Performance Engineering

Perf. Eng. Vs Optimisation

Performance Engineering :

a disciplined approach to achieving and maintaining performance goals throughout the entire system lifecycle

Optimisation :

an attempt to improve the performance of some part of an existing system by local re-designing or re-writing

Digression: Engineering Processes

- The real-world is not like here: you never just "start coding"
  - Stuff before coding: *requirements, spec. docs, cost-benefit*
  - Stuff after coding: *testing, deployment, maintenance*
  - We don't always do a good job of explaining this...

- Most companies have formal design processes
  - Old-school rigorous: waterfall, spiral
  - New-hotness dynamic: Agile, SCRUM
- Embedded systems companies tend towards rigorous
  - More reliable when shipping a discrete product
- Software/HPC companies tend towards dynamic
  - Ship early, ship often – e.g. Facebook don't have formal testing

Old-school : Waterfall

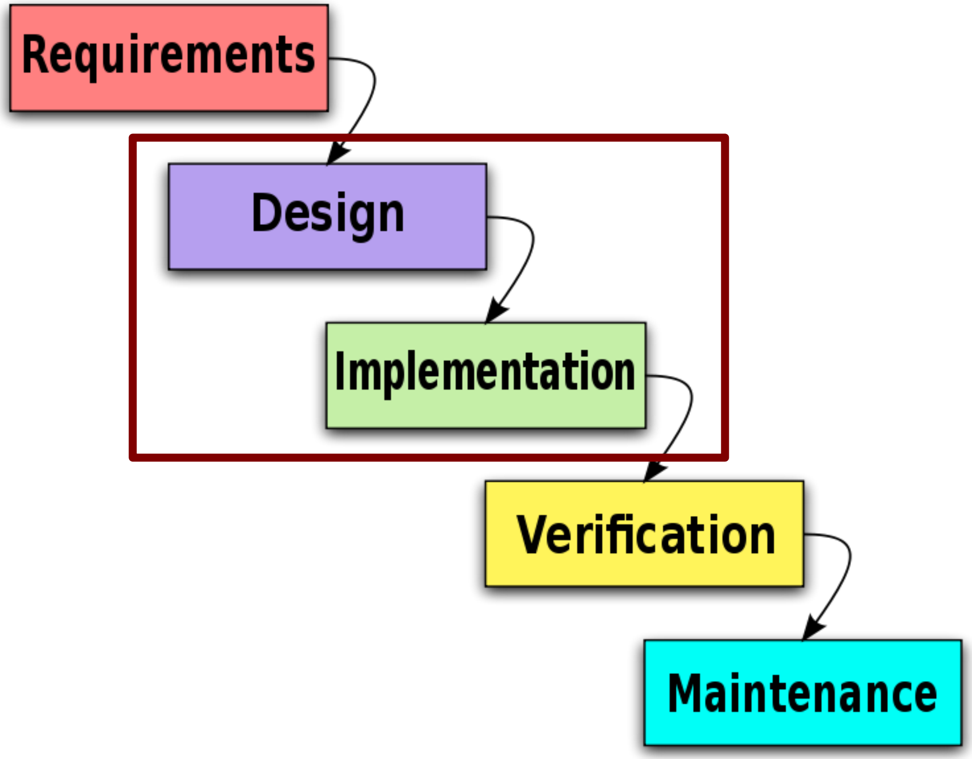
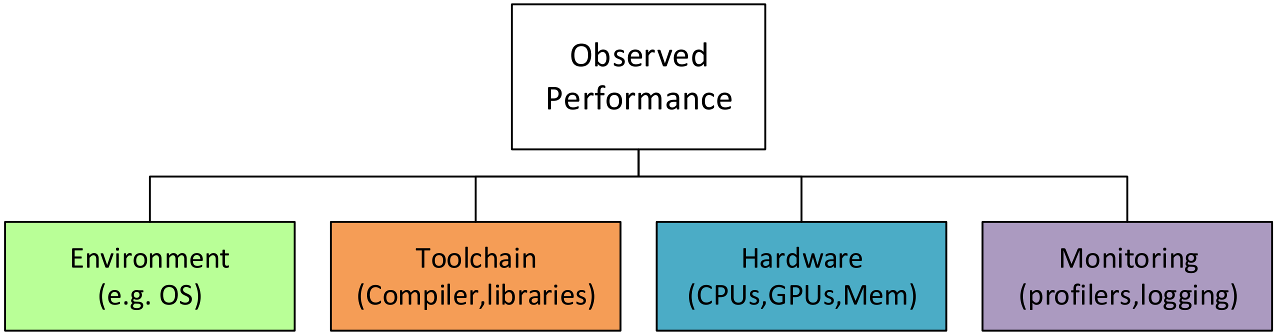
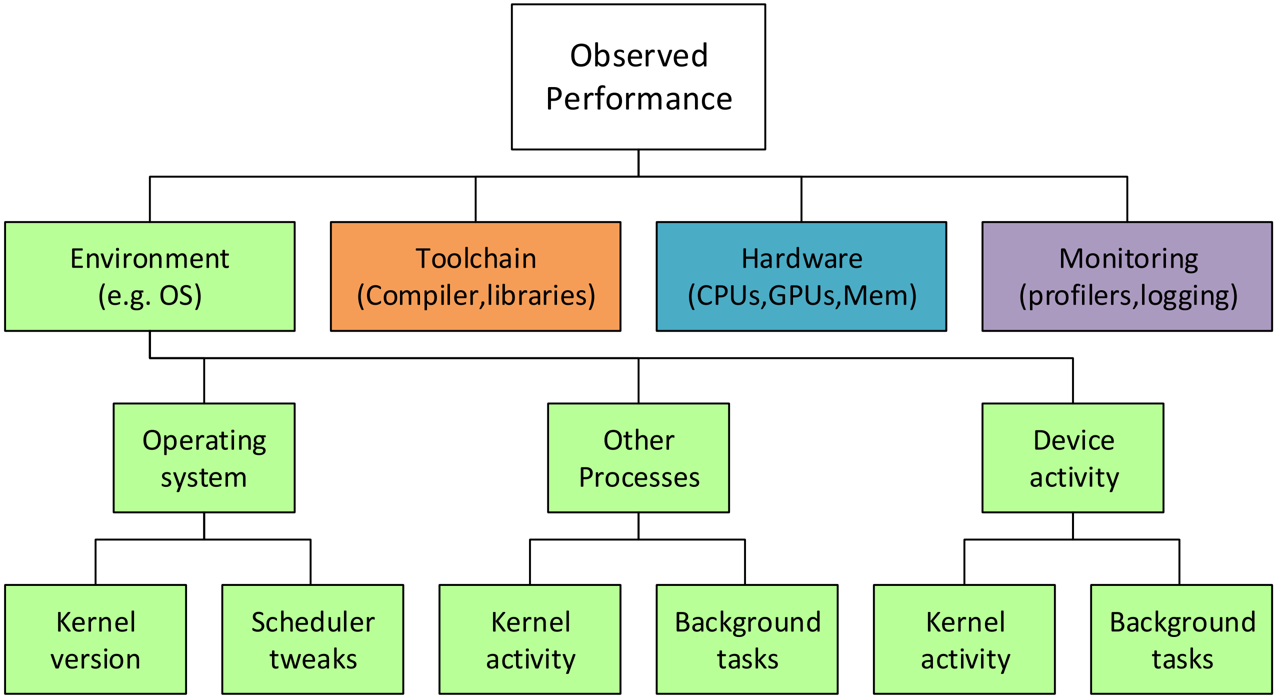


Image: Paul Smith; http://en.wikipedia.org/wiki/File:Waterfall_model_(1).svg

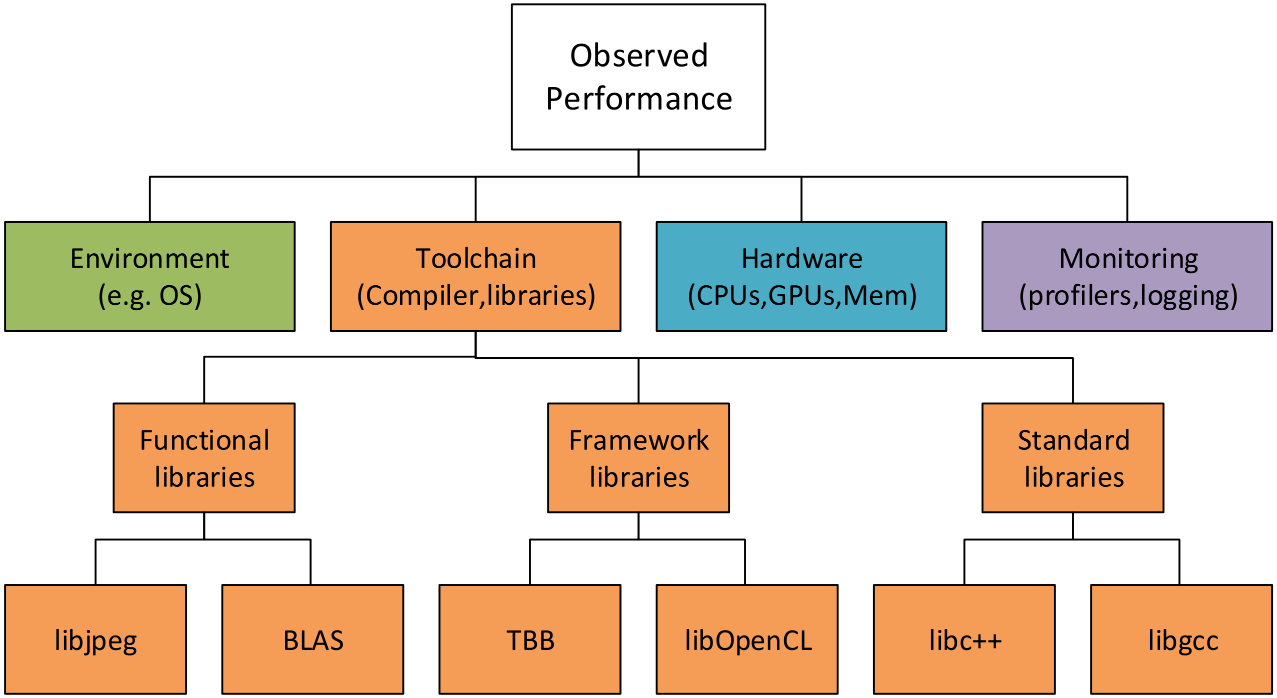
Isolating Performance



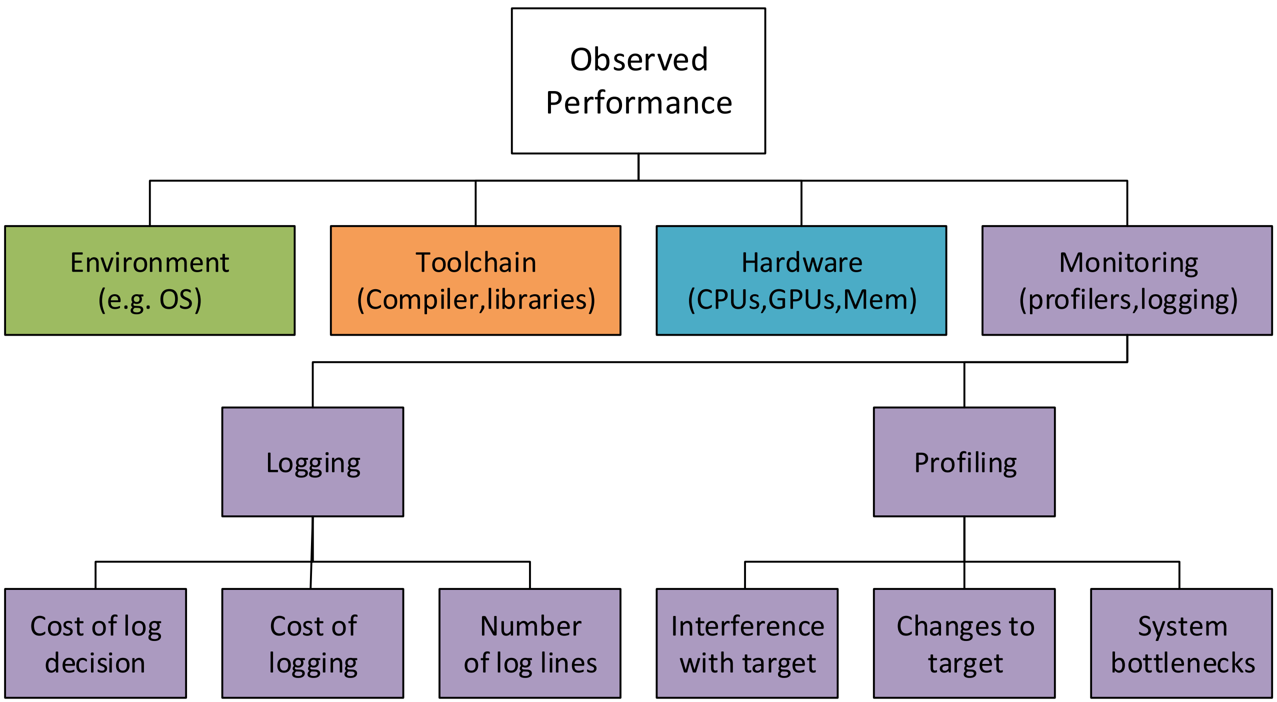
Isolating Performance : Environment



Isolating Performance : Toolchain



Isolating Performance : Observers



Be careful with logging

- Be very careful with logging on an inner loop
  - There is a cost to deciding whether to log
  - Branch predictor will help, but still costs
- Be aware that log statements affect optimisation
  - Compiler may decide not to inline due to logging
  - Possibility of log may force values to stay live
  - Possibility of log may stop calculations being removed
- Logging is good, but has a cost

Profilers

- *Profiling*: where does the program spend the most time?

- Profiling is broadly used for three purposes:
  - Exploration: “*I have no idea what this program does*”
  - Optimisation: “*We’ve done the big stuff, let’s look at details*”
  - Diagnosis: “*why has the 99% response time increased*”
- There are also three main types of profiler:
  - **Instrumented**: the program is modified to track activity
  - **Event-based**: important program events are tracked
  - **Sampling**: program state is sampled statistically as it runs
- Different purposes need different profilers

What should be made faster?

- CW5 is a function acceleration/optimisation task:
  1. Here is a function with 20 lines of code
  2. Make that single function faster

- Not very realistic
  - You know immediately where the slow part is
  - The important code all fits on one page
  - The important code is all in one file
  - There are no abstraction layers in the way
  - All inputs and outputs are visible and in memory

Exploring your own code

- Scenario 1 : ***You*** wrote the code
    1. Spend days/weeks on initial development
    2. Needed to integrate 12 different third party libraries
    3. Transform and change algorithms during debugging
    4. Now start optimisation it

- Problems occurring
    - How long do those third party libraries take?
    - Is most of the time spent on one stage, or split across multiple
    - Where does the “slow part” start?
    - Which functions are indirectly called by the slow part
    - Where are the any memory allocations happening?

Exploring existing code-bases

- Scenario 2 : ***Someone else*** wrote the code
  1. “Here is 100k lines of code. Do \`make all\`, then \`make test\`.
  2. Make it faster.

- Problems occurring:
  – How does it work? : “*we’re not too sure*”
  – What do the files do? : *“Steve knows that, but he’s away*”
  – How do I run it? : “*I already said, \`make test\`*”
  – Where should I look? : “*isn’t that your job?*”
  – What parts are slow? : “*I’m a doctor, not an engineer*”

Exploration via profiling

- Profiling can give you a big picture view of a program
  - What functions are actually called?
  - Which functions take most of the time?
  - Which functions ***contain*** most of the time?
  - How often are functions called?
  - Which external library functions are actually called?

- Exploration mainly done at the “function” level
  - Not so interested in any one statement yet

gprof : the classic profiler

- gprof comes with the standard GNU toolchain (gcc,g++,…)
- It is an *instrumenting* profiler
  - Extra instructions will be added to the program
  - Need to recompile the program for a profile build
  - Program will then generate profiling information

- Profiling process
  - Add `` `-pg` `` flag to g++ when compiling
  - Run your program as normal
  - Profiling results will be in `` `gmon.out` ``
  - Use `` `gprof gmon.out` `` to view the results

gprof : Limitations

- Optimising compilers _love_ to inline functions
  - gprof only instruments function calls
  - If everything is inlined, you'll see that `main` takes 100% of time
  - Can avoid this with ``__attribute__ ((noinline))``
    - Make sure you don't make things slower though…

- The output with C++ names is horrible
  - Looks a bit better in a text file
  - Graphical tools can help (e.g. gprof2dot)

- Instrumentation does slow down the program
  - A small but noticeable cost for each function call

perf : a sampling+event profiler

- perf is part of linux, supported at the kernel level
- You can use it in a number of ways:
  - Profile CPU time via sampling
  - Profile hardware usage by monitoring performance counters
  - Profile OS usage usage by hooking kernel events
- Unlike gdb, the binary does not need to be modified
  - The kernel will intermittently sample the program counter
  - Very low overhead as long as sampling is slow
- Usage:
  1. `perf record –g -- your-program arg1 arg2 arg3`
  2. `Perf report`

Profiling GPU code

- Measuring execution time on the GPU is hard
- When the API call finishes is *not* when the hardware finished

```
queue.enqueueWriteBuffer(buffState, CL_TRUE, 0, cbBuffer, &world.state[0]);

for(unsigned t=0;t<n;t++){
    kernel.setArg(3, buffState);
    kernel.setArg(4, buffBuffer);
    queue.enqueueNDRangeKernel(kernel, offset, globalSize, localSize);

    queue.enqueueBarrier();

    std::swap(buffState, buffBuffer);
    world.t += dt;
}

queue.enqueueReadBuffer(buffState, CL_TRUE, 0, cbBuffer, &world.state[0]);
```

clGetEventProfilingInfo : OpenCL events

- The OpenCL API is event oriented
  - Operations in a queue happen once dependencies are satisfied
  - They may be scheduled by software or hardware
- clGetEventProfilingInfo lets you see how long each part took
  - COMMAND_QUEUED : when it was added to the queue
  - COMMAND_SUBMIT : time it left queue and went to device
  - COMMAND_START : time it started on the device
  - COMMAND_END : time it finished on the device
- Can be used both for kernels and memory copies
  - Which part is actually taking the time?

```
void dumpEventTiming(cl::Event ev)
{
    uint64_t queue, start, finish;
    clGetEventProfilingInfo(ev, CL_PROFILING_COMMAND_QUEUED,
                                               sizeof(cl_ulong), &queue, NULL);
    clGetEventProfilingInfo(ev, CL_PROFILING_COMMAND_START,
                                               sizeof(cl_ulong), &start, NULL);
    clGetEventProfilingInfo(ev, CL_PROFILING_COMMAND_FINISH,
                                               sizeof(cl_ulong), &finish, NULL);

    // Print queue -> start and start -> end times
    fprintf(stdout, "%f %f\n", (start-queue)*1e-9, (finish-start)*1e-9);
};

for(unsigned t=0;t<n;t++){
    kernel.setArg(3, buffState);
    kernel.setArg(4, buffBuffer);
    queue.enqueueNDRangeKernel(
           kernel, offset, globalSize, localSize,
           NULL, &evKernels[t]
    );
    queue.enqueueBarrier();
    std::swap(buffState, buffBuffer);
    world.t += dt;
}
```

Profiling for optimisation

- Profiling for low-level optimisation can be difficult
  - Isolating bottle necks to a function is easy
  - Isolating it to source lines is hard
- More advanced profilers can do line-by-line analysis
  - AMD CodeAnalyst
  - Intel Vtune
- More advanced profilers have problems:
  - Sometimes limited to one CPU
  - Often rely on a GUI
  - Difficult to run remotely
- Profilers are not the panacea you might hope for

Tips for profiling

- Keep your baseline consistent
  - Same platform, same input, same compiler flags
- Change one thing at a time
  - Don’t change an algorithm ***and*** change inputs
- Record your history
  - Note down the main functions taking time
  - Track the changes as you apply optimisations (spreadsheet?)
- Make sure your program runs for long enough
  - Is most of the cost just startup cost?

Engineering : The PCAM Approach

- PCAM approach comes from traditional parallel programming
  - Book online: http://www.mcs.anl.gov/~itf/dbpp/text/node15.html
  - Works well in a multi-core + GPU (+ *FPGA*) world

- ***Partitioning***: split the problem into as many parts as possible

- ***Communication***: which tasks have dependencies?

- ***Agglomeration***: when to move from parallel to serial

- ***Mapping***: which tasks go to which device

Partitioning

- Try to identify all the parallelism available in the application
  - Identify true dependencies: X must occur before Y
  - Reduce false dependencies: convert `for` to `parallel_for`

- Don’t worry about task size
  - If you could have a task of one instruction, what could you do?
  - Pretend the synchronisation overhead is zero

- ***Do*** worry about total work and critical path
  - Lots of tasks in a linear chain is no good
  - Using an algorithm with poor big-O is dangerous

Partitioning : Checklist

1. Do you have ~10x more tasks than processors?
2. Does the partitioning avoid redundant computation?
3. Are tasks of similar size?
4. Does the number of tasks scale with problem size?

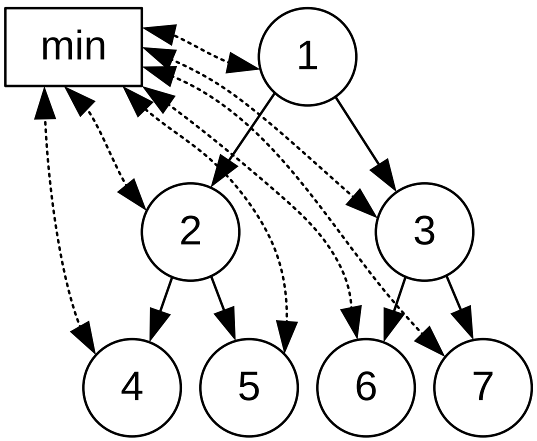
Communication

- Need to worry about two main types of communication
  - Data movement: data must move or be shared between tasks
  - Synchronisation: dependencies in task schedule

- How far is data moving and where is it moving to?
  - ***Local vs Global***: how many tasks communicate
  - ***Structured vs Unstructured***: are tasks a grid, a tree, arbitrary?
  - ***Static vs Dynamic***: is the communication known in advance

- How can you plan communication at a high-level?

Communications in a reduction




```
void Version1(
    int i, int N,
    atomic<int> *pMin
){
    if(2*i<N)
        spawn Version1(2*i,N,pMin);
    if(2*i+1<N)
        spawn Version1(2*i+1,N,pMin);

    int value=BigFunction(i);

    atomic_min(pMin, value);
}
```

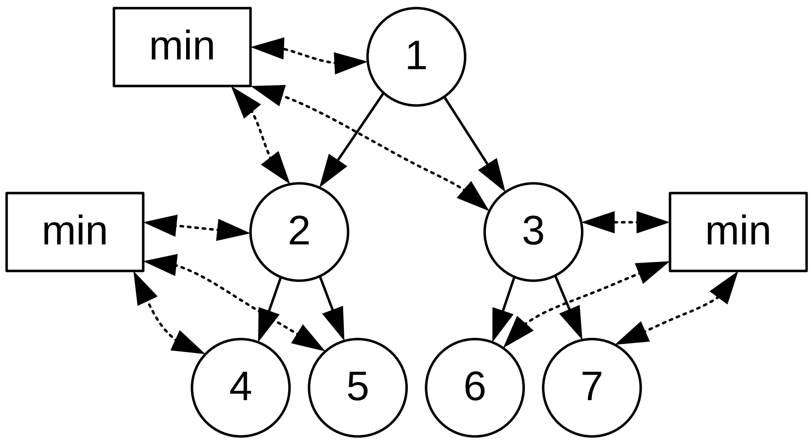
Finding a shared minimum

- Need to use `fetch_and_store` to atomically exchange value
  - Atomically reads the current value while writing the new value

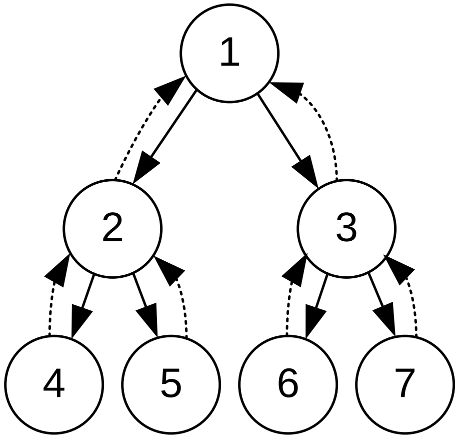
    `T tbb::atomic<T>::fetch_and_store(T value);`

```
void atomic_min(tbb::atomic<unsigned> *res, unsigned curr)
{
    while(true){
        unsigned prev=res->fetch_and_store(curr);
        if(prev>curr)
            break;
        curr=prev;
    }
}
```

  - OpenCL 1.1 has an `atomic_min` primitive, with HW support

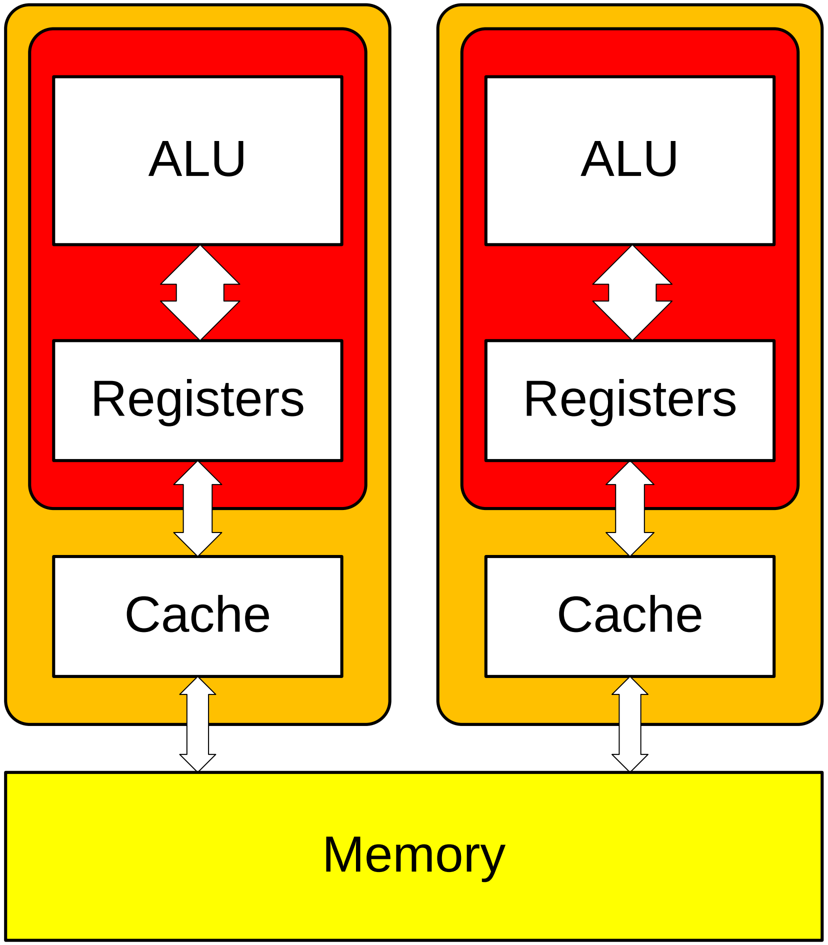



```
void Version2(
  int i, int N,
  atomic<int> *pMin
){
   atomic<int> lMin=0;

   if(2*i<N)
     spawn Version2(2*i,N,&lMin);
   if(2*i+1<N)
     spawn Version2(2*i+1,N,&lMin);

   int value=BigFunction(i)
   atomic_min(&lMin, value);

   atomic_min(pMin, (int)lMin);
}
```



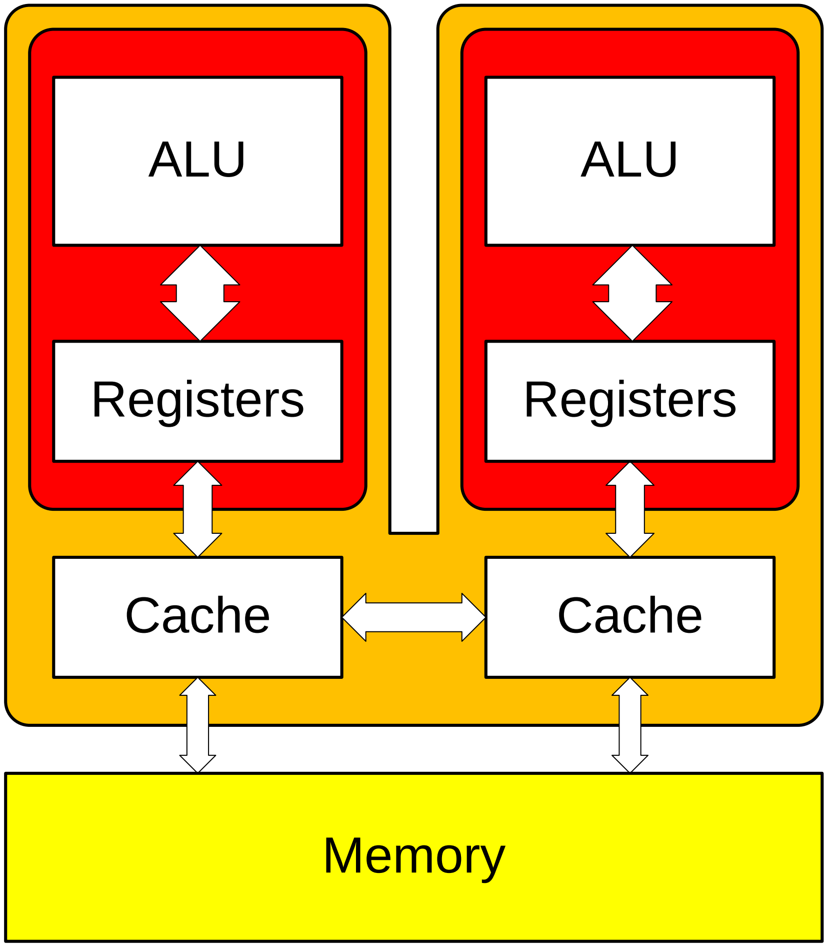

```
int Version3(int i, int N)
{
   int xC1=INT_MAX, xC2=INT_MAX;
   if(2*i<N)
     xC1=spawn Version3(2*i,N);
   if(2*i+1<N)
     xC2=spawn Version3(2*i+1,N);

   int value=BigFunction(i)

   sync;

   return min(value, xC1, xC2);
}
```

Parallel caches



- We have parallel CPUs
  - Each CPU has local caches
  - Shared underlying memory
- How does `tbb::atomic` work?

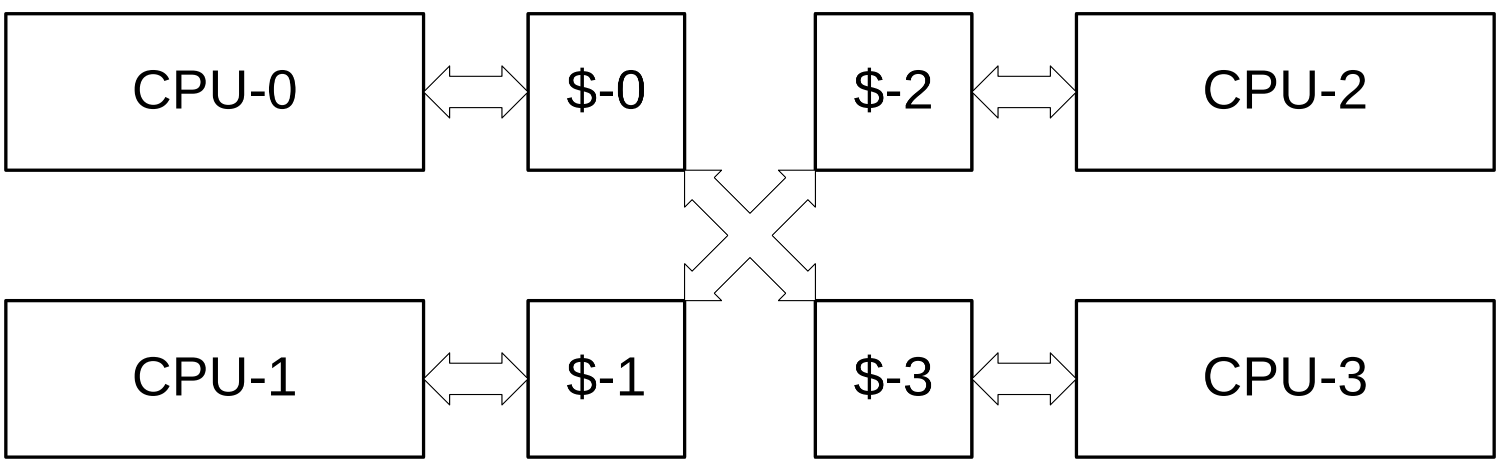
Parallel caches



- We have parallel CPUs
  - Each CPU has local caches
  - Shared underlying memory
- CPUs connect via caches
  - Memory is too dumb
- Same data in many caches
  - e.g. shared read-only data
- What about writes?
  - Lazy consistency
- Atomics: ***cache coherence***
  - Only one CPU can modify atomic data at a time

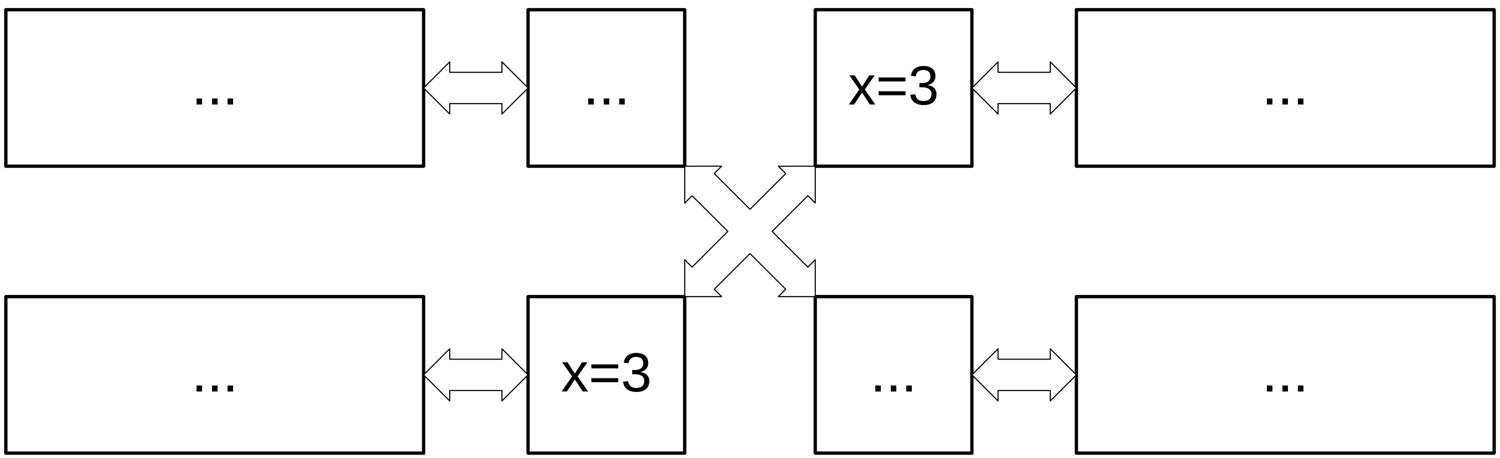
Atomics through locking

- Multiple processors are connected through caches



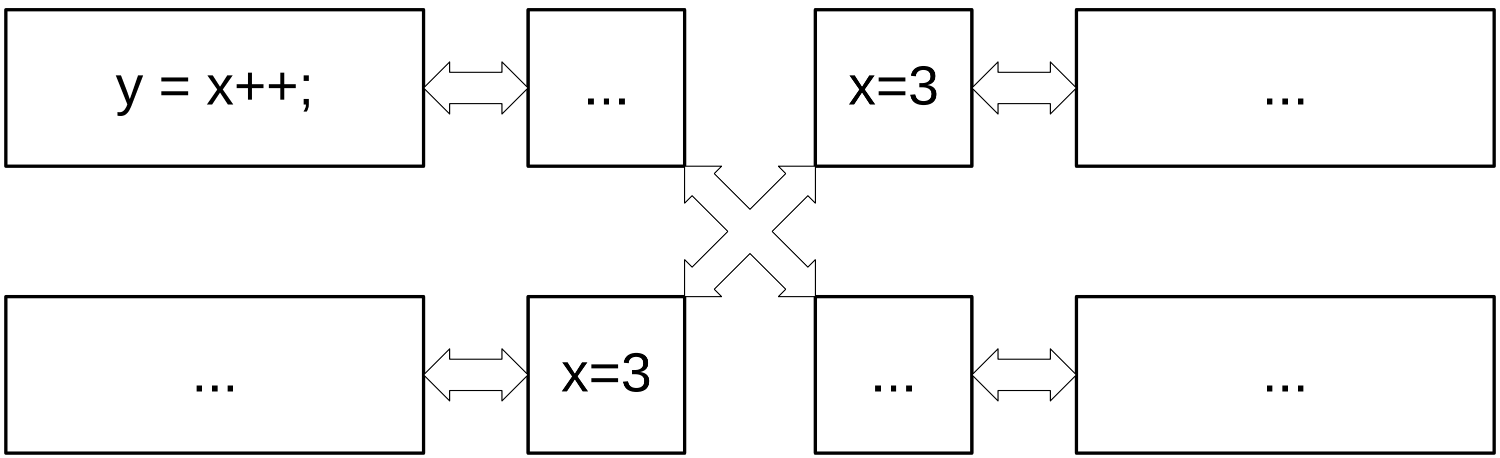
Atomics through locking

- Multiple processors are connected through caches
  - Identical data may be found in multiple caches



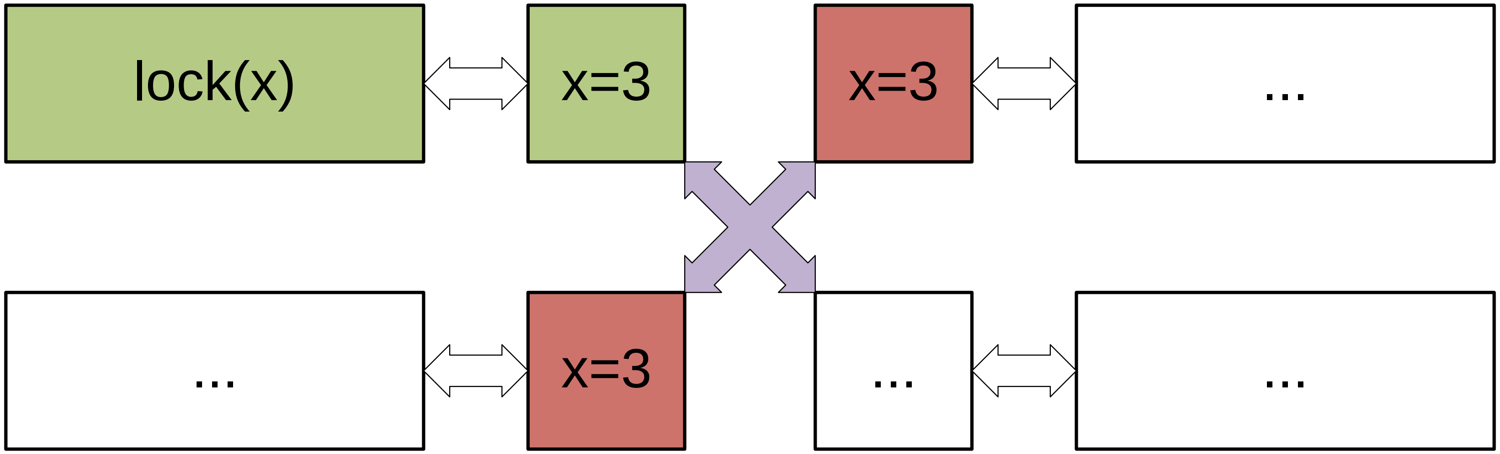
Atomics through locking

- Multiple processors are connected through caches
  - Identical data may be found in multiple caches
  - Atomic operations expand into a sequence of smaller operations



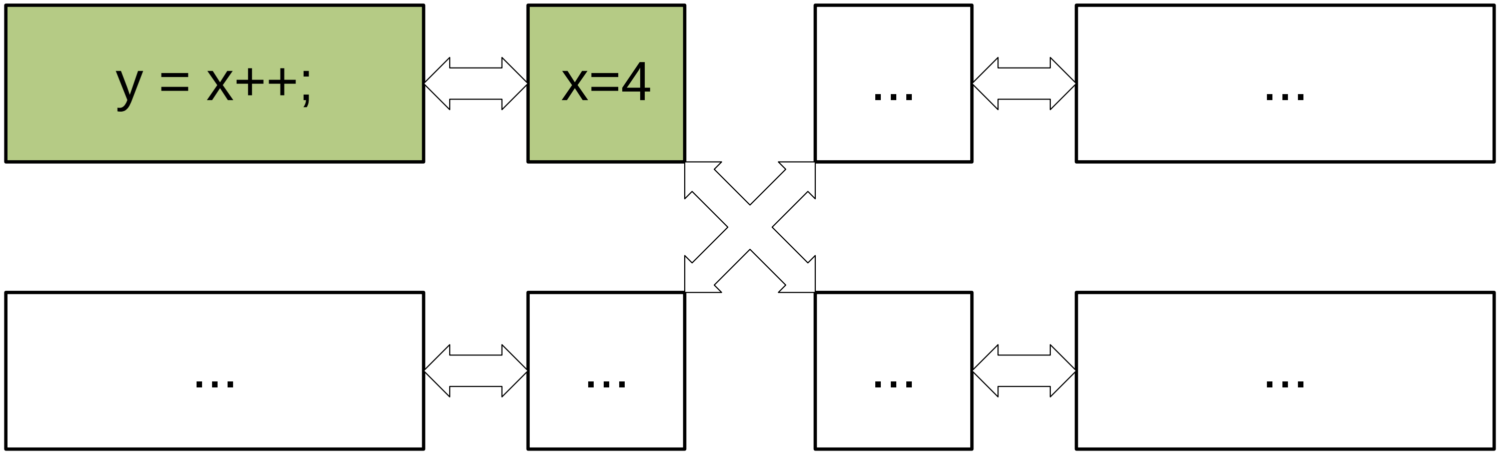
Atomics through locking

- Multiple processors are connected through caches
- Atomic operations expand to: *lock, update, release*
  - Locking ensures data is only present in *one* cache



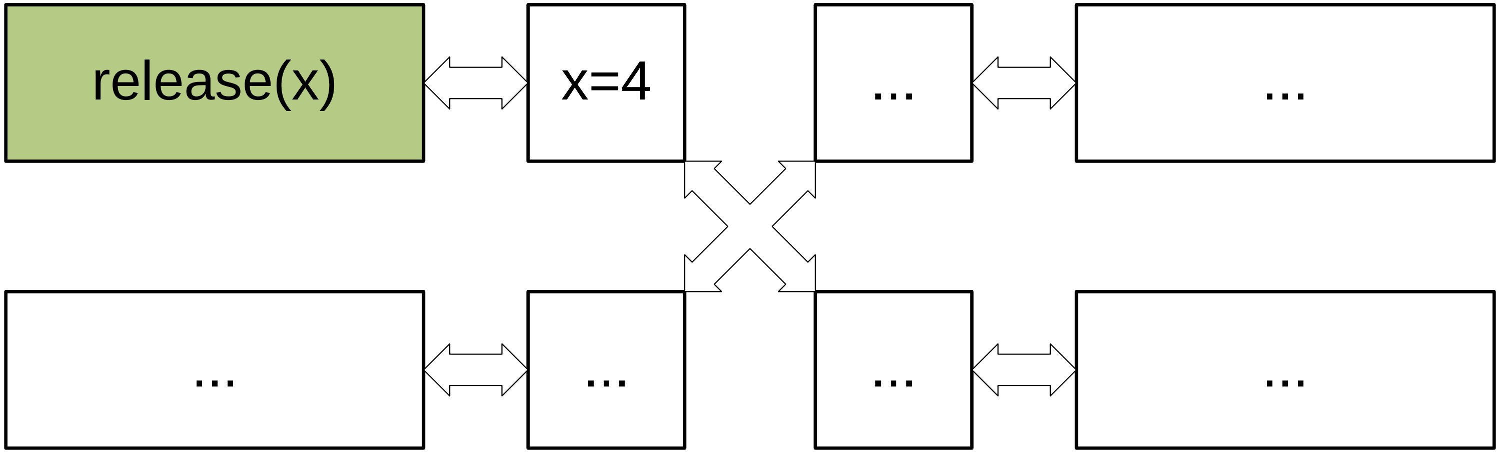
Atomics through locking

- Multiple processors are connected through caches
- Atomic operations expand to: *lock, update, release*
  - Locking ensures data is only present in *one* cache
  - While locked, data can be manipulated locally



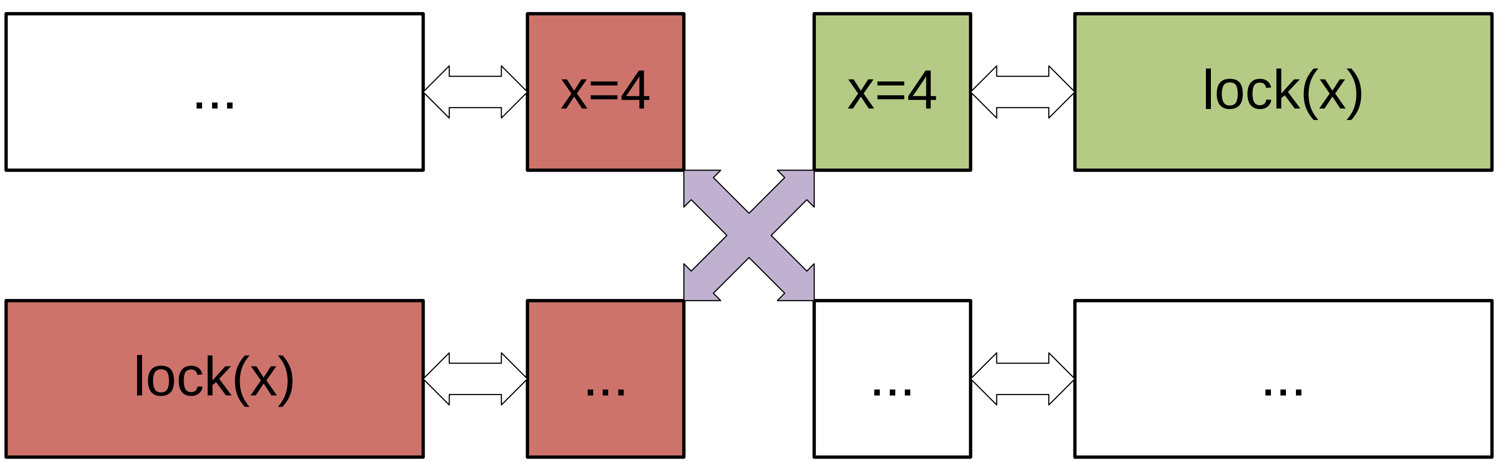
Atomics through locking

- Multiple processors are connected through caches
- Atomic operations expand to: *lock, update, release*
  - Locking ensures data is only present in *one* cache
  - While locked, data can be manipulated locally
  - Other CPUs can read the data once it is released



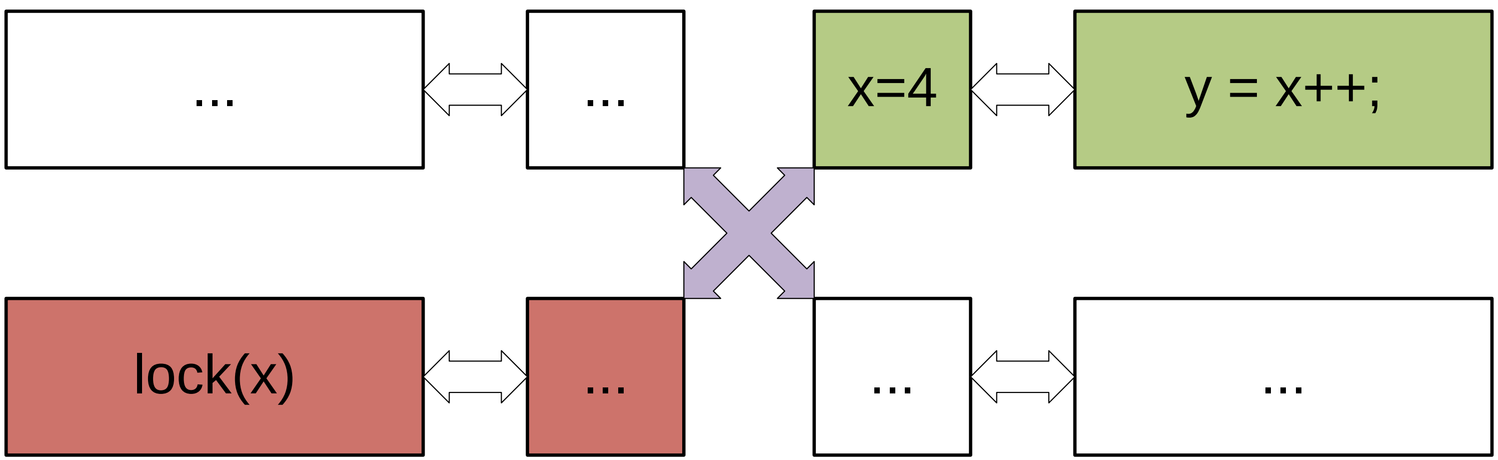
Atomics through locking

- Multiple processors are connected through caches
- Atomic operations expand to: *lock, update, release*
- Problem occurs if two CPUs want to modify same memory
  - Only one lock will succeed



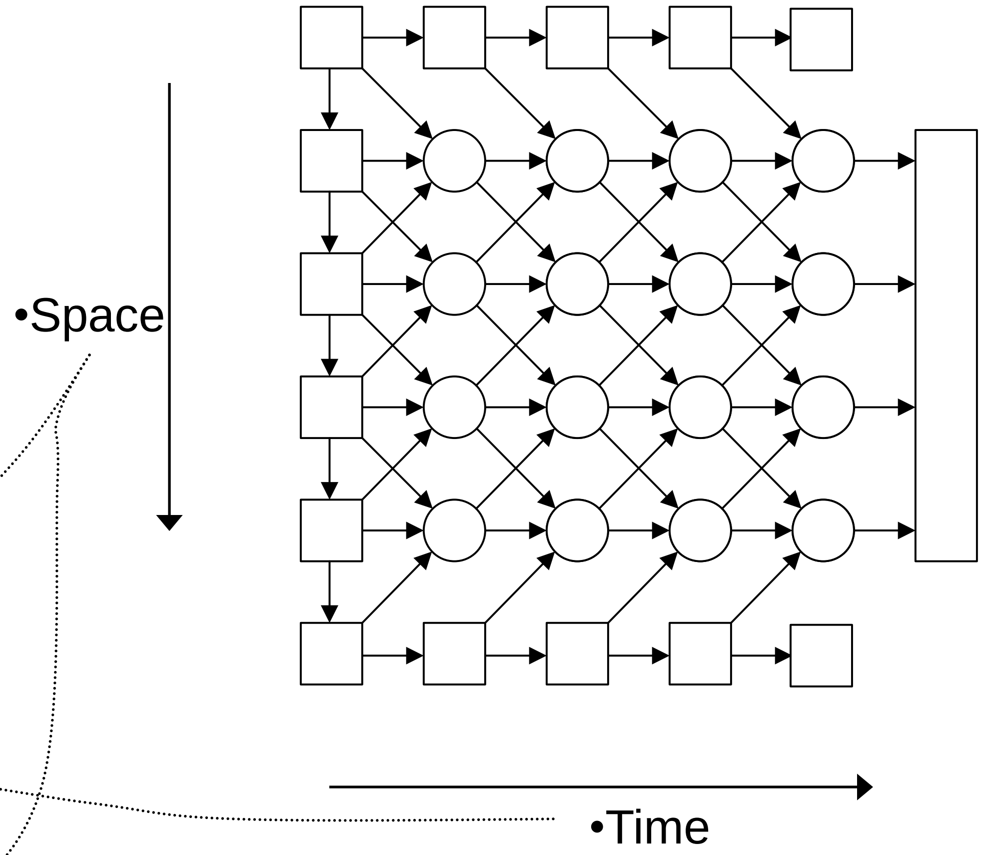
Atomics through locking

- Multiple processors are connected through caches
- Atomic operations expand to: *lock, update, release*
- Problem occurs if two CPUs want to modify same memory
  - Only one lock will succeed
  - Other CPU will block until it can acquire a lock



Potential Problems with Atomic Operations

- ***Lock contention*** : CPUs fight to lock the same location
  - Assume CPU performs atomic with probability p per cycle
  - Given *n* processors, probably of conflict per cycle is ~$1-(1-p)^n$
  - But *eventually* progress will be made
- ***Cache thrashing***: Locking a variable evicts entire cache line
  - Memory traffic increases even if conflicts don't occur
  - Still need to move data from cache to cache
- General guidelines for use:
  - atomic ops should be a low percentage of total instructions
  - try to ensure that each atomic only lives in one cache




```
•g[1,1]=F1(0)
for s=2..n
  g[s,1]=F1(g[s-1,1])
end

for t=1..n-1
  g[1,t+1]=g[1,t]
  for s=2..n-1
    g[s,t+1]=F2(g[s-1,t],g[s,t],g[s+1,t])
  end
  g[n,t+1]=g[n,t]
end

return F3(g[1..n,t])
```

Communication : Checklist

1. Do tasks do similar amounts of communication?
2. Do tasks communicate with “local” tasks?
3. Can communication occur concurrently?
4. Is there “hidden” hardware communication?

Agglomeration

This space intentionally left blank

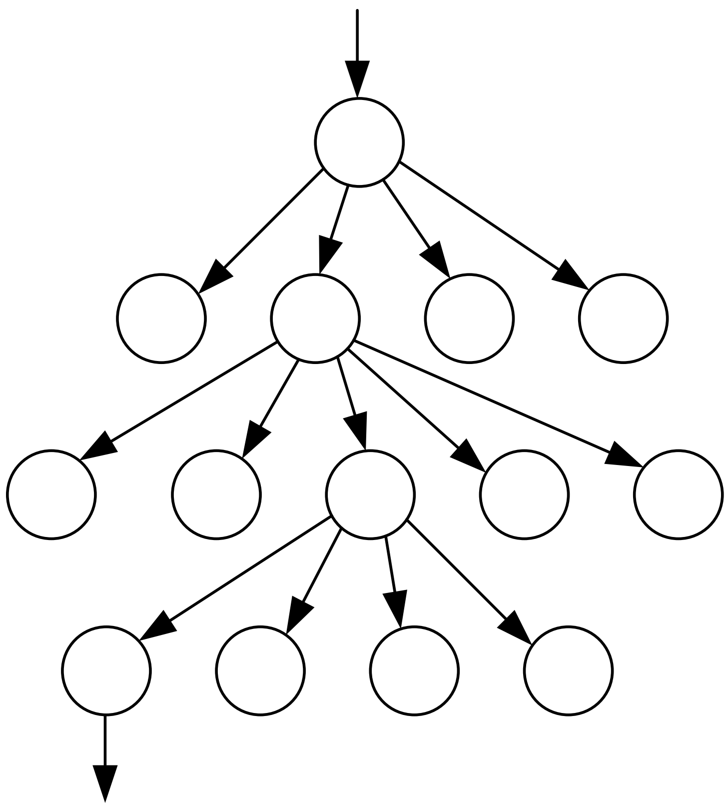
Agglomeration

1. Is agglomeration tailored towards locality of data?
2. Does task count still scale with problem size?
3. Have you left sufficient concurrency for future machines?
4. Can you reduce the grain-size any further?

Mapping

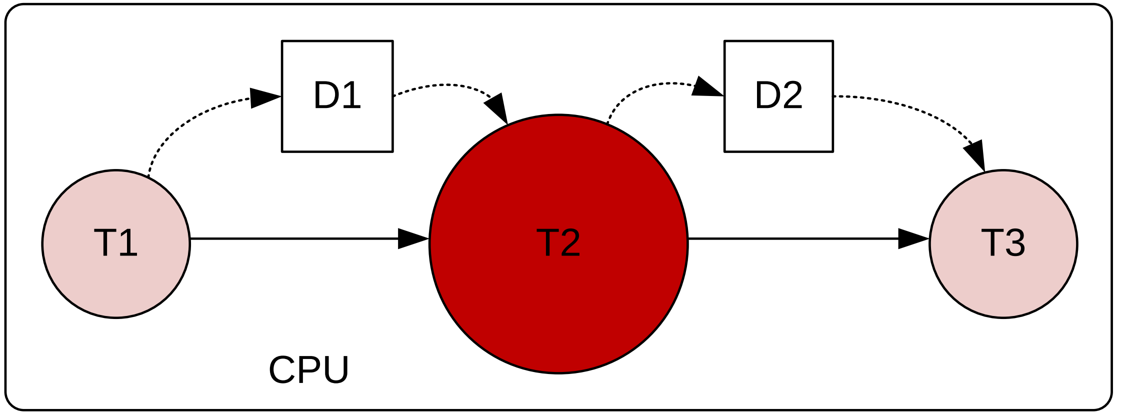
- We have parallelism at many levels
  - Low-level: SIMD, VLIW, super-scalar
  - Homogenous systems: multi-core CPU, multi-core GPU
  - Heterogeneous systems: CPU + GPU (+ *cloud* + *FPGA*)
- Need to map agglomerated tasks to different devices
  - Put independent tasks on different devices: ***increase parallelism***
  - Put communicating tasks on same device: ***increase locality***
  - Conflicting requirements, which should win?
- How is the mapping managed: ***static vs dynamic***
  - Dynamic: e.g. task-based work-stealing
  - Static: "*I'll put this part in a GPU kernel, and this part is host code*"
- Is there significant overhead due to the mapping?

Hidden communication costs

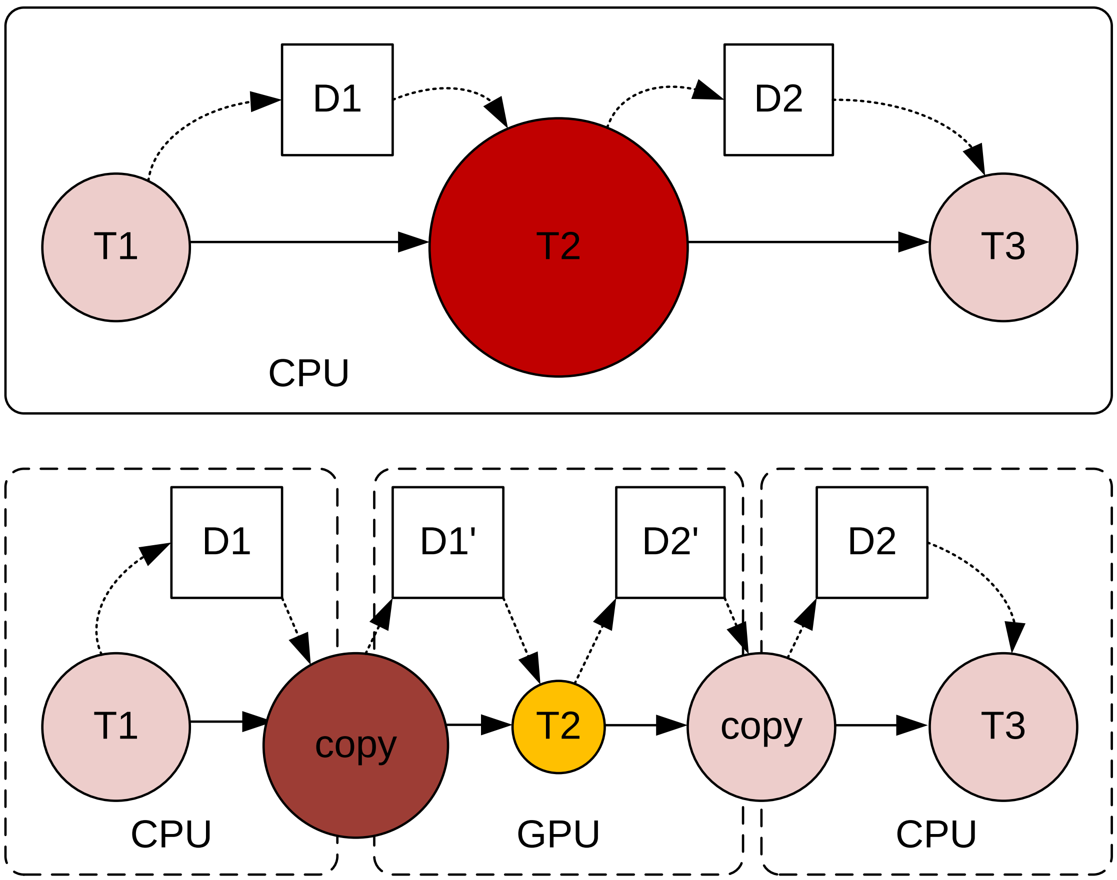


```
void MyFunc(node)
{
  if(accept(node)){
    foreach(c in children(node)){
      spawn MyFunc(c);
    }
  }
}
```

Beware the cost of IO



Beware the cost of IO



Mapping

1. Will the task-scheduler become a bottleneck?
2. Do you *still* have enough tasks (~10 per processor)?
3. Should you be using dynamic (or static) mapping?
4. How does the mapping change communication costs?

When to use PCAM

- PCAM is not always the right choice
  - Sometimes you need to be more flexible
  - Each application is different
- ***But***: if you don't use it, you need a reason why
  - Are you considering mapping before parallelism?
    - Might end up with a GPU kernel with only 4 threads
  - Are you ignoring communications and locality?
    - Can end up with cache contention between CPUs