Implementing high performance Synchronous Message Exchange
University of Copenhagen

Truls Asheim <truls@asheim.dk>

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla laoreet lobortis erat, ac fringilla quam accumsan ut. Nam eget tortor neque. Nulla dictum dapibus venenatis. Vestibulum interdum sem vestibulum enim vulputate ullamcorper nec ut velit. Suspendisse rhoncus a nisi eget viverra. Curabitur porttitor hendrerit elit vel porta. Phasellus elementum nulla aliquet, rhoncus sapien sit amet, maximus libero. Morbi ipsum sapien, faucibus et tempor sed, commodo at enim. Curabitur maximus pretium felis, ac elementum nisi tincidunt et. Sed facilisis lacus purus, sit amet blandit elit ultricies sit amet. Maecenas et cursus libero, ac tincidunt arcu. In at tincidunt ante. Pellentesque lacinia dui nec nulla accumsan auctor faucibus elit suscipit. Pellentesque quis justo turpiso.

Praesent in dignissim lacus, in auctor felis. Phasellus iaculis finibus sapien, vitae vestibulum leo rutrum ac. Etiam in tellus eget sapien suscipit ornare eget sit amet lectus. Vestibulum sagittis consectetur varius. Aenean consectetur ante quis dui iaculis fringilla. Etiam enim sem, facilisis et interdum sit amet, congue nec metus. Fusce eu imperdiet enim, non sollicitudin sem. Vestibulum malesuada mattis justo, vel dictum massa tristique ac. Vivamus pretium turpis ut ligula porta, eget facilisis eros pulvinar. Aenean elit mi, semper vitae ornare vel, pretium sit amet ex. Cras commodo justo quis dolor gravida auctor. Vivamus ultrices arcu dolor, vitae porta ante viverra et. Quisque at aliquet massa, vel gravida felis. Donec eu imperdiet mi. Aliquam in tortor a libero iaculis luctus.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project description

In this report, we describe the design and implementation of a highly efficient library for parallel execution of SME networks.

In this report, we describe a synchronous message passing framework intended to aid simulation of applications whose operating semantics are compatible with the SME model

1.2 Motivation

Why is this awesome?

1.3 Limitations

This report will not discuss details related to design of hardware

1.4 Related work

Discuss master thesis

Chapter 2

Analysis and Design

2.1 Motivation

2.2 Synchronous Message Exchange

Components

In this section, we describe the various components that make up the a SME network.

Process

Properties

The SME model has a number of special properties which influences how to efficiently represent and execute the network in our implementation.

Clock cycles One defining feature of hardware is that all processing is driven by a clock beat. In order to enable fulfillment of this goal we introduce a simulated clock beat in our implementation of SME and thus the defining property of hardware is preserved in the SME model.

Global synchronicity As a consequence of implementing the simulated clock best, all events and communications of the network occurs completely synchronous from the point of view of a process. FIXME

Shared nothing A process can be considered a completely autonomous and can only change state through receiving a message on its incoming bus. A process is also self contained in the sense

Execution model

Our execution is derived from the unique properties of the system

2.3 Public API

The API exposed to the user of the SME library should be designed with a focus on balancing expressiveness and ability to be easily understanidble....

The API implemented by VINTER and SKOVHEDE 2014 is heavily dependent of the highly dynamic nature of the Python programming language used to implement their prototype. Since our implementation is written in C++ we cannot directly mimic the python API in our implementation. Given the similarities between SME and CSP it seems obvious to let us inspire by

2.4 Problem characteristics

2.5 Paralellization model

Using OS-level threads for representing the SME processes was quickly ruled out during the design process. OS-level threads are severely limited in that the number of threads a process can have is highly limited and switching between OS-level threads is very expensive compared to, for instance, user-level threads. Sung et al. 2002.

FiXme Note: Find reference for number of threads per process

FiXme Note: Find more recent citation for the performance of OS vs. user-level threads

Using user-level threads, on linux implemented by using the (now deprecated) `setjump, longjump` functions or the more current `setcontext, getcontext`, is another possible way of implementing our networks. The notion of a user-level thread is highly compatible with our concept of a process, namely . User-level threads is the method that is used to implement many CSP libraries, including C++CSP2.

FiXme Note: finish

One important difference between the CSP and SME execution models is that CSP is highly asynchronous in nature while SME is, as previously mentioned, entirely synchronous. This allows us to take a significantly more simple approach to scheduling threads compared to C++CSP. A parallel CSP library needs to schedule a process for execution based on events generated by other processes Brown and Welch 2003. In SME we know that every process needs to run during every "clock cycle". This greatly simplifies our implementation of multi-threaded SME.

FiXme Note: This needs to be better described

Based on this, we propose an extremely and efficient model for parallel execution of a SME process network illustrated in. . The idea is to start one OS-level thread per available CPU-core and assign a subset of SME processes to each code. Then, for each iteration of the SME-network , the controlling thread will start by

FiXme Note: make figure similar to ??

FiXme Note: Define "iteration of SME-network" as a full clock-cycle of the simulated hardware or replace with different term

2.6 Identifying optimal process scheduling

In order to determine the efficiency of various methods of process scheduling we need to identify the optimality condition for our process scheduling. An illustration of our threading model can be seen in ??. The green boxes represents processes while the red boxes indicates core idle time. Notice, how we by redistributing the processes across threads could reduce the idle-time of our cores.

2.7 Process orchestration

As we discussed in the previous section, the primary limiting factor for our multi-threaded network is an uneven and suboptimal distribution of processes across CPU-cores. If orchestration is determined by the order of which processes were defined

The optimal way to do this depend

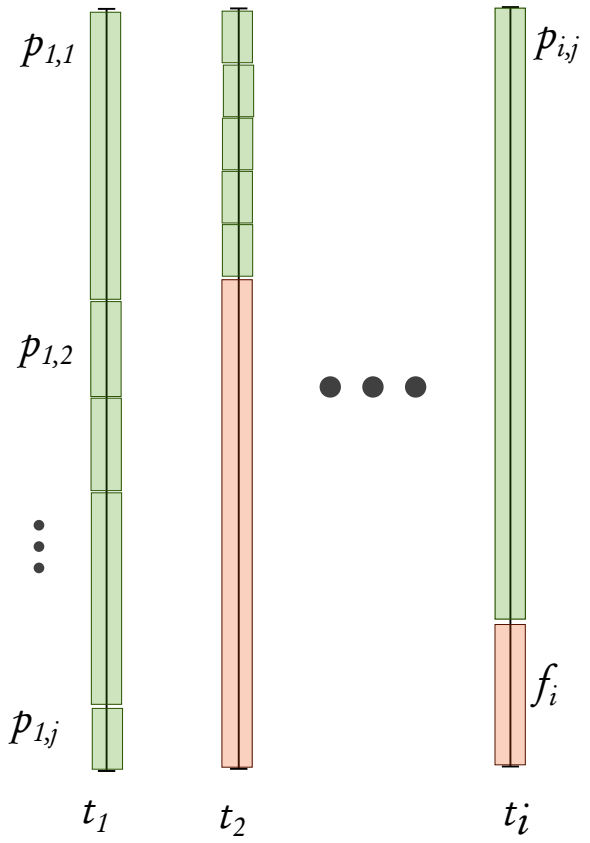

Figure 2.1: Example of suboptimal distribution of processes across processing threads. Green blocks represents processes while red blocks represents thread idle time. Threads are named $p_{i,j}$ where i is the number of the thread the process has been assigned to and f_i is the combined idle time for each thread. Threads are named t_i.

2.8 One-shot process orchestration

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

2.9 Monte Carlo orchestration

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

2.10 Optimization-based orchestration

Another way to orchestrate the processes is to use a

2.11 Adaptive process orchestration

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

2.12 Adaptive Monte Carlo process orchestration

2.13 Adaptive Optimization-based process orchestration

Chapter 3

Implementation

Chapter 4

Discussion

Bibliography

Brown, Neil CC and Peter H Welch (2003). "An introduction to the Kent C++ CSP Library". In: *Communicating Process Architectures 2003* 61, pp. 139–156 (cit. on p. 8).

Lawrence, Steve and C Lee Giles (1998). "Searching the world wide web". In: *Science* 280.5360, pp. 98–100.

Sung, Minyoung et al. (2002). "Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread". In: *Information processing letters* 84.4, pp. 221–225 (cit. on p. 8).

VINTER, Brian and Kenneth SKOVHEDE (2014). "Synchronous Message Exchange for Hardware Designs". In: (cit. on p. 8).

.1 Compiling and executing

This section will contain information about how to compile and execute cppsme