The Multicore Programming Course in Lund

Welcome to the Multicore Programming course in Lund

- 12 lectures and 6 labs
- You will implement the preflow-push algorithm with different methods of exploiting multicore hardware:
  1. Lab 1: Scala with actors
  2. Lab 2: Java and C with locks
  3. Lab 3: C with barriers
  4. Lab 4: C with atomic variables
  5. Lab 5: Rust
  6. Lab 6: Clojure and C with transactional memory
- At `forsete.cs.lth.se` you will upload your source code and see a highscore list
- The ten first are shown with stilid and after ten no stilid

Purpose of the course

- To learn how to write fast parallel programs on common hardware
- Understand pros and cons of different approaches to parallelizing an algorithm
- Understand that language differences for performance mostly is a matter of different syntax — what is important is how we compute and move data around in the machine
- But some languages result in more work for the computer when doing essentially the same thing
- Understand which languages make it easier to have a fast and bug free program
- Get a good skepticism against overly hyped new things including new hardware which requires new languages

Administration

- Lecturer is `Jonas.Skeppstedt@cs.lth.se` with office E:2190
- Course web page `cs.lth.se/edan26`
- You can do labs either in E:Venus or on Discord
- Book a time to ask questions and oral exam at `calendly.com/forsete`
- Or ask questions at Discord — and it is nice if you help others there
- Course files can be found with link to `tresorit.com` (see email)
- Email me your stilid if you have not already an account at `power.cs.lth.se` or `forsete.cs.lth.se`
- Each core in power has up to 8 hardware threads
- You can work on other machines if you wish but performance measurements are to be done on it.
- You can login with `ssh stilid@power.cs.lth.se` and to `forsete` with ssh from power

Lectures

F1 Introduction to multicore programming

F2 High-level parallel programming: Scala on the JVM

F3 Java and POSIX Threads details

F4 Multicore architectures

F5 Memory consistency models

F6 Threads and the memory model in C/C++ and Java

F7 OpenMP for C/C++, and Rust

F8 Transactional memory in Clojure and C

F9 Cache aware programming for multicores

F10 Parallelizing compilers

F11 Hardware accelerators

F12 Research trends

Introduction to Multicore Programming

Contents of Lecture 1

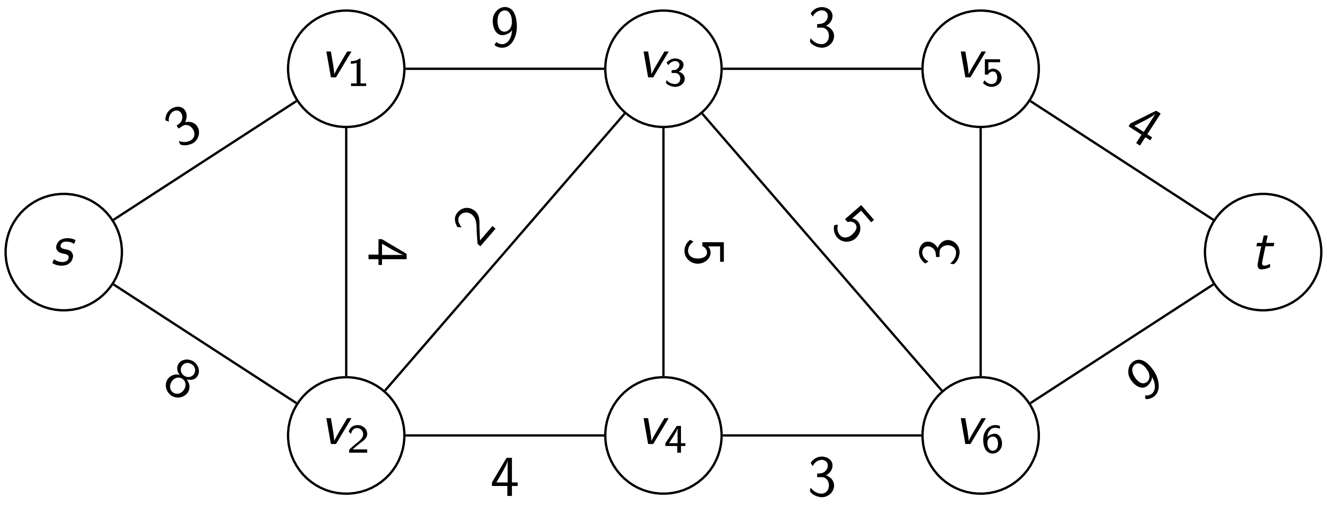
- The preflow-push algorithm
- Advantages with multithreading
  - Performance
  - Sometimes simpler programming
- Disadvantages with multithreading
  - Overhead
  - Usually more complex programming
- Two tools to help us avoid threading errors

A flow network

- The maximum flow problem
- The Ford-Fulkerson algorithm
- The preflow-push maximum flow algorithm

A flow network

- A graph $G(V, E)$ — directed or undirected — we will use undirected
- Each edge $e \in E$ has a nonnegative capacity $c(e)$
- A source node $s \in V$ with no incoming flow
- A sink node $t \in V$ with no outgoing flow
- An example:



- The flow on an edge can be up to the capacity and in either direction
- We want to have as much flow as possible from s to t

Flow conservation constraint

- The flow coming in to a vertex v must equal the flow going out from v
- This **flow conservation constraint** does not apply to the source s and the sink t

$$v \in V - \{s, t\} : \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out from } v} f(e)$$

The maximum flow problem

- The value of a flow f is $\sum\limits_{e \text{ out from } s} f(e)$
- The **maximum flow problem** is to find a flow f with maximum value

The Ford-Fulkerson algorithm: overview

- The basic idea is very simple
  1. Start with a flow $f(e) = 0$ for every $e \in E$
  2. Look for a simple path p from s to t such that on every edge (u, v) in p we can increase the flow in the direction from u to v
  3. If we could not find any such path, we have found the maximum flow
  4. Let each edge $e = (u, v)$ on p have a value $\delta(e)$, which means room for improvement, or how much we can increase the flow on that edge
  5. Let Δ be the minimum of all $\delta(e)$ on p
  6. Increase the flow by Δ along the path p
  7. `goto` 2
- Sometimes one selected path can block later paths so we need to be able to reduce the flow on edges as well.
- Reducing flow can be done with a so called residual graph, and if you have not taken EDAF05, you can learn more about it there.
- Ford-Fulkerson has not been easy to parallelize since it finds one s-t path at a time and parallel path finding is difficult to make fast.

The preflow-push maximum network flow algorithm

- Usually the fastest algorithm for maxflow
- Instead of maintaining a valid flow which satisfies both the conservation constraint and the capacity constraint, it uses a weaker type of flow which only satisfies the capacity constraint
- The weaker flow is called a **preflow**
- At algorithm termination, the preflow will have become a valid flow
- In addition, it uses a **height** function for each node
- It also uses the residual graph but you can actually ignore that
- See EDAF05 if you want to learn why preflow-push works
- The algorithm itself is really simple but why it works is more subtle but it is intuitive anyway :-)

The preflow

- For each edge $e \in E$ we have $0 \leq f(e) \leq c(e)$
- Thus the capacity constraint is always satisfied
- Instead of the conservation constraint, a node $u \neq s$ is allowed to have more incoming flow than outgoing
- Thus for each node $u \in V - \{s\}$ we have

$$\sum_{e \text{ into } u} f(e) \geq \sum_{e \text{ out from } u} f(e)$$

- The **excess preflow** of a node u is

$$e_f(u) = \sum_{e \text{ into } u} f(e) - \sum_{e \text{ out from } u} f(e)$$

- Only s has a negative excess preflow

The height function

- There is a height function $h : V \to \mathbb{N}$
- $h(s) = n$
- $h(t) = 0$
- A node u can only push flow to a node v if $h(u) > h(v)$
- For s and t the heights cannot change and for other nodes they start at 0 and can increase
- $0 \leq h(u) \leq 2n - 1$ for $u \notin \{s, t\}$

push

- Three conditions must be satisfied for a push:
  1. $e_f(v) > 0$
  2. $h(v) > h(w)$
  3. $(v, w) \in G_f$ this simply means there is more capacity to use on that edge in the direction from v to w

```
procedure push(f, h, v, w)
    assert $e_f(v) > 0$ and $h(v) > h(w)$ and $(v, w) \in G_f$
    $e \leftarrow (v, w)$
    $\delta \leftarrow \min(e_f(v), c(e) - f(e))$
    increase $f(e)$ by $\delta$
```

- This is somewhat simplified since we have to consider if the current flow goes from v to w or from w to v. A flow from v to w can be represented as positive and from w to v can be as negative.

relabel

- The purpose of a relabel is to increase the height of a node
- It is done when the node has excess flow but nowhere to push it due to neighbors have too big height

```
**procedure** *relabel*$(f, h, v)$
    **assert** $e_f(v) > 0$
    $h(v) \leftarrow h(v) + 1$
```

The preflow push algorithm

```
function $preflow\_push(G, s, t)$
    $h(s) \leftarrow n$
    for each node $u \neq s$ do $h(u) \leftarrow 0$
    for each edge $(s, v)$ do $f(s, v) \leftarrow c(s, v)$
    for each edge $(u, v)$ such that $u \neq s$ do $f(u, v) \leftarrow 0$
    while there is a node $v \neq t$ with $e_f(v) > 0$ do
        if there is a node $w$ such that $h(v) > h(w)$ and $(v, w) \in G_f$ then
            $push(h, f, v, w)$
        else
            $relabel(h, f, v)$
    return $f$
```

Lab 0

- Now you may want to check out lab 0 with a sequential implementation of preflow push
- The source is in Tresorit
- You can set `PRINT` to one in it to get output saying what is happening
- This can be a useful reference for debugging your programs

What makes it easy to parallelize preflow-push

- We can work on every node and edge concurrently
- Of course some locking may be needed
- No sequential loop as in Ford-Fulkerson
- Consider an edge between v and w. Both cannot change it at the same time since only the higher one can push
- Of course $h(v)$ and $h(w)$ can change in the future

What makes it difficult to parallelize preflow-push

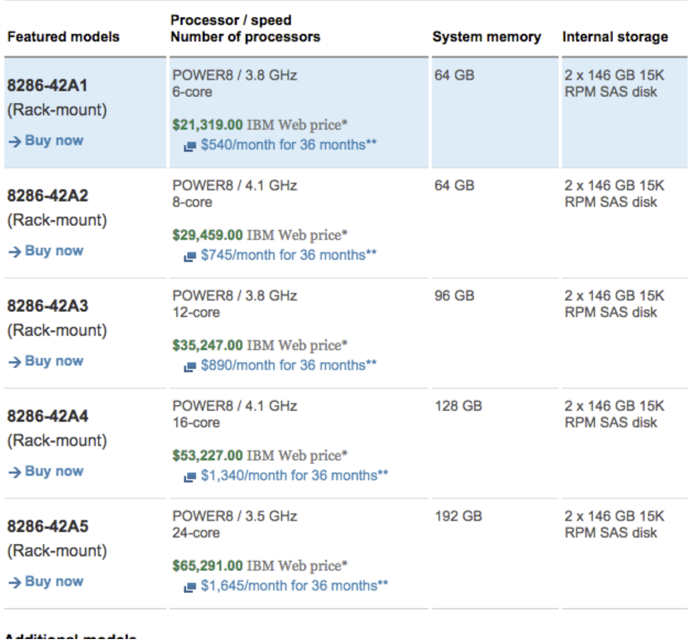
- The amount of work in a push or relabel is really not much
- Somehow we must make sure two threads do not work on the same node or edge
- There is a huge risk of slowing down the program due to overhead of managing the threads

Multicore Programming

- Parallel programming is much more fun than sequential programming!
- It's very nice to see many threads speeding up our code :-)
- The computer industry revolution to make parallel computing widespread has already happened and we need to learn this
    - More accurate and/or faster research results when solving big problems (medicine, climate, products,...)
    - Fun to know how your computer really works
    - Good to know advantages/disadvantages of different languages
    - Easier to sell products which are faster than the competition
- 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
- It can have 14 disks and 1 TB RAM but has 16 GB RAM
- 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

Additional models

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**.

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 a 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.
- Both Google sanitizer and Valgrind can detect threading problems.
- In ISO C/C++ 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 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

Non-Deterministic Execution

- 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?

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 certain locks in an inconsistent 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==
```