Parallel Programming
Exam notes and quiz answers

Daniel Machon

8. januar 2014

Indhold

1 Exam notes

1.1 Introduction

1.1.1 Parallel programming

Parallel programming is the use of multiple computational resources (CPU's, Cores), to solve a problem. A problem is divided into subproblems and solved concurrently.
The problem must be subdivideable, so that several instructions can be executed in parallel.

Examples of uses of parallel programming:

- Weather forecast
- Economic modelling
- Mathematics

Advantages of parallel computing:

- Save time and money. You can increase computation speed by adding more cheap hardware
- It is possible to do several things at a time
- Share computation resources across a network

Disadvantages of parallel computing:

- The speed of serial computation is limited by the amount of time it takes for data to move through a wire.
- At some point, the size of a transistor on the CPU cannot be made smaller.
- More expensive to increase the speed of a single core CPU.
- Increased amount of needed memory, due to overhead of creating the parallel environment.
- If the program is too small, running it in parallel can decrease performance, because of the associated overhead of creating the parallel environment (libraries etc).

A parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer. The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease.

Limiting hardware factors:

- Memory-cpu bus bandwidth on an SMP (Symmetric Multi-Processor) machine
- Communications network bandwidth
- Amount of memory available on any given machine or set of machines
- Processor clock speed

Most computers uses the Von Neumann architecture:

Figur 1: Von Neumann architecture

1.1.2 Flynn's taxonomy

SISD - Non parallel computation. single processor, executes a single instruction stream, to operate on data stored in a single memory. This corresponds to the von Neumann architecture.

SIMD - Parallel computation. All processing units execute the same instruction at any given clock cycle Each processing unit can operate on a different data element. GPU's uses this. Synchronous

MISD - Parallel computation. Each processing unit operates on the data independently via separate instruction streams. A single data stream is fed into multiple processing units. Rarely used.

MIMD - Parallel computation. Every processor executes a different instruction steam, and every processor executes a different data stream. Execution can be synchronous or asynchronous. Most common type of parallel computer - most modern supercomputers fall into this category. many MIMD architectures also include SIMD execution sub-component.

1.1.3 Amdahl's Law

Amdahl's law says that the speedup of a program depends on the fraction of the program code (P) that can be parallelized.

$$\text{Speedup} = \frac{1}{1-P}$$

$$\text{Speedup} = \frac{1}{\frac{P}{N}+S}$$ [1]

Shared memory

UMA - All processors have access to the same address space. Changes in the memory space are visible to all processors.

NUMA - Several SMP's linked together by physical busses. One SMP have access to another SMP's memory by a link (slower)

Advantages

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages

- Adding more CPU's will result on more traffic on the CPU to memory bus.
- Programmer responsibility for synchronization.

Distributed memory

Processors have their own private address space. A processor does not have direct access to another processors memory. If one processor needs data from another, they can exchange data using, for instance, sockets.

Advantages

- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.

Disadvantages

- The programmer is responsible for the inter-processor communication

[1] Where S = serial fraction, and N = number of processors

1.2 Multi-programming With Shared Memory

1.2.1 Processes and Threads

Parallel programming can be achieved with both processes and threads. The programmer needs to explicitly write the code to take advantage of parallelism. Some ways to write parallel code:

- Using an already parallel programming language (Ada)
- Modifying the syntax of an existing sequential programming language to create a parallel programming language. Example UPC
- Using an existing sequential programming language supplemented with compiler directives and libraries for specifying parallelism. Example: OpenMP and Pthreads are based on a library solution.

A process and a thread makes up of an **Identification**, and **Instance of all CPU register (also IP)**, a **Stack** and eventually a **Priority**

Figur 2: Process

Figur 3: Thread

If a thread is dependant on some other thread, it can call the join() function to wait for that thread to terminate. If two threads are independant and dont care when on another terminates, they are called **detached**.

1.2.2 Thread synchronization

Threads within a process have access to the same shared data. The shared data that two threads may be interested in manipulating, is called the **critical section**. There are several ways to ensure that only one thread has access to the critical section at once.[2]

1.2.3 User Level Threads (ULT) and Kernel Level Threads (KLT)

- In ULT, each process has a thread table and the kernel has a process table
- In KLT, the kernel has a process and a thread table

In a scenario where you want to sleep a single thread within a process; if using ULT, the whole process encapsulating that single thread will be slept. When using KLT, the kernel has a thread table and hence, only the thread will be slept and not the whole process.

Figur 4: ULT and KLT

[2]Described in chapter 4 - Pthreads

1.3 Parallel Programming With Shared Memory (2)

1.4 Pthreads - POSIX threads

Threads that adhere to the standardized implementation of IEEE POSIX are called POSIX threads or pthreads.
This standardization is to make threads portable Threads are called lightweight processes (less overhead), because the only require the most basic resources to exist. pthreads have the following characteristics:

- A thread lives within the scope of a process, and uses the resources (memory, stack) of the process.
- A thread has it's own **stack pointer, registers, thread specificdata**
- A thread terminates if its parent thread (or process) terminates
- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.
- A thread can be created using much less operating overhead that a process
- Serial application can also take advantage of threads, to simulate concurrency, when task need to progress independently

1.4.1 Thread programming models

- **Manager/Worker** - a single thread, the manager assigns work to other threads, the workers. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
- **Pipeline** - task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
- **Peer** - similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

1.4.2 Deadlocks

The following four conditions (Coffman; Havender) are necessary but not sufficient for deadlock.
Mutual exclusion: A resource can be assigned to at most one process at a time (no sharing).
Hold and wait: A processing holding a resource is permitted to request another.
No preemption: A process must release its resources; they cannot be taken away.
Circular wait: There must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain.

1.4.3 Synchronization techniques

1.4.3.1 Joining threads

Thread joining is a simple way of controlling multithreaded code. It allows a programmer to stop a thread, and wait until the calling thread has finnished its job. If you have two threads running concurrently, and you want one thread to wait until the other finnishes (maybe to accesss its computed data) you do so by joining them:

Figur 5: Synchronization by joining threads

When a thread is created, one can specidy whether its joinable or not. If its not, its called a detached thread. It is possible to change a thread from joinable to detached at runtime.

1.4.3.2 Condition variables

Condition variables allow threads to wait until some event or condition has occurred. A condition variable must always be used together with a mutex. A given condition variable can have only one mutex associated with it, but a mutex can be used for more than one condition variable.

- wait()
- signal()

Mutexes and condition variables are busy waiting techniques

1.4.3.3 Semaphores

Introducing a blocked state eliminates the need of busy waiting. a blocked process is unblocked by another process doing a signal(). all processes are allowed to signal() or wait() in a semaphore (as opposed to mutexes).

- A semaphore is a data structure with associated primitive operations
- A semaphore is a queue of blocked processes

Figur 6: Semaphore diagram

- Binary semaphore A binary semaphore is a synchronization object that can have only two states: **Taken** and **Not taken**

  Binary semaphores have no ownership attribute and can be released by any thread or interrupt handler regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize threads with external events implemented as ISRs, for example waiting for a packet from a network or waiting that a button is pressed.

  Because there is no ownership concept a binary semaphore object can be created to be either in the taken or not taken state initially.

- Counting semaphore A counting semaphore is a synchronization object that can hold more then one state, where each state represent an blocked process. N state represent the number of queued threads, if the number is negative there is -N queued threads, so a new thread will be placed in the queued. If the number is positive there is no waiting threads, and if its gets a signal or a wait no action will be taken.
- General semaphore Positive values shows number of free resources
  Negative values shows number of blocked processes

1.4.3.4 Monitors

1.4.3.5 Barriers

Barriers are a group of processes. With a barrier, each process need to reach the barrier before they can continue to run. Once all processes has reach the barrier, they will all be released simultaneously to start the next iteration.

2 Quiz answers

2.1 Links

http://www.youtube.com/channel/UC7jgpE3sllixozoA4GzVRqQ