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 actors
  3. C with Pthreads (two parts)
- There will also be labs on atomic types in C++, and transactional memory
- There is also a project and a competition which consists of choosing one of these versions and to tune as much as you can — also in groups of two.
Lecturer is Jonas.Skeppstedt@cs.lth.se with office E:2190
Office hours during LP1: every weekday at 12.30 - 13.00
Lectures at 13 on Mondays and at 10 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 10-core POWER/Linux machine
Each core has up to 8 hardware threads clocked at up to 3.5 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

<table>
<thead>
<tr>
<th>F1</th>
<th>Introduction to multicore programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Multicore architectures</td>
</tr>
<tr>
<td>F3</td>
<td>Memory consistency models</td>
</tr>
<tr>
<td>F4</td>
<td>High-level parallel programming: Scala on the JVM</td>
</tr>
<tr>
<td>F5</td>
<td>POSIX Threads details</td>
</tr>
<tr>
<td>F6</td>
<td>Threads in the new ISO C Standard: C11</td>
</tr>
<tr>
<td>F7</td>
<td>Parallelizing sequential C codes by hand</td>
</tr>
<tr>
<td>F8</td>
<td>Cache aware programming for multicores and Cell</td>
</tr>
<tr>
<td>F9</td>
<td>Transactional memory</td>
</tr>
<tr>
<td>F10</td>
<td>Parallelizing sequential C codes automatically</td>
</tr>
<tr>
<td>F11</td>
<td>Accelerators</td>
</tr>
<tr>
<td>F12</td>
<td>Research trends</td>
</tr>
</tbody>
</table>
Contents of Lecture 1

- A graph application: live variables analysis
- 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
a = u + v;
if (a > b) {
    y = u;
} else {
    a = u - v;
    b = a - 1;
}
y = a * b;
Control-flow graph: Basic Blocks and Branches

Basic block: sequence of instructions with no label or branch
CFG: directed graph with basic blocks as nodes and branches as edges
Control-Flow Graph: the CFG View

Special nodes:

- the first node is called $s$ — start
- the last node is called $e$ — exit
A variable $x$ is live at a point $p$ (instruction) if it may be used in the future without being assigned to.

- $a$ is live from the function start and up to and including the add, and then after $S_3$ and up to and including the negation.
- $b$ is live from the start and up to and including the subtraction.
- $c$ is live from $S_1$ and up to and including the multiplication.
Live Variables Analysis is used for different purposes.

For example an assignment to a local variable which is not used in the future can be removed.

Two variables live at the same point in the program are said to **interfere** and cannot be allocated the same CPU register.
Live variables analysis is performed in a local and a global analysis.

In the local analysis, each basic block (vertex) is inspected with the purpose of finding which variables are first used or first defined (assigned to).

The information that a variable is live propagates backwards in the control flow graph (CFG) from a use and to its definition.

The propagation of a use stops at a definition. The use in \( a + 13 \) is defined by the definition \( a = 14 \).

\[
\begin{align*}
  a &= 44; \\
  b &= a + 11; \\
  a &= 14; \\
  b &= a + 13;
\end{align*}
\]

In the global analysis the local information is combined to produce the complete view.

Sometimes use/def is call gen/kill.
procedure local_live_analysis
    for each vertex w do
        for each stmt s do /* forward direction */
            for each used variable x of s do
                if (x ∉ def(w))
                    add x to use(w)
            for each defined variable x of s do
                if (x ∉ use(w))
                    add x to def(w)
    end
Local Analysis Example

0 $a = 1$
1 $b = a+1$
2 $c = a+2$
3 $d = b+c$
4 $b = c+1$
5 $a = 2$
6 $b = a-1$
7 $c = a+3$
8 $\text{ret } a+b$

<table>
<thead>
<tr>
<th>vertex</th>
<th>use</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td>${a}$</td>
</tr>
<tr>
<td>1</td>
<td>${a}$</td>
<td>${b}$</td>
</tr>
<tr>
<td>2</td>
<td>${a}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>3</td>
<td>${b, c}$</td>
<td>${d}$</td>
</tr>
<tr>
<td>4</td>
<td>${c}$</td>
<td>${b}$</td>
</tr>
<tr>
<td>5</td>
<td>${a, b}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>6</td>
<td>${a}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>7</td>
<td>${a}$</td>
<td>${b}$</td>
</tr>
<tr>
<td>8</td>
<td>$\emptyset$</td>
<td>${a}$</td>
</tr>
</tbody>
</table>
procedure global_live_analysis
  change ← true
  while (change) do
    change ← false
    for each vertex w do
      out(w) ← \( \bigcup \{ \text{in}(s) | s \in \text{succ}(w) \} \)
      old ← in(w)
      in(w) ← use(w) \( \cup \) (out(w) − def(w))
      if (old \( \neq \) in(w))
        change ← true
    end
  end
Global Analysis Example: Iteration 1

\[
\begin{align*}
\text{out}(w) & \leftarrow \bigcup_{s \in \text{succ}(w)} \text{in}(s) \\
\text{in}(w) & \leftarrow \text{use}(w) \cup (\text{out}(w) - \text{def}(w))
\end{align*}
\]

<table>
<thead>
<tr>
<th>vertex</th>
<th>use</th>
<th>def</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>{a, b}</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{a, b}</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{b}</td>
<td>{a, b}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>3</td>
<td>{b, c}</td>
<td>{d}</td>
<td>{a, c}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>2</td>
<td>{a}</td>
<td>{c}</td>
<td>{a, b, c}</td>
<td>{a, b}</td>
</tr>
<tr>
<td>8</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a, b}</td>
<td>{b}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
<td>{b}</td>
<td>{a, b, c}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>6</td>
<td>{a}</td>
<td>{c}</td>
<td>{a, c}</td>
<td>{a}</td>
</tr>
<tr>
<td>1</td>
<td>{a}</td>
<td>{b}</td>
<td>{a, b}</td>
<td>{a}</td>
</tr>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Global Analysis Example: Iteration 2

\[
\begin{align*}
\text{out}(w) & \leftarrow \bigcup_{s \in \text{succ}(w)} \text{in}(s) \\
\text{in}(w) & \leftarrow \text{use}(w) \cup (\text{out}(w) - \text{def}(w))
\end{align*}
\]

<table>
<thead>
<tr>
<th>vertex</th>
<th>use</th>
<th>def</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>{a, b}</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{a, b}</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{b}</td>
<td>{a, b}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>3</td>
<td>{b, c}</td>
<td>{d}</td>
<td>{a, c}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>2</td>
<td>{a}</td>
<td>{c}</td>
<td>{a, b, c}</td>
<td>{a, b}</td>
</tr>
<tr>
<td>8</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a, b}</td>
<td>{b}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
<td>{b}</td>
<td>{a, b, c}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>6</td>
<td>{a}</td>
<td>{c}</td>
<td>{a, c}</td>
<td>{a}</td>
</tr>
<tr>
<td>1</td>
<td>{a}</td>
<td>{b}</td>
<td>{a, b}</td>
<td>{a}</td>
</tr>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Your task is to compute the in and out sets for each vertex.

See Dataflow.java in Lab 1.

Modify the liveness method.

You will later solve the same problem in Scala (Lab 2) and C (Lab 4).
High score list of assignment DATAFLOW in edan26

Wed Dec 13 14:27:06 2017 (Sweden time zone)

Top results --- execution time

<table>
<thead>
<tr>
<th>position</th>
<th>username</th>
<th>time/s</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>fys09akl</td>
<td>3.5</td>
<td>2015-10-19 10:30:41</td>
</tr>
<tr>
<td>2.</td>
<td>dat11vro</td>
<td>3.6</td>
<td>2015-10-16 21:04:29</td>
</tr>
<tr>
<td>3.</td>
<td>ref</td>
<td>3.6</td>
<td>2017-10-04 10:09:47</td>
</tr>
<tr>
<td>4.</td>
<td>linus</td>
<td>3.9</td>
<td>2015-10-14 08:56:03</td>
</tr>
<tr>
<td>5.</td>
<td>jesper</td>
<td>4.1</td>
<td>2017-10-18 10:41:57</td>
</tr>
<tr>
<td>6.</td>
<td>dat11msi</td>
<td>4.3</td>
<td>2015-10-15 16:03:32</td>
</tr>
<tr>
<td>7.</td>
<td>ada10pst</td>
<td>5.1</td>
<td>2015-10-05 22:33:53</td>
</tr>
<tr>
<td>8.</td>
<td>ada08lka</td>
<td>5.5</td>
<td>2015-10-14 13:25:24</td>
</tr>
<tr>
<td>9.</td>
<td>dat12mla</td>
<td>5.6</td>
<td>2015-10-09 11:56:06</td>
</tr>
<tr>
<td>10.</td>
<td>dat11ael</td>
<td>8.1</td>
<td>2015-10-07 11:30:58</td>
</tr>
</tbody>
</table>
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 about USD 11000 new in 2016.
- We bought it from `ebay.de` for EUR 299
- This machine *can* have 14 disks and 1 TB RAM
- Large scale servers cost much more of course.

![Table of featured models with processor speed, system memory, and internal storage details.](image-url)
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.
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 $S = \frac{T_{\text{slow}}}{T_{\text{fast}}}$
- Speedup $S_N = \frac{1}{(1-P)+P/N}$
- Assume $P = 0.9$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.8</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
</tr>
<tr>
<td>10</td>
<td>5.3</td>
</tr>
<tr>
<td>100</td>
<td>9.2</td>
</tr>
<tr>
<td>$\infty$</td>
<td>10</td>
</tr>
</tbody>
</table>

- 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:

```c
pthread_mutex_lock(L);
count += 1;
pthread_mutex_unlock(L);
```
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`. 
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.
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 a **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 below but first an unpleasant realization
One very important (and sometimes unpleasant) 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?
So: Helgrind does **not** detect deadlocks but does something much more useful.

Helgrind observes the lock-acquire order and complains when two threads acquire certain locks in the opposite order.

```c
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;
}
```
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==
```