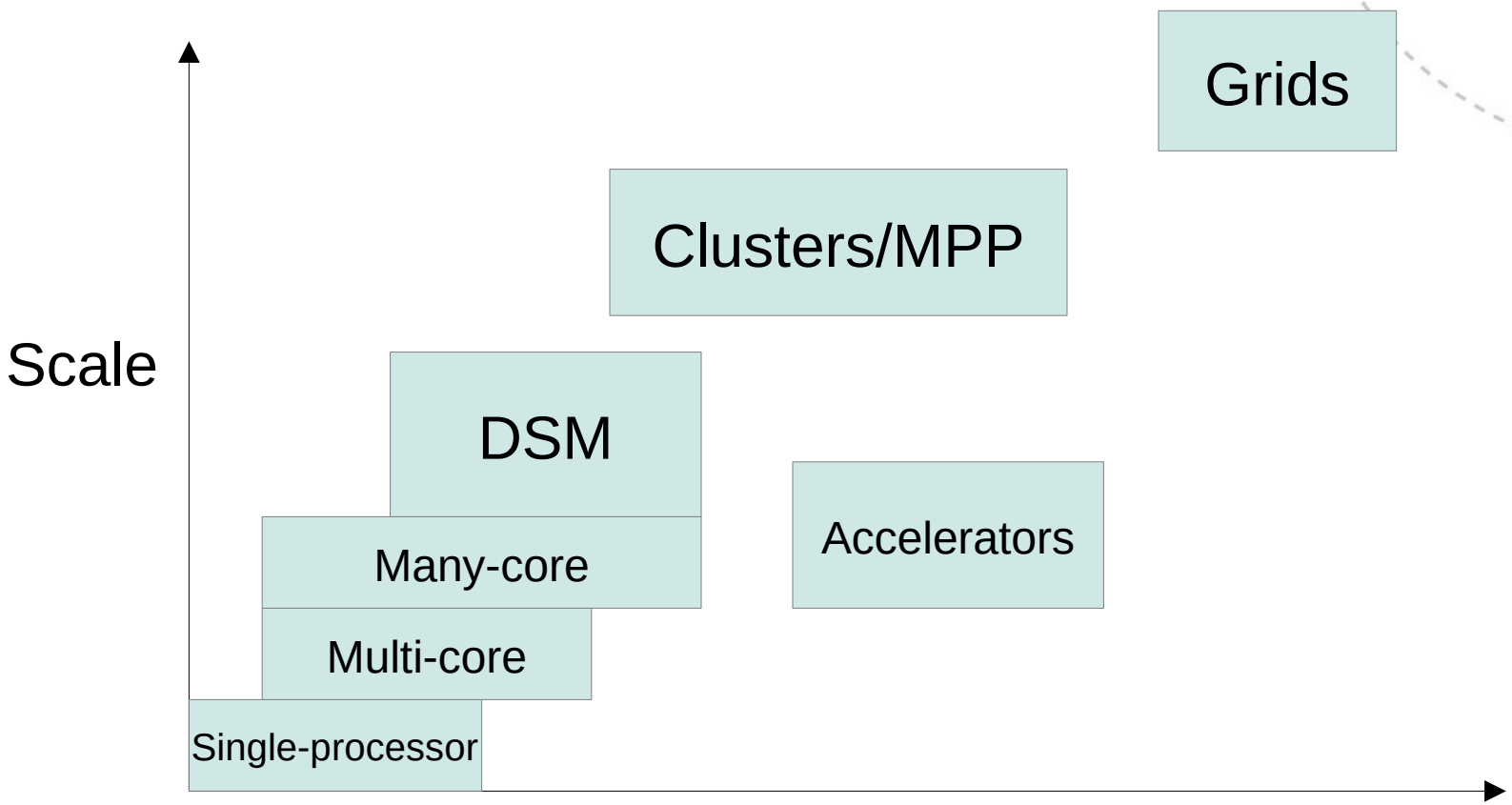
NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4200 Grand summary, pt.1

Jan.Christian.Meyer@ntnu.no

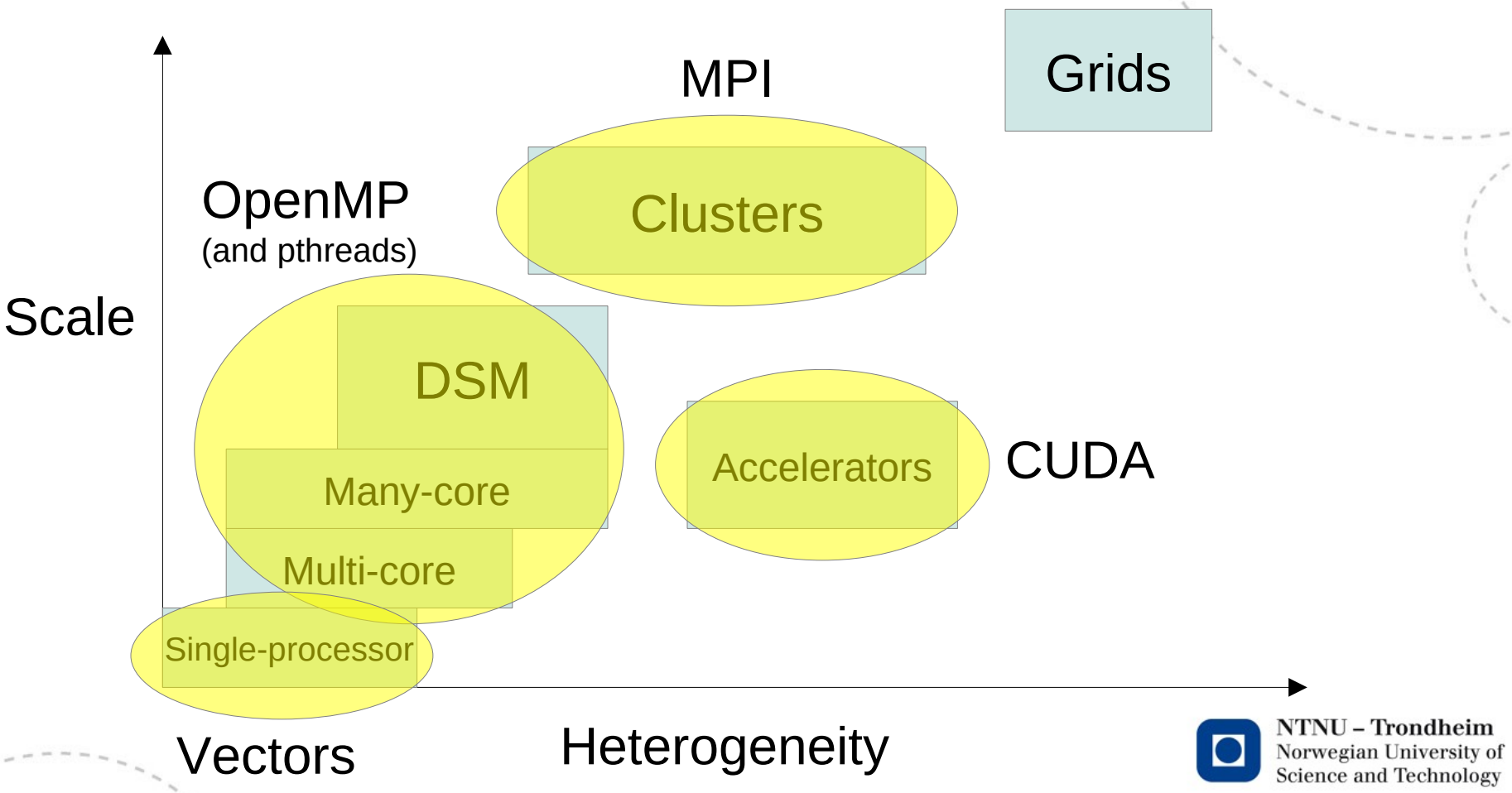
Approximate map of parallel systems



The system classes

- Single processor
  - Not *very* parallel, but has vector registers
- Multi-core
  - 2-100 superscalar cpu cores
- Many-core
  - 100+ simplified, but otherwise regular cpu cores
- DSM
  - 1000s of superscalar cpu cores, distributed memory concealed by directory-based cache coherence built into the interconnection network
- Accelerators
  - Graphics processors, FPGAs, signal processing units (and other application-specific circuitry)
- Clusters/MPP
  - Local networks with nodes taken from one of the previous classes, explicit communication
- Grids
  - International networks of connected clusters

Matching parallel programming models



These are not the only choices...

- ...but they’re the most popular at this time.
- We try to cover as much of the spectrum with as few tools as possible
- Hopefully, it’s given you a good starting point

Sequential computers

- We started out with the von Neumann computer
  - It has a CPU and some memory
  - The *control path* in the CPU fetches and decodes instructions
  - The *data path* moves data between memory and registers, and carries out operations on them
- Programs and data are all in the same memory
  - We distinguish between operations and operands by where we store them
  - We covered the structure of a *process image*, and indicated that the important parts are the
    - Text segment
    - Data segment
    - Stack
    - Heap

Improvements on the von Neumann model

- Since the von Neumann computer is only a model, actual hardware can support it without working exactly as it specifies
- Recognizing that its main bottleneck is that programs become long sequences of *read-modify-write* cycles, we can improve performance by second-guessing what is about to happen before it does
- We talked about
  - Cache memory
  - Instruction level parallelism

Cache memory

- Anticipating that programs will exhibit
  - *Spatial locality* (nearby values will be needed soon)
  - *Temporal locality* (same values will be re-used soon)

  we can speed up programs using small, low-latency memory buffers which
  - Fetch neighboring values along with single addresses when they are accessed
  - Keep them in the buffer as long as they are being re-used, unless the buffer overflows
- We briefly looked at *loop tiling* as a technique that can improve cache utilization

Cache coherence

- With multiple cores, caches must maintain a coherent view of memory when it is updated
- We looked at
  - *Snooping* (detecting updates from a shared part of the interconnect)
  - *Directory* (detecting updates by marking memory banks when they are updated)

Instruction level parallelism

- Instruction streams can be sped up in many ways, we looked at
  - *Pipelining*
    - starting the next op. before the previous finishes
  - *Out-of-order execution*
    - dispatching independent ops. simultaneously
  - *Prefetching & branch prediction*
    - collecting statistics on where the next op. is likely to come from
  - *Vectorization*
    - using special ops. that do the same thing to several data elements simultaneously

Vector operations

- When compilers don't detect that vector operations can be used, we can write them by hand
- This necessitates using explicit CPU-specific operations
  - *Intrinsics* are slightly more abstract than raw assembly code, but slightly less abstract than plain C
  - We looked at SSE2 instructions for x86-compatible CPUs
  - I mentioned Neon instructions for ARM-based CPUs
  - Both are a little old, but our example only needed length-2 vectors
  - Newer versions with longer vectors are available

Flynn's taxonomy

- This is a theoretical classification of parallel architectures:
  - SISD (Single Instruction, Single Data – sequential computers)
  - SIMD (Single Instruction, Multiple Data – vector computers)
  - MISD (Multiple Instruction, Single Data – not in practical use)
  - MIMD (Multiple Instruction, Multiple Data – threads & processes)
- Not a universal classification, but useful to know about
- We also mentioned two non-Flynn categories
  - SPMD (Single Program, Multiple Data – MPI/OpenMP style code)
  - SIMT (Single Instruction, Multiple Threads – CUDA style code)

Shared and distributed memory

- When starting multiple control flows, we can do it by adding
  - Processes
    - No shared memory, require explicit message passing
    - Work across networks of independent computers
  - Threads
    - Private stack memory, shared data and and heap
    - Implicit messaging, require protection against race conditions (locks, atomic operations)

Amdahl's and Gustafson's laws

- All programs have some inherently sequential fraction of their work, which we called f
- Speedup is the ratio of total sequential run time to total parallel run time
- With the same problem + more cores, we get Amdahl's law
  - Limit of speedup is 1/f
- With a problem that grows in proportion to the core count, we get Gustafson's law
  - Scaled speedup is f + p(1-f)
- We also mentioned *parallel efficiency*
  - Derived from speedup

MPI

- MPI parallelization works by making multiple processes
  - Since they don't share memory, they don't have to be on the same computer
  - Since they don't share memory, communication becomes very explicit, with function calls to transport data between processes
- In order to simplify common problems, lots of extra abstractions are available
  - We looked at point-to-point messaging, collective operations, derived data types, custom communicators, and parallel I/O

MPI: Point-to-point operations

- Each process has a *rank* within a *communicator*
- Linear arrays of data can be sent from one rank to another when the sender knows the recipient's rank
- They must be received with a matching call at the other end, where the receiver knows the sender's rank
- All MPI programs can be written in terms of
  - Init, Finalize (start and stop)
  - Comm_rank, Comm_size (rank and total number of ranks)
  - Send, Recv (pass messages from point to point)

MPI: Communication modes

- The semantics of sending and receiving differ by the *mode* of the sending operation
  - Standard
    - Default, usually buffers small messages and blocks until completion for large messages
  - Synchronized
    - Always blocks until completion for all messages
  - Ready
    - Doesn't buffer at all, but requires receive to be posted before send
  - Buffered
    - Allows programmer to specify the buffer space to use, instead of allocating new buffers for every message

MPI: Border exchanges

- We’ve looked at how physics simulations that split their work into smaller, local areas require communication between the parts
  - We saw it with the advection eq. in lecture
  - You’ve seen it with the heat eq. in homework
- Local areas must be padded with a small border of values taken from their neighbors
- Pairwise exchanges of values come with a potential for deadlock
  - Unified sendrecv or non-blocking send/recv calls mitigate this

MPI: Collective operations

- We looked at some operations that involve all active ranks simultaneously
  - Barrier
  - Broadcast
  - Scatter
  - Gather
  - Reduce
- The latter also have *non-rooted* versions
  - Allreduce
  - Allgather

MPI: Performance analysis

- We looked at the Hockney model of communication cost, consisting of
  - 1 *latency* per message sent (in seconds)
  - (message size) x (inverse bandwidth) additional seconds of data traffic
  - Gives estimate of communication cost when you count messages and sizes based on the program code
- Too simple for modern platforms
  - Complicated interconnects have more than one type of links inside, each with their own latencies and bandwidths
  - Hockney model must be adapted to the machine, but its basic observation still holds

MPI: Derived data types

- Because sending and receiving requires linear arrays, indexing complicated patterns becomes tricky
- Derived data types give a notation for indexing and offsets which lets MPI handle it for us
- We looked at how to construct derived data types as
  - Contiguous
  - Structured (variable-length lists of elements)
  - Vectors (regularly spaced lists of elements)
  - Subarrays (regularly spaced lists in multiple dimensions)

MPI: Communicators

- Additional communicators can be derived from MPI_COMM_WORLD
  - We can split it into sub-groups, by including/excluding specific ranks
  - We can structure it as a general graph, and find graph neighbors using the communicator
  - We can structure it as a cartesian grid, and find coordinates + grid neighbors using the communicator

MPI: Parallel I/O

- MPI-IO allows multiple ranks to open the same file simultaneously
- Derived data types can set a different *view* for each rank, to ensure that they don't read/write in the same places
- Collective read/write operations allow all ranks to engage in I/O at the same time
  - Saves us the trouble of appointing one of them to collect data from all the others
  - Runs faster when supported by the file system

Pthreads

- We looked at how pthreads share everything in a process image except for
  - Private stack memory
  - Private instruction counter
- Starting/stopping pthreads is connected to the call/return of a function
- They can produce race conditions unless the program logic prevents them from it
  - There is no automatic protection of shared memory

Pthreads operations

- Create
  - Makes a new thread out of a function call, returns a handle
- Join
  - Waits for the threaded function call to return, using the handle
- Mutex
  - Locking variable that can only be acquired by one thread at a time
- Cond
  - Signal mechanism attached to one or more threads waiting for a mutex
- Barrier
  - Synchronization mechanism that can wait for N threads to arrive at the same point in the program

OpenMP

- Programming model for the same operations as threads, but has
  - Higher abstraction level (easier to write)
  - Additional operations for things that are repetitive to write out explicitly
- Works by #pragma directives
  - If you write it carefully, the program will still work as a sequential implementation even if OpenMP support is turned off
  - Fork/join style parallelism wherever a lexical scope lies inside of a region marked with #pragma omp parallel

OpenMP: mutual exclusion

- Simple assignment statements with commutative operations (probably) have hardware support for mutual exclusion
  - X += 42, Y = Y * Z, Z = 64, ...and such
  - These can be made atomic with the #pragma omp atomic directive
- More complicated blocks can be marked for mutual exclusion with #pragma omp critical
  - This introduces locking/unlocking mechanisms behind the scenes
- We also have an explicit omp_lock_t variable type, which works like the pthread mutex constructs.
  - Explicit function calls omp_lock_set and omp_lock_unset

OpenMP: worksharing

- Worksharing directives partition some section of code according to its function or data
- Functional decomposition: #pragma omp sections
- Data decomposition: #pragma omp for
- Exclusive access: #pragma omp single
- Worksharing directives are followed by a barrier, unless it is disabled with the *nowait* clause

OpenMP: loop scheduling

- The paralle *for* directive divides the iteration space of a for loop among threads
- The parts of the iteration space are assigned according to a *schedule*
  - *Static* (equal parts for everyone)
  - *Dynamic* (list of equal-size parts, assigned as threads finish them)
  - *Guided* (list of initially large, but successively smaller parts, assigned as threads finish them)

OpenMP: tasks

- #pragma omp task creates a work-unit that can be assigned to a thread, and lists it for execution
  - Threads pick up tasks from the list when they are available
- Tasks create dependency graphs when they have a specific order of execution
- Main benefit: tasks can create additional tasks without making assumptions about the size of the thread pool
  - *Nested* parallelism is hard with worksharing directives, but easy with tasks
  - Wonderful for divide&conquer algorithms