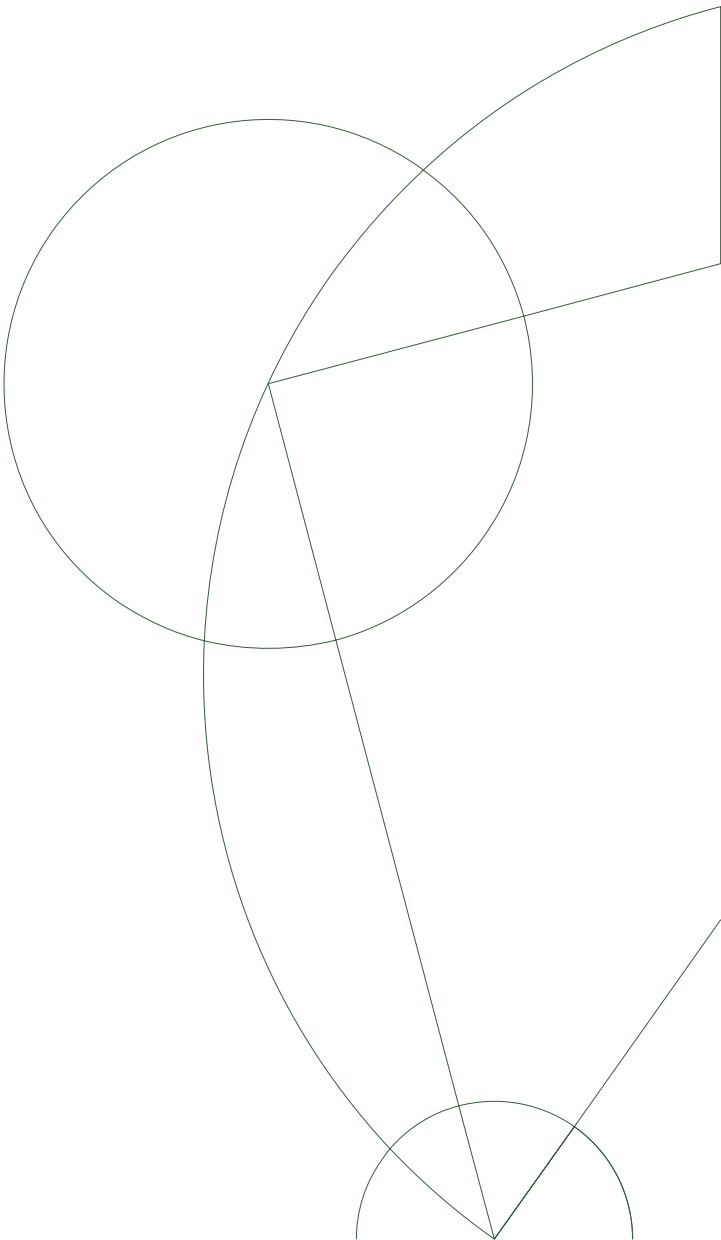
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

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 messaging framework similar to Concurrent Sequential Processes (CSP) aimed at modeling synchronous systems such as hardware. With the goal of faster execution of SME networks, various models for parallelized execution of SME networks have been designed, implemented and benchmarked.

We give an overview of the properties of SME and explain how they act in a parallelized environment. We then explain the details of our implementation and finally 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. However, one of our the models that we have implemented proved extremely successful and is capable of achieving near linear speedups.

Resumé

Synchronous Message Exchange (SME) er et Concurrent Sequential Processes (CSP) lignende messaging framework som er rettet imod modellering af synkrone systemer så som hardware. Med en målsætning om at muliggøre hurtigere eksekveringer af SME netværk, har vi designet, testet og implementeret forskellige modeller for parallelisering af SME modellen.

Vi vil give et overbliv over egenskaber ved SME modellen og hvordan de fungerer i et paralleliseret miljø. Vi vil derefter forklare detaljer ved vores implementering og endeligt vil vi undersøge ydelsen af vores implementering igennem forskellige målinger. Vores konklusion er at den mulige hastighedsforbedring i høj grad afhænger af hvilken type arbejde som bliver udført af de netværk vi kører. Dog har en af vores modeller vist sig yderst succesfuld og skalerer linært i forhold til antallet af processorer.

Contents

List of Figures

Chapter 1

Introduction

In this report, we describe a 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 explain the implementation process. Finally, we will show the benchmarks performed of our implementation before presenting our conclusions.

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 provide several advantages over using for processing work including a significantly improved performance-per-watt ratio, their prevalence are limited by the lack of tools which allows ordinary programmers to adapt their applications for FPGAs. Currently, 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].

In an attempt to improve this situation, a master's thesis[2] explored using PyCSP to define hardware designs and synthesize them into Hardware Description Languages. Since Concurrent Sequential Processes (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 [3].

The aim of this project is 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.

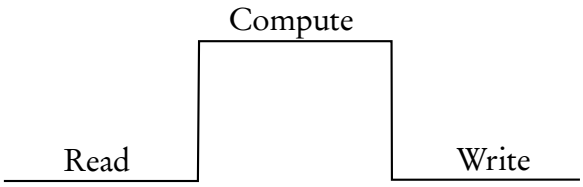


Figure 1.1 – The execution cycle of a SME process visualized as a hardware clock-cycle. Before every cycle data from a input bus is read into the process and after a cycle, data is written back to the gate. Reproduction of figure from [3]

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. A process execution cannot have any side effects, however, The internal state of a process is persistent between executions. Thus, the only way the execution of a process can alter the state of other processes is by bus communication. When we use the word "'process"' in the remainder of this report, this is what we refer to.

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 single 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 once". 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.

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

High level approaches to programming FPGAs are offered from at least two major FPGA manufacturers. Xilinx offers a language called Vivado-C as a part of their Vivado high Level Synthesis package which allows programs written in a C-like language to be synthesized into FPGAs. Another FPGA manufacturer, Altera, offers an OpenCL interface to their FPGAs

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. We will primarily focus on how the SME model can be parallelized.

2.1 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 [4] and memory cost. Furthermore, a much higher number of user-level threads can coexist on a system. Due to these limitations, implementing these kind 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[5] and the goroutines in the Go language[6]. 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.

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 the 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.

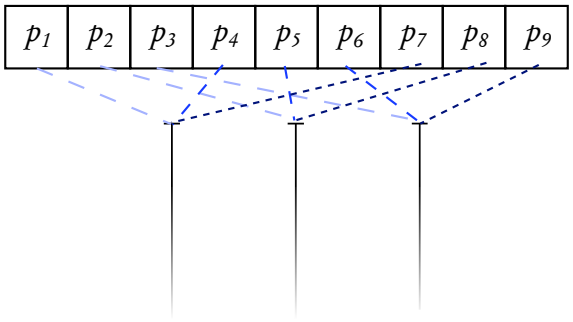


Figure 2.1 – Illustration of the work list model. The order of which processes are executed is shown as dashed arrows of increasing density

The basic idea that we base our design on is conceptually similar to a classic producer-consumer setup. In our case, the work “produced” are pointers to the processes in the network, and the consumers are the threads executing the processes. In this setup, a process is executed by a simple function call. Once again, this simplifies our implementation compared to an equivalent implementation using user-level threads.

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 models to achieve this goal under different circumstances.

2.1.1 Work list model

This approach most closely matches the aforementioned producer-consumer model where we have a number of workers which takes tasks off a circular queue and executes them. This model executes processes in a "round-robin"-like manner allowing it to "interlace" processes with different computational loads. Thus, we expect that this model to produce lower overall execution times for networks with uneven workloads (figure 2.1). 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.

2.1.2 Static orchestration model

In this model, we assign separate queues to each thread of execution and distribute (“orchestrate”) the processes amongst them (figure 2.2). 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

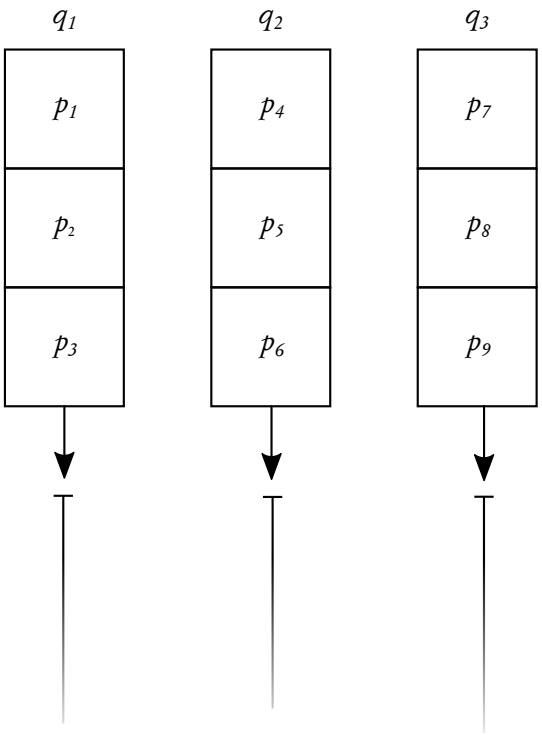


Figure 2.2 – Illustration of the statically orchestrated mode shown the separate processes queues q_n assigned to each thread. Processes are denoted p_n.

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 thread and large processes to another, the thread 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.

2.1.3 Comparison

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

2.1.4 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

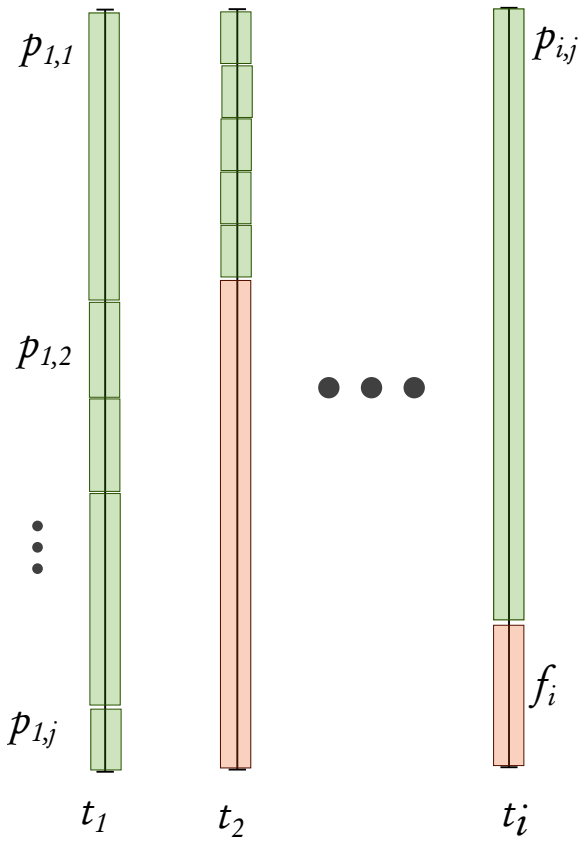


Figure 2.3 – Example of suboptimal distribution of processes across processing threads. Green blocks represents processes while red blocks represents thread idle time. Processes are named $p_{i,j}$ where i is the number of the thread the process has been assigned to and f_i is the combined idle time for each thread. Threads are named t_i.

of our threading model can be seen in figure 2.3. 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 reordering the processes.

2.2 Managing execution flow

In order to keep track of the execution of an SME network, 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 completed

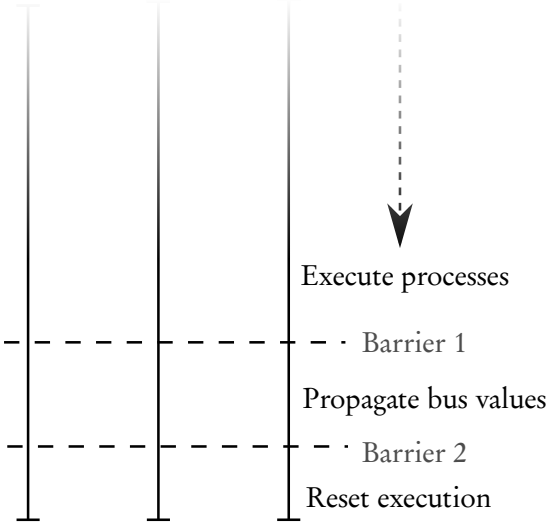


Figure 2.4 – Location of the synchronization barriers of the SME execution cycle relative to the phases of the cycle

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 other challenge 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. Due tho the shared nothing property of SME, the problem of executing the actual processes is embarrassingly parallel, however, the need for synchronization makes it less so. Therefore, the time spent on synchronization will significantly impact the overall performance of our network.

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. The second is after the bus value propagation, before the next cycle starts.

Inserting the barriers into the process execution flow relieves us from having to keep track of the number of processes that has been executed since we know that all processes int he network has been executed once we hit the first barrier (figure 2.4).

2.2.1 Synchronizing cycles

We will start this section by defining *process executors* as the functions running in each execution thread that are responsible for acquiring and executing the processes of the network.

A possible way of synchronizing the cycles at the previously mentioned barriers would be to perform the network state tracking, inside the process executors. Such state tracking include counting the number of executed cycles and executed processes. The problem with this approach is that it adds a fixed computational cost

to each process execution. This fixed cost would primarily arise from the fact that, we would need to keep the state of the network execution in a globally shared state. Global state in a parallel environment is problematic since it needs to be protected by locks in order to support concurrent access safely.

As an alternative to this, we propose a method for managing the execution of a network without keeping any global state. Our method essentially creates a self-managing execution network which is based on special processes.

The idea is that we, by placing special processes at the end of our process execution queues, can let these processes control the execution flow. These special processes have the ability to control the global execution state of the network when they are executed by a thread. We insert the following special processes into the process queues:

Locker This process blocks a thread until it gets released by a Syncer or Reset process.

Syncer blocks a thread until all threads have entered the barrier. This process is only executed by one thread.

BusStep propagates bus values for a preassigned range of busses. Every thread executes one of these processes, i.e., they enable us to also parallelize bus-value propagation.

Reset fills the same basic role as the Syncer process (and is used in conjunction with the aforementioned Locker processes) except that it moves the queue pointers back to the beginning so that a new cycle can start. This process is also responsible for terminating the network execution since it will do nothing and let the process executors fall-through to the Terminate processes if we have no more cycles to execute.

Terminate This final process is placed at the very end of the process queues of each thread. It will cause any thread that executes it to terminate unconditionally These processes are only reached if the Reset process doesn't reset the queue pointers.

We arrange the processes such that $n-1$ threads will be blocked by the Locker processes until the final thread, 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. In the static orchestration model, the Reset process moves the individual queue pointers of all the threads back to their respective beginnings. In the worker list model it will move the global queue pointer back to the start of the queue. The organization of these processes on the statically orchestrated model can be seen in figure 2.5. In the work list model, a similar organization is used except that all the

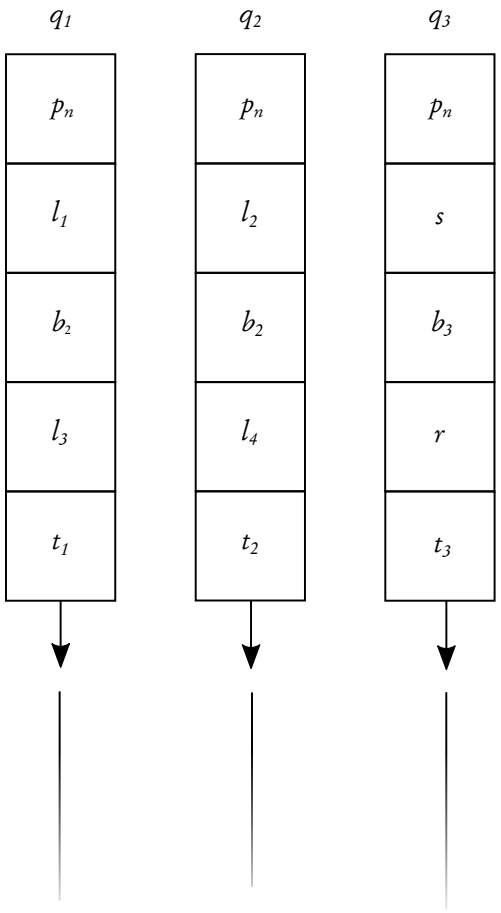


Figure 2.5 – The placement of special purpose execution controlling processes in the process' queues of the statically orchestrated model. The denotations of the figure are the following: p is a regular process of the network. l is a Locker process, s is a Syncer process, b is a BusStep process, r is a Reset process and finally, t are the Terminate processes.

special processes of the same type are laid out sequentially since this model only have one process queue.

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

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 [7] revision of the language allows for easy access to features that were previously hard to use. Examples of such, are the functions provided by the `<atomics>` header which enables the use of atomic instructions and enforced memory ordering without the need for inlining assembly instructions. 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 they provide a natural enclosure of the state maintained by a SME process.

We have implemented a library that is meant to be imported by applications implementing systems using the SME model.

3.1 Initial implementation

The initial version of the C++SME code was purely single-threaded and was implemented to play around with the C++ APIs and defining the user visible APIs for defining SME networks. This implementation served as a proof-of-concept which allowed us to gain experience and familiarity with the SME model. We therefore used as many high-level containers as possible such as `vector` and `map`

3.2 Multithreaded implementation

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 was that we wanted to move away from using high level containers and represent our queues using arrays. Since one of the goals of our design is to minimize the overhead related to executing processes, we want to use the list data structure which allow elements to be accessed in sequential order as fast as possible. In our case, this

is an array. More advanced data structures such as linked lists exhibits poor cache locality which slows element accesses.

3.2.1 Benchmarking support

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. We hadn't initially foreseen the need for these features since we wanted to statically generate the code for benchmark networks by using code generation scripts. This method, however, proved to be infeasible due to compilation times. Even relatively small networks consisting of 5000 processes, took in excess of one hour to compile. This problem persisted after disabling optimization features in GCC known for increasing compilation time.

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.

3.3 Queue implementation

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 describe some aspects of the actual implementation of the queues

3.3.1 Locking mechanisms

As specified by our design, we need locking in two areas of our implementation. The first, is to enforce the barriers in the cycle. The second, which is specific to the work list model, is to support concurrent accesses to the process queues.

We initially attempted to create all locks entirely by using atomic variables. After this effort turned out to be (partially) unsuccessful, we switched to using more traditional synchronization primitives.

Referring to our description of the special purpose processes in section 2.2.1, we ended up implementing the Locker, Syncer and Reset processes using Condition Variables [8]. This choice was made since Condition Variables are readily available in C++11 and performs the task that we require. Whenever a Locker process is executed by a thread it will block, waiting for the mutex associated with the condition variable to be released. The Syncer and Reset processes will release the waiting threads on the condition that $n - 1$ threads has entered the barrier.

In the work list model we use an atomic integer as a pointer to the current process in the execution queue. The atomic integer allows safe concurrent access to the process execution queue.

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)

3.3.2 Distributing processes across threads

In the static orchestration model, we need to distribute the processes in the network across threads. While this, in isolation, isn't a very interesting implementation detail, the specific of the algorithm that we use will help to explain some anomalies seen in our upcoming 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 thread. The example in table 3.1 shows what this unequal distribution looks like in practice.

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

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 computationally light and heavy processes.

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.

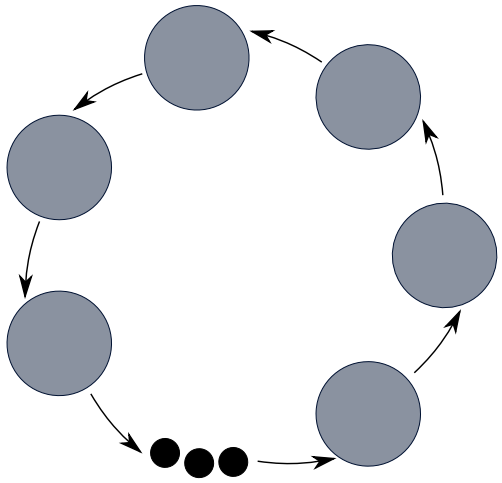


Figure 4.1 – Illustration showing the layout of the network used for benchmarking. The blue circles represents processes and the arrows represents busses

4.1 Testing methodology

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 have the following specs:

- 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

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

where S is the achieved speedup, T_{old} is the original (pre-improvement) speed and T_{new} is the new (post-improvement) speed [9].

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 and 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.

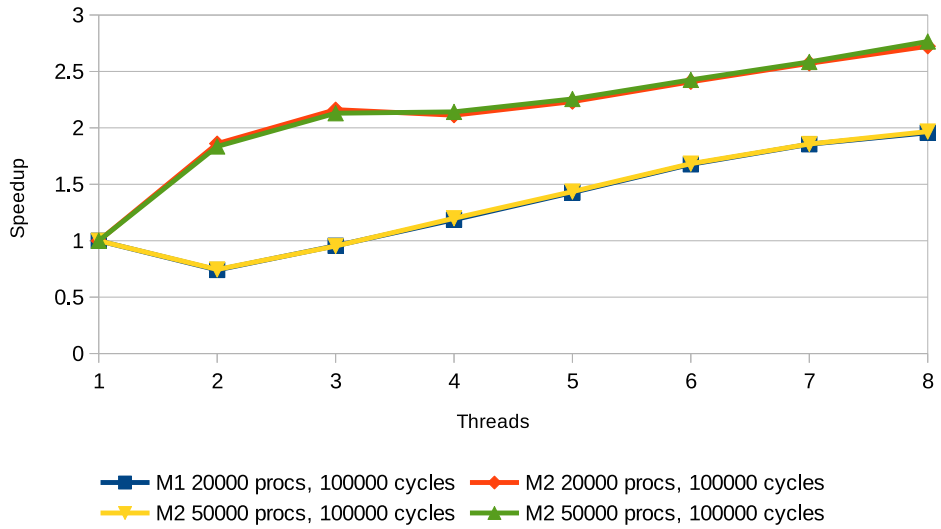


Figure 4.2 – Graph showing the speedup of a SME network consisting of 20000 and 50000 processes respectively when executed for 100000 cycles on an Intel Xeon CPU

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 more 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.

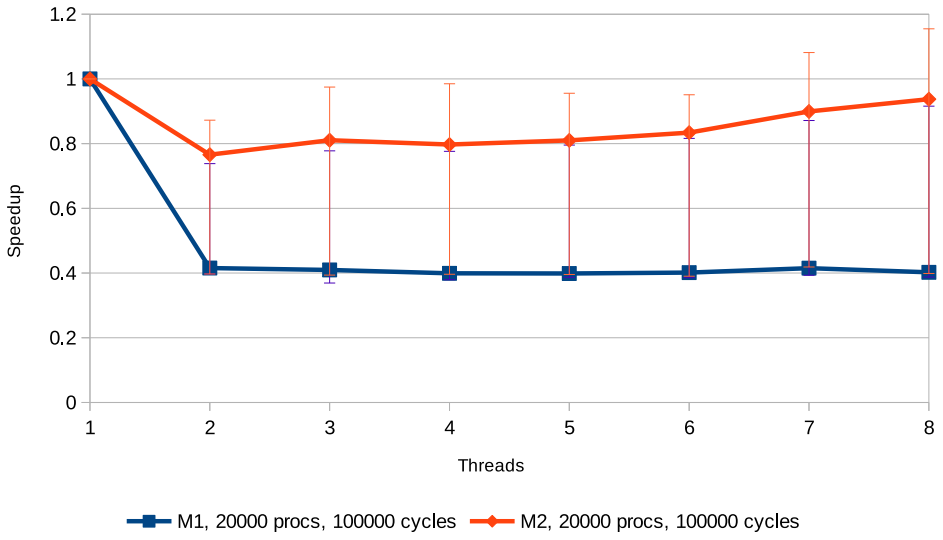


Figure 4.3 – Benchmark results for a network of 20000 processes running for 100000 iterations on the AMD cluster

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.

In addition to the time required by the atomic increment, which is performed before every process execution, we also need to wait for the threads to sync up at the end of a cycle. This means that this model is doubly penalized. What is slightly surprising, 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.

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[10]. 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.

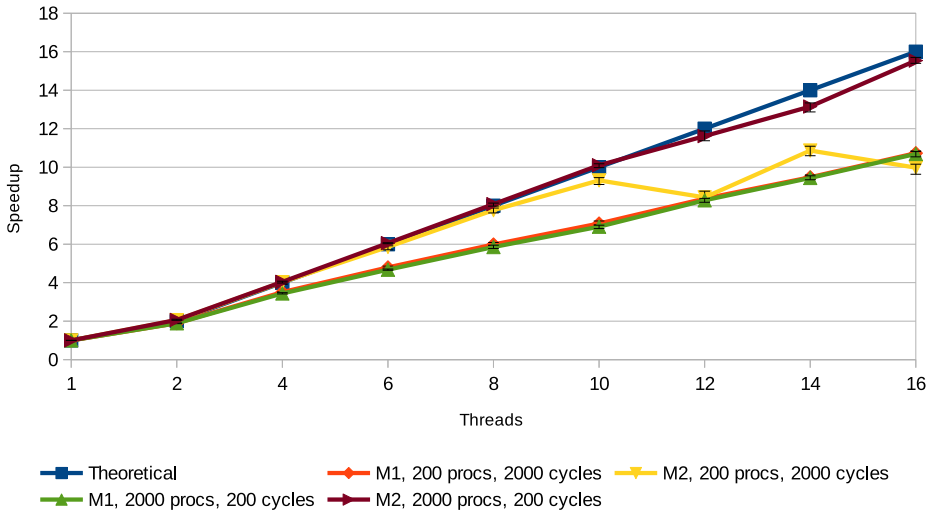


Figure 4.4 – Benchmark results for a network of 20000 processes running for 100000 iterations on the AMD cluster

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 is that our syncronization mechanisms is significantly less optimized on the AMD CPU compared to the Intel CPU.

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 Compute 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 be-

ing done with the value calculated, however, as long as our workload isn't being optimized away at compilation time this is irrelevant.

The size of the workload and the number of nodes were chosen such that the running times of the benchmarks would be reasonable. This is important, since benchmarks that run for a very short time produce less dependable results.

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 compute 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 better than when benchmarking a problem which to a larger extent is limited by memory bandwidth and/or CPU-cache misses. Additional benchmarking is needed to determine the performance of networks that depends on memory accesses;

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[11] 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

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 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 compute 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.

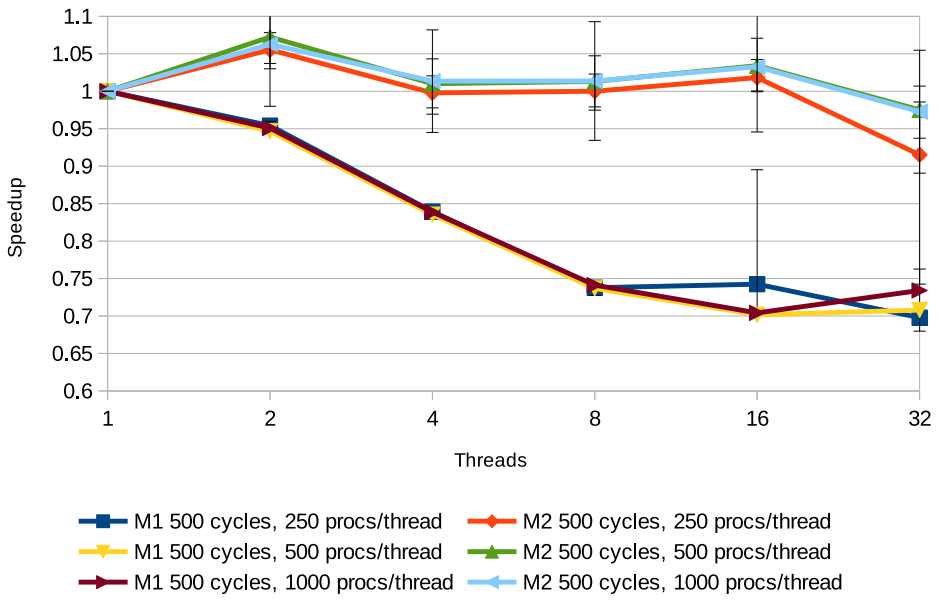


Figure 4.5 – The performance of the two parallelization models when performing a constant amount of work per thread.

4.5 Uneven workloads

With 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 compute 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 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

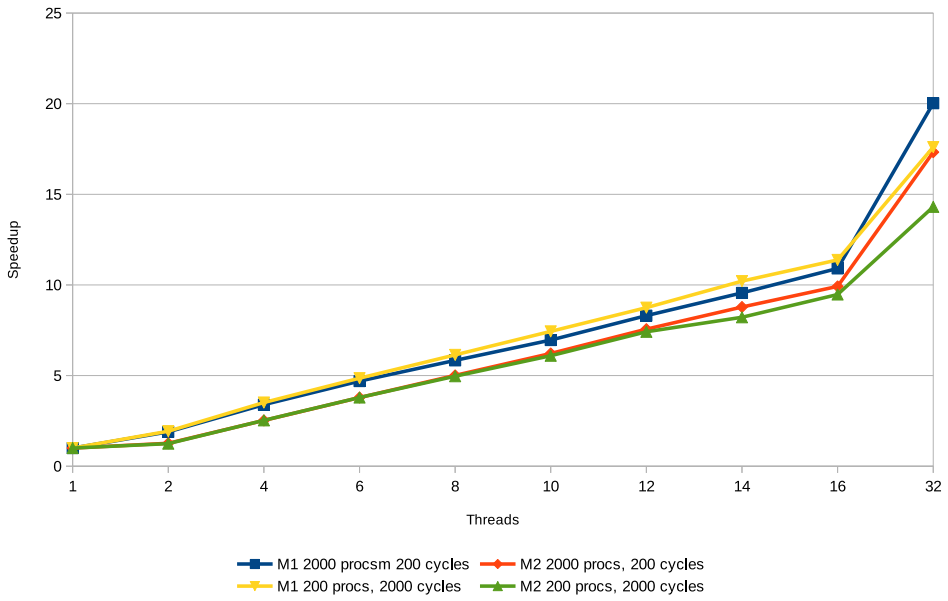


Figure 4.6 – The performance of the two parallelization models when executed on a network consisting of processes with a significant workload imbalance

its advantage over the statically orchestrated model even more if we increased the workload imbalances. Based on the current results, however, it seems likely that 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 and heavy processes but even randomizing the process ordering would be an improvement over status quo.

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

4.6 Future works

As these benchmarks has shown, it is possible to achieve significant speedups when executing SME in a parallelized environment. However, a lot of what we have concluded from the benchmarks is simply qualified guesses based on incomplete knowledge. Therefore, more in-depth testing is needed in order to confirm our conclusions. In particular, profiling our implementation using tools such as `gperftools` and `callgrind` would yield the information needed to focus further optimization efforts appropriately and confirm our current conclusions.

It should also be clears, that all of the benchmarks that we have performed has

been largely artificial since they have been targeted at exposing the advantages and disadvantages of the properties of the execution models that we have proposed. However, they are probably a poor approximation of the characteristics shown by actual applications implemented using the SME model. Given the recent conception of the SME model, such applications are currently hard to come by, but benchmarking them would help deciding the future directions of this project.

An obvious addition to the statically orchestrated model would be to add a work-stealing[12] scheme such that idle threads could fill up their work queues with work from adjacent non-empty queues. However as our benchmarks has shown, that imposing the cost of locking on access to the work queues (as in the work list model), severely reduces the overall performance of the execution. This makes the introduction of a work stealing scheme challenging. Furthermore, linearly reading items from an array makes efficient use of CPU cache and memory prefetching advantage. Using work stealing to pull from "the reverse" would reduce that advantage.

Another possibility is to dynamically orchestrate processes across the queues based on their computational load. Such a system could be implemented by performing a "'trail run"', measuring the time each process uses to perform a single execution. Then the processes could be orchestrated across CPU threads in a way that minimizes CPU core idle times.

Chapter 5

Conculsions

We have successfully showed that it is possible to introduce a parallelization model for SME which delivers significant speedups in environments where sufficient hardware parallelism is available.
The statically orchestrated model, in particular, proved to be extremely successful and is capable of achieving near-linear speedups for certain workloads. These results were made possible through our concept of letting the execution flow "'self manage"' by using special processes.
Our work list model, on the other hand, generally delivered a disappointing performance. We attribute this to the synchronization mechanism used which proved more expensive than anticipated.
Based on the benchmarks that we have performed, we conclude that there is a large difference between the speedups achieved by different workloads. This raises the question of whether our results are representative for SME networks implementing real-world systems.
Looking ahead, our results opens up increased use of the SME model for simulating hardware designs since we, through parallelization has been able to significantly decrease the time required to execute SME networks.
Overall, our work has laid a foundation for additional exploration into making parallelized SME implementations. Our results and conclusions sets a clear direction for future work aiming to improve this implementation even more.

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] Esben Skaarup and Andreas Frisch. *Generation of fpga hardware specifications from pycsp networks*. 2014 (cit. on p. 5).

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

[4] 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).

[5] 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).

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

[7] 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).

[8] 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).

[9] 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. 22).

[10] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers*. 2014 (cit. on p. 24).

[11] 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).

[12] Robert D Blumofe and Charles E Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748 (cit. on p. 30).

Appendix A

Benchmark data

The following contains the raw output data from our benchmarks
The data below follows the following format (from left to right)

- The execution model used. CQueue refers to our work list model, while BQueue refers to our statically orchestrated model.
- The number of threads used
- The number of processes
- The number of cycles
- Execution times in seconds

A.1 Intel machine synchronization dominated results

A.1.1 50000 processes

```
void ThreadedRun::start() [with T = CQueue] 1 50000 100000 212.424
void ThreadedRun::start() [with T = BQueue] 1 50000 100000 170.835
void ThreadedRun::start() [with T = CQueue] 2 50000 100000 284.392
void ThreadedRun::start() [with T = BQueue] 2 50000 100000 91.1839
void ThreadedRun::start() [with T = CQueue] 3 50000 100000 223.621
void ThreadedRun::start() [with T = BQueue] 3 50000 100000 86.0699
void ThreadedRun::start() [with T = CQueue] 4 50000 100000 176.322
void ThreadedRun::start() [with T = BQueue] 4 50000 100000 82.8004
void ThreadedRun::start() [with T = CQueue] 5 50000 100000 148.114
void ThreadedRun::start() [with T = BQueue] 5 50000 100000 76.0188
void ThreadedRun::start() [with T = CQueue] 6 50000 100000 126.422
void ThreadedRun::start() [with T = BQueue] 6 50000 100000 70.7396
void ThreadedRun::start() [with T = CQueue] 7 50000 100000 114.645
void ThreadedRun::start() [with T = BQueue] 7 50000 100000 66.2468
void ThreadedRun::start() [with T = CQueue] 8 50000 100000 107.442
void ThreadedRun::start() [with T = BQueue] 8 50000 100000 61.7085
```

```
void ThreadedRun::start() [with T = CQueue] 1 50000 100000 212.382
void ThreadedRun::start() [with T = BQueue] 1 50000 100000 171.201
void ThreadedRun::start() [with T = CQueue] 2 50000 100000 284.893
void ThreadedRun::start() [with T = BQueue] 2 50000 100000 91.8218
void ThreadedRun::start() [with T = CQueue] 3 50000 100000 220.363
void ThreadedRun::start() [with T = BQueue] 3 50000 100000 77.2614
void ThreadedRun::start() [with T = CQueue] 4 50000 100000 177.443
void ThreadedRun::start() [with T = BQueue] 4 50000 100000 77.0582
void ThreadedRun::start() [with T = CQueue] 5 50000 100000 148.064
void ThreadedRun::start() [with T = BQueue] 5 50000 100000 75.7833
void ThreadedRun::start() [with T = CQueue] 6 50000 100000 126.167
void ThreadedRun::start() [with T = BQueue] 6 50000 100000 70.411
void ThreadedRun::start() [with T = CQueue] 7 50000 100000 113.949
void ThreadedRun::start() [with T = BQueue] 7 50000 100000 66.3753
void ThreadedRun::start() [with T = CQueue] 8 50000 100000 108.216
void ThreadedRun::start() [with T = BQueue] 8 50000 100000 62.1015
void ThreadedRun::start() [with T = CQueue] 1 50000 100000 212.672
void ThreadedRun::start() [with T = BQueue] 1 50000 100000 171.327
void ThreadedRun::start() [with T = CQueue] 2 50000 100000 284.022
void ThreadedRun::start() [with T = BQueue] 2 50000 100000 91.5474
void ThreadedRun::start() [with T = CQueue] 3 50000 100000 220.238
void ThreadedRun::start() [with T = BQueue] 3 50000 100000 79.7654
void ThreadedRun::start() [with T = CQueue] 4 50000 100000 179.148
void ThreadedRun::start() [with T = BQueue] 4 50000 100000 80.9521
void ThreadedRun::start() [with T = CQueue] 5 50000 100000 148.603
void ThreadedRun::start() [with T = BQueue] 5 50000 100000 75.7782
void ThreadedRun::start() [with T = CQueue] 6 50000 100000 126.538
void ThreadedRun::start() [with T = BQueue] 6 50000 100000 70.2436
void ThreadedRun::start() [with T = CQueue] 7 50000 100000 114.802
void ThreadedRun::start() [with T = BQueue] 7 50000 100000 66.2705
void ThreadedRun::start() [with T = CQueue] 8 50000 100000 108.466
void ThreadedRun::start() [with T = BQueue] 8 50000 100000 62.2585
void ThreadedRun::start() [with T = CQueue] 1 50000 100000 213.262
void ThreadedRun::start() [with T = BQueue] 1 50000 100000 171.148
void ThreadedRun::start() [with T = CQueue] 2 50000 100000 286.483
void ThreadedRun::start() [with T = BQueue] 2 50000 100000 91.7057
void ThreadedRun::start() [with T = CQueue] 3 50000 100000 224.107
void ThreadedRun::start() [with T = BQueue] 3 50000 100000 84.1722
void ThreadedRun::start() [with T = CQueue] 4 50000 100000 175.741
void ThreadedRun::start() [with T = BQueue] 4 50000 100000 79.3511
void ThreadedRun::start() [with T = CQueue] 5 50000 100000 148.211
void ThreadedRun::start() [with T = BQueue] 5 50000 100000 76.0164
void ThreadedRun::start() [with T = CQueue] 6 50000 100000 126.814
void ThreadedRun::start() [with T = BQueue] 6 50000 100000 70.4807
void ThreadedRun::start() [with T = CQueue] 7 50000 100000 114.888
void ThreadedRun::start() [with T = BQueue] 7 50000 100000 66.2033
void ThreadedRun::start() [with T = CQueue] 8 50000 100000 108.507
void ThreadedRun::start() [with T = BQueue] 8 50000 100000 61.7935
void ThreadedRun::start() [with T = CQueue] 1 50000 100000 213.407
```

```
void ThreadedRun::start() [with T = BQueue] 1 50000 100000 171.772
void ThreadedRun::start() [with T = CQueue] 2 50000 100000 288.414
void ThreadedRun::start() [with T = BQueue] 2 50000 100000 100.939
void ThreadedRun::start() [with T = CQueue] 3 50000 100000 228.09
void ThreadedRun::start() [with T = BQueue] 3 50000 100000 75.7844
void ThreadedRun::start() [with T = CQueue] 4 50000 100000 179.198
void ThreadedRun::start() [with T = BQueue] 4 50000 100000 79.8707
void ThreadedRun::start() [with T = CQueue] 5 50000 100000 148.532
void ThreadedRun::start() [with T = BQueue] 5 50000 100000 75.8899
void ThreadedRun::start() [with T = CQueue] 6 50000 100000 126.364
void ThreadedRun::start() [with T = BQueue] 6 50000 100000 71.1819
void ThreadedRun::start() [with T = CQueue] 7 50000 100000 114.677
void ThreadedRun::start() [with T = BQueue] 7 50000 100000 66.2334
void ThreadedRun::start() [with T = CQueue] 8 50000 100000 108.338
void ThreadedRun::start() [with T = BQueue] 8 50000 100000 61.7831
```

A.1.2 20000 processes

```
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 84.4959
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 67.6345
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 113.879
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 36.5261
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 86.479
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 33.3521
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 73.3068
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 32.1204
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 59.4373
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 30.3505
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 50.604
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 28.5866
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 45.5982
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 26.4238
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.3921
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.7046
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 42.8081
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.7037
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 84.5724
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 68.2153
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 115.03
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 36.7033
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 90.0435
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 30.8818
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 71.6077
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 32.6052
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 58.9442
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 30.5422
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 50.3241
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 28.2932
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 45.7375
```

```
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 26.4915
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.1658
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.641
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 42.6855
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.634
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 84.6339
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 68.1926
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 114.451
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 36.8519
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 87.9081
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 30.5932
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 69.9262
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 32.9747
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 59.2815
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 30.6583
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 50.5024
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 27.966
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 45.6295
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 26.3479
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.1922
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 26.2256
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.3405
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.7107
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 84.6271
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 68.137
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 113.979
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 36.4507
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 88.8915
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 30.66
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 69.3957
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 32.3788
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 59.264
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 30.5412
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 50.5185
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 28.3272
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 45.4629
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 26.607
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.1818
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.7088
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 84.8217
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 68.286
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 114.136
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 36.4717
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 89.4556
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 32.2373
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 72.599
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 31.1283
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 59.6146
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 30.3972
```

```
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 50.34
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 28.1883
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 45.7168
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 26.5084
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 43.3032
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 24.8148
```

A.2 AMD cluster synchronization dominated results

```
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 189.001
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 167.628
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 460.95
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 196.712
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 491.463
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 184.611
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 484.111
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 183.055
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 459.277
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 204.2
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 484.126
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 179.548
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 466.902
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 166.766
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 185.214
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 170.526
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 183.591
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 166.263
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 462.447
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 197.089
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 458.156
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 194.759
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 484.055
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 203.366
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 474.525
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 187.098
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 476.062
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 178.376
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 467.122
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 168.038
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 473.722
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 157.414
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 185.201
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 166.066
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 460.237
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 196.726
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 501.734
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 188.781
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 485.103
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 200.208
```

```
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 461.502
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 183.025
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 441.836
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 174.6
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 417.909
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 153.542
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 475.502
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 157.565
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 185.693
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 166.324
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 459.646
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 199.016
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 184.584
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 166.358
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 464.964
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 196.999
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 468.952
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 183.169
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 465.889
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 200.086
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 466.793
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 203.837
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 473.069
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 181.364
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 440.795
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 167.11
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 463.073
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 158.667
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 195.65
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 163.176
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 441.518
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 187.013
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 442.487
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 171.525
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 468.158
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 165.644
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 473.792
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 170.741
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 490.699
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 171.851
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 484.84
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 160.438
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 475.297
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 159.51
void ThreadedRun::start() [with T = CQueue] 1 20000 100000 195.341
void ThreadedRun::start() [with T = BQueue] 1 20000 100000 161.414
void ThreadedRun::start() [with T = CQueue] 2 20000 100000 446.008
void ThreadedRun::start() [with T = BQueue] 2 20000 100000 185.061
void ThreadedRun::start() [with T = CQueue] 3 20000 100000 441.42
```

```
void ThreadedRun::start() [with T = BQueue] 3 20000 100000 165.584
void ThreadedRun::start() [with T = CQueue] 4 20000 100000 464.954
void ThreadedRun::start() [with T = BQueue] 4 20000 100000 168.502
void ThreadedRun::start() [with T = CQueue] 5 20000 100000 493.046
void ThreadedRun::start() [with T = BQueue] 5 20000 100000 178.766
void ThreadedRun::start() [with T = CQueue] 6 20000 100000 473.186
void ThreadedRun::start() [with T = BQueue] 6 20000 100000 170.399
void ThreadedRun::start() [with T = CQueue] 7 20000 100000 468.256
void ThreadedRun::start() [with T = BQueue] 7 20000 100000 162.991
void ThreadedRun::start() [with T = CQueue] 8 20000 100000 461.017
void ThreadedRun::start() [with T = BQueue] 8 20000 100000 139.761
```

A.3 Compute dominated networks

```
void ThreadedRun::start() [with T = CQueue] 1 200 2000 131.193
void ThreadedRun::start() [with T = BQueue] 1 200 2000 141.701
void ThreadedRun::start() [with T = CQueue] 2 200 2000 70.1725
void ThreadedRun::start() [with T = BQueue] 2 200 2000 67.4685
void ThreadedRun::start() [with T = CQueue] 4 200 2000 38.1573
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.7742
void ThreadedRun::start() [with T = CQueue] 6 200 2000 28.1738
void ThreadedRun::start() [with T = BQueue] 6 200 2000 23.3185
void ThreadedRun::start() [with T = CQueue] 8 200 2000 22.7004
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.4136
void ThreadedRun::start() [with T = CQueue] 10 200 2000 19.2553
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.921
void ThreadedRun::start() [with T = CQueue] 12 200 2000 16.0422
void ThreadedRun::start() [with T = BQueue] 12 200 2000 12.2933
void ThreadedRun::start() [with T = CQueue] 14 200 2000 14.0348
void ThreadedRun::start() [with T = BQueue] 14 200 2000 10.7561
void ThreadedRun::start() [with T = CQueue] 16 200 2000 12.4076
void ThreadedRun::start() [with T = BQueue] 16 200 2000 9.024
void ThreadedRun::start() [with T = CQueue] 1 200 2000 124.565
void ThreadedRun::start() [with T = BQueue] 1 200 2000 132.115
void ThreadedRun::start() [with T = CQueue] 2 200 2000 65.7137
void ThreadedRun::start() [with T = BQueue] 2 200 2000 64.6248
void ThreadedRun::start() [with T = CQueue] 4 200 2000 36.2324
void ThreadedRun::start() [with T = BQueue] 4 200 2000 32.5019
void ThreadedRun::start() [with T = CQueue] 6 200 2000 26.5717
void ThreadedRun::start() [with T = BQueue] 6 200 2000 21.9181
void ThreadedRun::start() [with T = CQueue] 8 200 2000 21.203
void ThreadedRun::start() [with T = BQueue] 8 200 2000 16.3755
void ThreadedRun::start() [with T = CQueue] 10 200 2000 17.8286
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.0131
void ThreadedRun::start() [with T = CQueue] 12 200 2000 15.1402
void ThreadedRun::start() [with T = BQueue] 12 200 2000 11.1163
void ThreadedRun::start() [with T = CQueue] 14 200 2000 13.0374
void ThreadedRun::start() [with T = BQueue] 14 200 2000 10.0523
void ThreadedRun::start() [with T = CQueue] 16 200 2000 11.5755
```

```
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.4984
void ThreadedRun::start() [with T = CQueue] 1 200 2000 133.56
void ThreadedRun::start() [with T = BQueue] 1 200 2000 140.349
void ThreadedRun::start() [with T = CQueue] 2 200 2000 70.2673
void ThreadedRun::start() [with T = BQueue] 2 200 2000 68.5057
void ThreadedRun::start() [with T = CQueue] 4 200 2000 38.366
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.8923
void ThreadedRun::start() [with T = CQueue] 6 200 2000 28.5655
void ThreadedRun::start() [with T = BQueue] 6 200 2000 23.0695
void ThreadedRun::start() [with T = CQueue] 8 200 2000 22.4444
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.2889
void ThreadedRun::start() [with T = CQueue] 10 200 2000 19.2034
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.9215
void ThreadedRun::start() [with T = CQueue] 12 200 2000 16.0043
void ThreadedRun::start() [with T = BQueue] 12 200 2000 12.0004
void ThreadedRun::start() [with T = CQueue] 14 200 2000 14.056
void ThreadedRun::start() [with T = BQueue] 14 200 2000 10.6136
void ThreadedRun::start() [with T = CQueue] 16 200 2000 12.4186
void ThreadedRun::start() [with T = BQueue] 16 200 2000 9.07162
void ThreadedRun::start() [with T = CQueue] 1 200 2000 131.186
void ThreadedRun::start() [with T = BQueue] 1 200 2000 140.915
void ThreadedRun::start() [with T = CQueue] 2 200 2000 70.0242
void ThreadedRun::start() [with T = BQueue] 2 200 2000 67.969
void ThreadedRun::start() [with T = CQueue] 4 200 2000 38.2683
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.6854
void ThreadedRun::start() [with T = CQueue] 6 200 2000 28.2442
void ThreadedRun::start() [with T = BQueue] 6 200 2000 23.2757
void ThreadedRun::start() [with T = CQueue] 8 200 2000 22.631
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.39
void ThreadedRun::start() [with T = CQueue] 10 200 2000 19.1543
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.9297
void ThreadedRun::start() [with T = CQueue] 12 200 2000 15.8736
void ThreadedRun::start() [with T = BQueue] 12 200 2000 12.1548
void ThreadedRun::start() [with T = CQueue] 14 200 2000 14.0129
void ThreadedRun::start() [with T = BQueue] 14 200 2000 10.5633
void ThreadedRun::start() [with T = CQueue] 16 200 2000 12.4489
void ThreadedRun::start() [with T = BQueue] 16 200 2000 9.04801
void ThreadedRun::start() [with T = CQueue] 1 200 2000 133.12
void ThreadedRun::start() [with T = BQueue] 1 200 2000 139.096
void ThreadedRun::start() [with T = CQueue] 2 200 2000 69.8381
void ThreadedRun::start() [with T = BQueue] 2 200 2000 67.5393
void ThreadedRun::start() [with T = CQueue] 4 200 2000 38.9875
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.9219
void ThreadedRun::start() [with T = CQueue] 6 200 2000 28.2791
void ThreadedRun::start() [with T = BQueue] 6 200 2000 23.0138
void ThreadedRun::start() [with T = CQueue] 8 200 2000 22.6845
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.4353
void ThreadedRun::start() [with T = CQueue] 10 200 2000 19.1454
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.9396
```

```
void ThreadedRun::start() [with T = CQueue] 12 200 2000 15.8852
void ThreadedRun::start() [with T = BQueue] 12 200 2000 12.2269
void ThreadedRun::start() [with T = CQueue] 14 200 2000 14.08
void ThreadedRun::start() [with T = BQueue] 14 200 2000 10.8055
void ThreadedRun::start() [with T = CQueue] 16 200 2000 12.3124
void ThreadedRun::start() [with T = BQueue] 16 200 2000 9.03456
void ThreadedRun::start() [with T = CQueue] 1 2000 200 127.186
void ThreadedRun::start() [with T = BQueue] 1 2000 200 129.666
void ThreadedRun::start() [with T = CQueue] 2 2000 200 66.3428
void ThreadedRun::start() [with T = BQueue] 2 2000 200 65.0229
void ThreadedRun::start() [with T = CQueue] 4 2000 200 36.1266
void ThreadedRun::start() [with T = BQueue] 4 2000 200 33.1527
void ThreadedRun::start() [with T = CQueue] 6 2000 200 26.1769
void ThreadedRun::start() [with T = BQueue] 6 2000 200 22.7047
void ThreadedRun::start() [with T = CQueue] 8 2000 200 20.8823
void ThreadedRun::start() [with T = BQueue] 8 2000 200 17.0051
void ThreadedRun::start() [with T = CQueue] 10 2000 200 17.6507
void ThreadedRun::start() [with T = BQueue] 10 2000 200 14.2438
void ThreadedRun::start() [with T = CQueue] 12 2000 200 15.1262
void ThreadedRun::start() [with T = BQueue] 12 2000 200 15.8068
void ThreadedRun::start() [with T = CQueue] 14 2000 200 13.2476
void ThreadedRun::start() [with T = BQueue] 14 2000 200 12.237
void ThreadedRun::start() [with T = CQueue] 16 2000 200 11.7258
void ThreadedRun::start() [with T = BQueue] 16 2000 200 13.4589
void ThreadedRun::start() [with T = CQueue] 1 2000 200 133.216
void ThreadedRun::start() [with T = BQueue] 1 2000 200 139.998
void ThreadedRun::start() [with T = CQueue] 2 2000 200 70.6244
void ThreadedRun::start() [with T = BQueue] 2 2000 200 68.5235
void ThreadedRun::start() [with T = CQueue] 4 2000 200 38.1185
void ThreadedRun::start() [with T = BQueue] 4 2000 200 34.8419
void ThreadedRun::start() [with T = CQueue] 6 2000 200 28.0139
void ThreadedRun::start() [with T = BQueue] 6 2000 200 23.6598
void ThreadedRun::start() [with T = CQueue] 8 2000 200 22.2323
void ThreadedRun::start() [with T = BQueue] 8 2000 200 18.1182
void ThreadedRun::start() [with T = CQueue] 10 2000 200 18.9129
void ThreadedRun::start() [with T = BQueue] 10 2000 200 15.2269
void ThreadedRun::start() [with T = CQueue] 12 2000 200 16.0067
void ThreadedRun::start() [with T = BQueue] 12 2000 200 16.6971
void ThreadedRun::start() [with T = CQueue] 14 2000 200 14.2292
void ThreadedRun::start() [with T = BQueue] 14 2000 200 12.7792
void ThreadedRun::start() [with T = CQueue] 16 2000 200 12.5874
void ThreadedRun::start() [with T = BQueue] 16 2000 200 13.87
void ThreadedRun::start() [with T = CQueue] 1 2000 200 134.356
void ThreadedRun::start() [with T = BQueue] 1 2000 200 142.953
void ThreadedRun::start() [with T = CQueue] 2 2000 200 70.7357
void ThreadedRun::start() [with T = BQueue] 2 2000 200 68.5412
void ThreadedRun::start() [with T = CQueue] 4 2000 200 38.265
void ThreadedRun::start() [with T = BQueue] 4 2000 200 35.2155
void ThreadedRun::start() [with T = CQueue] 6 2000 200 27.9101
```

```
void ThreadedRun::start() [with T = BQueue] 6 2000 200 24.1992
void ThreadedRun::start() [with T = CQueue] 8 2000 200 22.4867
void ThreadedRun::start() [with T = BQueue] 8 2000 200 18.146
void ThreadedRun::start() [with T = CQueue] 10 2000 200 19.1137
void ThreadedRun::start() [with T = BQueue] 10 2000 200 15.112
void ThreadedRun::start() [with T = CQueue] 12 2000 200 15.9995
void ThreadedRun::start() [with T = BQueue] 12 2000 200 16.9178
void ThreadedRun::start() [with T = CQueue] 14 2000 200 14.2052
void ThreadedRun::start() [with T = BQueue] 14 2000 200 13.1064
void ThreadedRun::start() [with T = CQueue] 16 2000 200 12.4361
void ThreadedRun::start() [with T = BQueue] 16 2000 200 14.3443
void ThreadedRun::start() [with T = CQueue] 1 2000 200 133.693
void ThreadedRun::start() [with T = BQueue] 1 2000 200 142.473
void ThreadedRun::start() [with T = CQueue] 2 2000 200 70.7313
void ThreadedRun::start() [with T = BQueue] 2 2000 200 68.832
void ThreadedRun::start() [with T = CQueue] 4 2000 200 38.0591
void ThreadedRun::start() [with T = BQueue] 4 2000 200 34.7144
void ThreadedRun::start() [with T = CQueue] 6 2000 200 27.9872
void ThreadedRun::start() [with T = BQueue] 6 2000 200 24.2434
void ThreadedRun::start() [with T = CQueue] 8 2000 200 22.5098
void ThreadedRun::start() [with T = BQueue] 8 2000 200 18.3515
void ThreadedRun::start() [with T = CQueue] 10 2000 200 18.833
void ThreadedRun::start() [with T = BQueue] 10 2000 200 15.1309
void ThreadedRun::start() [with T = CQueue] 12 2000 200 16.0216
void ThreadedRun::start() [with T = BQueue] 12 2000 200 17.018
void ThreadedRun::start() [with T = CQueue] 14 2000 200 14.1435
void ThreadedRun::start() [with T = BQueue] 14 2000 200 13.2138
void ThreadedRun::start() [with T = CQueue] 16 2000 200 12.4962
void ThreadedRun::start() [with T = BQueue] 16 2000 200 14.0269
void ThreadedRun::start() [with T = CQueue] 1 2000 200 133.882
void ThreadedRun::start() [with T = BQueue] 1 2000 200 142.786
void ThreadedRun::start() [with T = CQueue] 2 2000 200 70.9914
void ThreadedRun::start() [with T = BQueue] 2 2000 200 68.9167
void ThreadedRun::start() [with T = CQueue] 4 2000 200 38.2933
void ThreadedRun::start() [with T = BQueue] 4 2000 200 35.3306
void ThreadedRun::start() [with T = CQueue] 6 2000 200 28.1024
void ThreadedRun::start() [with T = BQueue] 6 2000 200 24.2582
void ThreadedRun::start() [with T = CQueue] 8 2000 200 22.5595
void ThreadedRun::start() [with T = BQueue] 8 2000 200 18.192
void ThreadedRun::start() [with T = CQueue] 10 2000 200 19.1155
void ThreadedRun::start() [with T = BQueue] 10 2000 200 15.1897
void ThreadedRun::start() [with T = CQueue] 12 2000 200 16.0055
void ThreadedRun::start() [with T = BQueue] 12 2000 200 16.3064
void ThreadedRun::start() [with T = CQueue] 14 2000 200 14.0636
void ThreadedRun::start() [with T = BQueue] 14 2000 200 12.8754
void ThreadedRun::start() [with T = CQueue] 16 2000 200 12.4864
void ThreadedRun::start() [with T = BQueue] 16 2000 200 14.2313
```

A.4 Weak scaling benchmarks

```
void ThreadedRun::start() [with T = CQueue] 1 1000 500 168.363
void ThreadedRun::start() [with T = BQueue] 1 1000 500 181.286
void ThreadedRun::start() [with T = CQueue] 2 1000 1000 175.882
void ThreadedRun::start() [with T = BQueue] 2 1000 1000 163.42
void ThreadedRun::start() [with T = CQueue] 4 1000 2000 200.433
void ThreadedRun::start() [with T = BQueue] 4 1000 2000 172.974
void ThreadedRun::start() [with T = CQueue] 8 1000 4000 226.761
void ThreadedRun::start() [with T = BQueue] 8 1000 4000 172.247
void ThreadedRun::start() [with T = CQueue] 16 1000 8000 238.368
void ThreadedRun::start() [with T = BQueue] 16 1000 8000 169.713
void ThreadedRun::start() [with T = CQueue] 32 1000 16000 221.205
void ThreadedRun::start() [with T = BQueue] 32 1000 16000 181.684
void ThreadedRun::start() [with T = CQueue] 1 1000 500 167.759
void ThreadedRun::start() [with T = BQueue] 1 1000 500 175.82
void ThreadedRun::start() [with T = CQueue] 2 1000 1000 175.298
void ThreadedRun::start() [with T = BQueue] 2 1000 1000 164.122
void ThreadedRun::start() [with T = CQueue] 4 1000 2000 199.54
void ThreadedRun::start() [with T = BQueue] 4 1000 2000 173.665
void ThreadedRun::start() [with T = CQueue] 8 1000 4000 226.586
void ThreadedRun::start() [with T = BQueue] 8 1000 4000 173.165
void ThreadedRun::start() [with T = CQueue] 16 1000 8000 238.239
void ThreadedRun::start() [with T = BQueue] 16 1000 8000 169.53
void ThreadedRun::start() [with T = CQueue] 32 1000 16000 219.255
void ThreadedRun::start() [with T = BQueue] 32 1000 16000 179.843
void ThreadedRun::start() [with T = CQueue] 1 1000 500 167.943
void ThreadedRun::start() [with T = BQueue] 1 1000 500 179.633
void ThreadedRun::start() [with T = CQueue] 2 1000 1000 176.192
void ThreadedRun::start() [with T = BQueue] 2 1000 1000 163.203
void ThreadedRun::start() [with T = CQueue] 4 1000 2000 200.126
void ThreadedRun::start() [with T = BQueue] 4 1000 2000 173.495
void ThreadedRun::start() [with T = CQueue] 8 1000 4000 226.194
void ThreadedRun::start() [with T = BQueue] 8 1000 4000 173.254
void ThreadedRun::start() [with T = CQueue] 16 1000 8000 237.976
void ThreadedRun::start() [with T = BQueue] 16 1000 8000 169.466
void ThreadedRun::start() [with T = CQueue] 32 1000 16000 220.748
void ThreadedRun::start() [with T = BQueue] 32 1000 16000 179.369
void ThreadedRun::start() [with T = CQueue] 1 1000 500 159.194
void ThreadedRun::start() [with T = BQueue] 1 1000 500 152.692
void ThreadedRun::start() [with T = CQueue] 2 1000 1000 170.187
void ThreadedRun::start() [with T = BQueue] 2 1000 1000 155.812
void ThreadedRun::start() [with T = CQueue] 4 1000 2000 190.136
void ThreadedRun::start() [with T = BQueue] 4 1000 2000 161.568
void ThreadedRun::start() [with T = CQueue] 8 1000 4000 214.539
void ThreadedRun::start() [with T = BQueue] 8 1000 4000 163.396
void ThreadedRun::start() [with T = CQueue] 16 1000 8000 226.909
void ThreadedRun::start() [with T = BQueue] 16 1000 8000 161.468
void ThreadedRun::start() [with T = CQueue] 32 1000 16000 229.964
```

```
void ThreadedRun::start() [with T = BQueue] 32 1000 16000 171.434
void ThreadedRun::start() [with T = CQueue] 1 1000 500 167.514
void ThreadedRun::start() [with T = BQueue] 1 1000 500 177.986
void ThreadedRun::start() [with T = CQueue] 2 1000 1000 176.098
void ThreadedRun::start() [with T = BQueue] 2 1000 1000 169.14
void ThreadedRun::start() [with T = CQueue] 4 1000 2000 200.296
void ThreadedRun::start() [with T = BQueue] 4 1000 2000 173.398
void ThreadedRun::start() [with T = CQueue] 8 1000 4000 226.672
void ThreadedRun::start() [with T = BQueue] 8 1000 4000 172.913
void ThreadedRun::start() [with T = CQueue] 16 1000 8000 238.243
void ThreadedRun::start() [with T = BQueue] 16 1000 8000 169.139
void ThreadedRun::start() [with T = CQueue] 32 1000 16000 242.379
void ThreadedRun::start() [with T = BQueue] 32 1000 16000 178.71
void ThreadedRun::start() [with T = CQueue] 1 250 500 41.8561
void ThreadedRun::start() [with T = BQueue] 1 250 500 43.3675
void ThreadedRun::start() [with T = CQueue] 2 250 1000 44.0077
void ThreadedRun::start() [with T = BQueue] 2 250 1000 41.0467
void ThreadedRun::start() [with T = CQueue] 4 250 2000 50.0676
void ThreadedRun::start() [with T = BQueue] 4 250 2000 43.3326
void ThreadedRun::start() [with T = CQueue] 8 250 4000 56.735
void ThreadedRun::start() [with T = BQueue] 8 250 4000 43.2841
void ThreadedRun::start() [with T = CQueue] 16 250 8000 59.4842
void ThreadedRun::start() [with T = BQueue] 16 250 8000 43.001
void ThreadedRun::start() [with T = CQueue] 32 250 16000 61.5678
void ThreadedRun::start() [with T = BQueue] 32 250 16000 58.4015
void ThreadedRun::start() [with T = CQueue] 1 250 500 41.9951
void ThreadedRun::start() [with T = BQueue] 1 250 500 42.0349
void ThreadedRun::start() [with T = CQueue] 2 250 1000 43.899
void ThreadedRun::start() [with T = BQueue] 2 250 1000 40.8124
void ThreadedRun::start() [with T = CQueue] 4 250 2000 49.916
void ThreadedRun::start() [with T = BQueue] 4 250 2000 43.362
void ThreadedRun::start() [with T = CQueue] 8 250 4000 57.705
void ThreadedRun::start() [with T = BQueue] 8 250 4000 43.1226
void ThreadedRun::start() [with T = CQueue] 16 250 8000 46.9096
void ThreadedRun::start() [with T = BQueue] 16 250 8000 41.6463
void ThreadedRun::start() [with T = CQueue] 32 250 16000 61.5892
void ThreadedRun::start() [with T = BQueue] 32 250 16000 45.1393
void ThreadedRun::start() [with T = CQueue] 1 250 500 42.1216
void ThreadedRun::start() [with T = BQueue] 1 250 500 42.4344
void ThreadedRun::start() [with T = CQueue] 2 250 1000 43.8792
void ThreadedRun::start() [with T = BQueue] 2 250 1000 40.8815
void ThreadedRun::start() [with T = CQueue] 4 250 2000 49.9778
void ThreadedRun::start() [with T = BQueue] 4 250 2000 43.1772
void ThreadedRun::start() [with T = CQueue] 8 250 4000 56.6129
void ThreadedRun::start() [with T = BQueue] 8 250 4000 43.1113
void ThreadedRun::start() [with T = CQueue] 16 250 8000 59.7697
void ThreadedRun::start() [with T = BQueue] 16 250 8000 42.4208
void ThreadedRun::start() [with T = CQueue] 32 250 16000 61.5229
void ThreadedRun::start() [with T = BQueue] 32 250 16000 44.8973
```

```
void ThreadedRun::start() [with T = CQueue] 1 250 500 41.8246
void ThreadedRun::start() [with T = BQueue] 1 250 500 43.9217
void ThreadedRun::start() [with T = CQueue] 2 250 1000 44.0754
void ThreadedRun::start() [with T = BQueue] 2 250 1000 40.9103
void ThreadedRun::start() [with T = CQueue] 4 250 2000 49.9123
void ThreadedRun::start() [with T = BQueue] 4 250 2000 43.2453
void ThreadedRun::start() [with T = CQueue] 8 250 4000 56.6262
void ThreadedRun::start() [with T = BQueue] 8 250 4000 43.2261
void ThreadedRun::start() [with T = CQueue] 16 250 8000 59.4477
void ThreadedRun::start() [with T = BQueue] 16 250 8000 42.6211
void ThreadedRun::start() [with T = CQueue] 32 250 16000 61.5317
void ThreadedRun::start() [with T = BQueue] 32 250 16000 45.2135
void ThreadedRun::start() [with T = CQueue] 1 250 500 41.9061
void ThreadedRun::start() [with T = BQueue] 1 250 500 44.1444
void ThreadedRun::start() [with T = CQueue] 2 250 1000 43.915
void ThreadedRun::start() [with T = BQueue] 2 250 1000 40.9423
void ThreadedRun::start() [with T = CQueue] 4 250 2000 49.9329
void ThreadedRun::start() [with T = BQueue] 4 250 2000 43.2564
void ThreadedRun::start() [with T = CQueue] 8 250 4000 56.5694
void ThreadedRun::start() [with T = BQueue] 8 250 4000 43.1536
void ThreadedRun::start() [with T = CQueue] 16 250 8000 59.4251
void ThreadedRun::start() [with T = BQueue] 16 250 8000 42.3516
void ThreadedRun::start() [with T = CQueue] 32 250 16000 54.9431
void ThreadedRun::start() [with T = BQueue] 32 250 16000 44.7874
void ThreadedRun::start() [with T = CQueue] 1 500 500 83.1189
void ThreadedRun::start() [with T = BQueue] 1 500 500 85.975
void ThreadedRun::start() [with T = CQueue] 2 500 1000 88.1401
void ThreadedRun::start() [with T = BQueue] 2 500 1000 81.5994
void ThreadedRun::start() [with T = CQueue] 4 500 2000 99.9364
void ThreadedRun::start() [with T = BQueue] 4 500 2000 86.4183
void ThreadedRun::start() [with T = CQueue] 8 500 4000 113.247
void ThreadedRun::start() [with T = BQueue] 8 500 4000 86.3196
void ThreadedRun::start() [with T = CQueue] 16 500 8000 119.046
void ThreadedRun::start() [with T = BQueue] 16 500 8000 84.6139
void ThreadedRun::start() [with T = CQueue] 32 500 16000 109.184
void ThreadedRun::start() [with T = BQueue] 32 500 16000 87.8672
void ThreadedRun::start() [with T = CQueue] 1 500 500 84.0183
void ThreadedRun::start() [with T = BQueue] 1 500 500 90.5925
void ThreadedRun::start() [with T = CQueue] 2 500 1000 88.4237
void ThreadedRun::start() [with T = BQueue] 2 500 1000 81.5329
void ThreadedRun::start() [with T = CQueue] 4 500 2000 99.8559
void ThreadedRun::start() [with T = BQueue] 4 500 2000 86.837
void ThreadedRun::start() [with T = CQueue] 8 500 4000 112.871
void ThreadedRun::start() [with T = BQueue] 8 500 4000 86.5041
void ThreadedRun::start() [with T = CQueue] 16 500 8000 118.878
void ThreadedRun::start() [with T = BQueue] 16 500 8000 84.611
void ThreadedRun::start() [with T = CQueue] 32 500 16000 124.647
void ThreadedRun::start() [with T = BQueue] 32 500 16000 89.9755
void ThreadedRun::start() [with T = CQueue] 1 500 500 83.5706
```

```
void ThreadedRun::start() [with T = BQueue] 1 500 500 84.5381
void ThreadedRun::start() [with T = CQueue] 2 500 1000 88.3843
void ThreadedRun::start() [with T = BQueue] 2 500 1000 81.5122
void ThreadedRun::start() [with T = CQueue] 4 500 2000 99.8409
void ThreadedRun::start() [with T = BQueue] 4 500 2000 86.4574
void ThreadedRun::start() [with T = CQueue] 8 500 4000 113.407
void ThreadedRun::start() [with T = BQueue] 8 500 4000 86.3389
void ThreadedRun::start() [with T = CQueue] 16 500 8000 118.938
void ThreadedRun::start() [with T = BQueue] 16 500 8000 84.5905
void ThreadedRun::start() [with T = CQueue] 32 500 16000 122.987
void ThreadedRun::start() [with T = BQueue] 32 500 16000 90.1947
void ThreadedRun::start() [with T = CQueue] 1 500 500 83.9774
void ThreadedRun::start() [with T = BQueue] 1 500 500 88.4746
void ThreadedRun::start() [with T = CQueue] 2 500 1000 88.0771
void ThreadedRun::start() [with T = BQueue] 2 500 1000 81.6183
void ThreadedRun::start() [with T = CQueue] 4 500 2000 99.9577
void ThreadedRun::start() [with T = BQueue] 4 500 2000 86.9623
void ThreadedRun::start() [with T = CQueue] 8 500 4000 113.655
void ThreadedRun::start() [with T = BQueue] 8 500 4000 86.475
void ThreadedRun::start() [with T = CQueue] 16 500 8000 119.044
void ThreadedRun::start() [with T = BQueue] 16 500 8000 84.6138
void ThreadedRun::start() [with T = CQueue] 32 500 16000 123.212
void ThreadedRun::start() [with T = BQueue] 32 500 16000 90.3859
void ThreadedRun::start() [with T = CQueue] 1 500 500 83.6114
void ThreadedRun::start() [with T = BQueue] 1 500 500 88.0567
void ThreadedRun::start() [with T = CQueue] 2 500 1000 87.9343
void ThreadedRun::start() [with T = BQueue] 2 500 1000 81.847
void ThreadedRun::start() [with T = CQueue] 4 500 2000 100.129
void ThreadedRun::start() [with T = BQueue] 4 500 2000 86.5778
void ThreadedRun::start() [with T = CQueue] 8 500 4000 113.298
void ThreadedRun::start() [with T = BQueue] 8 500 4000 86.3343
void ThreadedRun::start() [with T = CQueue] 16 500 8000 119.098
void ThreadedRun::start() [with T = BQueue] 16 500 8000 84.74
void ThreadedRun::start() [with T = CQueue] 32 500 16000 111.487
void ThreadedRun::start() [with T = BQueue] 32 500 16000 90.3981
```

A.5 Uneven benchmarks

```
void ThreadedRun::start() [with T = CQueue] 1 200 2000 81.7011
void ThreadedRun::start() [with T = BQueue] 1 200 2000 87.2387
void ThreadedRun::start() [with T = CQueue] 2 200 2000 43.4842
void ThreadedRun::start() [with T = BQueue] 2 200 2000 68.4883
void ThreadedRun::start() [with T = CQueue] 4 200 2000 24.233
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.7374
void ThreadedRun::start() [with T = CQueue] 6 200 2000 17.4999
void ThreadedRun::start() [with T = BQueue] 6 200 2000 22.9268
void ThreadedRun::start() [with T = CQueue] 8 200 2000 14.2396
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.3638
void ThreadedRun::start() [with T = CQueue] 10 200 2000 11.687
```

```
void ThreadedRun::start() [with T = BQueue] 10 200 2000 14.0112
void ThreadedRun::start() [with T = CQueue] 12 200 2000 9.90297
void ThreadedRun::start() [with T = BQueue] 12 200 2000 11.3942
void ThreadedRun::start() [with T = CQueue] 14 200 2000 8.64078
void ThreadedRun::start() [with T = BQueue] 14 200 2000 9.95279
void ThreadedRun::start() [with T = CQueue] 16 200 2000 7.63082
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.68502
void ThreadedRun::start() [with T = CQueue] 32 200 2000 4.15271
void ThreadedRun::start() [with T = BQueue] 32 200 2000 4.4722
void ThreadedRun::start() [with T = CQueue] 1 200 2000 81.8213
void ThreadedRun::start() [with T = BQueue] 1 200 2000 83.3596
void ThreadedRun::start() [with T = CQueue] 2 200 2000 43.5368
void ThreadedRun::start() [with T = BQueue] 2 200 2000 68.6851
void ThreadedRun::start() [with T = CQueue] 4 200 2000 24.2326
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.6869
void ThreadedRun::start() [with T = CQueue] 6 200 2000 17.649
void ThreadedRun::start() [with T = BQueue] 6 200 2000 22.7347
void ThreadedRun::start() [with T = CQueue] 8 200 2000 14.0494
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.1352
void ThreadedRun::start() [with T = CQueue] 10 200 2000 11.8189
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.883
void ThreadedRun::start() [with T = CQueue] 12 200 2000 9.94973
void ThreadedRun::start() [with T = BQueue] 12 200 2000 11.4752
void ThreadedRun::start() [with T = CQueue] 14 200 2000 8.65818
void ThreadedRun::start() [with T = BQueue] 14 200 2000 9.81431
void ThreadedRun::start() [with T = CQueue] 16 200 2000 7.56
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.78712
void ThreadedRun::start() [with T = CQueue] 32 200 2000 4.17004
void ThreadedRun::start() [with T = BQueue] 32 200 2000 4.519
void ThreadedRun::start() [with T = CQueue] 1 200 2000 81.4249
void ThreadedRun::start() [with T = BQueue] 1 200 2000 85.6086
void ThreadedRun::start() [with T = CQueue] 2 200 2000 43.3684
void ThreadedRun::start() [with T = BQueue] 2 200 2000 67.6822
void ThreadedRun::start() [with T = CQueue] 4 200 2000 24.1663
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.323
void ThreadedRun::start() [with T = CQueue] 6 200 2000 17.6981
void ThreadedRun::start() [with T = BQueue] 6 200 2000 22.9296
void ThreadedRun::start() [with T = CQueue] 8 200 2000 14.3253
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.1307
void ThreadedRun::start() [with T = CQueue] 10 200 2000 11.9754
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.8993
void ThreadedRun::start() [with T = CQueue] 12 200 2000 9.83654
void ThreadedRun::start() [with T = BQueue] 12 200 2000 11.4571
void ThreadedRun::start() [with T = CQueue] 14 200 2000 8.56921
void ThreadedRun::start() [with T = BQueue] 14 200 2000 9.82036
void ThreadedRun::start() [with T = CQueue] 16 200 2000 7.49006
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.69124
void ThreadedRun::start() [with T = CQueue] 32 200 2000 4.0791
void ThreadedRun::start() [with T = BQueue] 32 200 2000 5.53054
```

```
void ThreadedRun::start() [with T = CQueue] 1 200 2000 79.1773
void ThreadedRun::start() [with T = BQueue] 1 200 2000 81.7319
void ThreadedRun::start() [with T = CQueue] 2 200 2000 41.015
void ThreadedRun::start() [with T = BQueue] 2 200 2000 64.4704
void ThreadedRun::start() [with T = CQueue] 4 200 2000 22.9126
void ThreadedRun::start() [with T = BQueue] 4 200 2000 31.3272
void ThreadedRun::start() [with T = CQueue] 6 200 2000 16.5962
void ThreadedRun::start() [with T = BQueue] 6 200 2000 21.3093
void ThreadedRun::start() [with T = CQueue] 8 200 2000 13.0953
void ThreadedRun::start() [with T = BQueue] 8 200 2000 16.0237
void ThreadedRun::start() [with T = CQueue] 10 200 2000 11.0096
void ThreadedRun::start() [with T = BQueue] 10 200 2000 12.9005
void ThreadedRun::start() [with T = CQueue] 12 200 2000 9.41183
void ThreadedRun::start() [with T = BQueue] 12 200 2000 10.7057
void ThreadedRun::start() [with T = CQueue] 14 200 2000 7.95327
void ThreadedRun::start() [with T = BQueue] 14 200 2000 9.2149
void ThreadedRun::start() [with T = CQueue] 16 200 2000 7.00233
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.19717
void ThreadedRun::start() [with T = CQueue] 32 200 2000 3.77077
void ThreadedRun::start() [with T = BQueue] 32 200 2000 4.87226
void ThreadedRun::start() [with T = CQueue] 1 200 2000 82.1613
void ThreadedRun::start() [with T = BQueue] 1 200 2000 89.0809
void ThreadedRun::start() [with T = CQueue] 2 200 2000 43.4323
void ThreadedRun::start() [with T = BQueue] 2 200 2000 68.3615
void ThreadedRun::start() [with T = CQueue] 4 200 2000 24.1525
void ThreadedRun::start() [with T = BQueue] 4 200 2000 34.228
void ThreadedRun::start() [with T = CQueue] 6 200 2000 17.1444
void ThreadedRun::start() [with T = BQueue] 6 200 2000 22.8861
void ThreadedRun::start() [with T = CQueue] 8 200 2000 13.9594
void ThreadedRun::start() [with T = BQueue] 8 200 2000 17.5972
void ThreadedRun::start() [with T = CQueue] 10 200 2000 11.9319
void ThreadedRun::start() [with T = BQueue] 10 200 2000 13.9844
void ThreadedRun::start() [with T = CQueue] 12 200 2000 9.8232
void ThreadedRun::start() [with T = BQueue] 12 200 2000 11.4759
void ThreadedRun::start() [with T = CQueue] 14 200 2000 8.69023
void ThreadedRun::start() [with T = BQueue] 14 200 2000 9.84553
void ThreadedRun::start() [with T = CQueue] 16 200 2000 7.57736
void ThreadedRun::start() [with T = BQueue] 16 200 2000 8.69038
void ThreadedRun::start() [with T = CQueue] 32 200 2000 4.13416
void ThreadedRun::start() [with T = BQueue] 32 200 2000 5.42253
void ThreadedRun::start() [with T = CQueue] 1 2000 200 84.7598
void ThreadedRun::start() [with T = BQueue] 1 2000 200 87.1647
void ThreadedRun::start() [with T = CQueue] 2 2000 200 43.7021
void ThreadedRun::start() [with T = BQueue] 2 2000 200 70.7417
void ThreadedRun::start() [with T = CQueue] 4 2000 200 24.0171
void ThreadedRun::start() [with T = BQueue] 4 2000 200 35.5799
void ThreadedRun::start() [with T = CQueue] 6 2000 200 17.3466
void ThreadedRun::start() [with T = BQueue] 6 2000 200 23.2679
void ThreadedRun::start() [with T = CQueue] 8 2000 200 13.7178
```

```
void ThreadedRun::start() [with T = BQueue] 8 2000 200 17.6445
void ThreadedRun::start() [with T = CQueue] 10 2000 200 11.5486
void ThreadedRun::start() [with T = BQueue] 10 2000 200 14.227
void ThreadedRun::start() [with T = CQueue] 12 2000 200 9.75763
void ThreadedRun::start() [with T = BQueue] 12 2000 200 11.9995
void ThreadedRun::start() [with T = CQueue] 14 2000 200 8.17686
void ThreadedRun::start() [with T = BQueue] 14 2000 200 10.5989
void ThreadedRun::start() [with T = CQueue] 16 2000 200 7.49118
void ThreadedRun::start() [with T = BQueue] 16 2000 200 9.21562
void ThreadedRun::start() [with T = CQueue] 32 2000 200 4.76837
void ThreadedRun::start() [with T = BQueue] 32 2000 200 6.28576
void ThreadedRun::start() [with T = CQueue] 1 2000 200 84.3118
void ThreadedRun::start() [with T = BQueue] 1 2000 200 86.1925
void ThreadedRun::start() [with T = CQueue] 2 2000 200 43.9611
void ThreadedRun::start() [with T = BQueue] 2 2000 200 69.1773
void ThreadedRun::start() [with T = CQueue] 4 2000 200 23.9996
void ThreadedRun::start() [with T = BQueue] 4 2000 200 34.4667
void ThreadedRun::start() [with T = CQueue] 6 2000 200 17.453
void ThreadedRun::start() [with T = BQueue] 6 2000 200 22.912
void ThreadedRun::start() [with T = CQueue] 8 2000 200 13.7123
void ThreadedRun::start() [with T = BQueue] 8 2000 200 17.6182
void ThreadedRun::start() [with T = CQueue] 10 2000 200 11.4776
void ThreadedRun::start() [with T = BQueue] 10 2000 200 14.3256
void ThreadedRun::start() [with T = CQueue] 12 2000 200 9.69794
void ThreadedRun::start() [with T = BQueue] 12 2000 200 11.7176
void ThreadedRun::start() [with T = CQueue] 14 2000 200 8.34937
void ThreadedRun::start() [with T = BQueue] 14 2000 200 10.3241
void ThreadedRun::start() [with T = CQueue] 16 2000 200 7.39294
void ThreadedRun::start() [with T = BQueue] 16 2000 200 8.92471
void ThreadedRun::start() [with T = CQueue] 32 2000 200 4.64752
void ThreadedRun::start() [with T = BQueue] 32 2000 200 5.64033
void ThreadedRun::start() [with T = CQueue] 1 2000 200 84.6157
void ThreadedRun::start() [with T = BQueue] 1 2000 200 87.8467
void ThreadedRun::start() [with T = CQueue] 2 2000 200 43.72
void ThreadedRun::start() [with T = BQueue] 2 2000 200 68.3313
void ThreadedRun::start() [with T = CQueue] 4 2000 200 23.9997
void ThreadedRun::start() [with T = BQueue] 4 2000 200 33.893
void ThreadedRun::start() [with T = CQueue] 6 2000 200 17.472
void ThreadedRun::start() [with T = BQueue] 6 2000 200 22.9082
void ThreadedRun::start() [with T = CQueue] 8 2000 200 13.8302
void ThreadedRun::start() [with T = BQueue] 8 2000 200 17.6223
void ThreadedRun::start() [with T = CQueue] 10 2000 200 11.2675
void ThreadedRun::start() [with T = BQueue] 10 2000 200 14.3205
void ThreadedRun::start() [with T = CQueue] 12 2000 200 9.76004
void ThreadedRun::start() [with T = BQueue] 12 2000 200 11.5957
void ThreadedRun::start() [with T = CQueue] 14 2000 200 8.34503
void ThreadedRun::start() [with T = BQueue] 14 2000 200 10.684
void ThreadedRun::start() [with T = CQueue] 16 2000 200 7.51115
void ThreadedRun::start() [with T = BQueue] 16 2000 200 9.04845
```

```
void ThreadedRun::start() [with T = CQueue] 32 2000 200 4.67215
void ThreadedRun::start() [with T = BQueue] 32 2000 200 5.68089
void ThreadedRun::start() [with T = CQueue] 1 2000 200 83.5017
void ThreadedRun::start() [with T = BQueue] 1 2000 200 87.1321
void ThreadedRun::start() [with T = CQueue] 2 2000 200 43.8307
void ThreadedRun::start() [with T = BQueue] 2 2000 200 70.6668
void ThreadedRun::start() [with T = CQueue] 4 2000 200 24.0021
void ThreadedRun::start() [with T = BQueue] 4 2000 200 33.5607
void ThreadedRun::start() [with T = CQueue] 6 2000 200 17.4481
void ThreadedRun::start() [with T = BQueue] 6 2000 200 22.7772
void ThreadedRun::start() [with T = CQueue] 8 2000 200 13.7986
void ThreadedRun::start() [with T = BQueue] 8 2000 200 17.2507
void ThreadedRun::start() [with T = CQueue] 10 2000 200 11.3382
void ThreadedRun::start() [with T = BQueue] 10 2000 200 14.1975
void ThreadedRun::start() [with T = CQueue] 12 2000 200 9.72429
void ThreadedRun::start() [with T = BQueue] 12 2000 200 11.6274
void ThreadedRun::start() [with T = CQueue] 14 2000 200 8.28332
void ThreadedRun::start() [with T = BQueue] 14 2000 200 10.6321
void ThreadedRun::start() [with T = CQueue] 16 2000 200 7.48619
void ThreadedRun::start() [with T = BQueue] 16 2000 200 9.51968
void ThreadedRun::start() [with T = CQueue] 32 2000 200 4.65486
void ThreadedRun::start() [with T = BQueue] 32 2000 200 5.91993
void ThreadedRun::start() [with T = CQueue] 1 2000 200 80.7267
void ThreadedRun::start() [with T = BQueue] 1 2000 200 81.4627
void ThreadedRun::start() [with T = CQueue] 2 2000 200 41.5124
void ThreadedRun::start() [with T = BQueue] 2 2000 200 66.5264
void ThreadedRun::start() [with T = CQueue] 4 2000 200 23.0099
void ThreadedRun::start() [with T = BQueue] 4 2000 200 32.5625
void ThreadedRun::start() [with T = CQueue] 6 2000 200 16.358
void ThreadedRun::start() [with T = BQueue] 6 2000 200 21.6068
void ThreadedRun::start() [with T = CQueue] 8 2000 200 13.0217
void ThreadedRun::start() [with T = BQueue] 8 2000 200 16.4714
void ThreadedRun::start() [with T = CQueue] 10 2000 200 10.5904
void ThreadedRun::start() [with T = BQueue] 10 2000 200 13.4578
void ThreadedRun::start() [with T = CQueue] 12 2000 200 8.88021
void ThreadedRun::start() [with T = BQueue] 12 2000 200 11.0395
void ThreadedRun::start() [with T = CQueue] 14 2000 200 7.78125
void ThreadedRun::start() [with T = BQueue] 14 2000 200 10.0851
void ThreadedRun::start() [with T = CQueue] 16 2000 200 6.86252
void ThreadedRun::start() [with T = BQueue] 16 2000 200 8.68666
void ThreadedRun::start() [with T = CQueue] 32 2000 200 5.02742
void ThreadedRun::start() [with T = BQueue] 32 2000 200 6.67077
```