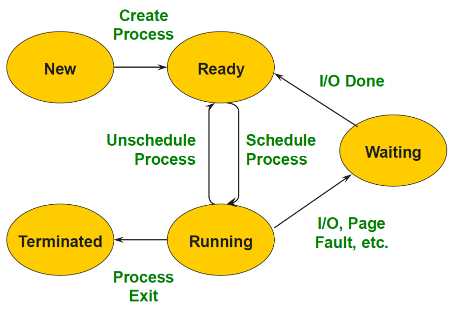
# CS3210 Parallel Computing

## Processes

- **Time slicing**: Same core shared by multiple processes



- `fork()` – useful when child relies on parent's data
- **Disadvantages**: creation of new process is costly (all data structures must be copied), communication costly (goes through the OS)

## Threads

- **User-level**: OS is unaware – fast context switching, but cannot map to different execution resources (no parallelism), blocking I/O will block all threads
- **Kernel**: OS is aware
- **Mapping**:
  Many-to-one: All user-level threads mapped to one process, thread library is responsible for scheduling
  One-to-one: Each user-level thread is mapped to exactly one kernel thread, no library scheduler needed
  Many-to-many: Library scheduler assigns user-level threads to a set of kernel threads, library may move user threads to different kernel threads during program execution
- Number of threads should be suitable to parallelism degree of application, suitable to available parallelism resources, not too large to keep overheads small

## Synchronization

- **Race condition**: two concurrent threads access a shared resource without any synchronization
- **Critical section (CS)** requirements:
  Mutual exclusion: If one thread is in CS, then no other is
  Progress: If thread T is not in CS then T cannot prevent other threads from entering CS; threads in CS will eventually leave it
  Bounded wait: All waiting threads will eventually enter
  Performance: Overhead of entering/exiting CS is small w.r.t. work being done within it
  *Safety property: Mutual exclusion*
  *Liveness property: Progress, Bounded wait*
  *Performance requirement: Performance*
- **Locks**: can spin (spinlock) or block (mutex)
  Software lock:
  - either use hardware atomics (test-and-set) or disable/enable interrupts
  - give up CPU: call yield() or sleep() instruction
- **Semaphores**: Wait()/P() and Signal()/V()
  - mutex sem. (binary sem.) or counting sem. (general sem.)
  - semaphores can be signalled by different thread – no connection to data being controlled
- **Monitor**: allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true (implemented using mutex + cond. variable)
- **Condition variable**: supports three operations:
  Wait: release monitor lock and blocks
  *(Condition variables have wait queues too)*
  Signal: wake one waiting thread
  Broadcast: wake all waiting threads
- **Barrier**: blocks until specified number of threads arrive
- **Starvation**: process is prevented from making progress because some other process has the resources required, e.g.:
  - a high priority process always prevents low priority process from using CPU
  - one thread always beats another to acquire lock
- **Deadlock**: amongst a set of processes, every process is waiting for an event that can be caused only by another process in the set
  Deadlock can exist iff all four conditions hold:
  1. mutual exclusion (at least one non-shareable resource)
  2. hold and wait (at least one process holding resource but waiting for another resource)
  3. no preemption (CS cannot be aborted externally)
  4. circular wait (there must exist a set of processes $\{P_1, \ldots, P_n\}$ such that $\forall i$, $P_i$ is waiting for $P_{(i+1)\%n}$
  Dealing with deadlock:
  - Ignore
  - Prevent (make it impossible for deadlock to happen)
  - Avoid (control resource allocation)
  - Detect & Recover (look for a cycle in dependencies)
- **Producer-consumer with finite buffer**:

```
Producer                          Consumer
■ event = waitForEvent ()         ■ items.wait ()
■ spaces.wait ()                  ■ mutex.wait ()
■ mutex.wait ()                       ❑ event = buffer.get ()
    ❑ buffer.add ( event )        ■ mutex.signal ()
■ mutex.signal ()                 ■ spaces.signal ()
■ items.signal ()                 ■ event.process ()
```

- **Lightswitch**: lock/unlock system to acquire another semaphore if there are nonzero threads in CS (implemented with private mutex and counter)
- **Reader-writer without writer starvation**:

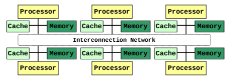
```
Writers                               Readers
■ turnstile.wait ()                   ■ turnstile.wait ()
    ❑ roomEmpty.wait ()               ■ turnstile.signal ()
        ■ # critical section for writers   ■ readSwitch.lock ( roomEmpty )
■ turnstile.signal ()                     ❑ # critical section for readers
■ roomEmpty.signal ()                 ■ readSwitch.unlock ( roomEmpty )
```

## Parallel Computing Platforms

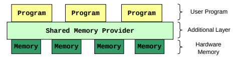
- **Execution time**:
  $$\text{CPU Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$
- **Levels of Parallelism** Bit: word size
  Instruction: execute instructions in parallel:
  - Pipelining (parallelism across time)
  - Superscalar (parallelism across space): duplicate pipeline (scheduling can be dynamic (hardware) or static (compiler))
  Thread: hardware support for multiple thread contexts (PC, registers, etc), e.g. simult. multithreading (SMT)
  Process: independent memory space, use IPC mechanisms
  Processor: multiple processors
- **Thread level multithreading implementations**:
  switch after each instruction → fine-grained multithreading
  switch on stalls → coarse-grained multithreading
  - switch after predefined timeslice → timeslice m.t.
  - switch when proc. wait for event → switch-on-event m.t.
  - sched. inst. from diff threads in same cycle → simult. m.t.

### Architectures

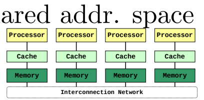
- **Flynn's Taxonomy: SISD, SIMD, MISD, MIMD**
  - MISD: no actual implementation
  - SIMD+MIMD: Stream processor (e.g. NVIDIA GPUs)
- **Memory organization**:
  Distributed-Mem. (Multicomputers) → cannot directly access other mem



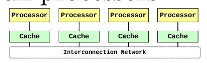
  Shared-Mem. (Multiprocessors) →



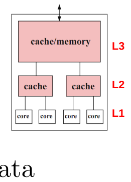
  - Uniform Memory Access (UMA)
    - memory latency same for every processor
  - Non-Uniform Memory Access (NUMA)
    - physically dist. mem. mapped to shared addr. space
    - Cache coherent NUMA (ccNUMA): each node has cache memory that keeps a consistent memory image for all processors

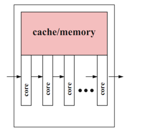


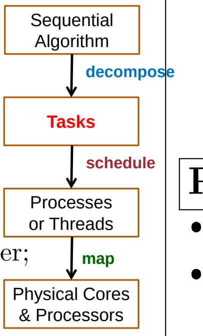
  - Cache-only Memory Access (COMA) →
    - data migrates dynamically



  Hybrid (Distributed-shared mem.)
- **Advantages/Disadvantages of shared memory**:
  - no need to partition code/data, no need to physically move data among processors → efficient communication)
  - but special synchronization constructs are required, and lack of scalability due to contention
- **Multicore architecture**:
  Hierarchical design: multiple caches, slower cache shared by more cores
  Pipelined design: same computation steps to be applied on sequence of data
  Network-based design: cores and their local caches and memories are connected via an interconnection







## Parallel Programming Models

- **Program Parallelization Steps**
  Decomposition: *num. tasks* $\geq$ *num. cores*;
  *task size* $>>$ *parallelism overhead*
  Scheduling: find an efficient task execution order; load balancing among tasks; minimize shared memory access or communicaton operations
  Mapping: focus on performance: equal utilization and minimal communication between processors

  *Decomp. and scheduling can be* static (compile-time or program start) *or* dynamic (during program execution)
- **Types of parallelization**
  Instruction: instructions executed in parallel unless inhibited by data dependencies:
  - Flow (true) dependency (read-after-write)
  - Anti-dependency (write-after-read)
  - Output dependency (write-after-write)
  Loop: indep. iterations run in parallel (e.g. OpenMP for-loop)
  Data: same op. applied to diff. data in parallel (e.g. SIMD, data partitioning into ranges)
  Task (functional parallelism): diff tasks in parallel
- **Task dependence graph**: DAG of tasks and dependencies
  Critical path length: length of longest path
  Degree of concurrency $= \frac{\text{total work}}{\text{critical path length}}$
- **Representation of parallelism**:
  Implicit parallelism:
  - Automatic: compiler automatically decomp. & schedule
  - Functional programming: side-effect-free
  Explicit parallelism:
  - Implicit scheduling: OpenMP
  - Explicit scheduling:
    - Implicit mapping: BSPLib
    - Explicit mapping:
      - Implicit communication & synchronization: Linda
      - Explicit comm. & synchronization: MPI, pthread
- **Parallel programming patterns**:
  Fork–Join: explicit fork()/join() to work in parallel
  Parbegin–Parend: specify sequence of statements to be executed in parallel (e.g. OpenMP for-loop)
  SIMD: same instruction operating on different data
  SPMD: same *program* on diff. processors & data (e.g. MPI)
  Master–Slave: master assigns work to slaves
  Client–Server: MPMD model where server computes requests from multiple client tasks concurrently (can use multiple threads for the same request); a task can generate requests to other tasks (client role) and process requests from other tasks (server role); used in heterogeneous systems e.g. cloud & grid computing
  Task Pools: num. threads is fixed; during processing of a task, new tasks can be generated and inserted into task pool; useful for non-fixed task size, must synch. pool access
  Producer–Consumer: shared data buffer/queue
  Pipelining: stream parallelism (a form of functional parall.)
- **Data distribution**:
  Blockwise: for homogeneous load over data structure
  Cyclic: for inhomogeneous load, to improve load balancing
  Block-cyclic: reduce overhead for cyclic but keep its benefits
  2D arrays: either group row-wise/column-wise or apply data distribution on both dimensions (checkerboard)
- **Information exchange**:
  Shared variables: used for shared address space
  - need to synchronize (e.g. mutex) to avoid race condition (computation result depends on execution order of threads)
  - each thread might also have private variables
  Communication operations: used for distributed addr. space
  - dedicated (explicit) communication operations

## Performance of Parallel Systems

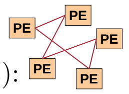
- **Goals**: small response time vs high throughput
- user CPU time $= N_{cycle} \times$ time per cycle
  user CPU time = num. inst. × avg. cycles per inst. *(CPI)* × time per cycle
  - num. inst. and CPI are also compiler-dependent
  Refinement with memory access time (one-level cache):
  user CPU time $= (N_{cycle} + N_{mm\_cycle}) \times$ time per cycle
  (where $N_{mm\_cycle} =$ num. add$^{\text{n}}$ cycles due to memory access
  $N_{mm\_cycle} = N_{read\_cycle} + N_{write\_cycle}$
  $N_{read\_cycle} = N_{read\_op} \times Rate_{read\_miss} \times N_{miss\_cycles}$
- **Average memory access time**:
  $T_{read\_access} = T_{read\_hit} + Rate_{read\_miss} \times T_{read\_miss}$
- **Benchmarks**: SPECint, SPCfp, SPECjvm2008, NAS
- **Parallel execution time** $p :=$ num. processors;
  $n :=$ problem size; $T_p(n) :=$ execution time $(end - start)$
  Cost: processor-runtime product; $C_p(n) = p \times T_p(n)$
  Cost-optimal par. prog. has same cost as *fastest* seq. prog.
  Speedup: $S_p(n) := \frac{T_{best\_seq}(n)}{T_p(n)}$; theoretically $S_p(n) \leq p$
  Superlinear speedup: $S_p(n) > p$ (cache locality, early term., etc)
  Difficulties with measuring speedup: best seq. alg. may not be known; algorithm with optimum asymptotic complexity is slower in practice; seq. alg. implementation is complex
  Efficiency: $E_p(n) := \frac{T_{best\_seq}(n)}{C_p(n)} = \frac{S_p(n)}{p}$; ideal efficiency = 1
- **Amdahl's Law**: Constant unparallelizable fraction of algorithm then: $S_p(n) \leq \frac{1}{f}$ $(\lim_{p\to\infty} S_p(n) = \frac{1}{f})$
  - $f$ $(0 \leq f \leq 1)$ is the *sequential fraction* of algorithm
  - also known as *fixed-workload performance*
  - rebuttal: in many cases $f$ is non-const, dependent on $n$
- **Gustafson's Law**: Constant execution time for sequential part then $S_p(n) \leq p$ $(\lim_{n\to\infty} S_p(n) = p)$
  - const exec. time of seq. part with increasing problem size

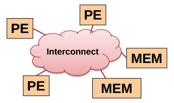
- **Grosch's Law (rebuttable)**: the speed of a computer is proportional to the square of cost → bigger processor better
- **Minksy's Conjecture (rebuttable)**: the speedup of a parallel computer increases as the logarithm of num. of processing elements → large-scale parallelism unproductive

## Coherence & Consistency

- **Write policy**:
  Write-through: immediately transferred to main memory
  Write-back: dirty bit, only transfer on cache replacement
- **Cache coherence definitions**:
  1: $P$ write to $X$, n.f.w., then read from $X$ – should get same value
  2 (Write Propagation): $P_1$ write to $X$, no further write to $X$, then $P_2$ read from $X$ – should get same value
  3 (Write Serialization): Any processor write $V_1$ to $X$, any processor then write $V_2$ to $X$ – all processors should never read $X$ as $V_2$ then later as $V_1$ (i.e. writes seen in same order)
- **Hardware cache coherence tasks**:
  - track sharing status and update shared cache line
  Snooping based: no centralized directory, cache monitors (snoops) on the bus to update its cache line
  Directory based: sharing status kept at centralized location, common with NUMA
- **Memory consistency**: Each processor has consistent view of memory through its local cache
  Sequential consistency (SC): all reads/writes are serializable
  Relaxed consistency: read may be reordered before write of different variable
  - Total store ordering (TSO): writes seen by all other processors at the same time, in instruction order
  - Processor consistency (PC): writes seen by *each* processor in instruction order (like N×N pipes)
  - Partial store ordering (PSO): writes seen by all other processors at the same time, out of instruction order (i.e. write-write reorder of different variables can happen)

## Interconnections

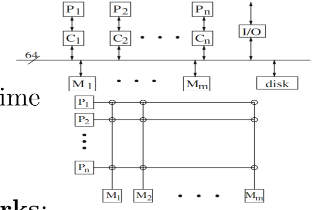




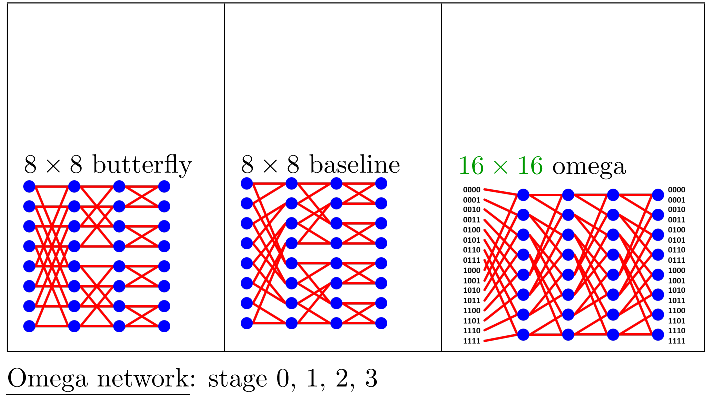
- **Major types**:
  Direct (or Static, Point-to-Point): each endpoint is a processing element
  Indirect (or Dynamic): interconnect formed by switches
- **Embedding**: Can embed $G'$ into $G \iff \exists\sigma: V' \to V$ s.t. $\sigma$ is injective, and if $(u,v) \in E'$ then $(\sigma(u), \sigma(v)) \in E$
- **Direct interconnections**:

| $n$ **nodes** | **degree** | **diameter** | **edge conn.** | **bisect.** |
|---|---|---|---|---|
| complete graph | $n-1$ | $1$ | $n-1$ | $\left(\frac{n}{2}\right)^2$ |
| linear array | $2$ | $n-1$ | $1$ | $1$ |
| ring | $2$ | $\left\lfloor\frac{n}{2}\right\rfloor$ | $2$ | $2$ |
| $d$-d mesh ($n = r^d$) | $2d$ | $d\,(r-1)$ | $d$ | $n^{\frac{d-1}{d}}$ |
| $d$-d torus ($n = r^d$) | $2d$ | $d\left\lfloor\frac{r}{2}\right\rfloor$ | $2d$ | $2n^{\frac{d-1}{d}}$ |
| $k$-d hypercube ($n = 2^k$) | $\log n$ | $\log n$ | $\log n$ | $\frac{n}{2}$ |
| $k$-d CCC ($n = k2^k$) | $3$ | $2k-1+\left\lfloor\frac{k}{2}\right\rfloor$ | $3$ | $\frac{n}{2k}$ |
| complete bin. tree ($n=2^k-1$) | $3$ | $2\log\frac{n+1}{2}$ | $1$ | $1$ |
| $k$-ary $d$-cube ($n = k^d$) | $2d$ | $d\left\lfloor\frac{k}{2}\right\rfloor$ | $2d$ | $2k^{d-1}$ |

  - XY-routing (2D mesh): move in $X$-dir until $X_{src} = X_{dest}$ then move in $Y$-dir
  - E-cube routing (hypercube): compare coord. tuples of src and dest; start with MSB or LSB, take link to correct bit if bit differs
- **Topology Metrics**:
  Diameter: max dist. between any two nodes
  - small diameter → small distance for message transmission
  Degree: max node degree in graph
  - small degree → small node hardware overhead
  Bisection width: min edges removed to divide network into *equal* halves
  - determines max node messaging rate network can support
  - required link data rate $= \frac{\text{num. nodes}}{2} \times \frac{\text{node messaging rate}}{\text{bisection width}}$
  Node connectivity: min num. nodes that must fail to disconnect network (determines robustness)
  Edge connectivity: min num. edges that must fail to disconnect network (determines num. indep. paths between any pair of nodes)



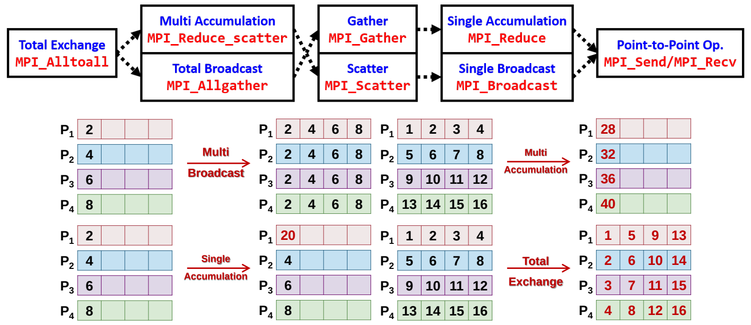
- **Indirect interconnections**:
  Bus network: only one pair of devices can communicate at a time
  Crossbar network: switch state: *straight* or *direction change*
- **Multistage switching networks**:



  Omega network: stage 0, 1, 2, 3
  - n/2 switches per stage
  - switch $(\alpha, i)$ [$\alpha$=pos. of *switch* within stage (e.g. $\in [0, 8)$); $i$=stage number] has edge to switch $(\beta, i+1)$ where $\beta \in \{\alpha \text{ by cyclic left shift}, \alpha \text{ by cyclic left shift} + \text{inversion of LSB}\}$
  XOR-tag routing for omega network:
  Let $T = sourceID \oplus destID$ (e.g. $sourceID, destID \in [0, 16)$)
  At stage $k$: go straight if $k^{\text{th}}$ bit of $T$ is 0, crossover otherwise
- **Evaluate** indirect interconnections by: cost (num. switches/links), num. allowable concurrent connections
- **Routing algorithm classification**:
  Minimal/Non-minimal: whether shortest path is always chosen
  Adaptive/Deterministic: adjust by network status/congestion

## Message Passing

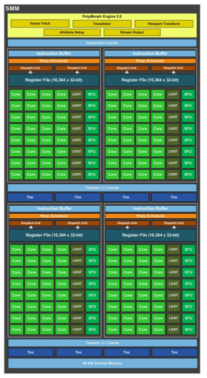
- **Loosely synchronous**: processes synchronize to perform interactions; apart from that tasks execute fully asynchronously
- **Protocol possibilities**:
  Buffered: uses a temporary buffer (instead of tx. directly to network)
  Blocking: original array can be reused after function returns
  - non-buffered blocking operation misuse can cause idling/deadlocks
  - idling due to mismatch in timing between sender and receiver
  - non-blocking op. hides communication overhead, and usually accompanied by a check-status op.
  - blocking send can pair with non-blocking recv and vice-versa
  Synchronous (MPI only): op. does not complete until both sender and receiver have started communication op.
- **MPI message**: Data: buffer, count, datatype;
  Envelope: src/dst, tag, communicator
  - Group = set of processors
  - Communicator = communication domain for one or two groups of processes
    - Intra-communicators: communicate within single group
    - Inter-communicators: communicate between two groups
    - `MPI_COMM_WORLD` is an intra-communicator
- **Virtual topologies**: e.g. Cartesian, Graph – easier to address neighbours or by 2D pair of coords
- **"Proc. consistency"**: order not guaranteed with more than two processes; but same src to same dst → in order
- **Deadlock** can happen (xchging data between 2 proc) when:
  - Two processes have blocking recv before send
  - Two processes have unbuffered blocking send before recv
  Instead, *even* proc should send while *odd* proc recv first
- **Collective Communication**

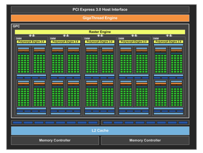


- **Duality**: same spanning tree can be used for both ops.

## CUDA Programming

- **Drawbacks of shader GPGPU (general purpose GPU)**:
  - awkward programming interface unnecessarily dependent on graphics pipeline  - no scatter
  - hard to transfer data from host to device
  - no communication between threads
  - coarse thread synchronization





- **GPU architechture**:
  - multiple streaming multiprocessors (SMs)
  - SM consists of multiple compute cores, memories, schedulers
- **CUDA programming model**:
  - SPMD model
  - transparently scales to arbitrary num. cores
  - programmers focus on parallel algorithms
  - enable heterogeneous systems (i.e. CPU+GPU)
  - threads need not be completely independent – can share results and memory accesses, atomic operations
  - in same block: can use shared mem, barrier syncthreads
  - each block assigned an SM and cannot migrate
  - several blocks can reside concurrently on one SM
  - register file partitioned among resident threads
  - shared memory partitioned among resident blocks
- **Execution mapping to architecture**:
  - SIMT (single instruction, multiple thread) execution model
  - multiprocessor sched. & exec. threads in SIMT warps (32 threads)
  - threads in a warp start tgt at same program address
  - each warp executes one common instruction at a time
  - scheduler groups threads with same exec. path in same warp

| **Memory** | **On/off chip** | **Cached** | **Access** | **Scope** | **Lifetime** |
|---|---|---|---|---|---|
| Register | On | N/A | R/W | 1 thread | Thread |
| Local | Off | No | | | |
| Shared | On | N/A | | All threads in block | Block |
| Global | Off | No | | All threads + host | Host allocation |
| Constant | | Yes | R | | |
| Texture | | | | | |

- Constant memory – can only read one int32 per cycle
- Shared memory – divided into banks, diff. banks can be accessed simultaneously

- **Compilation**:
  - NVCC outputs C host code (to be compiled using another compiler), and PTX code (interpreted at runtime)
- **Device code restrictions**: can only access GPU memory, no varargs, no static variables, (old versions) no recursion
- **Variable qualifiers**:
  - `__device__`: global memory, use `cudaMemcpyToSymbol`
  - `__constant__`: constant (cached), use `cudaMemcpyToSymbol`
  - `__shared__`: on-chip shared memory (very low latency)
- `int atomicCAS(int* address, int expected, int newval);` (returns `oldval`, overloads for uint and ull avail.)
- **Optimization strategies**:
  - maximize par. exec. to expose maximum data parallelism
  - optimize memory usage to maximize memory bandwidth
  - optimize instruction usage to maximize instruction throughput (e.g. avoid divergent warp, low-precision floats)
- **Memory optimizations**:
  - minimize data transfer between host and device
  - coalesce global mem access (simult. access to global memory by threads in a half-warp can be coalesced into single memory transactions of 32/64/128 bytes; requirements (alignment, random) based on compute capability)
  - prefer shared mem to global mem where possible
  - minimize shared mem bank conflicts

## Parallel Algorithm Design

- Consider machine-independent issues first
  Task/Channel model: task = code & data needed for computation; channel = message queue from one task to another
- **Foster's design methodology**:
  Partitioning of problem into small independent tasks
  - *Data-centric (domain decomposition)*: divide data into pieces (≈ equal size), associate computations with data
  - *Computation-centric (functional decomposition)*: divide computation into pieces, associate data with computations
  Communication between tasks
  - *local comm.*: task needs data from small number of other tasks only (create channels illustrating data flow)
  - *global comm.*: significant num. of tasks contribute data for calculation (don't create channels for them early in design)
  Agglomeration: combine tasks to larger tasks
  - reduce overheads (task creation + communication)
  - (In MPI, usually one agglomerated task per processor)
  Mapping of tasks to processors to maximize processor utilization (place tasks on different processors) but minimize inter-processor communication (place tasks that communicate frequently on the same processor)
  - done by OS for centralized multiprocessor
  - done by user for distributed memory systems

## Energy-Efficient Computing

- **Heterogeneous computing**:
  - Programs: OpenMP+MPI, OpenMP+CUDA, MapReduce
  - Systems:
    - Inter-node: diff. CPU generations, brawny+wimpy
    - Intra-node:
      - Inter-chip: CPU+VPU ("vision", AI accelerator), CPU+GPU
      - Intra-chip: CPU+GPU, ARM big.LITTLE
  *Heterogeneity: more power-efficient*
  Functional heterogeneity: different ISA
  Performance heterogeneity: same ISA, different speed
- **Costs of computing**: Higher performance → More/faster computers → Power → Heat → Cooling → Space → Money/Env. cost; Cooling → Power → Money/Env. cost
- **Cloud computing**:
  - Characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity, measured service
  - Service models: SaaS (software), PaaS (platform), IaaS (infra.)
  - Deployment models: private/community/public/hybrid clouds

- **Virtualization**: server/storage/network/services(e.g.DB)
- **Power use effectiveness (PUE)** = $\frac{\text{total energy used}}{\text{energy used for processors}}$
  Increase energy efficiency by: - building custom servers (minimize AC/DC conversions, remove unnecessary parts, strategic positioning, decrease fan speed)
  - control temp. of equipment (raise temp. to 26°C, manage airflow, thermal modeling, hot/cold aisles, seawater cooling)