NTNU – Trondheim
Norwegian University of
Science and Technology

Loose ends

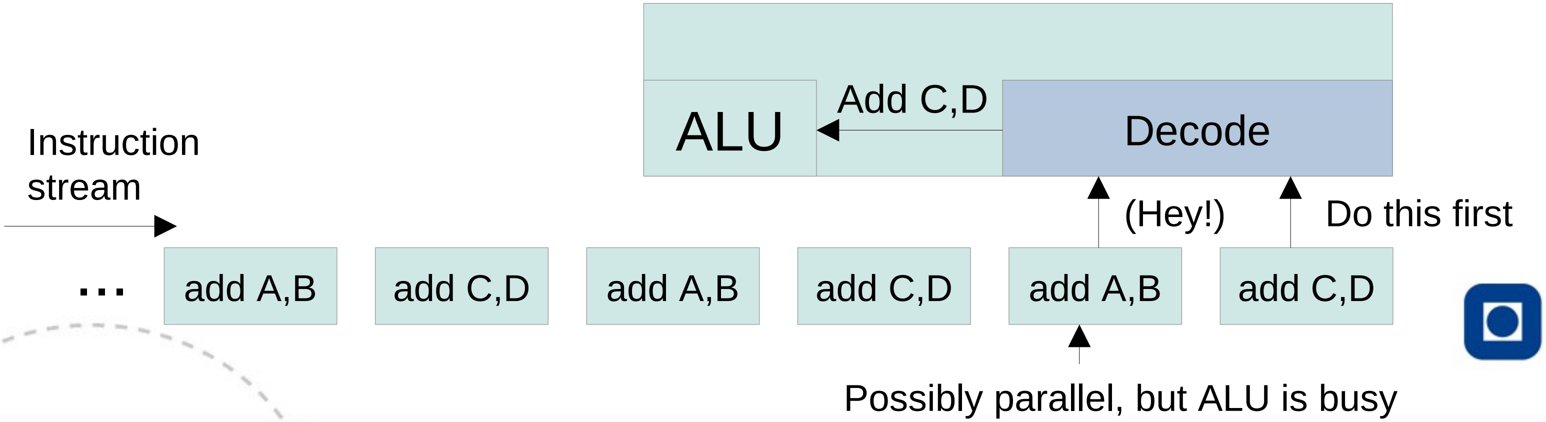
Jan.Christian.Meyer@ntnu.no

I wanted to say a few things...

- ...but the opportunity never arose.
- I’m saying them today, we’ll talk briefly about
  - Simultaneous MultiThreading (SMT)
  - Superlinear speedup
  - Load balancing
  - Hybrid programming

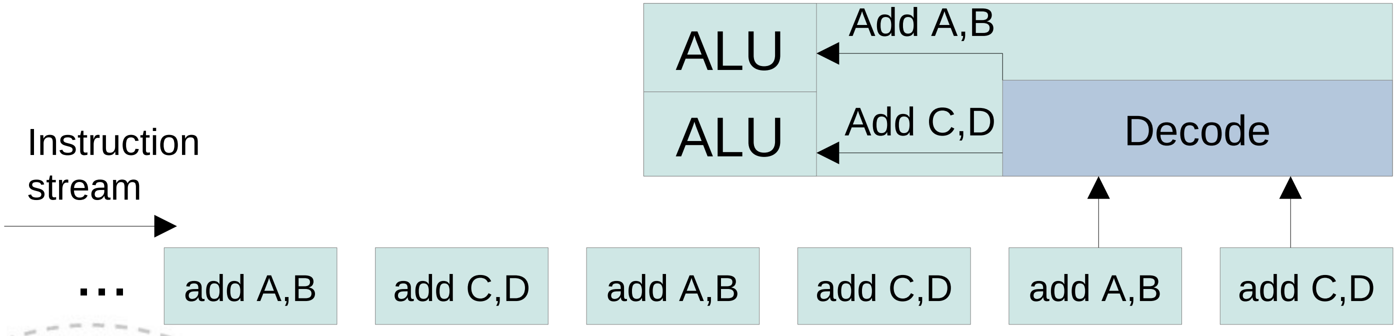
Decoding multiple instructions

- We started out with von Neumann machines, and modern modifications to them
  - Back in lecture #4, we were talking about automatic exploitation of *instruction-level parallelism*
  - Specifically, with multiple instructions on their way through a pipeline, we can detect whether they are independent (or not)
  - When they are, they can (in principle) be run simultaneously



Superscalar processors

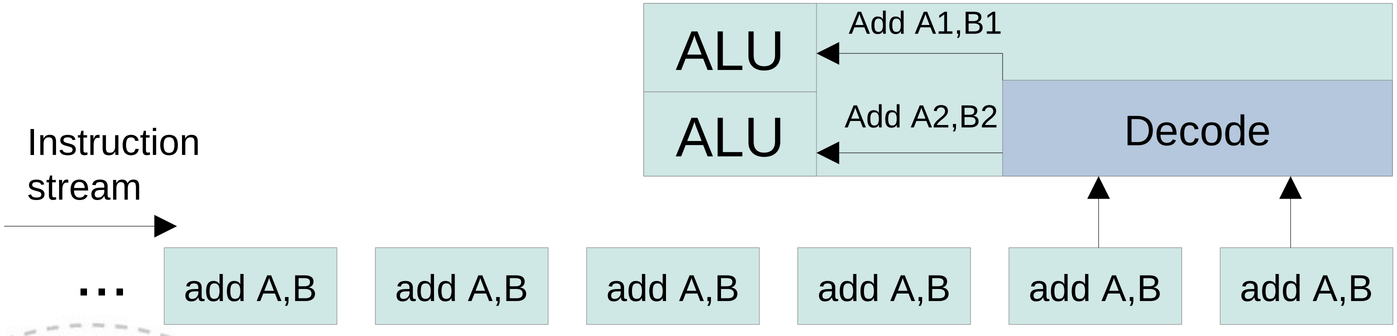
- If we replicate the unit that adds numbers, we can extend the decoder logic to dispatch several (independent) instructions simultaneously
- We called it *multiple issue*



There's even more:

Register renaming

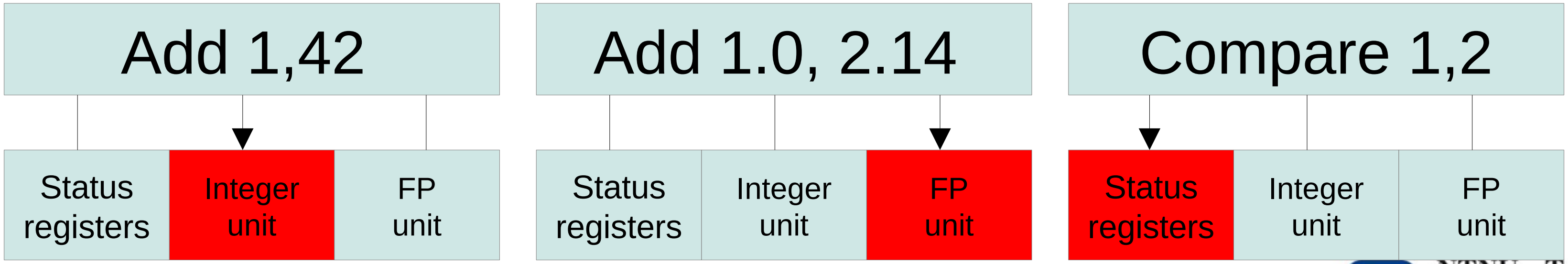
- With a window of several instructions, we can also detect whether use of the same registers is a *true* data dependence, or if it's "just" a *name* dependence
    - When it's a name dependence, it could be resolved by a machine with more registers
    - Many superscalar designs feature duplicated registers, but only expose one set in the instruction set / assembly language
    - The remaining *renaming registers* are used for multiple issue



NTNU – Trondheim
Norwegian University of Science and Technology

Inside the ALU

- Different instructions trigger different components to do different things
  - Adding (e.g.) a pair of memory addresses requires one part of the unit
  - Adding (e.g.) some numbers with decimals requires another, because different bits of the representation have to be flipped
  - Comparisons, jump instructions, *etc.* use yet another part, with separate registers



In a sequential run, there is some unused capacity here

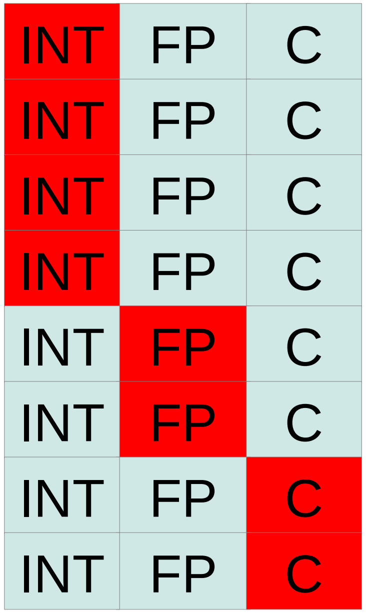
The under-utilization issue

- Only one part of the ALU is active at a time
  - Can we fill it up with simultaneous instructions?
- In principle: Yes!
  - Just map multiple-issue instructions that use different parts of it to the same ALU
- In practice: Not Really
  - Sequences of instructions that contain a balanced mix of integer, FP and control operations don’t appear often in programs
  - How often do you write programs where every 3rd statement does something entirely unrelated to the previous 2?
  - We actively discourage people from interleaving unrelated code in their programs, it’s terrible to read and understand

Threads to the rescue!

- Two independent control flows can easily contain entirely unrelated instructions at the same time:

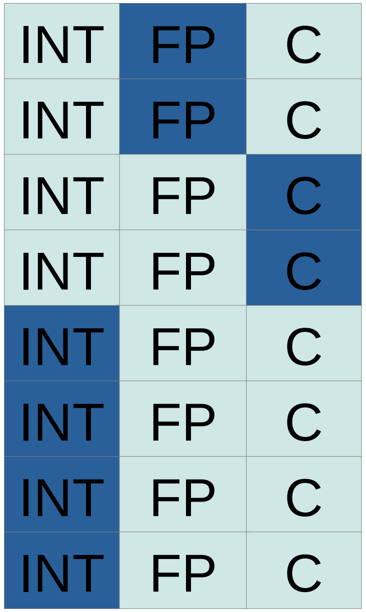
Required unit



INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C

Thread 1

Required unit



INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C

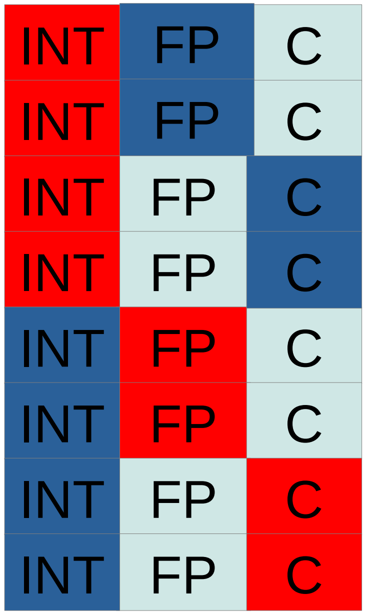
Thread 2

(For example: If this instruction mix is in a loop, two copies can be at different stages)

When the stars align

- When control flows with complementary requirements line up in time, they can be served by the same hardware:

Required unit



INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C
INT	FP	C

Thread 1 & Thread 2

Simultaneous MultiThreading (SMT)

- In an otherwise superscalar processor, it is a (relatively) minor extension to support this fortunate coincidence
  - Replicate instruction pointer / decoding unit
  - Pretend to be 2 processors, and receive 2 instruction streams
  - Merge them together when their needs don't conflict
- If your CPU says it has 4 cores but supports 8 threads, this is what it's doing

SMT exploits happy coincidences

- The actually simultaneous part only happens when the instruction streams interleave without conflict
- When threads 1 and 2 both need the integer unit simultaneously, one of them has to wait

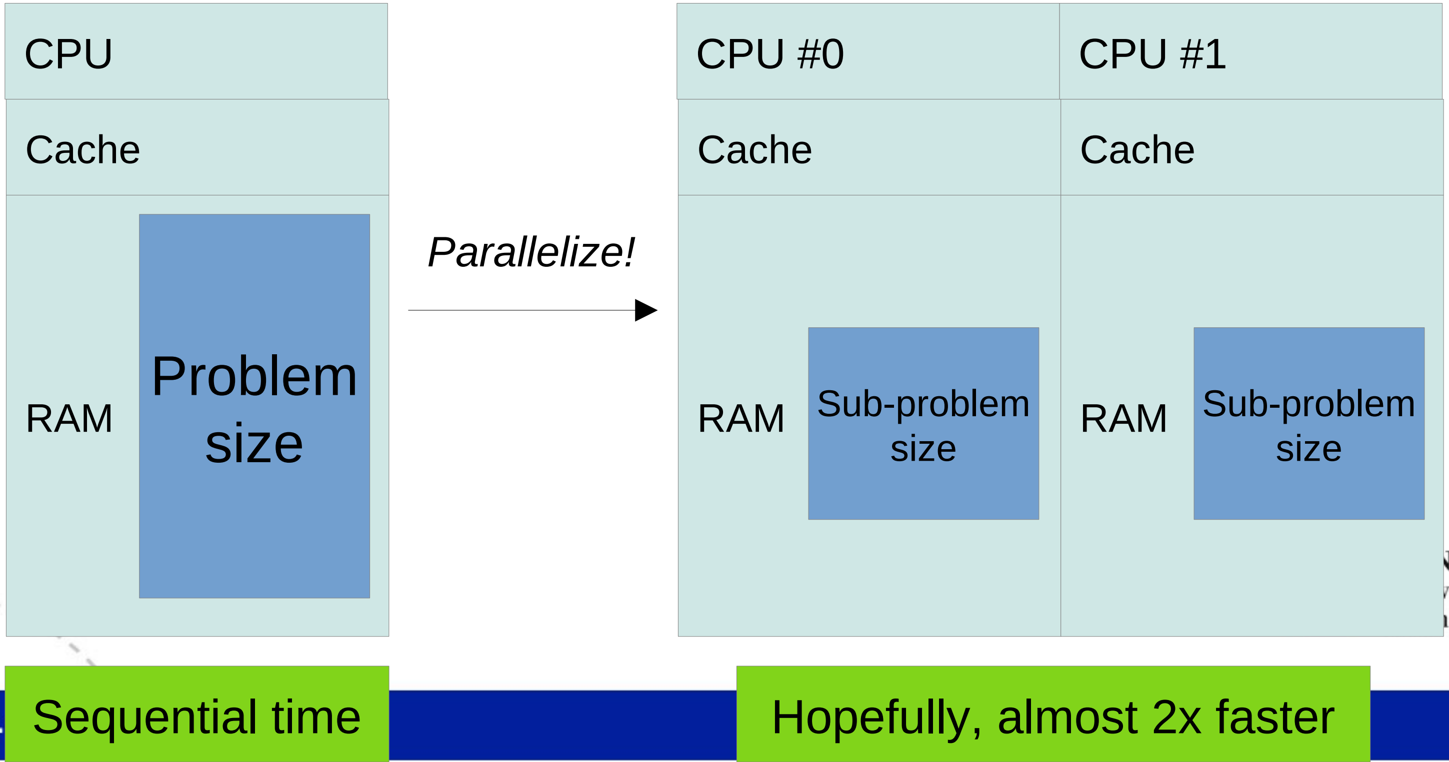
    (and we’re back to sequential interleaving)
- Statistically speaking, independent threads coincide every so often and make utilization a little better
    - If you have two threads that *e.g.* both constantly need the integer unit, however, they won’t speed up when scheduled on the same physical core
- It is very difficult to *plan* for your program to utilize this type of parallelism

Superlinear Speedup

- Amdahl’s law tells us that
  - The run time of a program has a fraction f that can’t be parallelized
  - Even if f could be 0, the speedup would only be S(p) = p at best
- The assumption is that we have a fixed-size problem, and increase p
  - In other words, the scalability-experiment we’re talking about here is carried out in the *strong scaling mode*
  - That’s when Amdahl’s law applies
- Sometimes, we can still measure S(p) > p
  - What is going on?

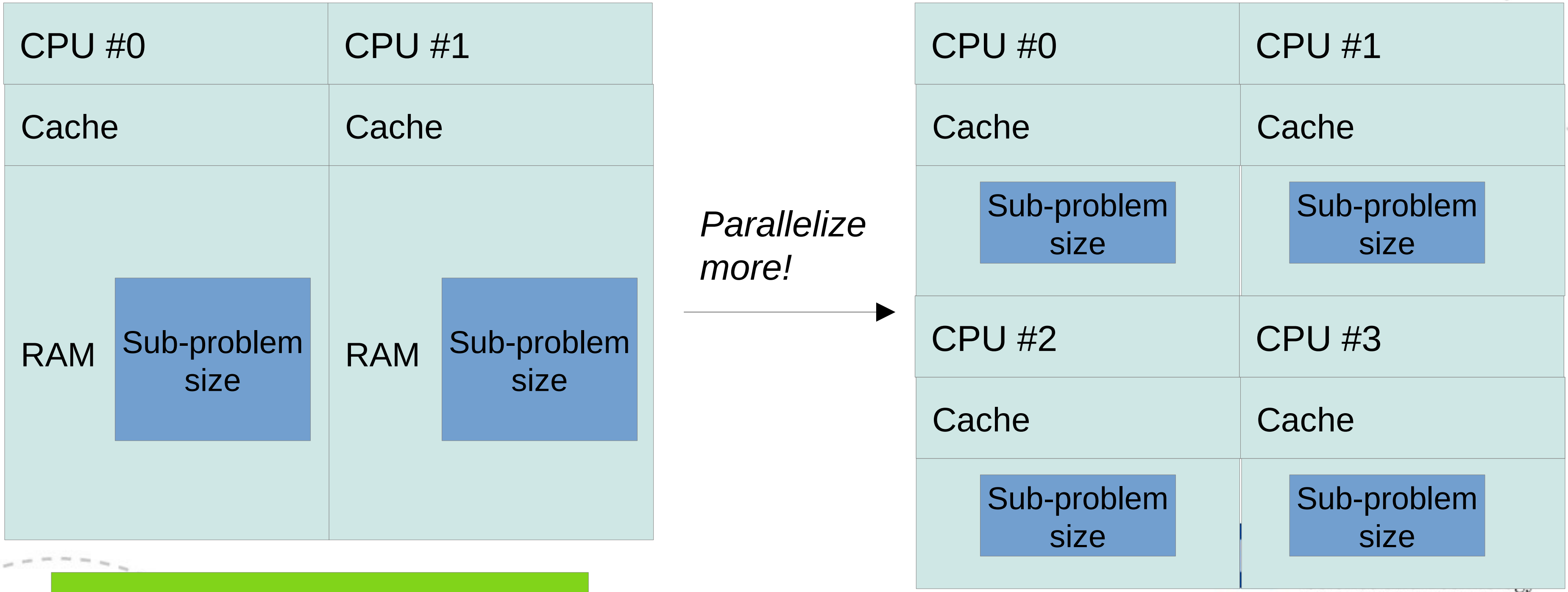
Split the problem

- Since we know about the memory hierarchy, we can illustrate a 2-way splitting of a constant problem size like this:



Split the problem again

- Remember, we’re not changing the global problem size:



CPU #0
CPU #1
Cache
Cache
RAM
Sub-problem size
RAM
Sub-problem size

Parallelize more!

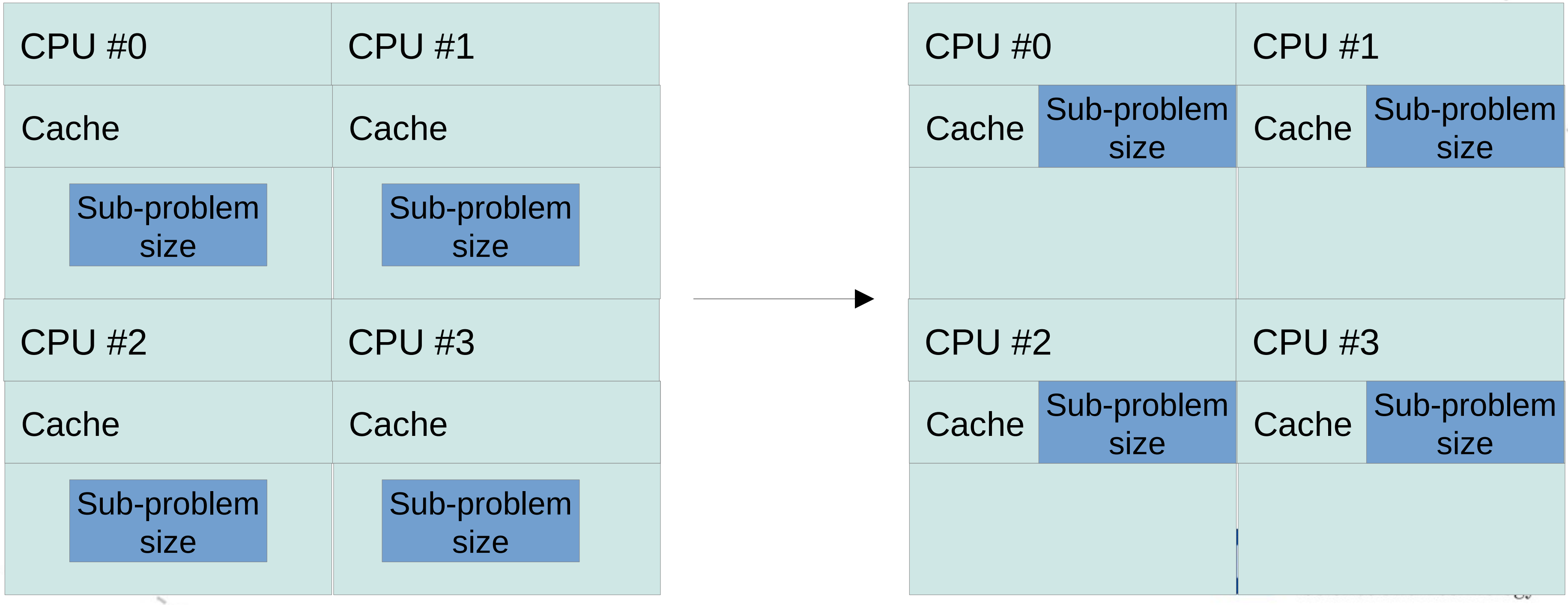
CPU #0
CPU #1
Cache
Cache
Sub-problem size
Sub-problem size
CPU #2
CPU #3
Cache
Cache
Sub-problem size
Sub-problem size

Hopefully, almost 2x faster than sequential

...almost 4x faster…?

When the magic happens

- At some point, we have split the problem into small enough parts that each fits in a faster class of memory
- This will give you speedup figures of S(p) > p



CPU #0
Cache
Sub-problem size

CPU #1
Cache
Sub-problem size

CPU #2
Cache
Sub-problem size

CPU #3
Cache
Sub-problem size

CPU #0
Cache
Sub-problem size

CPU #1
Cache
Sub-problem size

CPU #2
Cache
Sub-problem size

CPU #3
Cache
Sub-problem size

...almost 4x faster…?

MORE than 4x faster!

Load Balancing

- We’ve seen how parallel computations often lead to periodic synchronization points
- It works best when every participant has exactly the same amount of work
  - That way, nobody has to wait for long at a barrier
- It gets worse when the work is unevenly distributed
  - The collective can’t go faster than its slowest participant
  - When 1 process is late, P-1 processes are wasting time
- In a way, a little imbalance is unavoidable
  - Some process will always be the last to reach a synch. point, but we try to make it *almost* simultaneous

Load balancing in 3 flavors

Roughly speaking, there are 3 kinds of strategies to mitigate an unbalanced workload:

- Static
  - Embed the partitioning of the problem directly into the source code
    (this is what we've done in the problem sets, I won't illustrate it now)
- Semi-static
  - Examine the workload when the program starts, divide it then, and run with the initial partitioning until finished
- Dynamic
  - Adapt to the workload by shifting work around between participants while the program is running

Semi-static technique:

Recursive orthogonal bisection

- Suppose we have domains with irregular shapes
  - These images are extracted from map data
  - There is fluid motion to compute in the water (bright sections)
  - There is nothing to do on land (dark sections)

Trondheimsfjord[1]

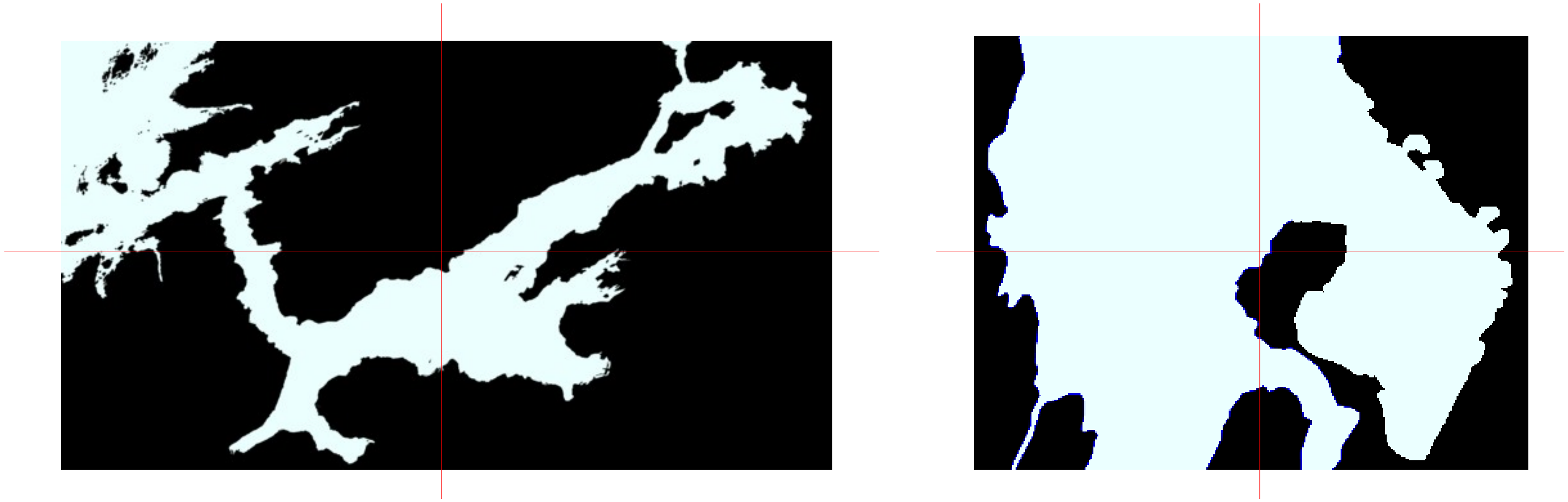
Mehamn[2]

[1] *"Performance Modeling of Finite Difference Shallow Water Equation Solvers with Variable Domain Geometry"*, Richard Bachmann, NTNUOpen 2021
[2] *"Performance Modeling of a Finite Volume Method for the Shallow Water Equations"*, Jenny Veronika Ip Manne, NTNUOpen 2022

NTNU – Trondheim
Norwegian University of Science and Technology

Recursive orthogonal bisection

- With a static Cartesian split, we get uneven workloads

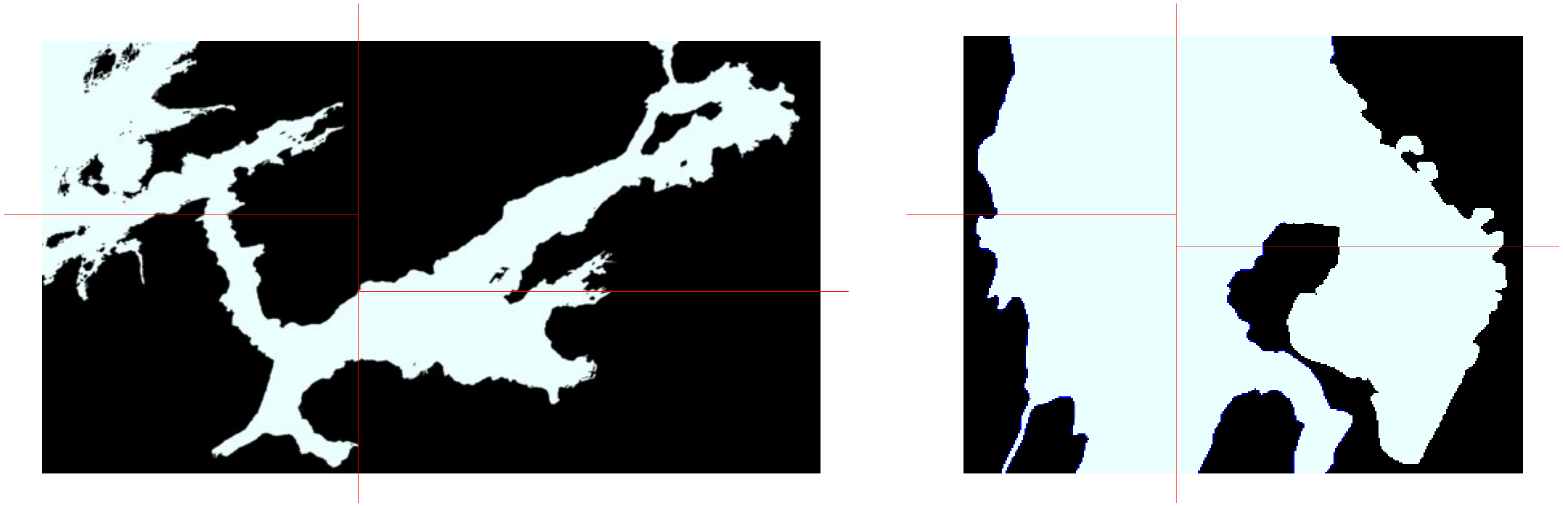


Recursive orthogonal bisection

- Recursive orthogonal bisection starts by scanning along one axis, and finding the 50% mark of cells that have actual work in them

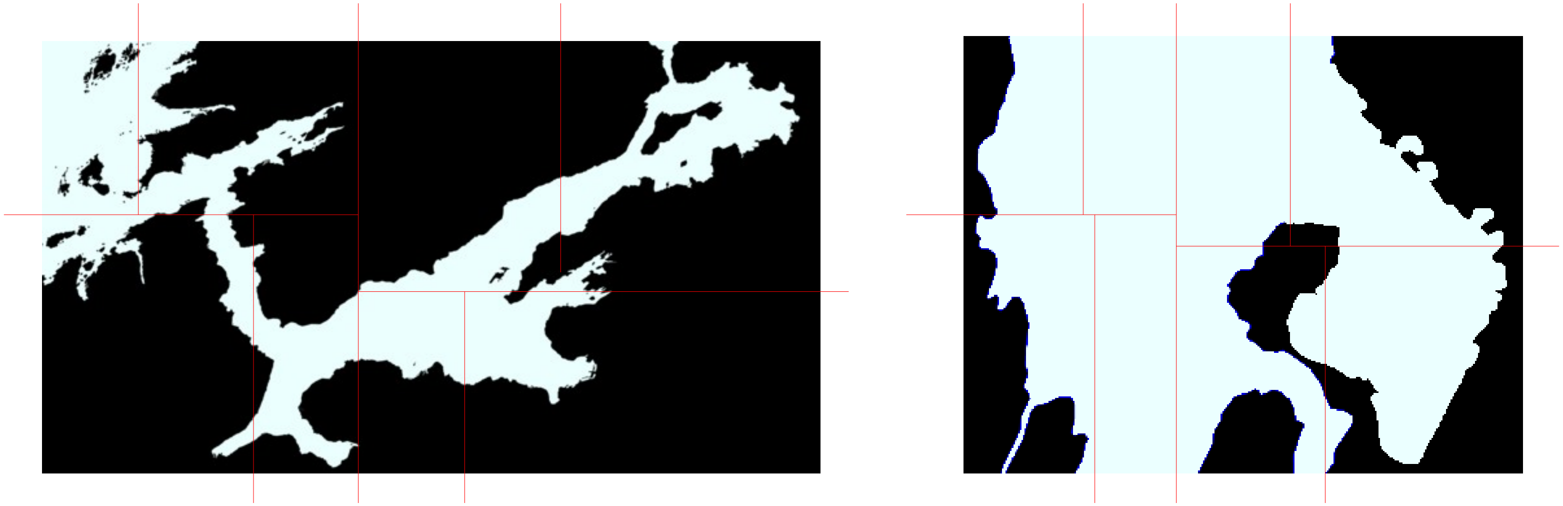
Recursive orthogonal bisection

- Next, it changes directions and finds 50% marks in the two parts from the previous step



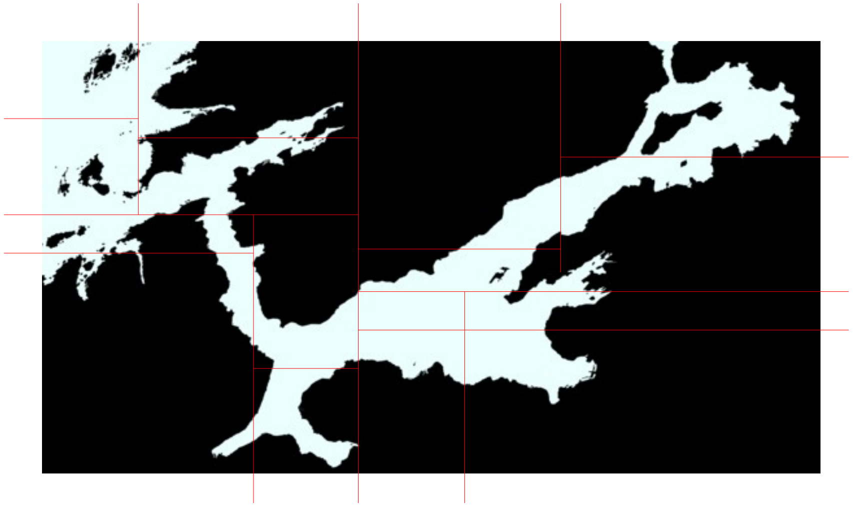
Recursive orthogonal bisection

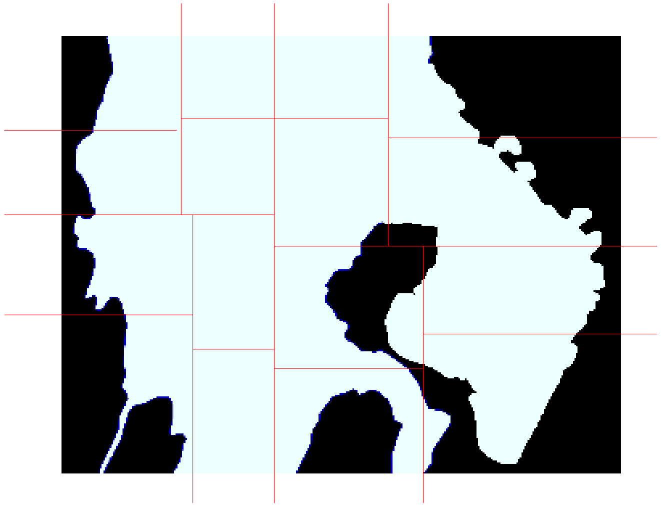
- The procedure repeats until we have enough parts to parallelize for the machine we want to use



Recursive orthogonal bisection

- The sub-domains get trickier to do border exchanges with, but they end up containing about the same amount of work





(Disclaimer: both of the referenced theses solve their load balancing problems using other techniques, but recursive orthogonal bisection is a good place to start)

Dynamic technique:

Master/worker pattern

- We touched upon this with the OpenMP schedules
- Nominate one rank/thread/whatever to be the *master*
  - This one maintains a queue of similar-sized tasks
- The rest of the ranks/threads/whatevers are *workers*
  - The master assigns them tasks

    or
  - They take tasks from the queue, and inform the master
- Pro: simple to understand
  - This is a very popular design in transaction-serving systems
- Con: centralized control = limited scalability
  - You can *always* imagine a number of workers that is large enough to overwhelm the master with requests for work

Dynamic technique:

Work stealing

- This approach is similar to the master/worker solution
  - Each participant maintains a queue of tasks that it has been assigned
  - Tasks can be assigned-to or taken-by unemployed fellow participants
- The difference is that it’s distributed
  - Each participant is both a “master” and a worker to its immediate neighbors
  - Unemployed participants receive/take a task from a neighbor
- Pro: scales to any number of participants
- Con: if there’s an overload of work at one end of the system and a shortage at the other, it takes a while (and many requests) before it evens out

Hybrid programming

- As you may have noticed, the four programming models we cover in this class can be combined
  - MPI enables communication between multi-core SMP systems
  - Within each SMP system, we have several cores
    - They can run Pthreads
    - They can run OpenMP threads
  - Within each SMP system, we may also have one or more GPUs
    - They can run CUDA kernels
- 15 years ago, studies of how to best combine separate programming models called it “hybrid programming”
  - Nobody calls it anything special anymore, because everyone is doing it now
- The only reason we’ve worked with each model separately is because it is easier for me to talk about one thing at a time

Tradeoffs in hybrid programming

Threads vs. processes

- With, say, 4 nodes that have 2 CPU sockets with 8 cores on each, you can
  - Run 64 MPI ranks
  - Run 4 MPI ranks (1 per node), and 16 threads in each rank
  - Run 8 MPI ranks (1 per cpu socket in each node), and 8 threads in each rank
  - Run 32 ranks with 2 threads in each…
- What is the best combination?
  - It depends on how your program uses memory
  - Try it out and measure the effect

Tradeoffs in hybrid programming

Threads vs. processes

- When you have threads in an MPI rank, you can
  - Make 1 thread responsible for communication, and have it do all the MPI calls
  - Let all the threads make MPI calls whenever they want

    (NB – Send and Recv are guaranteed to be thread-safe, but many of the more complicated MPI calls aren't, tread carefully)
  - Let all the threads use MPI, but enforce mutual exclusion with locks
- What is the best combination?
  - There is a very strong argument that only allowing one master thread to handle MPI is optimal*
  - You can create exceptions, but it's a good rule of thumb

* *"Comparison of Parallel Programming Models on Clusters of SMP Nodes"*, R. Rabenseifner and G. Wellein, Proceedings of the International Conference on High Performance Scientific Computing, 2003

Tradeoffs in hybrid programming

Processes and GPUs

- When you have multiple GPUs in one system, there is a similar tradeoff
  - You can create 1 process that controls all GPUs
  - You can create 1 process per GPU, and get the processes to talk via MPI
  - With the right kind of GPUs, you can get them to talk without involving the hosting processes
  - With the right kind of MPI, you can send and recv messages directly in device memory, without moving it via the hosting process
- What's best?
  - See the similar entry on balancing threads with processes