The Multicore Programming Course in Lund

Welcome to the Multicore Programming course in Lund

- 12 lectures
- 6 labs on parallel programming — work in groups of two.
- During the labs you will make different implementations of an algorithm (dataflow analysis) on different platforms:
  1. Java
  2. Scala with Akka
  3. C with Pthreads (two parts)
- There will also be labs on atomic types in C++, and lock-free data structures
- The first and third might look similar but they are not!
- There is also a project and a competition which consists of chosing one of these versions and to tune as much as you can — also in groups of two.

Administration

- Lecturer is `Jonas.Skeppstedt@cs.lth.se` with office E:2190
- Office hours during LP1: every weekday at 12.30 - 13.00
- Lectures at 10 on Mondays and Thursdays
- Labs start next week.
- Course web page `http://cs.lth.se/edan26`
- You will get an account on the machine **power.cs.lth.se** — a 4 processor POWER/Linux machine 2.5 GHz
- You can work on other machines if you wish but performance measurements are to be done on it.
- You can access it with **ssh -Y user@power.cs.lth.se**...

Contents of the course

F1 Introduction to multicore programming
F2 Multicore architectures
F3 Memory consistency models
F4 High-level parallel programming: Scala on the JVM
F5 POSIX Threads details
F6 Threads in the new ISO C Standard: C11
F7 Parallelizing sequential C codes by hand
F8 Lock-free data structures
F9 Cache aware programming for multicores
F10 Low-level parallel programming: explicit DMA transfers on the Cell
F10 Parallelizing sequential C codes automatically
F12 Research trends

Contents of Lecture 1

- Advantages with multithreading
  - Performance
  - Sometimes simpler programming
- Disadvantages with multithreading
  - Overhead
  - Usually more complex programming
- Parallel programming models
  - Message passing to compute nodes with private memories — e.g. the Cell (Sony PS3)
  - Shared memory with coherent caches — most common machine type
  - In Scala we will use message passing built on shared memory

Multicore Programming

- Parallel programming is much more fun than sequential programming!
- It's really cool to see many threads speeding up our code :-)
- The computer industry revolution to make parallel computing widespread is happening right **now**.
- In the 1990's researchers in industry and academia came to a consensus on how to build "easily" programmable parallel machines.
- What do such machines cost?
- The price of a phone, laptop or a desktop.

More expensive machines

- Our `power.cs.lth.se` cost USD 3299 as new in 2005.
- Now they are a few thousand krona on ebay (ours is from there)
- Large scale servers cost much more of course.

Featured models	Processor / speed Number of processors	System memory	Internal storage
8286-42A1 (Rack-mount) → Buy now	POWER8 / 3.8 GHz 6-core **$21,319.00** IBM Web price* $540/month for 36 months**	64 GB	2 x 146 GB 15K RPM SAS disk
8286-42A2 (Rack-mount) → Buy now	POWER8 / 4.1 GHz 8-core **$29,459.00** IBM Web price* $745/month for 36 months**	64 GB	2 x 146 GB 15K RPM SAS disk
8286-42A3 (Rack-mount) → Buy now	POWER8 / 3.8 GHz 12-core **$35,247.00** IBM Web price* $890/month for 36 months**	96 GB	2 x 146 GB 15K RPM SAS disk
8286-42A4 (Rack-mount) → Buy now	POWER8 / 4.1 GHz 16-core **$53,227.00** IBM Web price* $1,340/month for 36 months**	128 GB	2 x 146 GB 15K RPM SAS disk
8286-42A5 (Rack-mount) → Buy now	POWER8 / 3.5 GHz 24-core **$65,291.00** IBM Web price* $1,645/month for 36 months**	192 GB	2 x 146 GB 15K RPM SAS disk

Machines for multithreaded Java, C, C++

- These machines are called **cache coherent shared memory multiprocessors**, and are also often called multicores.
- Multicore actually means a machine with multiple processors, which do not necessarily have private cache memories.
- Obviously, just because we have multiple processors our program does not become faster automatically.
- We need to divide it into threads which can compute partial results.

Potential Sources of Troubles in Multicore Programming

- What can go wrong in this process?
  - **Amdahl's Law:** limited speedup if we cannot find enough parallel tasks the threads can work on.
  - Multiple threads modify the same data concurrently and corrupt the result, ie we have created **data races**.
  - Threads wait for each other in a circular way and we have a **deadlock**.
  - Our program becomes slower than we hoped for because the memory access penalty is much longer and **we failed to exploit the cache memories**.
  - One processor modifies data in its cache and another reads **obsolete data** from memory and the program crashes :-(
- We will next go through these in more detail.

Amdahl's Law

- Assume a sequential program has an execution time 1 time units, and a fraction P can be parallelized.
- What is the maximum speedup with N processors?
- Speedup $S = \frac{T_{slow}}{T_{fast}}$
- Speedup $S_N = \frac{1}{(1-P)+P/N}$
- Assume $P = 0.9$

N	S
2	1.8
3	2.5
4	3.1
10	5.3
100	9.2
∞	10

- Parallel programming is interesting only for sufficiently large P!

Data Races

```
count += 1;
```

- If two or more threads can modify the variable concurrently there is a data race and the final value written to memory is not predictable.
- It would have been predictable if += was implemented as an atomic instruction which modified a certain memory location but it's not.
- The Valgrind tool Helgrind can detect data races.
- In the new version of ISO C (aka C11) data races are undefined behavior (= very serious programmer bugs).
- To avoid data races we need to assure that only one thread can modify the data in a **critical region** which are typically created with some form of a lock. In Pthreads we can write:

```
pthread_mutex_lock(L);
count += 1;
pthread_mutex_unlock(L);
```

Locks

- A lock is a data structure (possibly as simple as only an integer variable) which only one thread at a time can hold, like a unique door key.
- There are at least two operations:
  - **Acquire** the lock. In Pthreads this function is called `pthread_mutex_lock` and it takes a pointer to a `pthread_mutex_t` as parameter. If some other thread already has taken the lock the second must wait.
  - **Release** the lock. In Pthreads this function is called `pthread_mutex_unlock` and it also takes a pointer to a `pthread_mutex_t` as parameter.
- In Pthreads it is also possible to see if the lock is currently free and only take it then while cancel the operation if the lock is taken in order to avoid waiting. It is called `pthread_mutex_trylock`.

Using Locks

- Thus, locks are used to protect data so that only one thread at a time can modify it in a critical region.
- Locks are thus used to achieve **mutual exclusion** which means that only one thread can do something with some data at a time.
- In Java, every object has such a lock, called a **mutex**, which is used with **synchronized** blocks or methods.

Let us try Helgrind!

- Let us run a program with a data race to see what Helgrind tells us!
- Without going into details the program increments a counter outside any critical region.
- We run it with 10 threads as follows:

```
$ gcc -g datarace.c -lpthread
$ valgrind --tool=helgrind a.out 10
```

```
==14599== Possible data race during read of size 4 at 0x10011130 by thread #3
==14599==    at 0x10000900: work (datarace.c:67)
==14599==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==14599==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==14599==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==14599==  This conflicts with a previous write of size 4 by thread #2
==14599==    at 0x10000910: work (datarace.c:67)
==14599==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==14599==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==14599==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
```

- Such messages can be invaluable.

Deadlocks

- A deadlock occurs when threads wait for each other in a circular way so that none can proceed.
- There are four requirements that all must hold for a deadlock to exist:
  1. **Mutual exclusion**, i.e. only one thread can use a resource R at a time.
  2. **No preemption**, i.e. it's not possible to take away the resource from a thread which currently holds it.
  3. **Hold and wait**, a thread which holds a resource R_1 may request another resource R_2 which may currently be held by another thread.
  4. There must be the **circular wait**, e.g. T_1 waiting for T_2 and T_2 waiting for T_1.

How can we prevent deadlocks in our programs?

- Mutual exclusion and no preemption may be difficult to avoid in general.
- In some cases we can, however, permit only one thread modifying some data, or multiple threads reading that data. This is called a **read-write lock**.
- To avoid hold-and-wait, we can use the `pthread_mutex_trylock` if it makes sense in our program.
- To avoid the circular wait, we can have rules which specify the order in which multiple locks may be acquired. If all threads follow these rules, there cannot be any circular wait.
- Helgrind does not detect deadlocks but it actually detects something better.
- What could be better than pointing out deadlocks?
- See next slide.

Non-Deterministic Execution

- One **very** important aspect of parallel programming is that execution normally is not deterministic.
- A deadlock might happen in one of 1,000,000 executions so testing as for sequential programs is not sufficient.
- By observing the order in which the threads acquire the different locks, one can check if the programmer had no rule for which order should be used.
- How can we observe that?

Helgrind and Deadlocks 1(2)

- So: Helgrind does **not** detect deadlocks but does something much more useful.
- Helgrind observes the lock-acquire order and complains when two threads acquire a lock in the opposite order.

```
void* work(void* p)
{
        arg_t*  arg = p;

        if (arg->i == 0) {
                pthread_mutex_lock(&A);
                pthread_mutex_lock(&B);

                printf("got both!\n");

                pthread_mutex_unlock(&B);
                pthread_mutex_unlock(&A);
        } else {
                pthread_mutex_lock(&B);
                pthread_mutex_lock(&A);

                printf("got both!\n");

                pthread_mutex_unlock(&A);
                pthread_mutex_unlock(&B);
        }

        return NULL;
}
```

Helgrind and Deadlocks 2(2)

- There is only a small probability that there will be a deadlock when running the program since the threads are started one at a time and the first most likely finishes before the second is even started.
- Helgrind reports the following, however:

```
==17471== Thread #3: lock order "0x100112AC before 0x100112C4" violated
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x10000A13: work (deadlock.c:84)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==   Required order was established by acquisition of lock at 0x100112AC
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x100009C7: work (deadlock.c:71)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==   followed by a later acquisition of lock at 0x100112C4
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x100009D3: work (deadlock.c:74)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==
```

Deadlocks in Java!?

- What should you do if you have made a very complex Lab 1 which has deadlocks?

C11 Atomic Types

- As we will see in Lecture 6, we can do as follows in C11:

```
_Atomic int      count;


/* Thread 1 */
count += 1;
```

```
                                /* Thread 2 */
                                count += 1;
```

Cache Memories

- Cache memories are extremely important on uniprocessors and even more important on multicores.
- If all threads would perform each memory access from system RAM, programs would simply be too slow. For example, the bus would be overloaded.
- **Unfortunately**, it's even more difficult on multicores than on sequential machines to exploit cache memories.
- Put simply, on sequential machines we get cache misses because the data does not fit (or was never accessed before) in the cache.
- On multicores, we also get cache misses when new values are communicated between different caches, called **true sharing misses**.
- In addition, we can also get cache misses due to some other processor wrote to a variable we are **not** interested in! These are called **false sharing misses**.

Caches in Multicores

- Suppose processor P_1 has read the value of a variable X.
- A copy of X will be in the cache of P_1.
- Assume next P_2 writes a new value to X.
- That new value, X', will be in the cache of P_2.
- What happens if P_1 wants to read X again???
- Without caches P_1 would probably see the new value (if it has reached memory).
- Unless we take special actions, depending on the time between the write by P_2 and the second read by P_1, it is more or less likely that P_1 will read an **obsolete** value of X from its cache!
- There are two special actions that the programmer needs to use:
  - P_2 must have released a lock L after writing X, and
  - P_1 must have acquired the lock L before reading X.
- The release by P_2 must happen before the acquire by P_1.
- We will go into details about this in Lecture 3 on **memory consistency models**.

A First Model of a Multicore Machine

- The simplest model of a multicore machine has a shared memory and no private caches.
- Without cache memories we do not have to be concerned about whether all threads see a consistent memory.
- Of course we must still avoid data races and deadlocks!
- Unfortunately, as we know, such a machine becomes too slow due the huge memory access times.
- However, the first multicore machine, from 1962, is similar to this, but the relative cost of accessing memory was less at that time.

A Second Model of a Multicore Machine

- A simple (and fastest for some applications) multicore machine is actually the Cell processor used e.g. in the Playstation 3
- The Cell processor has a normal 64-bit POWER processor (actually, it is multithreaded but ignore that for the moment) and a number of special compute nodes called **Synergistic Processing Units** or SPUs.
- Each SPU has a small private memory of 256 KB, called the **local store**.
- View the local store as a software managed cache of system RAM.
- The SPUs are programmed in C or C++ and on Linux the kernel schedules the SPU threads as it wishes... i.e. you can have any number of SPU threads regardless of how many SPUs your machine actually has.
- Normally the POWER processor controls the SPU threads and tells them what to do.

Computing on the Cell

- To process data on the SPUs we do the following:
  1. Copy input data from system RAM to an SPU's local store.
  2. Let the SPU compute a partial result.
  3. Copy the partial result to system RAM.
- Simple isn't it?

Yes, the Cell is simple and easy to understand

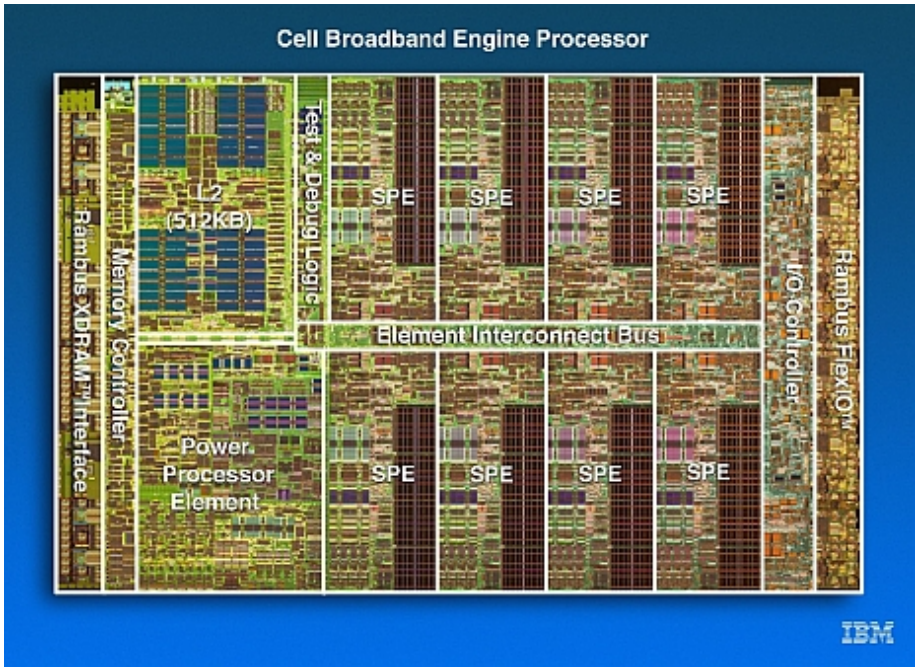
- The Cell is simple to understand for the programmer.
- The Cell is simple to make ***FAST*** in hardware.
- It's possible to write very fast software on the Cell due to its clever architecture (not because it's simple, of course)
- In addition the Cell uses relatively little energy — the most energy efficient supercomputers are all built with Cell processors.
- Here are some Gflops/s performance numbers:

Algorithm	**Cell**	**Cray X1E**	**AMD Opteron**	**Intel Itanium2**
dense float matrix multiply	204.7	29.5	7.8	3.0
sparse float matrix multiply	7.68	-	0.80	0.83
2-D FFT	40.5	8.27	0.34	0.15

There must be a catch...or?

- Being simple to understand is **not** the same as being simple to use.
- The copying of data to/from the local stores must be done using messages created by the programmer.
- You are all used to programming machines with cache memories, and the local store of an SPU can be seen as a cache memory, except that you must manually transfer the data!
- So why are there no hardware managed cache memories for the SPU's then?
- The researchers at IBM concluded that a sufficiently large number of applications can execute faster if we use the transistors for computing and not for cache memories — i.e. instead of only **one huge cache** we can have a number of small SPUs with their local stores.
- By letting the programmer manage the data transfers using a clever feature, very good performance can be achieved on the Cell.

A Cell Chip



- An SPE contains an SPU and a local store of 256 KB.
- On the Sony PS3 you can only use six of eight the SPU's.

A Third Model of a Multicore Machine

- The third model of a multicore machine consists of a number of nodes which can send control and data messages to each other through a general interconnection network.
- Each node has a processor, e.g. two levels of caches and a portion of the RAM memory.
- Each node implements in hardware a protocol for transferring data between caches and memory.
- The purpose of this protocol is to fetch data from a remote node at a read cache miss, and when writing to tell the other nodes which have a copy of that data that their copies have become obsolete.
- The protocol is called a **cache coherence protocol**.
- It is completely implemented in hardware.

Memory Ordering

- Consider a parallel program with two threads, T_1 and T_2, and two variables a and f, where f is a flag.
- Is the following code correct (in some sense) ? Both variables are initially zero.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        f = 1;
}
```

```
// called by T2
void w(void)
{
        while (!f)
                ;
        printf("a = %d\n", a);
}
```

- Can T_2 print out zero? (unfortunately, yes)
- T_2 might in fact not print anything!

Why T_2 May Print Zero

- The compiler can legally reorder the writes to a and f.
- The compiler can allocate both a and f to registers (across multiple functions — not common but HP have done so for 20 years) and may never have to write back f to memory if v is called in an infinite loop...
- Even if both of T_1's writes are performed in source code order, a and f may be located in RAM in different nodes and it may take longer time for the write to a to reach its node.
- What is needed are strict rules about what may or may not happen. We will look at such in Lecture 3 but until then, the following rule should be used:

Don't use normal variables for synchronization — instead use locks!

More about Unlock/Lock

- Essentially, an unlock by T_1 will have the side effect of forcing the processor to wait until all previous writes have been noticed by all other processors.
- A lock has the side effect of performing all pending incoming invalidations so the old value of a is removed.
- Then the code will work. Unlock and lock execute special instructions for this — see Lecture 5.

Parallel Programming in Java

- Creating Java threads
- Java synchronization
- The Java memory model and `volatile` attributes.

Creating Java Threads

- Either extend the `Thread` class or implement the `Runnable` interface.
- Your thread needs a `public void run()` method.
- Don't call `run()` — you should instead call `start()`.
- To wait for a thread to terminate, you use `join` in a try-catch block.

Java Synchronization

- A program which uses synchronization properly to avoid data races is said to be **thread safe**.
- Every Java object has a lock. Before entering a method declared `synchronized`, the JVM checks if the calling thread is the owner of the lock. If the lock is owned by another thread, the calling thread is blocked and is put into an **entry set** for the object.
- If no thread owns the lock the calling thread becomes the owner.
- Using `synchronized` we can avoid data races but we may end up with a deadlock, as illustrated on the next page.

Deadlock with Synchronized Methods

```
public synchronized void insert(Object item)
{
        while (count == BUFFER_SIZE)
                Thread.yield();
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
}
```

- The `remove` and `insert` methods use the same lock to avoid data races.
- Suppose the producer calls `insert`, finds the buffer full, and calls `yield()` which lets another thread run.
- When the consumer calls `remove` it is blocked since the producer still owns the lock and a deadlock has occured.

The Java Object Wait-Set

- There are two methods `wait()` and `notify()` which are used to avoid the previous deadlock problem.
- In addition to the `entry set` there is also a `wait set`.
- Calling `wait()` releases the lock, blocks the thread, and puts it into the `wait set`.
- Calling `notify()` wakes up one thread in the `wait set` if there was any there, puts it it into the `entry set` and sets its thread state to Runnable.
- Calling `notify()` does **not** release the lock.
- One can also call `notifyAll()` which wakes up all threads in the `wait set`.

More Details

- The Java object lock is a so called **recursive lock** which means that we can call other synchronised methods for the same object without being blocked.
- Of course, a thread may own multiple object locks (ie, call synchronised methods for different objects).
- A notification for an object with an empty `wait set` has no effect (ie, the object does not remember the number of notify calls, as it does remember the number of V calls for a semaphore).

Using `wait()`

- A thread waiting in the `wait set` can be interrupted, ie another thread can call its `interrupt()` method. This results in the `InterruptedException` being thrown:

```
try {
        wait();
}
catch (InterruptedException ie) { /* ignore */ }
```

The Java Memory Model

- The Java Memory Model has been changed after William Pugh identified problems with it several years ago.
- A relatively new keyword is `volatile`. It's meaning has changed since it was introduced.
- Initially, the accessing of a particular volatile attribute of an object was serialized, i.e. all threads saw these accesses in one total order.
- These accesses were, however, unrelated to accesses of other variables.
- If you updated some variables and then set a volatile flag to indicate you were done, your program was buggy since the accesses were not ordered, i.e. the updates may be seen after the new value of the flag!

New Semantics of Volatile in Java

- In more recent versions of Java an access to a volatile attribute is ordered with respect to other accesses in the same way as synchronized blocks:
  - When entering a synchronized block, or reading a volatile attribute, the cache is conceptually made invalid.
  - When leaving a synchronized block or writing a volatile attribute the cache is, again conceptually, copied to memory.
- With this semantics using a `volatile int flag` as in the previous C code works as expected.

Introducing Pthreads

- Pthreads stands for POSIX Threads and is available on all UNIX machines, including Linux and MacOS X.
- POSIX stands for Portable Operating System Interface and is an API for UNIX programmers.
- Pthreads are quite similar to Java threads so nothing dramatic will happen to you...
- In the next slides we will show you how to start a thread and synchronize threads.

Getting started

- `#include <pthread.h>`
- `pthread_t` is the type of a thread.
- Create threads using `pthread_create()`.
- Wait for a thread using `pthread_join()`.
- Terminate a thread using `pthread_exit()`.

pthread_create() 1(2)

```
int pthread_create(
        pthread_t*                  thread,          // output.
        const pthread_attr_t*       attr,            // input.
        void*                       (*work)(void*), // input.
        void*                       arg);            // input.

struct { int a, b, c } arg = { 1, 2, 3 };
status = pthread_create(&thread, NULL, work, &arg);
```

- The thread identifier is filled in by the call and attributes are optional.
- The created thread runs the `work` function and then terminates.
- A class is called a `struct` in C — no methods and everything public.
- Typically multiple arguments are passed in a struct as above.

pthread_create() 2(2)

- A zero return value from `pthread_create()` indicates success, while a nonzero describes an error printable with `perror`.
- The `work` function can return a void pointer.
- Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer.
- A thread can only be joined once.

pthread_join()

```
int pthread_join(
        pthread_t       thread,         // input.
        void**          result);        // output.
```

- The call causes the caller to wait for the termination of a thread.
- If non-NULL, the terminated thread's return value is stored in result.
- A thread can only be joined by one thread.

pthread_exit()

```
void pthread_exit(void*);        // return value from work.
```

- Either use this or a return from the work function to terminate a thread.
- At termination of the main thread using `exit` or return, all other threads are killed, so use `pthread_exit` by the main thread instead.
- After a thread has terminated, the Pthreads system waits until some other thread joins with it. Then the terminated thread's resources are recycled.
- If a thread will never be joined, it should have been detached so that the system can recycle resources at termination.

pthread_detach()

```
void pthread_detach(pthread_t thread);  // recycle at exit.
```

- A thread can be detached from the beginning by specifying an attribute saying so at `pthread_create`.
- Or, any thread can call `pthread_detach`.
- A detached thread cannot be joined.

sum.c and Dataflow.java

- Next we will look at two programs:
  - `sum.c` demonstrates a Pthread program.
  - You will parallelize `Dataflow.java` in Lab 1.
- You will later translate your `Dataflow.java` to Scala and then to C.

Parallelization of an Application

- Quite difficult usually!
- The programmer must identify parallel parts of a program, eg:
  - One outer loop iteration, or
  - one function call.
- The parallel parts which read or write the same data must communicate and be synchronised:
  - Either using message passing to communicate and wait for each other, or
  - communicate data through shared memory and synchronise with eg locks.
- Communication and synchronisation should be **minimised** to improve performance.

The Three Main Issues

- Ignore for the moment that memories are slow and caches are useful.
- Now the two main issues are:
  - Program correctness, and
  - Load-imbalance: some processors might have less work to do and must wait for others before proceeding.
- Unfortunately, memories can be hundreds of times slower than microprocessors:
  - Problem 1: communication usually creates new cache misses, and
  - Problem 2: caches in multiprocessors can introduce obscure bugs.
- All multiprocessors have some form of caches: tuning usually means exploiting them better and this is the third main issue.

Programming on a Multiprocessor

- Global variables and data allocated with malloc/calloc are shared
- Data allocated on the stack is private to a thread (but nothing prevents you from giving away a pointer to stack variables but doing so is uncommon).
- The most convenient way to parallelize a program is to do it ***incrementally***, one loop at a time.
- There is a standard for this which is an extension to C/C++/Fortran called OpenMP.
- The extensions are specified by the programmer as comments in Fortran and as pragma (preprocessor directives) for C/C++.
- GCC 4.4 supports OpenMP 3.0 from 2008.

An Example: Simulating Ocean Currents 1(2)

- For climate modelling of the earth, we can study interactions between oceans and the athmosphere.
- Our example program simulates the motion of water currents in an ocean.
- Predicting the state of the ocean at any time requires simulating:
  - athmospheric effects,
  - wind,
  - friction with the ocean floor and walls.
- An ocean is represented as a set of cross-sections, like chess boards put on each other.

An Example: Simulating Ocean Currents 2(2)

- The Atlantic is about 2,000 km $\times$ 2,000 km.
- Using 100 $\times$100 points in a cross-section gives 20 km between each point.
- We actually want to simulate at a much finer granularity.
- Time is represented by steps at which all variables are re-calculated.
- Simulating eg five years with 8 hours steps requires 5,500 steps.
- The computations are enourmous but can be parallelised.

Terminology

- A **task** is a unit of work that should be performed, and which can be performed in parallel with other tasks.
- A **thread** is a software entity which runs on a processor and works on tasks, (one at a time)
- A **processor** is the real hardware which executes one thread.

Parallelization of a Sequential Application

Four main steps in performance programming for a multiprocessor

1. Decomposition — dividing the work into parallel tasks.
2. Assignment — deciding which thread should do which tasks.
3. Orchestration — communication and synchronisation among the threads.
4. Mapping — deciding which threads should run on which processors.

- Decomposition and assignment are called **partitioning**.

Decomposition

- The number of tasks available at any time limits the amount of concurrency.
- We might want our program to have as many tasks as possible: for Ocean we can have
  - one task per set of cross-sections,
  - one task per cross-section,
  - one task for a subset of a cross-section, or
  - one task per grid point.
- What is best cannot be stated without knowing more details!
- ***We want our program to have as many tasks as can be efficiently handled.***
- Unfortunately, one partitioning may be best for one computer but not for others!
- ***Performance portability*** is complicated by varying cache sizes, number of CPU's and how the CPU's and memories are interconnected.

An example related to Ocean 1(3)

Assume a program has two phases.

- In phase 1 $n \times n$ operations are performed on grid points in parallel.
- In phase 2 the sum of values of the $n \times n$ points are accumulated.
- With p processors phase 1 can be done in time n^2/p.
- Without considering time for communication and synchronisation, phase 2 can be done such that each processor adds n^2/p values to the global sum.
- What will the speedup be if we consider synchronisation?

An example related to Ocean 2(3)

- Phase 1 can be done in parallel, taking time n^2/p.
- Since a single variable is incremented n^2 times in phase 2, it must be incremented in a critical region to avoid race conditions. This results in time n^2.
- The sequential execution time is $2n^2$.
- The parallelised execution time is $n^2/p + n^2$.
- The speedup will be $\frac{2n^2}{n^2/p+n^2} = \frac{2p}{1+p} < 2$.
- So, we need to find a better decomposition. See next slide.

An example related to Ocean 3(3)

- Let each processor sum up its n^2/p points to a private variable in phase 2.
- Add a phase 3 which adds all theses private variables, taking p time.
- New speedup becomes linear in the number of processors, p.

Assignment of tasks to threads

- Balance computation, I/O, data access, and communication.
- Easiest if it can be done well statically (at compile-time or just when the program starts and has read some specific input parameters), think eg of outer loops in matrix multiplication with a fixed amount of work for each processor.
- For other programs, eg a parallel chip simulator, it can be difficult to divide the work. A too sophisticated scheme for load balancing can also lead to too much run-time overhead.
- Run-time assignment is called dynamic assignment.
- Partitioning (decomposition and assignment) is an algorithmic step in the parallelisation and is mostly independent of the details of the machine.

Orchestration

- It is now we write source code.
- The programming model (message passing vs shared memory) determines how to do this.
- Questions in orchestration include:
  - How to organise data structures to reduce cache misses?
  - How to reduce the cost of communication and synchronisation?
  - How to reduce serialisation of access to shared data?
  - How to schedule tasks to satisfy dependences early?

Mapping

- Mapping is done in cooperation with the operating system at run-time.
- Mapping specifies which thread should run on which processor.
- Sometimes the programmer can specify on which processor a thread should run, called to **pin** it.
- Pinning can be important for large multiprocessor with a non-trivial topology (something other than a bus).
- It may very well be useful to have more threads than processors.

Owner computes

- So far we have parallelised a program with respect to computation.
- Sometimes it is better to parallelise it with respect to data (or both).
- The basic rule then is **owner computes** (and modifies) and other threads read the data.
- An extension to Fortran, called High Performance Fortran allows the programmer to distribute data structures to threads.
- In Ocean, it is natural to use the owner computes rule.