Parallelization of sequential programs

1. Operating systems and multicore programming (1DT089)
Project proposal for group 1: Jonatan Waern (920303-5598), Daniel Engh (900331-2155), Adam Hernod Olevall (900114-2554), Mikael Holmberg (911218-0519)
Version 1.5, 2013-04-22

Table of contents

1. Introduction
2. System architecture
3. Concurrency models
4. Development tools
5. Research & Development
6. Process evaluation

1. Introduction

During recent years parallelization of programs has been increasingly relevant due to the rise of multi-core CPUs as a means to increase performance. Therefore algorithms and techniques to safely and efficiently optimize programs for multiple cores are desirable.
We wish to program a scheduler that can parallelize the execution of a sequential program. If done well, this would essentially optimize a program for execution on machines with a variable number of cores.

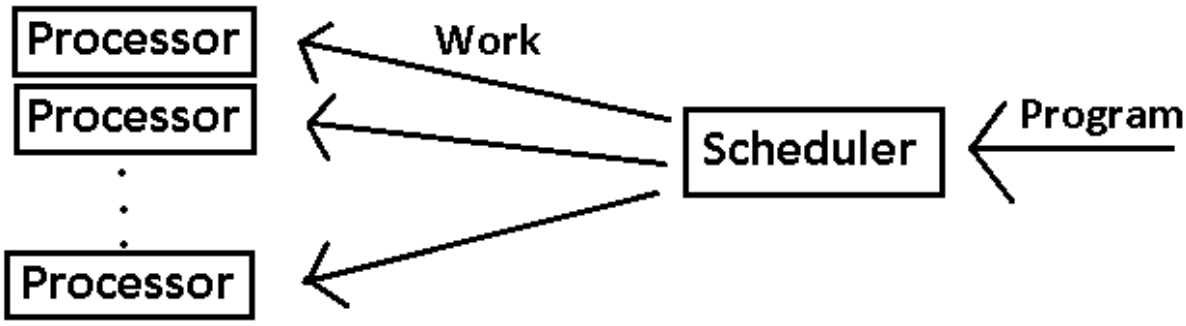
Of course, we are not actually planning to make an operating system with a custom-made scheduler. Rather, we plan to simulate this scheduler on a regular operating system. Given the behavior of multiple processes on certain operating systems, this should be close enough to determine effective techniques.

The main challenges of parallelizing a program are dividing instructions into reasonable chunks, resolving dependencies and minimizing overhead. But these are all reasonable obstacles.

2. System architecture

The obvious parts that are required are the simulated processors and the scheduler. The scheduler will receive some program as input (either a C program or a program written in some form of more easily handled code) and divide this up to the processors, which will each have some form of internal queue system, to select which process to perform next.

Possibly we may want an interpreter that translates C code into more easily digested code. Or we will want to design our own low-level language which can be easily handled by the scheduler.



This seemed like a very reasonable way to implement the system, it is an almost direct translation of the algorithmical problem to an implementation, and thus, easy to reason about.

3. Concurrency models

We plan to use multiple threads to implement both the scheduler and the multiple cores. Basically this is because the bulk of the planning work is done by the scheduler, so it would be preferred to let the scheduler have its own process. Then we have one thread for each core, which is because the cores do no need to communicate with each other but rather need only accept incoming jobs from the scheduler and report back with results, and the shared memory makes this communication fast.

The programing language we have chosen is C, this is because it is efficient for calculations, because we have a high degree of memory control which will lessen the overhead and because we can control on which core is each process is executed.

4. Development tools

Our code editor of choice is emacs, due to extensive plug-ins for code highlighting and due to personal familiarity. We plan to use git with github for our source code management, because we do need some sort merging code handler, and github has a number of nice features to use.

The build tool standard for C is Make, and we see no reason to change that.

For our testing needs, we have chosen the Check framework, as this has been in use for a long time and have a fairly large documentation. Also it runs more protected tests which will be useful seeing as dicking around with cores and memory may result in rather nasty errors.

Documentation will be written within the code and extracted into a more readable format using Doxygen, as is suggested.

5. Research & Development

Before any actual code is written we will need to do some research, we will look into previous attempts at similar problems. We will decide on what input language to use and depending on that we will need to do additional research pertaining to that specific language. We are already strongly considering a functional pseudo-language, and in such a case we will need to look into the parsing techniques of such a language and the stack building for evaluation of a functional program.

6. Process evaluation

We had relatively few suggestions, as we rather quickly decided on this one. The other suggestions were discarded due to their relatively small association with the course contents, while this one has a much bigger relevance.