FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Bachelor's Thesis

Implementing High Performance Synchronous Message Exchange

Truls Asheim <truls@asheim.dk>

Supervisor

Brian Vinter <vinter@nbi.dk>

Abstract

Synchronous Message Exchange (SME) is a CSP-like messaging framework aimed at modeling synchronous systems(/systems) such as hardware. With the goal of faster execution of SME networks, various models for parallelized execution has been design, implemented and benchmarked

We give an overview of the properties of SME and how they act in a parallelized envirnonment. Then we explain the details of our implementation and finally we measure our implementation through various benchmarks. We have found that the achievable speedups are highly dependent on the kind of work performed by a network, but in optimal cases, we were able to achieve near linear speedups.

Resumé

Synchronous Message Exchange (SME) er et CSP-lignende messaging framework som er rettet imod modellering af synkrone systemer så som hardware. Vi har i dette projekt forsøgt at ..

Contents

List of Figures

Chapter 1

Introduction

In this report, we describe the design and implementation of a highly efficient library for parallel execution of new, globally synchronous, message passing framework called Synchronous Message Exchange (SME). We will present a parallel, compiled framework, named C++SME, which can be used to implement and execute applications using the SME model.

The remainder of this chapter, will describe and define the SME model. The second chapter will describe the process behind designing C++SME. In the third chapter we will describe the implementation process and finally, we will show the benchmarks performed of our implementation.

1.1 Background and Motivation

In the pursuit of performance and energy efficiency, alternatives to traditional CPU's has been extensively researched in recent years. A lot of this research has been centered around taking advantage of the massive parallelism supported by GPUs in general purpose computing. Another technology which can be used to achieve this goal is Field-programmable Gate Arrays (FPGA). As their name suggests, FPGAs are integrated circuits whose function can be altered after manufacturing

While FPGAs provides several advantages over using for processing work including a significantly improved performance-per-watt ratio, their widespread adoptation are limited by the lack of tools which allows ordinary programmers to adapt their applications to run a FPGA. Currently, Today, FPGAs are programmed using Hardware Description Languages which enable programmers to specify the design of the FPGA in a low-level manner. Due to this, their use are largely restricted to engineers with working knowledge in the field of hardware design. In order for software developers to take advantage of FPGAs, improved high-level hardware design utilities are required [1].

FiXme Note: find right word

In an attempt to improve this situation, a master thesis explored using PyCSP to define hardware designs and synthesize them into Hardware Description Languages. Since CSP is based on the idea that any process can communicate at any time, the primary challenge that arose was how to model the clock-driven global synchrony that exists in hardware using CSP. Their solution was to add a central clock process which all processes in the network had to read from in order to know when communication was allowed. Furthermore, latch-processes had to be inserted between processes in order to control value propagation. This required the addition of channels from the clock process to every process in the network. This explosion of the number of channels proved difficult to manage and even though they successfully managed to synthesize simple CSP networks into HDL's, the feasibility of taking a pure CSP approach to hardware design was discredited.

Some properties of CSP, however, proved to be useful in relation to hardware design. Particularly the shared-nothing property of CSP processes.

These experiences and observations lead to the conception of a new messaging framework called Synchronous Message Exchange (SME) which aims to preserve the properties of CSP that proved useful in hardware designs (such as shared-nothing) and combining them with properties that enforces a hardware-like paradigm such as global synchrony and an implicit clock [2].

The aim of this project is to attempt to implement a parallelized execution environment for SME in order to improve the speed of which the systems that is modeled can be simulated.

1.2 Synchronous Message Exchange

In this section, we elaborate on the description of SME from the previous section.

We would like to start by defining the terms used henceforth in this report to avoid any ambiguities.

1.2.1 Definitions

Network A network is the highest level structure in the SME-model. It is simply a network of processes connected by buses

Cycle During a cycle (figure 1.1, all processes executes and all busses propagates their values. A cycle goes through two distinct phases which we will refer to as the process execution phase and the value propagation phase. The process execution phase will activate all the execution units of the processes in the network to allow them to perform data calculation. The value propagation phase will transfer the output-values generated by processes in the current cycle, to serve as input-values in the next cycle.

1.2.2 Components

Compared to CSP, a much smaller and simpler set of components are used to model the process network. In this section, we describe those components

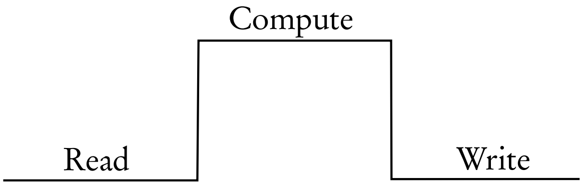

Figure 1.1 – The execution cycle of a SME process visualized as a hardware clock-cycle. Before every cycle data from a input bus is read into the process and after a cycle, data is written back to the gate. Reproduction of figure from [2]

Process A process is an execution unit performing a unit of work. A process is defined by input and output busses used for communicating with other processes in the network and function which is called when the process is executed. The internal state of a process persistent between executions, but it's execution cannot have any side-effects. Thus, the only way the execution of a process can alter the state of other processes is by bus communication.

FiXme Note: Isn't it really the other way around?

Bus A bus enables communication between processes and should be considered analogous to buses found in actual hardware. A bus consists of a writing-slot and a reading-slot, both of which can hold a (signed integer/single FIXME) value. A bus in SME implements the CSP-equivalent of a one-to-all channel with a one message overwrite-buffer which means that only the final value written to the writing-slot will persist in the next cycle. The value of the reading-slot, on the other hand, can be read by all connected processes during a cycle. The value of the reading-slot is idempotent and is guaranteed to remain constant during the process execution phase of a cycle. During the value-propagation phase of a cycle the value of the writing-slot is copied to the reading-slot. From the point of view of the processes, the value-propagation phase is atomic, meaning that the values of all buses can be observed changing "at conce". If no value is written the writing-slot of a bus during a cycle, the value of its writing-slot will be `0` in the subsequent cycle.

1.2.3 Properties

The SME model has a number of special properties which must be maintained in order to ensure correct execution of the network. These properties also influences the design of our execution model.

Property 1 (Implicit clock)**.** One defining feature of hardware is that all processing is driven by a clock beat. In order to maintain this feature in SME, we introduce a simulated clock beat in our implementation of SME and thus the defining property of hardware is preserved in the SME model.

Property 2 (Global synchrony)**.** As a consequence of implementing the simulated clock best, all events and communications of the network occurs completely synchronous from the point of view of a process. FIXME

Property 3 (Shared Nothing)**.** A process is completely autonomous and can only change state through receiving a message on its incoming bus. A process is also self contained in the sense

1.3 Limitations

This report will not discuss details related to design of hardware

1.4 Related work

Chapter 2

Analysis and Design

In this chapter, we will describe and justify the design of our library and the thoughts and considerations that went into producing the final design and ideas that was discarded along the way.

2.1 Overall goal and success parameters

2.2 Paralellization model

A common way of parallelizing CSP-like networks is to use user-level threads to represent a process. In comparison with OS-level threads, user-level threads has a significantly lower overhead both with regards to context switching penalty [3] and memory cost. Furthermore, a much higher limit of user-level threads can coexist on a system. Due to these limitations, implementing these kinds of message passing systems using only OS-level threads are generally not feasible. Therefore, user-level threads are used by other message passing systems such as the C++CSP library[4] and the goroutines in the Go language[5]. However, implementing a user-level threading library would add complexity to our program since we would need to implement a scheduler, for scheduling processes on top of OS-level threads.

Comparing, once again, to CSP, the concurrency in CSP is inherently asynchronous while SME is entirely synchronous in nature. This means that a CSP library needs to implement a scheduler which decides when to give control to a process based on certain events, e.g. a process wishing to communicate or a process receiving a message from another process. .

FiXme Note: Repeats some of previous paragraph

We initially considered a similar design for C++SME, however, due to the enforced synchrony of SME we don't have the same need to schedule processes "intelligently" since we know that, during a cycle, all processes needs to run and all busses have to propagate their values. This, in addition to he shared nothing property of SME, allows us to specify a much simpler parallelization model for SME compared to the techniques used by the aforementioned message passing network implementations.

The basic idea that we base our design on is conceptually similar to a classic producer-consumer setup. In our case, the work "produced" is the processes to be executed, and the consumers are the threads executing the processes. In this setup, a process is run simply by calling a function, whereas user-level threads are usually implemented using the `setcontext` and `getcontext` library functions, which, while extremely fast, still causes a slightly larger overhead compared to a simple function call.

FiXme Note: Switching between OS-level processes is actually really fast, since context-switching from one process to another usually only involves moving CPU registers, i.e. the stack pointer, to an appropriate location, TODO write something like that

by simply letting a number of OS-threads run SME processes in a worker-consumer like manner. This approach also make it simple enforce the synchrony property of SME, since

In this project, we have explored two different variations of this basic idea. Both models are based on the idea described in the previous paragraph. Our overall goal in parallelizing execution of SME-networks is to minimize the amount of core idle time. We expect the hereafter presented model to achieve this goal under different circumstances .

FiXme Note: different networks

2.2.1 Work list model

This approach is similar to a classic producer-consumer model where we have a number of workers which takes tasks off a circular queue and executes them. The main advantage of this model is that it allows processes with different execution times to "interleave" leading to a lower overall execution time. The primary problem of this model is that we need to make the queue thread-safe. The locking mechanism needed to do this isn't free, and could therefore become a dominant factor in the execution speed of networks consisting of many (small) processes.

When shortness is needed, we will refer to this model as Model 1 or M1.

2.2.2 Static orchestration model

In this model, we assign separate queues to each thread of execution and distribute ("orchestrate") the processes amongst them. Due to the properties of SME, this distribution of processes only needs to happen once, before we start network execution. The main advantage of this model is that eliminates any shared state in our network, and therefore we don't need to consider the thread-safety of our queues. This reduces the fixed cost of executing a process significantly. This model, however, is more sensitive to uneven distributions in process workloads. For instance, if we end up assigning predominantly small processes to one core and large processes to another, the core executing the small processes would be left idle until the other core has finished executing it's part of the cycle.

When shortness is needed, we will refer to this model as Motel 2 or M2.

2.2.3 Comparison

Overall, we expect the latter model to have a significant advantage in executing networks with small processes while the queue-locking cost of the former model will perform better when executing networks with large or unevenly distributed workloads since the queue-locking costs will be amortized and allow its process-interleaving ability to become visible.

The optimal way of showing

2.2.4 Identifying optimal process scheduling

In order to determine the efficiency of various methods of process scheduling we need to identify the optimality condition for our process scheduling. An illustration of our threading model can be seen in figure 2.2. Our primary goal is to keep up CPU core utilization and avoid wasting potential processing time by letting a core remain idle for any period of time. Notice, how the idle-time spent by each thread could be reduced by changing the execution order of the processes.

2.3 Managing execution flow

In order to keep track of an SME execution cycles, we need monitor the following metrics during a cycle.

1. In order to know when to stop executing, we need to count the number of cycles performed

2. In order to know when a cycle is complete, we need to keep track of the number of processes that has been executed.

The next problem of executing a SME network is to make sure that the cycles are synchronized and that all processes arrive at the preestablished meeting points (i.e. the phases) of the cycle. While the problem of executing the actual processes, due to the shared-nothing property of SME, is embarrassingly parallel, the need for synchronization makes it not quite so. Therefore, the time spent on synchronization will significantly impact the overall performance of our network.

In order to maintain the properties of SME, we need to insert two "‘meeting points"’, or barriers, into the execution cycle where all threads need to wait for each other before continuing. The first, is after process execution, before buss-value propagation and the second is after the bus value propagation, before the next cycle starts.

These two elements of state keeping are inherently connected since keeping track of how far our execution has come

FiXme Note: harmonize language

An alternative way of handling synchronization is to use thread barriers which makes all threads wait for each other at a certain point before continuing figure 2.1

FiXme Note: Talk about barriers somewhere, which probably describes some of what we're talking about more accurately

In order to monitor the progression of a network execution we need to track a number of metrics during execution.

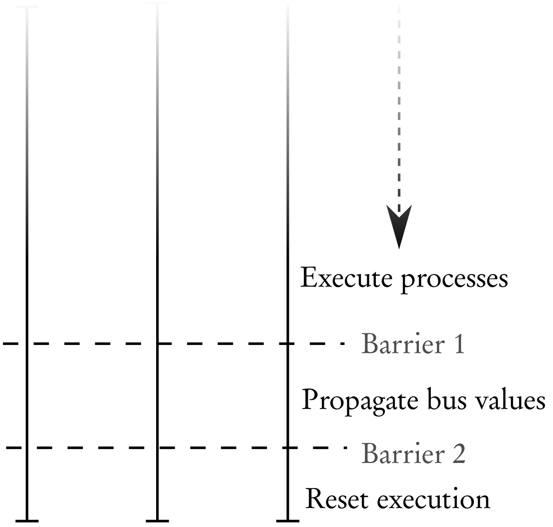

Figure 2.1 – Location of the synchronization barriers of the SME execution cycle relative to the phases of the cycle

2.3.1 Synchronizing cycles

We solve the aforementioned problem by inserting special purpose processes into the execution queue at certain times. The special processes enables us to manage the execution flow of our network without affecting the performance of the individual process invocations. The processes that we insert are the following

Locker This process halts the execution of a thread until getting noticed by a Syncer process

Syncer

The idea, is that $n—1$ processes will be caught in the Locker process until process n reaches the Syncer process which will release all of the waiting threads. After this, the bus value propagation is performed and another set of Locker processes will be reached by the threads. This time, instead of a Syncer process a Reset process will be reached by the final thread entering the barrier a Reset process will be reached. In the static orchestration model, the reset process resets the individual queue pointers of all the threads and in the worker list model it will reset the global queue pointer.

This creates a self-managing execution flow which controls the execution flow of a network without adding a constant overhead to the individual process executions.

2.3.2 Cost of synchronization

The two models has slightly different costs of synchronization

The cost of synchronization arises from two areas

There are two places in the network execution where this "accounting" could be preformed. We could either place a "guard" around each ... An alternative way of performing synchronization is to insert special p

"Naive" way of doing would be to let each execution thread count the number of processes that has been executed. The problem with this approach is that it adds a fixed computational cost to each process execution. This would become especially pronounced in model 1 which would require

Furthermore, we would need to keep the progression of the network execution as a globally shared state which would need to be protected by locks when accessed by a thread. Both of these factors would significantly limit the concurrent scalability of the parallel execution.

Instead, we will propose a model where

One of the central parts of managing an execution cycle is how we synchronize our threads before leaving each cycle phase . In order to maintain the previously described synchrony property, ... Furthermore, the network execution must be controlled so that we are able to stop the execution after a specified number of cycles has completed.

FiXme Note: elaborate

An alternative method, which allows the

2.4 Implementing the queues

An actual circular linked list where the last element points to the first would be the most natural representation of the conceptual circular queue that we just described. The usual advantage of using a linked-list structure is that it allows for O(1) addition of elements. The disadvantage if that element access is slower, even though we would never actually be exposed to the linear time required to find an item in a linked list, even simply accessing the elements of a linked list in order, one by one, is much more expensive t

A straight-forward array is much better suited for the task z7

2.4.1 Locking primitices

Classic locking mechanisms such as semaphores and mutexes needs no introduction. We will, however, spend a little bit of time on explaining the new kid on the block – atomic operations. Atomic operations....

2.4.2 Process orchestration

FiXme Note: This section doesn't really belong anywhere, remove or insert into other sections

As we discussed in the previous section, the primary limiting factor for our multi-threaded network is an uneven and suboptimal distribution of processes across CPU-cores. If no attempt is made to optimize process distribution, the order of process execution will depend on the order of which processes are defined in the source code. Due to property 3 and property 2 of SME networks there is no scenario where it would be necessary or beneficial for a programmer to exercise ultimate control

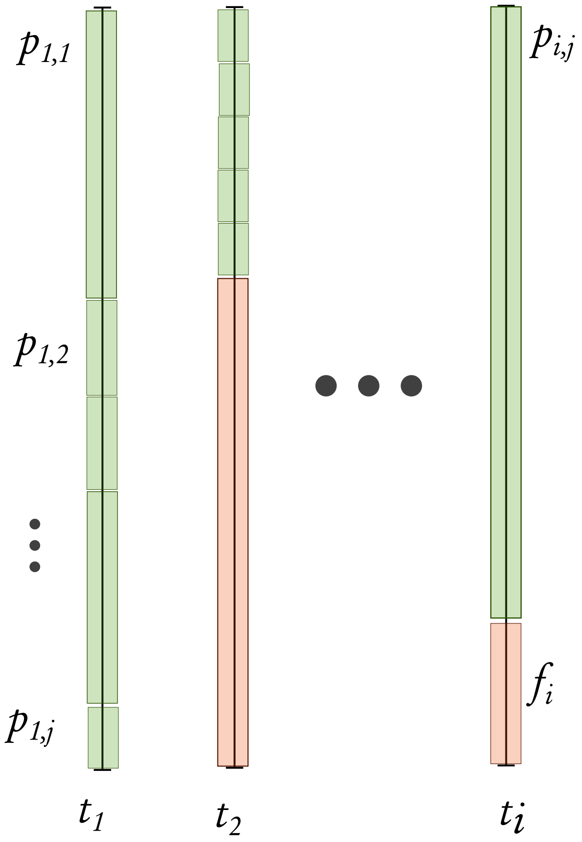

Figure 2.2 – Example of suboptimal distribution of processes across processing threads. Green blocks represents processes while red blocks represents thread idle time. Threads are named $p_{i,j}$ where i is the number of the thread the process has been assigned to and f_i is the combined idle time for each thread. Threads are named t_i.

over the order of process execution. Therefore, maximizing CPU-core utilization would be an unreasonable burden to put on the programmer, especially since their optimization efforts would be specific to a certain number of CPU-cores.

FiXme Note: is predictability a better word?

The optimal method and timing of process orchestration depends on the dynamicity of the work performed by the network we are executing. A network where each process performs a fixed amount of work per iteration will only need to be orchestrated once, while a network where the workload of the processes are variable will need to be continuously evaluated at runtime in order to maintain our optimality condition. These various methods will be discussed for the remainder of this section.

2.4.3 Round-robin orchestration

Processes will be executed on the first available core as seen in figure 2.3

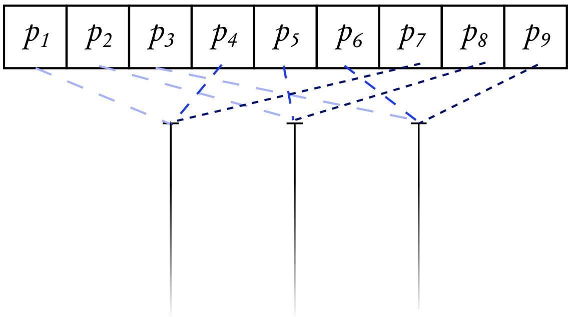

Figure 2.3 – Illustration of round-robin process orchestration. Progressive iterations are shown as increasing color intensity of arrows

Chapter 3

Implementation

We have chosen to implement the SME library in the C++ language. C++ combines the availability of high-level structures, such as classes, with the ability to, when needed, assert low-level control over the code generated. Furthermore, the C++11 [6] revision of the language allows for easy access to features that were previously hard to use. as the functions provided by the `<atomics>` header which enables the use of atomic instructions and enforced memory ordering without the need for inline assembly and similar. Having access to atomics is a highly desire able feature for us since they can be used as high-performance synchronization primitives. Furthermore, classes in C++ are well suited for representing SME constructs and specifically, they provide a natural enclosure of the state maintained by a SME process.

The library is meant to be imported by applications implementing systems using the SME model.

The implementation was performed in several phases. The initial version of the C++SME code was purely single-threaded and was implemented to play around with the C++ API's and for defining the API used to define SME networks.

Adding support for multi-threading required a lot of the code from the initial single-threaded implementation to be refactored and rewritten. The main reason for this, is that the single threaded version was done using high level C++ containers such as maps and vectors which are.... FIXME. Furhtermore, we wanted to be able to switch between the sequential and parallel versions of the execution framework, which meant that they had to be split into two separate classes which inherited common code.

It's also worth noting that these C++ features proved to be significantly slower than using a straight-forward array.

Since the networks that we benchmark are large enough that it would be tedious to write them by hand, features were also added mainly for the purpose of supporting benchmarking. Initially, we attempted to generate our benchmark networks by generating their code using python scripts. this method, however, quickly proved to be in infeasible since GCC's compilation time increases, seemingly exponentially with the amount of objects defined in the code being compiled. We therefore had to add support for runtime definition of networks in the C++SME library. Mind you, that networks are still statically defined in the sense that the orchestration of processes must be performed before the start of network execution. Networks that change at runtime is beyond the scope of SME since it simply isn't possible in hardware, which SME is intended to map

FiXme Note: rewrite

We want the API to be as seamless as possible, that is, it should get int he way of the programmer as little as possible. Several phases of refinement led to the current API which reduces the amount of boilerplate code required significantly compared to the original version

3.1 Initial implementation

Our initial implementation was a sequential implementation of the SME execution environment. This implementation was done simply, as a proof of concept and to experiment with different API's for defining SME networks.

3.2 Queue implementation

We implemented the processes

How we performed process orchestration and, in particular, the workqueue mechanism got a lot of attention in the previous chapter. In this section, we will how we made the actual implementations of the work queues

The locking mechanisms used in the special processes are Condition Variables [7]

3.2.1 Locking mechanisms and atomics

Atomics, and particularlily lockless algorithms have recently been made availible for "casual" use by programmer following their inclusion in recent language revisions.

https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/

3.2.2 Distributing processes across threads

In the static orchestration model, we need to distribute the processes in the network across the threads. While this, in isolation, isn't a very interesting design detail, the specific way that the algorithm that we use works will help to explain some anomalies seen in our benchmarks.

Let p be the number of processes in our network and t be the number of threads. The algorithm then works by distributing $\lfloor p/t \rfloor$ processes across the first $t-1$ threads and then the remaining $p-(t-1)\cdot\lfloor p/t \rfloor$ will end up on the final process. The example in table 3.1 shows what this unequal distribution looks like in practice.

Thread	1	2	3	4
Optimal distribution	2	2	2	1
Actual distribution	1	1	1	4

Table 3.1 – Actual and optimal distributions of 7 processes across 4 threads (in terms of evenness)

The same algorithm is used for distributing bus propagation across threads, however, since bus propagation doesn't require any particular work, the created imbalance has less impact.

3.3 Design goals

The library takes advantage of the fact that the initial process orchestration is only executed once and thus can be implemented with focus on code clarity rather than performance. This

3.4 Public API

TODO: How the SME constructs are exposed by the framework and which operations that can be performed on them.

Chapter 4

Benchmarks and Discussion

We will present a number of benchmarks designed to compare and quantify the differences in performance of the parallelization models that we have implemented.

Since the execution time is only dependent on the total amount of work that a network performs and not how the processes in the network are connected, all of our benchmarks will use a ring-shaped (figure 4.1) network with the participating processes performing varying amounts of work.

We conjecture that the scalability of our implementation will depend strongly on the nature of the workload performed by the SME-networks benchmarked. We will therefore benchmark both light and heavy

As our previously presented hypotheses states, we expect our benchmarks to show that the effects of syncing becomes more pronounced as we decrease the amount of work performed by our processes while it, on the other hand, will become amortized as the amount of work performed by each process increases.

4.1 Testing methodology

All of the time measurements shown were performed inside the SME framework itself using the C++11 `<chrono>` functions, and measures only the actual execution time of the network. It therefore does not include the constant time required to generate the benchmarked networks. Two different hardware platforms has been used for performing the benchmarks: One AMD and one Intel platform.

The Intel machine has the following specs

- CPU: 1x Intel Xeon E3-1245 V2 @ 3.40GHz, 4 cores (8 threads)
- RAM: 32GB
- OS: Linux

and the AMD machines used are part of the eScience cluster at NBI and boasts the following specs:

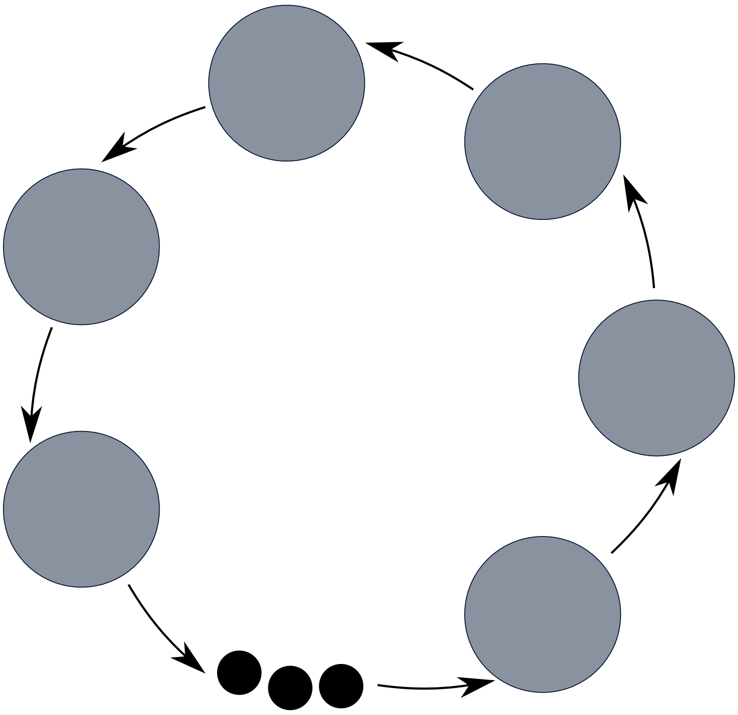

Figure 4.1 – Illustration showing the layout of the network used for benchmarking. The blue circles represents processes and the arrows represents busses

- CPU: 2x AMD Opteron 6272 @ 2.1GHz, 16 cores
- RAM: 128GB

Since the instruction set used by the two CPU's support incompatible optimizations, code generated for one of the CPU's will not run unmodified on the other. Therefore, code executed on the AMD CPU were compiled with the GCC flags `-mtune=barcelona -march=barcelona`, while code executed on the Intel CPU were compiled with `-march=native` on a Core i7 machine. GCC 4.9 was used in both cases. Furthermore, due to incompatible versions of `libstdc++` on the test machines, all benchmarks has been performed using statically linked binaries.

All of the benchmarks has been executed 5 times and the graphs are based on the averages of these. Error bars showing the minimum and maximum deviation from the average has been added to all graphs, however, in some cases the deviations between benchmark runs were too small for the error bars to be visible. We have tried to size the workloads such that the running times are kept within reasonable bounds We calculate our speedup using the formula

FiXme Note: elaborate

$$S = \frac{T_{\text{old}}}{T_{\text{new}}}$$

where S is the achieved speedup, T_{old} is the original (pre-improvement) speed and T_{new} is the new (post-improvement) speed [8].

As a final note, when we talk about the workload of a cycle we refer to the combined work of all processes in the network, not the work performed by an individual process. As such, we define the workload of the network as a function of both the work performed by the individual processes and the number of processes participating in the network. The raw benchmark data can be found in appendix A.

4.2 Synchronization dominated

In this section, we present a benchmark where the performance is predominantly determined by the efficiency of the synchronization mechanisms.

We perform this benchmark by creating a ring which does nothing other than passing an integer value from process to process. Sine each process only takes a few clock cycles to execute, we expect that this benchmark will expose the overhead caused by synchronization.

The following source code used in the execution unit of the process

```
void step() {
  int val = in->read();
  out->write(++val);
}
```

Listing 1 – Source code for the execution unit of the processes participating in the network used for sync-dominated benchmarking

4.2.1 Discussion

We can observe a number of things from the results that can bee seen in figure 4.2. This benchmarks shows the performance of two different networks, one with 20000 processes and one with 50000 processes. Both networks executed 100000 cycles.

When looking at the benchmark of the worker list model, one thing that is clearly visible from this benchmark is the overhead produced by the atomic increment that is required.. This model is doubly penalized when running the benchmark since we, addition to then time required by the atomic increment, which is performed before every process execution, also need to wait for all of the threads to sync up at the end of a cycle. What is slightly surprising, however, is the actual performance that this method shows. It performs significantly worse when going from one to two threads. The most likely explanation for this result is CPU optimizations which make atomic updates of a variable less costly when these updates only occures from one thread.

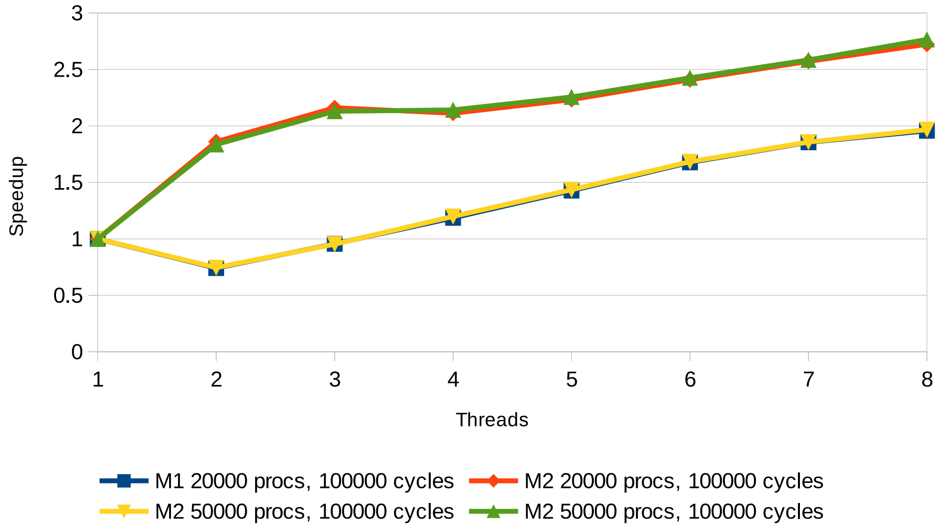

Figure 4.2 – Graph showing the speedup of a SME network consisting of 20000 and 50000 processes respectively when executed for 100000 cycles on an Intel Xeon CPU

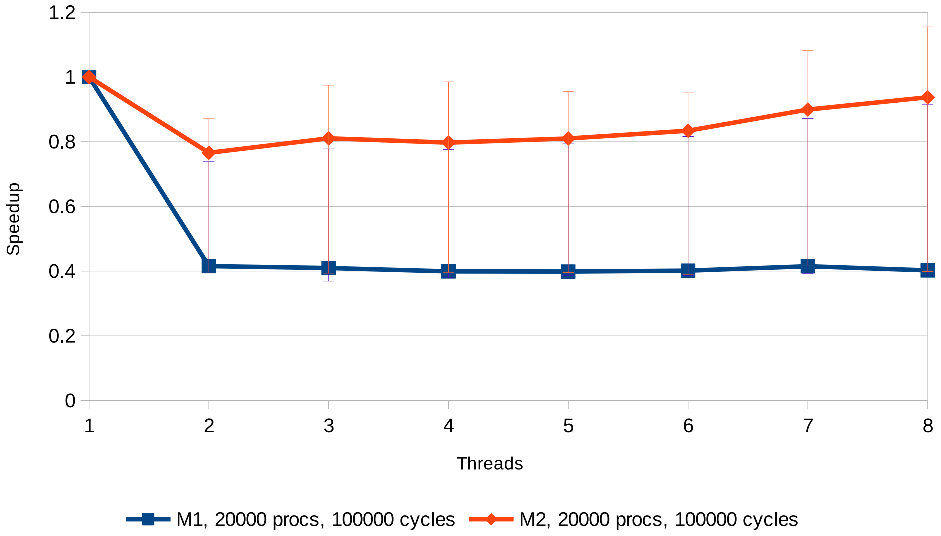

Figure 4.3 – Benchmark results for a network of 20000 processes running for 100000 iterations on the AMD cluster

Our static orchestration model performs quite decently and produces almost 2x speedup when going from 1 to 2 threads. When adding additional threads, the speedup decreases, which is expected since the time spent synchronizing is increased,

Common for both of the models, is that the size of the network executed seems to have no impact on the relative speedups achieved.

Since the Xeon CPU that the benchmarks were performed on only have 4 cores with Hyper Threading, another interesting observation is that Hyper Threading seems to give a significant additional speedup. One hypothesis for explaining the cause of this is that branch-prediction isn't very effective at predicting which functions we're going to call in our SME network. A branch mis-prediction causes the CPU-pipeline to be cleared, creating an optimal condition for Hyper Threading to make use of the empty pipeline-stages[9] While branch-predictors This hypothesis could be tested by running the program through a profiler in order to measure the number of mis-predictions occurring. At this time, these results are not available.

Figure 4.3 show the results of the smallest version of the benchmark running on the AMD cluster. The results are significantly worse compared to the results of the Intel Xeon CPU, both in absolute running times and speedup. Early possible explanations was that, due to the extremely long running time of the benchmark, we were seeing the effects of the process being moved between CPU-cores. However, the results remained unchanged after pinning the threads to CPU-cores placed on the same NUMA-unit. Thus, the only reasonable explanation explanation is that our syncronization mechanisms is significantly less optimized on the AMD CPU compared to the Intel CPU.

FiXme Note: Is it OK to show the numbers?

Another thing standing out from this benchmark is the huge variability between the different benchmark runs as shown by the Y-axis error bars.

Due to these very poor initial benchmark results, we didn't attempt benchmark the synchronization dominated network with different problem sizes on the AMD-cpu.

4.3 Cycle dominated

In this benchmark, the processes in the network performs a significant amount work. We expect that this will, to some extent, amortize the synchronization overhead inherent in the SME model. Combined with the fact that the individual processes contain no shared state, we conjecture that this benchmark will scale significantly better than the previous synchronization dominated benchmark that we performed.

The unit of work being performed by every process in every cycle is simply to divide a `double` floating point number by 3, `10000` times. Since the busses in our SME-implementation only supports transporting integer values nothing is being done with the value calculated, but as long as our workload isn't being optimized away at compilation time this is irrelevant.

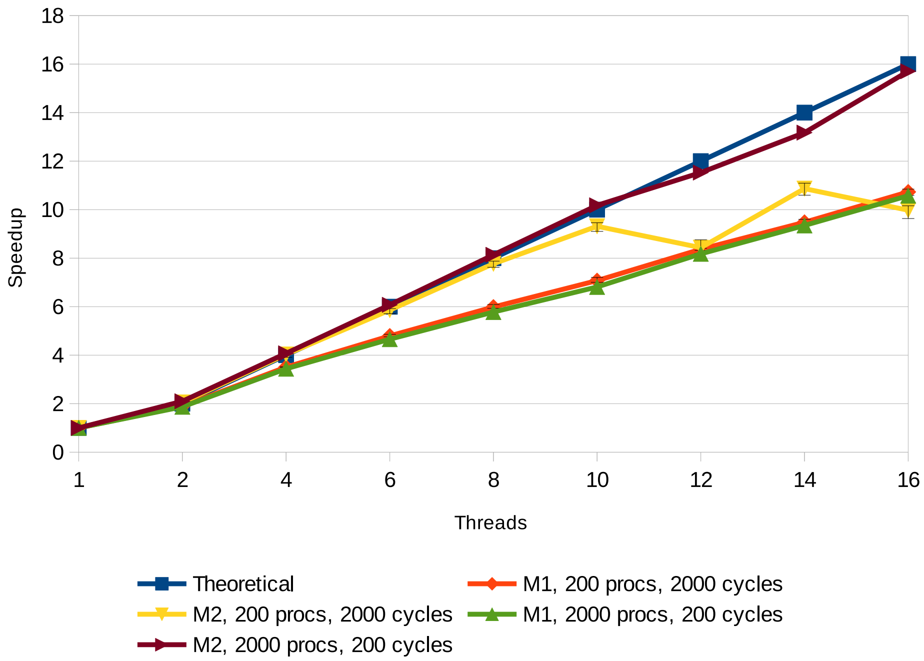

Figure 4.4 – Benchmark results for a network of 20000 processes running for 100000 iterations on the AMD cluster

The size of the workload and the number of nodes were chosen such that the running times of the benchmarks would be reasonable FiXme Note: elaborate

We use floating point numbers as values as opposed to integer values simply because floating point operations are more expensive.

The following code is used as workload in our processes

```
private:
  double n;
  int i;
protected:
void step() {
  n = 533.63556434;
  for (i = 0; i < 10000; i++) {
    n = n/3;
  }
  int val = in->read();
   out->write(++val);
}
```

Listing 2 – Code used for generating work in the cycle-dominated benchmarks

4.3.1 Discussion

The overall conclusion that we can draw from this benchmark it that, for workloads that are sufficiently large, the static orchestration model exhibits significantly better speedups than the work list model.

The work list model, however, appears to be less sensitive to variations in problem sizes since it performs produces similar speedups in the two benchmarks that we have performed. It is interesting that the overhead related to incriminating the atomic pointer still has a noticeable negative impact on its performance. The relative invariance of the results can probably be attributed to the continuous synchronization required by the work list which causes neither of the threads to fall behind or speed ahead of the other threads. Therefore, the combined time that the threads spends in the synchronization barrier is smaller compared to the statically orchestrated model.

Another thing that stands out in the graph is the curious zig-zag pattern that that the statically orchestrated model in the benchmark with the least amount of work forms when running across more than 10 threads. We assume this to be caused by the uneven distribution of processes performed by the method described in the implementation chapter which, in the case of 200 processes distributed across 12 threads, would assign 8 additional processes to one thread compared to the others.

A problem with this benchmark is that the work that we perform can be performed entirely within the cache of a CPU-core. This allows us to scale more strongly than when benchmarking a problem which to a larger extent is limited by memory bandwidth and/or CPU-cache misses. It is, however, still an open question how

4.4 Weak scaling

In the previous benchmarks, we have investigated the strong scaling properties of our implementation, i.e., we have kept the amount of work constant and varied the number of execution threads. In the weak scaling benchmark, we will keep the workload *per thread* constant by performing an amount of work defined to be a fixed multiple of the number of threads that we use. The purpose of this benchmark, is to determine if there is a hard limit on our achievable scalability or whether we can scale to an "‘infinite"’ amount of threads simply by adding more work. This is what is referred to as the *weak scaling* properties[10] of our implementation.

Since the previous benchmarks has shown that the achievable speedup depends on ratio between cycles (synchronizations) and work, we have performed our weak scaling benchmarks using the following cycles-to-processes ratios, 2:1, 1:1 and 1:2.

Ideally, we want this benchmark to have a speedup of 1 for any network with a constant work-per-thread ratio. This would mean that our ability to scale is only limited by how many CPUs that we can add to our system.

The results of this benchmark can be seen in figure 4.5

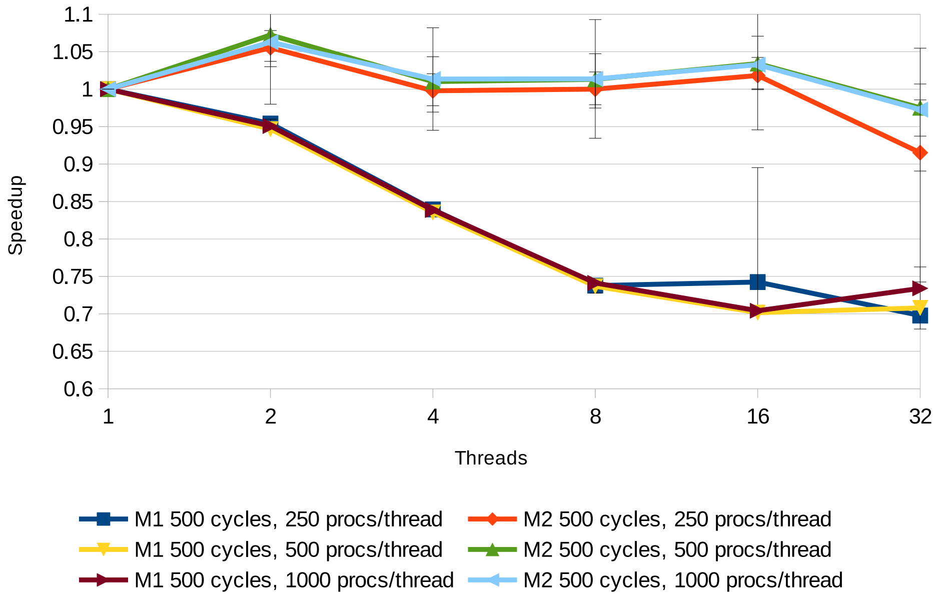

Figure 4.5 – The performance of the two parallelization models when performing a constant amount of work per thread.

4.4.1 Discussion

We can see that the continuous overhead associated with the atomic variable used in the worker list model is still clearly visible and appears to increase with the level of parallelization. The statically orchestrated model, on the other hand, produces much better results and appear to show weak scaling properties with speedups within +/- 0.05. The performance of this model, however appears to drop when going from 16 to 32 cores. This could be explained by the fact that we performed the benchmarks on a on a 2x16 core machine. This means, that CPU boundaries are crossed when we go from 16 to 32 cores.

Contrary to our expectations, the work-per-cycle ratio is not clearly visible in the results. The explanation for this may simply be that the work-per-cycle ratios that we used in the benchmark weren't sufficiently large. In the cycle dominated benchmark we used ratios of 10:1 and 1:10 respectively while in this benchmark we used, much smaller, ratios of 2:1, 1:1 and 1:2.

4.5 Uneven workloads

This benchmark, we attempt to confirm our conjecture that the work list model will perform better than the statically orchestrated model when executing networks consisting of processes with uneven workloads. This benchmark essentially repeats the cycle dominated benchmark except that we divide the processes on the ring evenly into two groups where one group performs exactly one fourth as much work

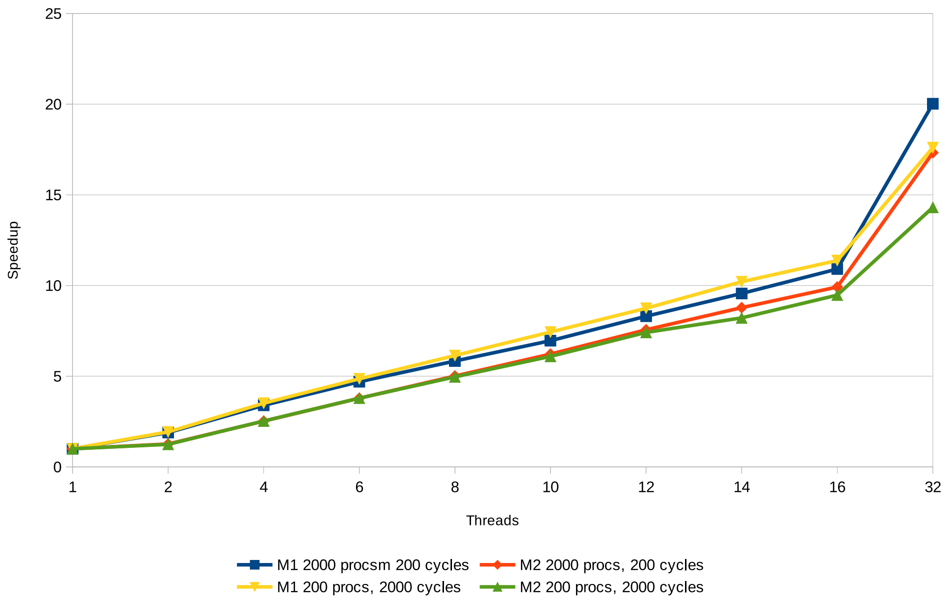

Figure 4.6 – The performance of the two parallelization models when executed on a network consisting of processes with a significant workload imbalance

as the other. So, we use the same process code as listing 2 except that half of the processes will only run for 2500 iterations instead of 10000. It's important to note that the processes in each of these groups are laid out sequentially on the ring, one after the other. The purpose of this orchestration, is to force the statically orchestrated model into its worst-case workload distribution.

The statically orchestrated model will thus spend a lot of time in the synchronization barrier while the work list model will be able to distribute load of the network much more evenly across its threads. This benchmark is indeed very artificial, but it serves the purpose of showing the strengths of the work list model

4.5.1 Discussion

This is the first benchmark where the work list model performs better than the statically orchestrated model so the fundamental conclusion is that our conjecture was confirmed. However, the difference between the two models aren't as significant as we expected. It remains to be seen whether the work list model would increase its advantage over the statically orchestrated model if we increased the workload imbalances between the processes even more but presumably, this would be the case.

This benchmark allows us to propose the interesting conjecture that if the light and heavy processes where placed on the ring (the benchmarked network) in a way that would be more optimal for the statically orchestrated model, then it would perform better that the work list model. The optimal organization would, in this case, probably be alternating order of light ant heavy processes but even randomizing the process ordering would presumably be an improvement over status quo.

So, once again, the continuous synchronization of the work list model proves disproportionately expensive.

4.6 Future works

BQueue with work stealing would be nice to implement, but as shown by relevant benchmark using “advanved” datastructures for the process queue comes at a significant cost. A way to implement work stealing without increasing the cost of getting a process from the queue is needed. Linearly pulling processes of a queue has a significant cache advantage. Using work stealing to pull from “the reverse” would reduce that advantage.

All benchmarks has been largely artificial since they have been targeted at exposing the advantages and disadvnatages of the properties of the execution models that we have proposed We need more real-world samples to know where to actually tune the performance of system

More benchmarks:

The results that we have shown, although reasonable, can not be easily explained by

4.6.1 One-shot process orchestration

In this model, we orchestrate the processes in our network as soon as possible after execution start and

4.6.2 Monte Carlo orchestration

In this approach, we simply randomize the order of the processes. The main advantage of this approach is that is computationally cheap compared to

4.6.3 Optimization-based orchestration

Another way to orchestrate the processes is to use a

4.6.4 Adaptive process orchestration

The benefits of using a oneshot orchestration approach diminishes when we execute process networks where the processes performs a variable amount of work per iteration. In these kinds of networks, CPU-core load distribution will gradually become uneven and suboptimal as the network execution progresses. In order to keep this from happening and maximize CPU-core utilization, we need to monitor process execution time and core idle time as the network execution progresses. This is what we refer to s adaptive orchestration. This approach, however introduces another trade-off that we need to consider. producing an

4.6.5 Adaptive Monte Carlo process orchestration

4.6.6 Adaptive Optimization-based process orchestration

Chapter 5

Conculsions

Bibliography

[1] David F Bacon, Rodric Rabbah, and Sunil Shukla. "FPGA Programming for the Masses". In: *Communications of the ACM* 56.4 (2013), pp. 56–63 (cit. on p. 5).

[2] Brian Vinter and Kenneth Skovhede. "Synchronous Message Exchange for Hardware Designs". In: *Communicating Process Architectures 2014* (2014) (cit. on pp. 6, 7).

[3] Minyoung Sung et al. "Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread". In: *Information processing letters* 84.4 (2002), pp. 221–225 (cit. on p. 9).

[4] Neil CC Brown and Peter H Welch. "An introduction to the Kent C++ CSP Library". In: *Communicating Process Architectures 2003* 61 (2003), pp. 139–156 (cit. on p. 9).

[5] Neil Deshpande, Erica Sponsler, and Nathaniel Weiss. "Analysis of the Go runtime scheduler". In: () (cit. on p. 9).

[6] ISO/IEC. *Working Draft, Standard for Programming anguage C++*. Feb. 28, 2011. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf (cit. on p. 17).

[7] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (Oct. 1974), pp. 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161. URL: http://doi.acm.org/10.1145/355620.361161 (cit. on p. 18).

[8] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012, pp. 46–47. ISBN: 978-0-12-383872-8 (cit. on p. 23).

[9] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers*. 2014 (cit. on p. 25).

[10] I.J.Bush and W.Smith. "The Weak Scaling of DL_POLY 3". In: (2014). URL: https://web.archive.org/web/20140307224104/http://www.stfc.ac.uk/cse/25052.aspx (cit. on p. 27).

.1 Compiling and Running

The project is built using GNU Autotools and thus can be built using the standard configure make command sequence. The standard make target will build the entire library including test cases.

The test cases are configured dynamically using the envirnomnet variables SME_SCHEDS, SME_ITERATIOSN, SME_PROCS amd SME_THREADS

Appendix A

Benchmark data

The tables below contain the raw results from our benchmarks.