 40, 41].

2.4 Out-of-Order Execution

In the 1960's Robert Tomasulo from IBM developed an algorithm to exploit multiple floating point execution units [42] on a *System/360*[5] model 91 mainframe computer system with a CISC[6] instruction set architecture. Today every dynamic processor in personal computers, workstations, servers and mainframes makes use of it.

The algorithm combines a clever tagging scheme together with super-

[5]See "IBM System/360", *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/63mF3aiJP` (accessed Dec 8, 2011).

[6]Complex Instruction Set Computer

scalarity[7] – fetching multiple instructions and executing them in multiple execution units – and dynamic scheduling at the instruction level – detecting dependencies and scheduling instructions to the execution units. The algorithm can easily be generalized, so that it works not only for floating point operations.

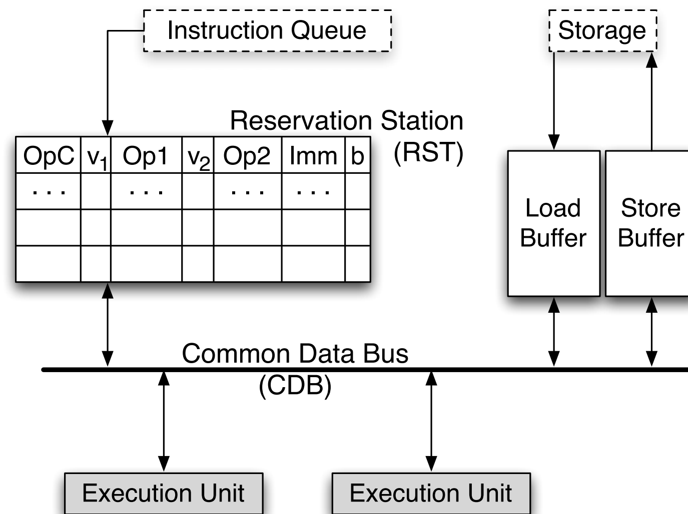

Figure 2.4: Basic structure of a computation unit using Tomasulo's algorithm. The example shows two execution units, e.g. an adder and a multiplier; a load buffer fetching load/store instructions; a store buffer reordering the results of the computations and storing them to the memory or a register; the RST consisting of operation-code *OpC*, operands *Op1* and *Op2* with valid tags v_1 and v_2, optional immediate operand *Imm*, busy flag b; the CDB interconnecting all parts; the abstract control unit symbolizing the communication protocol, that uses forwarding references and register tags.

Tomasulo's algorithm basically can be described in an abstracted view with the following parts, as shown in Figure 2.4:
The **Reservation Station** (RST) is a table, where instructions are stored with the operation, operands and additional tags for detecting dependencies, resolving conflicts[8] and checking if the instruction is in use (busy) and if operands are available (valid). There can be more than one RST, e.g.

[7]Considered to be a Multiple-Instruction-Multiple-Data (MIMD) architecture by Flynn's taxonomy [43].

[8]Read-after-Write (RAW) and Write-after-Read (WAR) conflicts, see "Hazard", *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/64FvroCql` (accessed Dec 27, 2011).

one for different types of operations (fixed point/floating point or R-Type/I-Type[9] operations), but this is not necessary. The RST also waits for updated operands. Figure 2.4 shows an universal RST which is able to administrate all kinds of instructions.
The **Execution Units** can be adders, multipliers or dividers with fixed or floating point arithmetic that perform the actual computation when an instruction at the RST is ready. Since the execution units may differ in the calculation time, the results of the calculations may return out-of-order to the RST and the store buffer.
The **Load and Store Buffers** enable the memory/storage access, when the RST reads operands or the execution units writes results. The store buffer writes results in chronological order (old instructions first), preserving the original program order.
The **Common Data Bus** is a fast interconnection for the RST, execution units and load and store buffers, which all listen on the bus. A clever tagging scheme takes care, that every bus member receives the right messages.

This algorithm or concept enables the exploitation of multiple execution units scheduling instructions as soon as all required operands are available, which increases the throughput and the performance of the execution pipeline. The approach in this thesis implements several parts of this concept (Chapter 3).

2.5 Message Passing Interface

This is the name of a standardized interface of the *Message Passing* programming paradigm for parallel computers or systems. It was developed by the MPI forum, a wide number of people from industry organizations and universities. They tried to benefit from available message-passing systems at that time and provide the most attractive features in one system [44]. There are several open source and commercial implementations available[10], portable or environment specific implementations. A widely spread and open source implementation is *MPICH2* that is also used in this thesis. MPI comes with C and Fortran libraries and compilers providing almost all functions of the specification.

The features of MPI from the MPI version 2.2 report:

[9] R-Type: register type operations, I-Type: immediate type operations; see "Machine code", *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/63mFFvfQB` (accessed Dec 8, 2011).

[10] A detailed list is available at `http://www.mcs.anl.gov/research/projects/mpi/implementations.html`, alternatively at `http://www.webcitation.org/62dUph5Xp` .

> "The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined [45]."

As MPI has many features, only the most important are presented here. The communication between MPI nodes can either be blocking[11], where the MPI node stops until the communication is completed successfully, or non-blocking[12], where the MPI node initializes the communication, but continues with other computation and tests later if the communication is finished and successful[13]. MPI supports buffered and unbuffered communication and provides primitive data types[14]. It is also possible to define own derived data types with a more regular structure[15]. MPI nodes communicate via communicators, that specify regions within the network. The default communicator is `MPI_COMM_WORLD`, where all MPI nodes belong to.

An MPI node is an instance of an MPI application and manifests itself as a thread in the underlying OS. Usually, one MPI node is instantiated for every processor core, as shown in Figure 2.5:

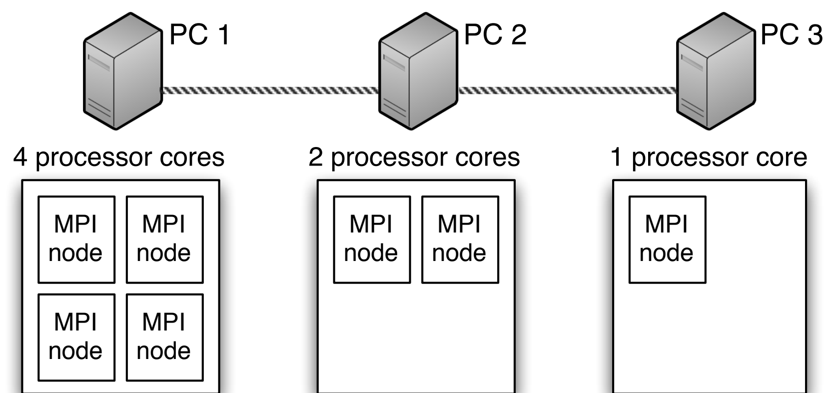

Figure 2.5: Mapping of MPI nodes to processor cores.

[11] `MPI_Send();`, `MPI_Recv();`
[12] `MPI_ISend();`, `MPI_IRecv();`
[13] `MPI_Test();`
[14] `MPI_BYTE,MPI_INT,MPI_FLOAT,MPI_DOUBLE,etc.`
[15] See `http://www.webcitation.org/64G9ZtOw8` for an introduction.

It is also possible to instantiate more MPI nodes per processor core, but this is usually not efficient. MPI usually communicates via TCP/IP[16] and in fact, MPI communication is not lightweight, but very portable.

Every MPI node runs the same application, so the complete behavior of the system has to be programmed in one single application, as shown in Figure 2.6. To distinguish between MPI nodes, every MPI node gets a unique rank or identity number within its communicator. The master node, from where the application is started, usually has the rank 0.

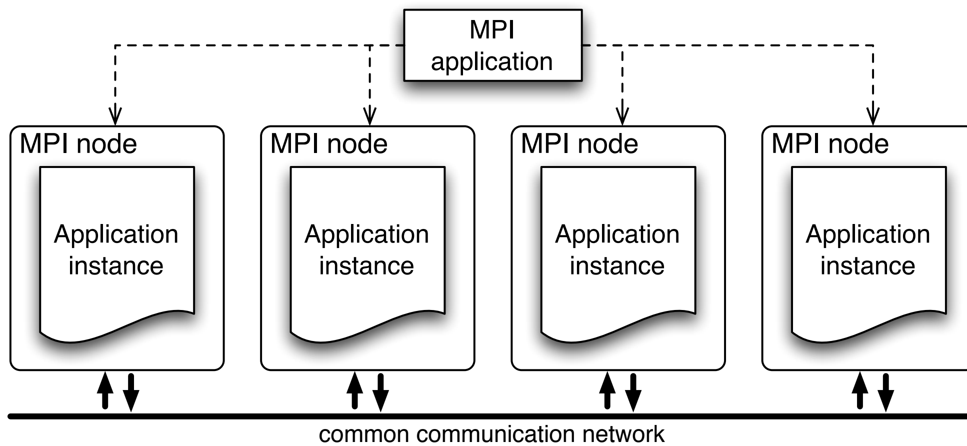

Figure 2.6: MPI application instances among MPI nodes.

[16] See http://www.webcitation.org/64GB3jvaP for an introduction to TCP/IP.

Chapter 3

Out-of-Order Execution of Data-Flow Process Networks

The main idea of this thesis is to exploit superscalarity with out-of-order (OOO) execution and dynamic scheduling of a DPN to improve the performance. The behavior of a DPN is given by the behavior of its nodes. When translating synchronous applications to synchronous GAs and to a synchronous DPN, a DPN node can be described with pseudo code as follows in Figure 3.1. As DPN nodes run decoupled, each node can run in a separate loop in a single thread. In general, in each iteration, the node starts with reading tokens from its input queues. Afterwards, it does some computations to calculate outputs, that are finally written to the output queues. Let's assume the nodes are named $\mathrm{T_N}$ with $N \in \{0, 1, .., n\}$ and n the number of nodes in the DPN.

```
1  Thread T_N {
2      loop{
3          a = A.pop();
4          b = B.pop();
5
6          (x,y) = fun_T_N(a,b);
7
8          X.push(x);
9          Y.push(y);
10     }
11 }
```

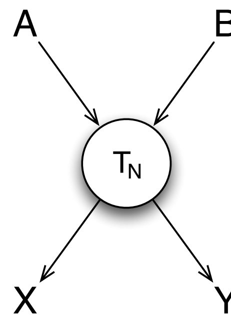

Figure 3.1: A DPN and its behavior in pseudo code.

The queues can be implemented using existing FIFO buffers, that are

available in different parallel APIs (application programming interface), e.g. Intel Threading Building Blocks [19, 1] (`tbb::concurrent_bounded_queue`). In this example, the node T_N has two input FIFO buffers A and B and two output FIFO buffers X and Y. First inputs are read from the input buffers A and B in local variables a and b (one token from each input buffer; line 3 and 4). An internal function $\mathbf{fun}_{T_N}$ computes the outputs, depending on the firing rules of the node T_N and the given inputs, and saves them to the local variables x and y (line 6). Finally the results are written to the output buffers X and Y (one token to each buffer; line 8 and 9). Hence, every DPN node is executed exactly once for a single input set from the environment of the application to produce an output set to the environment.

Recall that reads from an input buffer are blocking in case that the buffer is empty. In contrast, writes to an output buffer are non-blocking according to the Kahn semantics [37]. In the example of Figure 3.1, T_N would stall until both, A and B would have tokens, that can be consumed. The writes to X and Y take place immediately, so T_N would never stall. The order of reads or writes is negligible, because the communication is buffered, e.g. it does not matter whether A or B is read first.

As we see in the following example, mapping each DPN node to a single thread is not a perfect solution to occupy clusters or SMPs. Additionally the given parallelism on the data dependency level is often not sufficient for utilizing clusters to capacity. Consider the following situation, as shown in Figure 3.2 on the left-hand side: in a DPN the node T_k depends on node T_j and can be fired twice as often as T_j. As a consequence, T_j can not produce the tokens necessary for T_k fast enough, so that T_k has to wait every second time it can be fired. In general, the speed of the DPN is ruled by the speed of its slowest node. This problem potentially effects the succeeding nodes of T_k, so it can be characterized as a bottleneck in DPNs.

As this simple example can easily be made more complicated, the appearance of this scenario in big DPNs is very likely. The different firing frequencies of T_j and T_k can appear due to a better cache behavior or a better CPU of T_k's compute node compared to T_j's compute node, or when reading inputs from a slower application environment. Another reason is the typically unbalanced and possibly varying computational effort of the DPN nodes, as shown in the following example.

Consider the scenario shown in Figure 3.2 on the right-hand side: the DPN node T_j computes an exponential function f with time complexity $O(e^x)$, where e is Euler's number and x the input value. All other nodes, that are in parallel to T_j, compute the same polynomial function g with time complexity $O(x)$. The computational effort of T_j and parallel nodes is varying depending on the input x. While function g may be calculated

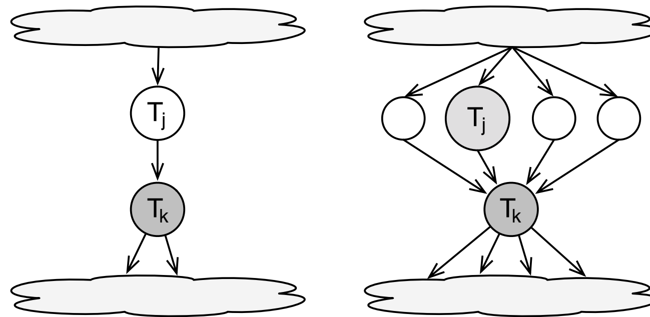

Figure 3.2: Left: simple waiting scenario, T_k can fire twice as often as T_j. Right: unbalanced computational effort of T_j and parallel nodes, T_k has to wait for the slowest predecessor to fire.

in linear time, the calculation time of function f grows exponentially in its input value. Successor node T_k only fires if all inputs from its predecessors are available and therefore can not run faster than the firing frequency of the slowest predecessor and this also effects T_k's successors. In fact the execution of the whole DPN depends on the DPN node with the slowest average firing rate.

For a perfect use of all computer units, a balanced DPN is required, i.e. all nodes of the DPN have the same computational effort. For this reason one can try to partition the DPN or aggregate DPN nodes, so that all parts or nodes have similar computational efforts. Here it must be said, that partitioning is not trivial. A *good* partitioning depends on the used algorithm and on other factors [46, 47]. When it comes to real computer systems, there are some additional issues:

1) We have a fixed amount of compute nodes in a cluster that we want to utilize for processing the DPN nodes. If the amount of DPN nodes is smaller than the amount of compute nodes, the cluster can not be fully utilized and vice versa: having more DPN nodes than compute nodes requires that compute nodes execute more than one DPN node. However, since the computational effort of DPN nodes may vary, the balancing of computational effort between compute nodes can hardly be achieved.

2) In practice it is simply not efficient to have thousands of threads (one thread per DPN node) running in parallel with hundreds per compute node, because the context switches of threads on a CPU are expensive operations. Usually for every compute node or processor, only one thread is initiated.

3) The input/output buffers are always bounded, because computer memory is always limited. A consequence of unbalanced DPN nodes (with respect

to computational effort) is that some output buffers (e.g. from DPN nodes with high firing frequencies) will later or sooner become full. Therefore, the writing DPN node become idle and as a consequence it reads nothing from its input buffers, which leads to a chain reaction that may cause a deadlock.

A simple solution to achieve more parallelism is to just "copy" or multiply the nodes with slower firing frequencies, to increase the produced output tokens by the number of copies per firing. Accordingly Figure 3.3 shows a modification of Figure 3.1 towards a superscalar execution of DPN nodes. T_N reads two input sets at once, then executes its function $\mathbf{fun}_{T_N}$ two times in parallel for the read input sets, and finally writes two output sets at once.

```
1  Thread T_N {
2      loop{
3          a_0 = A.pop();
4          b_0 = B.pop();
5          a_1 = A.pop();
6          b_1 = B.pop();
7
8          parallel {
9              (x_0,y_0) = fun_T_N(a_0,b_0);
10             ||
11             (x_1,y_1) = fun_T_N(a_1,b_1);
12         }
13
14         X.push(x_0);
15         Y.push(y_0);
16         X.push(x_1);
17         Y.push(y_1);
18     }
19 }
```

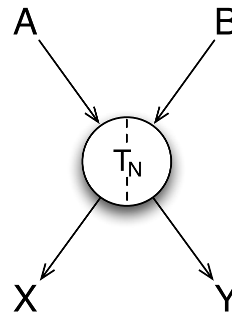

Figure 3.3: A simple superscalar DPN and its pseudo code.

In principle, this simple solution is possible, but not sufficient. It is only useful if no dependencies exist between subsequent inputs, because otherwise the instances of $\mathbf{fun}_{T_N}$ can not run independently from each other and therefore have to communicate with each other. Usually one would produce much more copies to see an effect as the difference of firing frequencies are not just multiples of each other. Hence, the number of DPN nodes can quickly explode. Further, this solution requires that the input buffers of T_N contain at least two input sets, hence, they have to be filled fast enough by T_N's predecessors, which can not be guaranteed. In general, it can be said

that dependencies between subsequent inputs exist in applications, so we are forced to test how many input sets can be read. The solution for it, of course, should be elegant.

However, this simple solution leads to the next idea. One technique to achieve an automatic balancing is e.g. task based scheduling known from OpenMP[5, 4] or Intel TBB[19, 1]). These approaches make use of a fork-join concept as described in Section 1.2. There is one main thread at the beginning, that generates tasks and puts them onto a task queue. In the following multiple worker threads execute the queued tasks in parallel as much as possible.

Instead of relying on a good partitioning, we attempt to create more parallelism by executing a DPN node as soon as the next inputs are available, while previous executions of the same node work. This also enables the OOO execution of the DPN node. Consider the new structure shown in Figure 3.4.

```
1  Thread T_N {
2      loop{
3          a = A.pop();
4          b = B.pop();
5
6          Schedule( Task_N(a,b) );
7      }
8  }
9
10 Task_N (a,b) {
11     (x,y) = fun_T_N(a,b);
12     X.push(x);
13     Y.push(y);
14 }
```

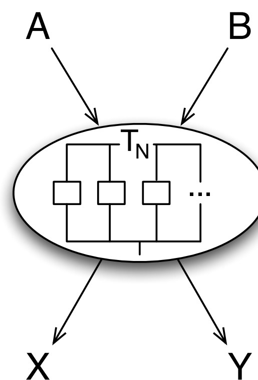

Figure 3.4: A superscalar DPN with enabled OOO execution and its pseudo code.

In the first step, each DPN node T_N first reads its inputs as usual. The second step issues the task that has to be executed. A *task* basically describes a function of a DPN node that has to be applied to given arguments and is scheduled among available compute nodes in the cluster, which are called *workers* in the following. The invocation of "`Schedule`" with T_N's inputs as arguments immediately returns from this function call, so that T_N can concurrently schedule the task for the next input set without waiting for the result of other tasks. To sum up T_N simply initiates the execution while the execution itself is done by one of the workers. This enables a program to

execute an arbitrary number of iterations of a single node at the same time – provided there are enough inputs, that can be proceeded for that node – which makes the DPN node superscalar. Although the scheduling of tasks of a single node T_N is done in order, the results may be out-of-order. The execution time of tasks of a single node may differ due to the reasons that already have been mentioned (e.g. varying computational effort, performance and cache behavior of workers). As a consequence a sorting mechanism is needed to ensure the correct order of outputs in the output buffers that is indicated with *Sort* in Figure 3.5. In fact, it is enough, if the sorting occurs when the affected outputs are read from the applications environment.

The scheduling does not have to be managed by every DPN node itself, but a separate scheduler unit as shown in Figure 3.5. This unit schedules tasks dynamically that means, it manages the states of the workers and checks their availability at runtime.

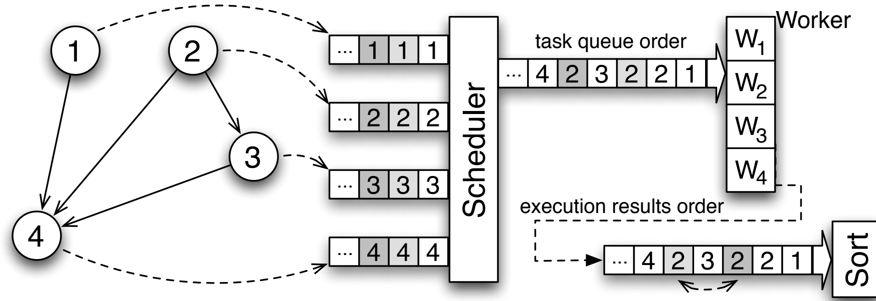

Figure 3.5: DPN task queuing and out-of-order execution. Tasks with the same shade are related to the same computation respectively the same macro step in the task queues. Task 2 is scheduled more often than the other tasks, but due to different execution times of the workers the results come out-of-order and must be reordered.

3.1 Components

As mentioned in the introduction the target language for translating DPNs into an application for distributed systems will be C code using MPI. The input of the synthesis tool is a DPN obtained by translation of synchronous GAs (Figure 1.1, the coherences are described in Chapter 2).

Recall that all compute nodes, worker nodes and master node, run the same executable using MPI, as described in Section 2.5. Synthesized applications have four basic parts: a dispatcher, task functions generated from the

DPN, an input reading and an output writing part. The dispatcher runs on a single master node and schedules the tasks dynamically and out-of-order among worker nodes that do the computations for the tasks. The master node administrates the reservation station (RST) and every worker has its own copy of the table that only holds the values, which are used by the particular worker (more in Section 3.1.3).

3.1.1 Task Functions

For each of the previously described tasks, one task function is implemented, as shown in Figure 3.6. This function can be divided into three parts, in analogy to DPN nodes: *receive the inputs* from the master, *perform a calculation* based on the inputs, and *send outputs* to the master.

```
1  function Task_N (int m) {
2      MPI_Recv( rst[m].varA );      // get inputs from Master
3      rst[m].varA_valid = true;
4      MPI_Recv( rst[m].varB );
5      rst[m].varB_valid = true;
6
7      if (rst[m].varA == 5) {       // computation
8          rst[m].varB++;
9          rst[m+1].varC = rst[m].varA;
10     }
11
12     MPI_Send( m );                // send outputs to Master
13     MPI_Send( rst[m].varB );
14     MPI_Send( rst[m+1].varC );
15 }
```

Figure 3.6: General structure of a task function in pseudo code.

The task functions are invoked with the argument m representing the macro step for which the calculation is done. Then, they are waiting for the needed inputs with a blocking `MPI_Recv` (more in Section 3.1.2). So the task functions do not actively check their input queues like DPN nodes do, they just wait until they receive needed variables from the master. Every input value received – lines 2 and 4 – is stored into the RST and the corresponding validity flag is set to *true* – lines 3 and 5 – by the computing worker. The computation is done according to the GAs of the corresponding DPN node, as shown in the example: **if** the *guard* “varA == 5” holds for macro step

m, **then** *action* increments varB and sets varC to varA – lines 8 and 9. After calculating the output values, Task_N first initializes communication by sending the current macro step to the master node, followed by the actual outputs – lines 12 to 14.

3.1.2 Dispatcher

The dispatcher runs on the master node and has the purpose of managing the workers, reading/writing to/from the applications environment, distributing tasks and their arguments to the workers and managing all occurring variables (local and in-/output variables), as well as control flags. Therefore the master has two tables according to Figure 3.7.

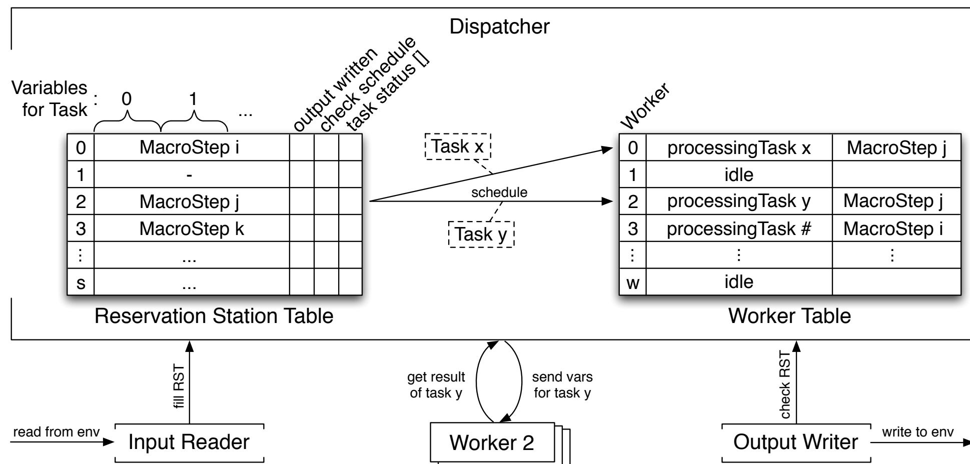

Figure 3.7: Schematic design of the dispatcher with RST and worker table (master MPI node).

The reservation station (RST) table – comparable to the reservation station in dynamic processors – consists of a fixed amount of entries. Each entry stands for a specific macro step holding:

1) the input/output variables of all tasks. Since we have no register sets like dynamic processors, the RST also contains local task variables.

2) the control flag "output written" indicating that the overall output was written to the applications environment, so the entry can be cleared and used for another macro step as soon as all tasks were executed for this macro step.

3) the control flag "check schedule" indicating that values in the RST have changed for this macro step, so the master checks for tasks that can be scheduled.

4) the array “task states”, where an element with index $i \in \mathbb{N}$ indicates, that task i has one of the following status values: *Pending* if the task have not been executed, *InProgress* if a worker processes the task, *Executed* if task was already executed. If all task status values are “Executed” and the “output written” flag is set to true, the corresponding RST entry can be removed and used again for another macro step.

The worker table also consists of a fixed amount of entries, where each entry belongs to one of the workers and holds two integer values: the task number in *processingTask* indicates, which task is processed by this worker, the macro step in *MacroStep* for which the task is executed. If the worker does not process any task, *processingTask* holds the value -1.

```
1  while (true){
2  for (worker in WORKERS) do
3       if (worker.processingTask != -1) {
4       // worker is busy with task
5            if (results_available(worker)) {
6                 get_results(worker);
7                 // worker is now free again
8                 worker.processingTask = -1;
9            }
10      } else { // worker is free
11           for (macrostep in RST) do
12                for (task in TASKS) do
13                     if (macrostep.task_status[task] == Pending
                           && task_inputs_available)
14                     {
15                          // send task arguments to worker
16                          schedule(task, worker);
17                          // update worker and process status
18                          worker.processingTask = task;
19                          macrostep.task_status[task] =
                               InProgress;
20                          // initialize receive
21                          init_nonblocking_receive(macrostep,
                               worker);
22                     }
23      }
24 }
```

Figure 3.8: Very simplified scheduler in pseudo code.

The heart of the dispatcher is the scheduler that checks the RST for values needed to execute the tasks, sends task instructions to free workers updating the worker table and receives results from the workers storing them into the RST. Figure 3.8 describes the main behavior of the scheduler. The send and receive operations for task arguments are implemented as blocking MPI communication (`MPI_Send();`, `MPI_Recv();`), this makes sure that the arguments arrive ordered at the worker, respectively master. After scheduling a task to a worker, the scheduler initializes a non-blocking receive operation (`MPI_Irecv();`) for the scheduled task. This enables the scheduler to schedule other tasks to free workers while simultaneously waiting for results.

3.1.3 Worker

Every MPI node has a working thread (worker) that constantly waits for a message (in particular the task number value) from the dispatcher/master (blocking `MPI_Recv` operation). The waiting process is not active that means the worker does not consume CPU time while waiting. The amount of workers in the cluster is determined at runtime, before the actual application starts its computation. Figure 3.9 shows the structure of a worker.

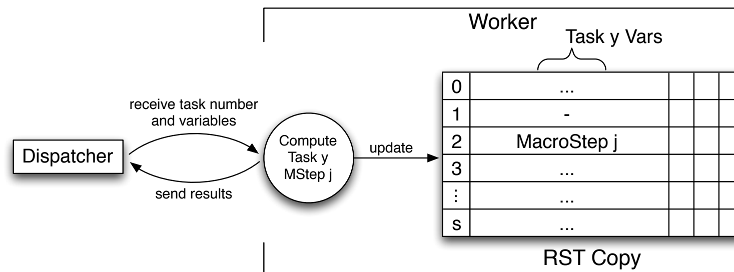

Figure 3.9: Schematic design of the worker and RST table copy (worker MPI node).

After receiving the task number from the master, the worker invokes the corresponding task function that waits for the input values, performs the computation and sends the results back to the master as explained in Section 3.1.1, Figure 3.6. Every worker has its own RST that is not synchronized with the one from the master. The worker only updates the RST variables that are received from the master or computed by the tasks, that were executed.

3.1.4 In- and Outputs

As shown in Section 3.1.2, Figure 3.7 there are input reading and output writing parts, that work in parallel to the dispatcher. Nevertheless, the current version of the synthesis tool implements them as a part of the dispatcher instead of concurrent threads.

The input reader fills the RST of the dispatcher as soon as it gets values from the environment and as long as the RST has free entries. Otherwise it waits until there is a free entry available. The output writer checks the RST entries for the tasks, that generate outputs for the applications environment. If all environment output values are available for a RST entry, the output writer sets the corresponding “output written” flag to true and writes the output ordered to the environment. “Ordered” means, the oldest macro step comes first. This is important, since the approach in this thesis also allows the environment outputs to be out-of-order. Therefore the output writer can be compared to the “Reorder Buffer” in dynamic processors.

The connection of the synthesized application to the environment must be established by the programmer manually in a header file provided by the synthesis tool.

Chapter 4

Benchmarks

The benchmark was performed on a small, “home-made” heterogeneous cluster consisting of 6 Pentium 4 machines (either dual-core or with Hyper-Threading[1]) with different clock speeds and 3 GB of RAM using an ethernet interconnect running at 100 Mbit/s for the communication. As mentioned before, the cluster runs with the open-source MPI implementation *MPICH2*. Each machine instantiates two MPI nodes, one per logical or physical processor core, so we have a total of 12 workers.

First, possible time variations due to unpredictable network delays are analyzed. As supposed, the communication between the MPI nodes takes most of the time and therefore has a great impact on the results, as shown in Figure 4.1. The greatest difference in time variation in the example is about 52% from the avarage (pass 20).

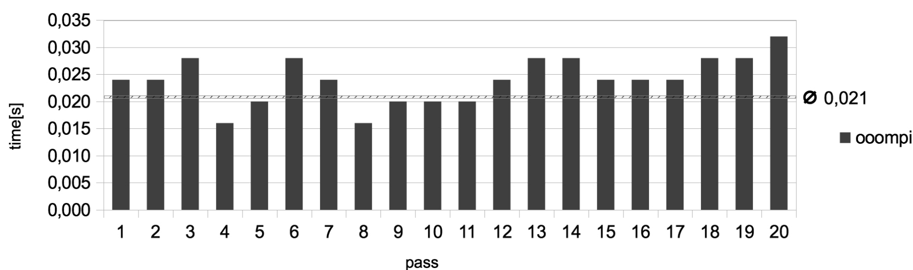

Figure 4.1: Time variation for a 4x4 matrix multiplication. In each pass, 20 multiplications were performed.

The solution from this thesis with out-of-order execution and dynamic scheduling (OOOMPI) is compared to another MPI solution (MPI), which

[1]See “Hyper-threading”, *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/64Fo9jaZt` (accessed Dec 27, 2011).

does not use these techniques. Unfortunately, CompSs/GridSs could not be run on our cluster despite all efforts, so no benchmarks could be made. If it can be run at a later point of time, a comparison to the approach from this thesis should give a better insight of the potential of this approach. Another problem was that some benchmarks likely failed due to timing problems of the MPI cluster – with debug information turned on the benchmarks ran well, without debug information a reproducible deadlock occurred and therefore those benchmarks are excluded.

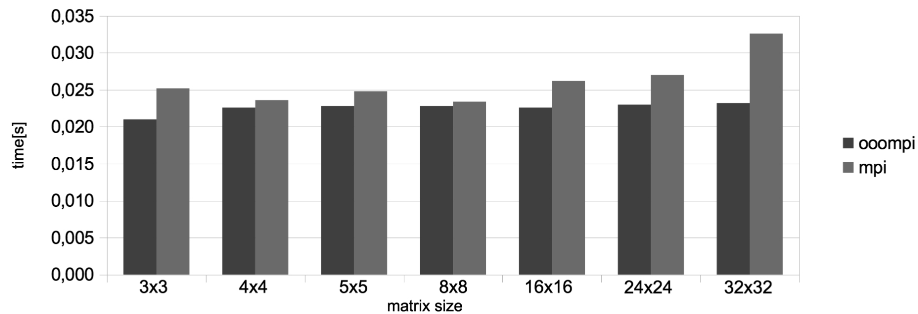

Figure 4.2: Matrix multiplication with OOOMPI and MPI.

Figure 4.2 shows matrix multiplications of different sizes. Each benchmark is run 20 times doing 10 matrix multiplications in a row for the given matrix size. The results from Figure 4.2 show that the size of the matrices barely effects the computation time, which is another indication that the network is the bottleneck of the cluster. Larger matrix multiplications than with 32x32 matrices were be synthesized, but could not compiled by MPI, because they needed too much memory. Furthermore, the benchmarks showed that the master node schedules the tasks very fast to the workers, so the master possibly can utilize much larger clusters.

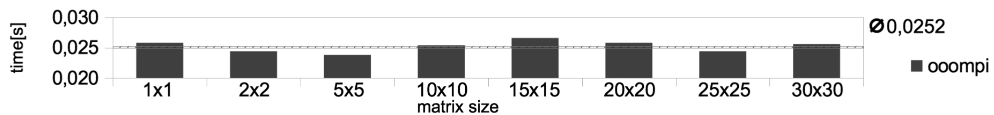

Figure 4.3: Cholesky decomposition with OOOMPI.

Figure 4.3 shows the benchmarks of cholesky decompositions of different sizes. Like in Figure 4.2, each benchmark is run 20 times doing 10 cholesky decompositions in a row for the given matrix size. All benchmarks took about 0,025s, independent from the matrix size, which strengthens the previous statement about the network as the bottleneck.

Chapter 5

Conclusion

This thesis presents a working task-level out-of-order approach with dynamic scheduling based on synchronous guarded actions for the utilization of distributed heterogeneous computer clusters using MPI. The benchmarks showed that the master node schedules tasks much faster to worker nodes than the network connection let him do. Since currently only the network connection limits the performance, the approach from this thesis may have high potential, because this first implementation allows further optimizations and extensions (presented in Chapter 6), but has to be tested on a more performant cluster.

As the current state-of-the-art approaches work task-based, further investigation for the approach from this thesis is very promising.

Chapter 6

Future Work

As this first implementation of the approach from this thesis is very elementary but promising, it can be optimized in different ways:

First MPI can be combined with PThreads[1] or OpenMP creating one MPI instance per machine – one thread per processor core of the machine – and using queues for the communication between the master and the worker nodes. This can improve the performance of multi/many-core or SMP machines in the cluster. The current implementation treats every processor as a MPI node, which slows down the communication speed between processor cores on the same machine. To reduce the impact of network delays on the master, the current blocking communication between the master and the workers can be changed to non-blocking communication. Therefore, an extended message tagging scheme has to be implemented.

Further, each worker can broadcast its computed task results instead of sending them only to the master. Hence, every other worker can update its own RST with necessary inputs and don't have to wait for the master node to start the computation of a task. This also implies that the master only sends data to the worker if the worker does not have it already.

For a better scheduling in heterogeneous clusters, the entries of the worker table of the dispatcher can be extended by hardware specific details. The dispatcher is then able to balance the computational effort even better and schedule tasks in advance.

Next, the out-of-order execution can be extended by speculative execution, also known from dynamic processors. Instead of waiting for all inputs, some inputs can be guessed with a certain probability depending on different factors and the computation of the task can be executed immediately. In

[1] See "POSIX Threads", *Wikipedia, The Free Encyclopedia*, `http://www.webcitation.org/64EY8LiQW` (accessed Dec 27, 2011)

case the later arriving inputs differ from the guessed ones, the results of the speculative execution are dropped.

Finally, the input reader and output writer can be run as separate threads in parallel to the dispatcher on the master node. The master node itself can also create a working thread for processing tasks, since the benchmarks have shown that the master was not fully utilized with scheduling.

List of Figures

1.1 Synthesis flow for the generation of MPI code from a synchronous system. 10

2.1 Quartz code example. 16
2.2 Basic structure of a synchronous system described with guarded actions. 17
2.3 Simple example of a DPN. 18
2.4 Basic structure of a computation unit using Tomasulo's algorithm. 20
2.5 Mapping of MPI nodes to processor cores. 22
2.6 MPI application instances among MPI nodes. 23

3.1 A DPN and its behavior in pseudo code. 25
3.2 DPN waiting scenarious. 27
3.3 A simple superscalar DPN and its pseudo code. 28
3.4 A superscalar DPN with enabled OOO execution and its pseudo code. 29
3.5 DPN task queuing and out-of-order execution. 30
3.6 General structure of a task function in pseudo code. . 31
3.7 Schematic design of the dispatcher with RST and worker table (master MPI node). 32
3.8 Very simplified scheduler in pseudo code. 33
3.9 Schematic design of the worker and RST table copy (worker MPI node). 34

4.1 Time variation for a 4x4 matrix multiplication. 37
4.2 Matrix multiplication with OOOMPI and MPI. 38
4.3 Cholesky decomposition with OOOMPI. 38

Bibliography

[1] Developing parallel programs - a discussion of popular models. Technical report, Oracle Corporation, 2010. Joint work with Liang T. Chen and Deepankar Bairagi.

[2] Parallel programming models for heterogeneous multicore architectures. *Micro, IEEE*, 30(5):42 –53, sept.-oct. 2010.

[3] Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In: *Proceedings GSPx Multicore Applications Conference*, Oct 2006.

[4] The openmp api specification for parallel programming. `http://www.openmp.org`.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. Using openmp: Portable shared memory parallel programming (scientific and engineering computation). The MIT Press, 2007.

[6] Cluster openmp for intel compilers. `http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers`.

[7] Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In: *In 18th Int. Parallel & Distributed Symposium*, page 15, 2004. Joint work with Nikolaos Drosinos and Nectarios Koziris.

[8] Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: , editor, *Proceedings of PDP 2009*, pages 1–10, Weimar, 2009. Joint work with Rolf Rabenseifner and Georg Hager and Gabriele Jost.

[9] Clusterss: a task-based programming model for clusters. In: *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 267–268, New York, NY, USA, 2011. ACM. Joint work with Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, and Gheorghe Almasi, and Jesus Labarta.

[10] An extension of the starss programming model for platforms with multiple gpus. In: *Proceedings of the 15th International*

Euro-Par Conference on Parallel Processing, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. Joint work with Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, and Rafael Mayo, and Enrique S. Quintana-Ortí.

[11] Overlapping communication and computation by using a hybrid mpi/smpss approach. In: *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM. Joint work with Vladimir Marjanović, Jesús Labarta, and Eduard Ayguadé, and Mateo Valero.

[12] Poster: programming clusters of gpus with ompss. In: *Proceedings of the international conference on Supercomputing*, ICS '11, pages 378–378, New York, NY, USA, 2011. ACM. Joint work with Javier Bueno, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, and Rosa M. Badia, and Jesús Labarta.

[13] A dependency-aware task-based programming environment for multi-core architectures. In: *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008. Joint work with J.M. Perez and R.M. Badia and J. Labarta.

[14] Handling task dependencies under strided and aliased references. In: *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. Joint work with Josep M. Perez and Rosa M. Badia and Jesus Labarta.

[15] Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In: *Proceeding of the 4th international workshop on Multicore software engineering*, IWMSE '11, pages 17–24, New York, NY, USA, 2011. ACM. Joint work with Martin Sandrieser and Siegfried Benkner and Sabri Pllana.

[16] Cellss: a programming model for the cell be architecture. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. Joint work with Pieter Bellens, Josep M. Perez, and Rosa M. Badia, and Jesus Labarta.

[17] Task superscalar: An out-of-order task pipeline. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 89–100, Washington, DC, USA, 2010. IEEE Computer Society. Joint work with Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta, and Mateo Valero.

[18] Comp superscalar: Bringing grid superscalar and gcm together. In: *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 185–193, Washington,

DC, USA, 2008. IEEE Computer Society. Joint work with Enric Tejedor and Rosa M. Badia.

[19] Multicore desktop programming with intel threading building blocks. *IEEE Softw.*, 28:23–31, January 2011. Joint work with Wooyoung Kim and Michael Voss.

[20] Ooojava: An out-of-order approach to parallizing java. In: *Hot Topics in Parallelism*, Jun 2010.

[21] The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64 – 83, January 2003. Joint work with A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, and P. Le Guernic, and R. de Simone.

[22] B. Houssais. The synchronous programming language signal, a tutorial. IRISA, April 2002.

[23] The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. Joint work with G. Berry and G. Gonthier.

[24] The foundations of Esterel. In: G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[25] The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, sep 1991. Joint work with N. Halbwachs, P. Caspi, and P. Raymond, and D. Pilaud.

[26] Lustre: a declarative language for real-time programming. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. Joint work with P. Caspi, D. Pilaud, and N. Halbwachs, and J. A. Plaice.

[27] Klaus Schneider. The synchronous programming language quartz. Department of Computer Science, University of Kaiserslautern, 2.0 edition, November 2010.

[28] Imperative synchronous languages, quartz introduction. `http://www.averest.org/documentation/Quartz-Introduction.pdf`, Febuary 2011. Embedded Systems Group, Department of Computer Science, Technical University of Kaiserslautern.

[29] A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006. Joint work with K. Schneider and J. Brandt and T. Schuele.

[30] Modular compilation of synchronous programs. In: B. Kleinjohann, L. Kleinjohann, R.J. Machado, C. Pereira, and P.S. Thiagarajan, editors, *Distributed and Parallel Embedded Systems (DIPES)*, vol-

ume 225 of *IFIP Advances in Information and Communication Technology*, pages 75–84, Braga, Portugal, 2006. Springer. Joint work with K. Schneider and J. Brandt and E. Vecchi©.

[31] DIFFERENT KINDS OF SYSTEM DESCRIPTIONS AS SYNCHRONOUS PROGRAMS. In: S.A. Huss, editor, *Advances in Design and Specification Languages for Embedded Systems*, pages 243–263. Springer, 2007. ISBN: 978-1-4020-6147-9. Joint work with J. Brandt and K. Schneider.

[32] SEPARATE COMPILATION FOR SYNCHRONOUS PROGRAMS. In: H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM. Joint work with J. Brandt and K. Schneider.

[33] N. Halbwachs. SYNCHRONOUS PROGRAMMING OF REACTIVE SYSTEMS. Kluwer, 1993.

[34] A NEW METHOD FOR COMPILING SCHIZOPHRENIC SYNCHRONOUS PROGRAMS. In: *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM. Joint work with K. Schneider and M. Wenz.

[35] GUARDED COMMANDS, NONDETERMINACY AND FORMAL DERIVATION OF PROGRAMS. *Commun. ACM*, 18:453–457, August 1975.

[36] DATAFLOW PROCESS NETWORKS. In: *Proceedings of the IEEE*, pages 773–799, 1995. Joint work with Edward A. Lee and Thomas Parks.

[37] THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING. In: J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[38] COROUTINES AND NETWORKS OF PARALLEL PROCESSES. In: *Information Processing 77*, pages 993–998. North Holland Publishing Company, 1977. Joint work with Gilles Kahn and David B. Macqueen.

[39] PETRI NETS: PROPERTIES, ANALYSIS AND APPLICATIONS. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[40] STATIC SCHEDULING OF SYNCHRONOUS DATA FLOW PROGRAMS FOR DIGITAL SIGNAL PROCESSING. *IEEE Transactions on Computers (T-C)*, 36(1):24–35, January 1987. Joint work with E.A. Lee and D.G. Messerschmitt.

[41] SYNCHRONOUS DATA FLOW. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. Joint work with E.A. Lee and D.G. Messerschmitt.

[42] AN EFFICIENT ALGORITHM FOR EXPLOITING MULTIPLE ARITHMETIC UNITS. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[43] SOME COMPUTER ORGANIZATIONS AND THEIR EFFECTIVENESS. *IEEE Trans. Comput.*, 21:948–960, September 1972.

[44] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J Dongarra. Mpi: The complete reference. MIT Press, Cambridge, MA, 1996.

[45] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. High Performance Computing Center Stuttgart (HLRS), September 2009.

[46] Translating synchronous systems to data-flow process networks. In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society. Joint work with D. Baudisch and J. Brandt and K. Schneider.

[47] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to algorithms. 2. MIT Press, heimatbooks edition, 2001.