Course 02158

Parallel Computation

Hans Henrik Løvengreen

DTU Compute

Ever Growing Need for Computational Power

Modelling and analysis

- Climate prediction
- System biology — whole genome sequencing
- Particle collission analysis
- Market analysis

Entertainment

- Computer animation
- Virtual/augmented reality
- Speech recognition

Technology

- Machine learning

Performance Improvement Measures

Rely on Moore's Law?

- *"Size and speed grows with a factor 2 every 2 years"*
- Speed of a single processor has stalled at 2–3 GHz for the last 15 years
- Why? Heat!

Solution

- Many (releatively slow) processors on a single chip
- Such on-chip processing elements are often called *cores*
- In this course: processing element = core = processor = CPU
- We also assume processors to be *uniform* (having same capabilities)
- Recent development: *performance cores* vs. *efficiency cores* — not covered

Performance

Notions

- p number of processors
- T_1 Time for sequential computation
- T_p Time for parallel computation (using p processors)

Measures

- *Speedup*

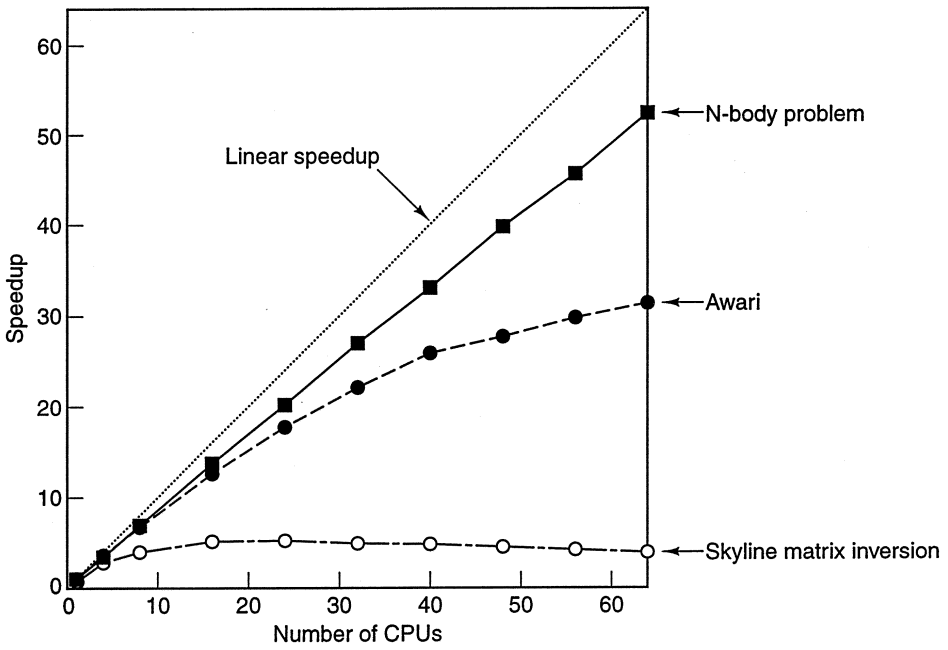
$$S \triangleq \frac{T_1}{T_p}$$

- *Efficiency*

$$E \triangleq \frac{S}{p} = \frac{T_1}{p \times T_p}$$

Speedup

- Examples



Amdahl's Law

- Let f be the *"sequential fraction"* of a problem

f	

- Then

$$T_p = f \times T_1 + \frac{(1-f) \times T_1}{p}$$

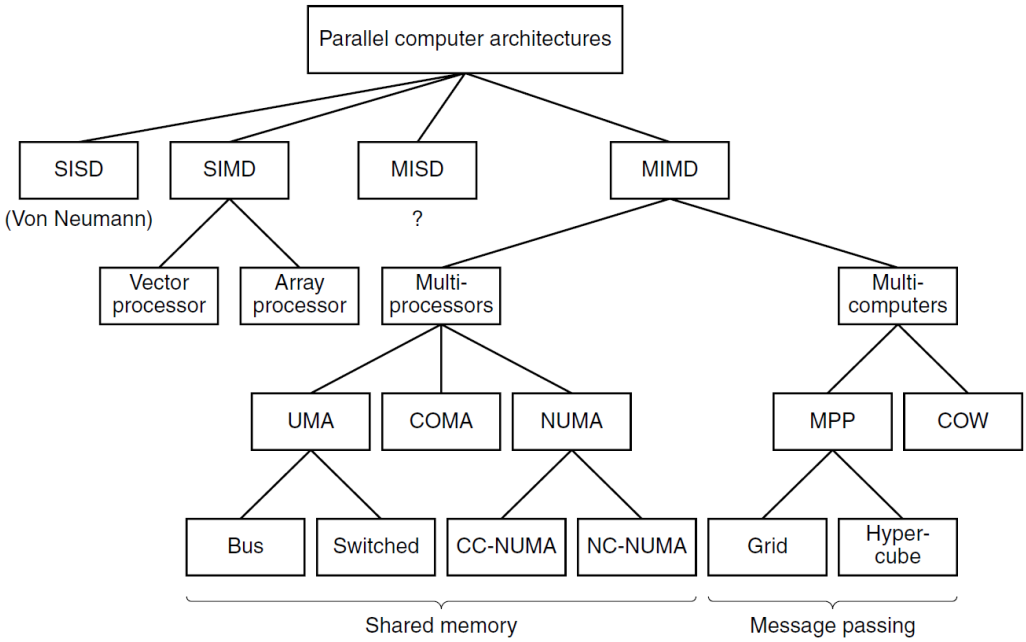
- and

$$S = \frac{T_1}{T_p} = \frac{T_1}{f \times T_1 + \frac{(1-f) \times T_1}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

- For $p \to \infty$,

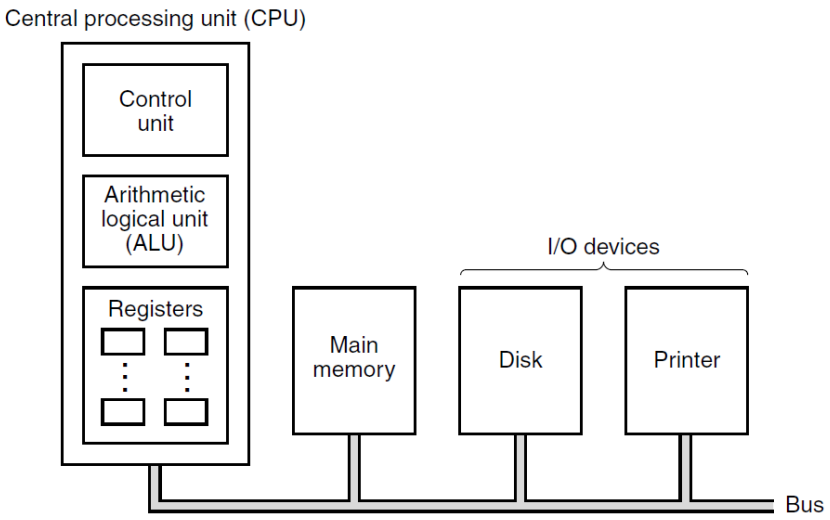
$$S \to \frac{1}{f}$$

Flynn's Taxonomy

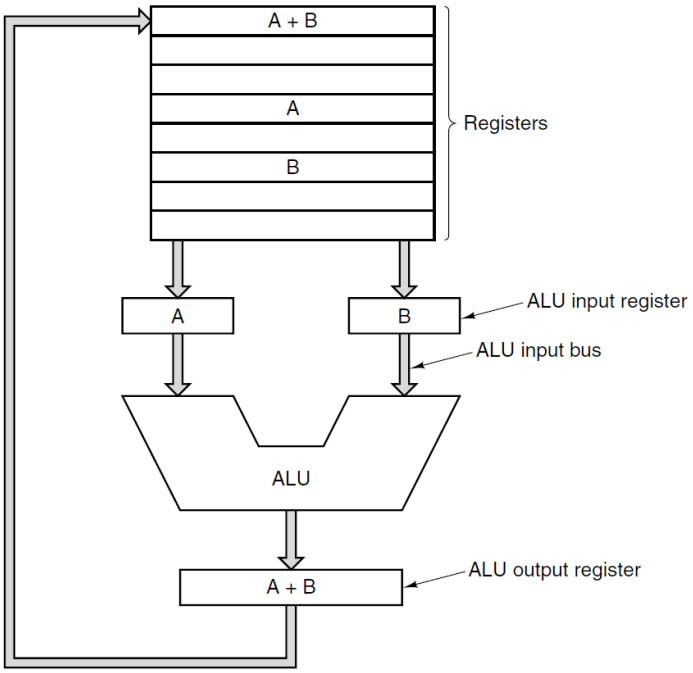


Von Neumann Machine

- IAS ∼1950



Data Path



Performance Improvement Tricks

Caching

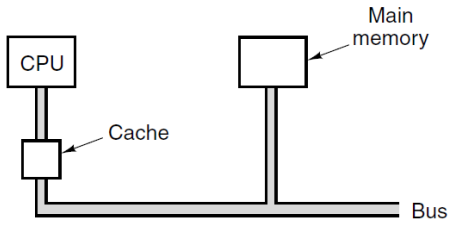
- Bulk memory is slow relative to processor speed
- Exploit *spatial* and *temporal locality* of memory accesses
- Hold *lines* of memory in fast memory — the *cache*
- Many levels — *memory hierarchy*

Instruction-level parallelism

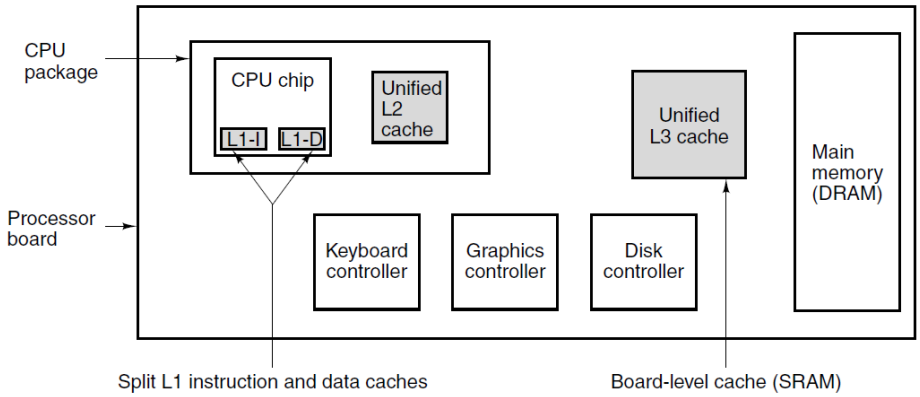
- Exploit *sequential nature* of instruction execution
- Overlap execution of several instruction at a time — *pipelining*
- Extended with *branch prediction*, *speculative*, and *out-of-order execution*
- Exploit the *stalling periods* of instruction execution
- Interleave execution of several instruction sequences — *hyper-threading*
- Requires *register replication*

Caching

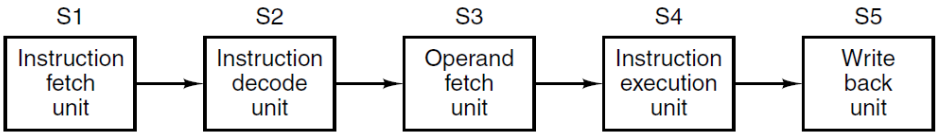
- Basic idea:



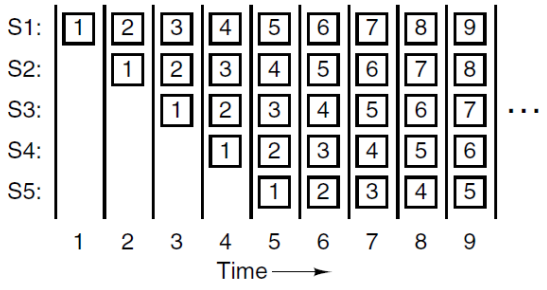
- Many levels



Pipelining

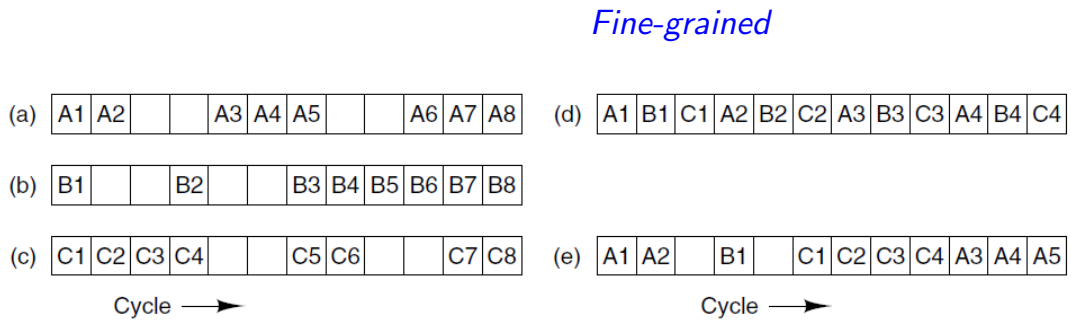


(a)



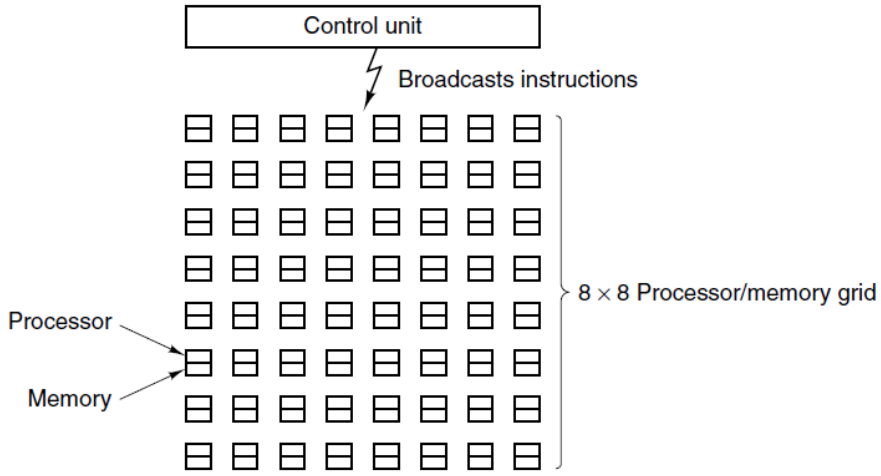
(b)

Hyper-Threading



SIMD

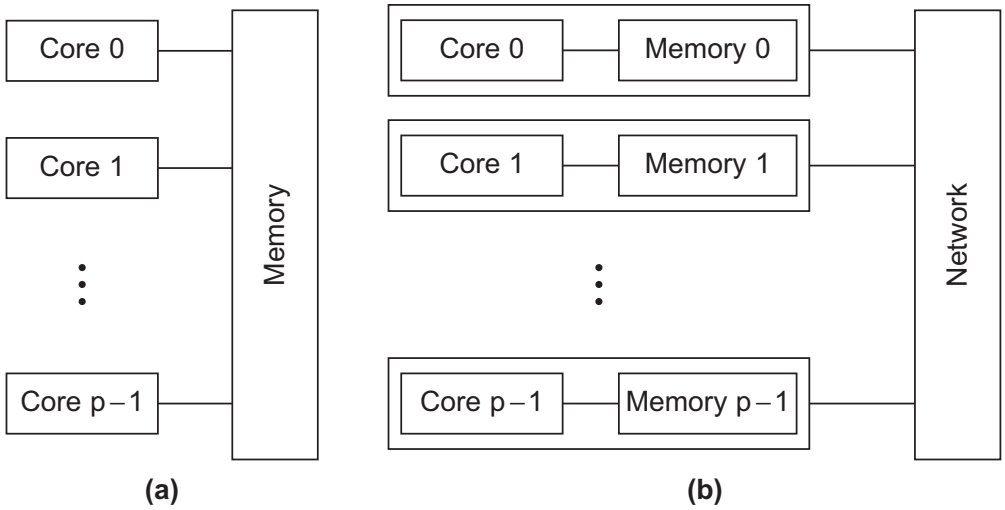
- Array Processor:



- Vector processors, vector instructions
- Graphical Processing Units (GPUs)

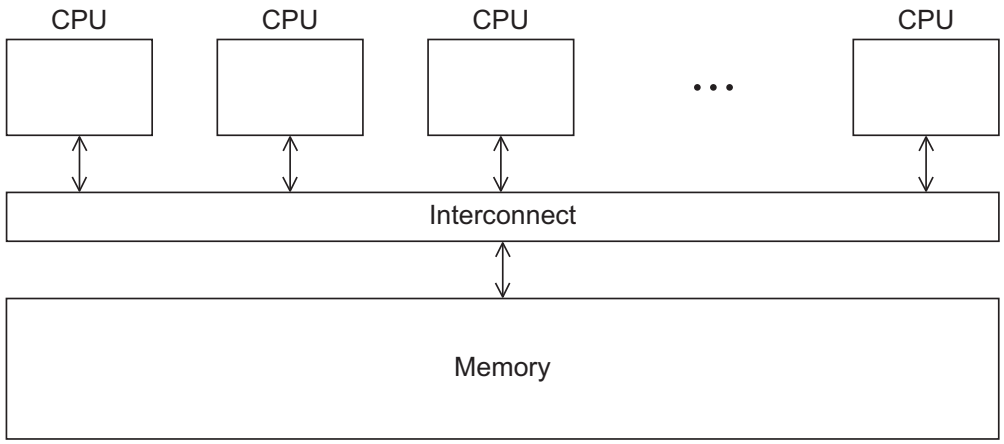
MIMD Architectures

Principal Division



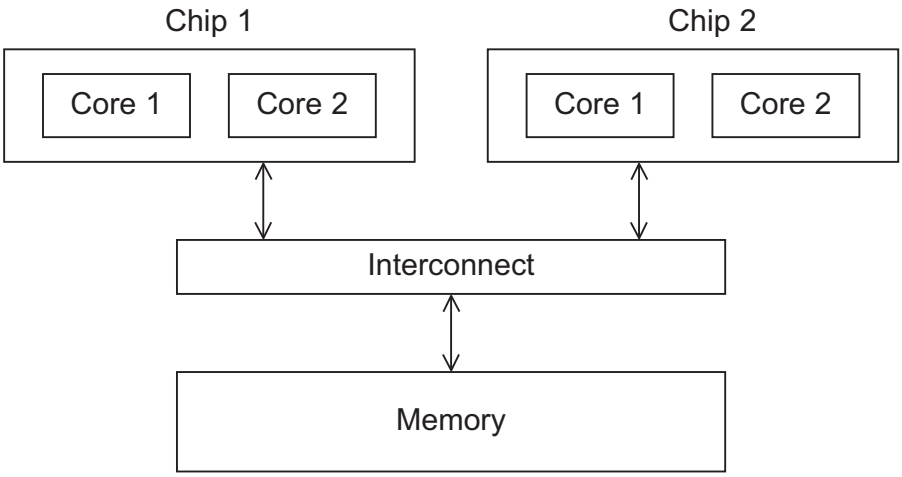
Shared memory *Distributed memory*

Shared-Memory Systems — Multiprocessors

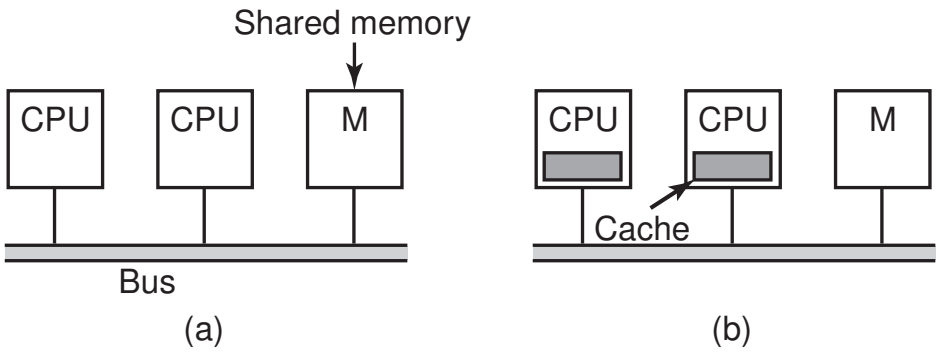


UMA Multiprocessors

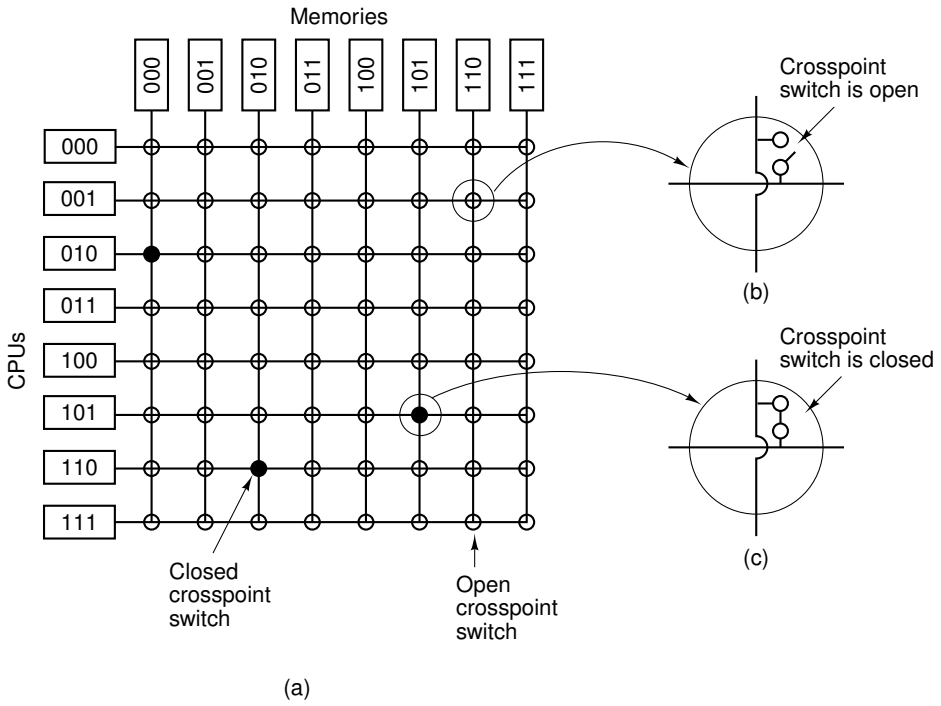
- Uniform Memory Access



Interconnect: Bus

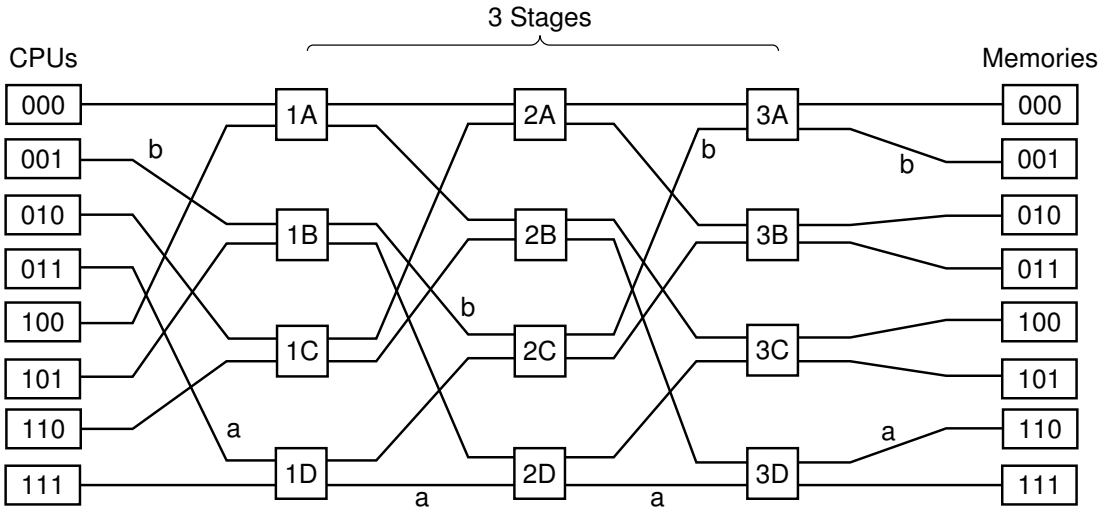


Interconnect: Crossbar Switch



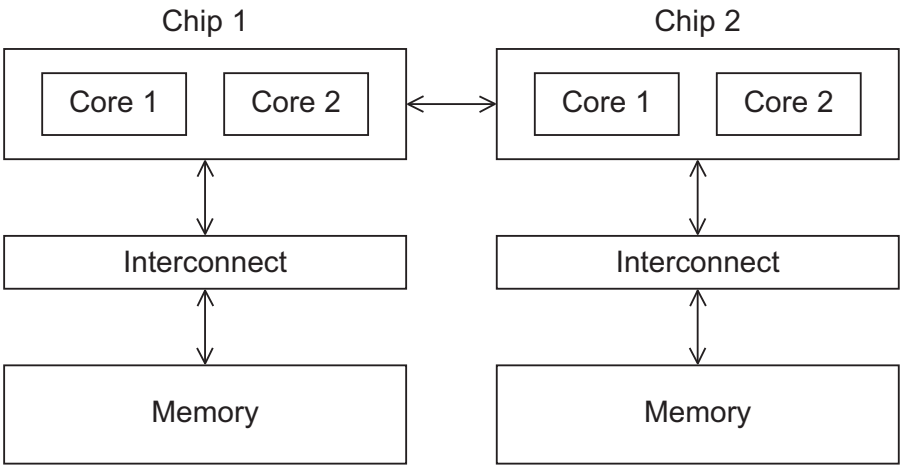
Interconnect: Switching Network

- Omega network

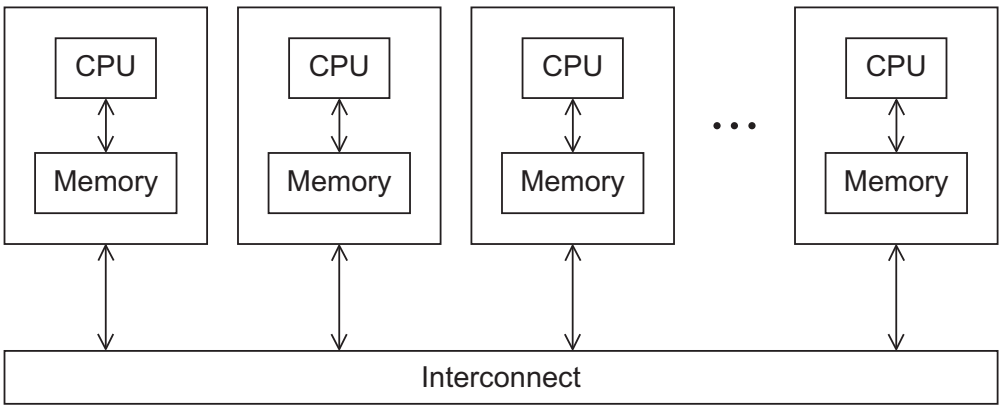


NUMA Multiprocessors

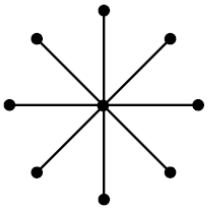
- Non-Uniform Memory Access



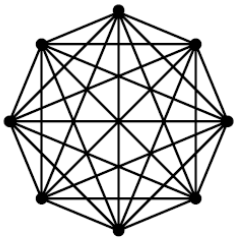
Distributed Memory Systems — Multicomputers



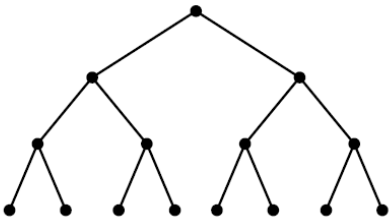
Interconnection Network Topologies I



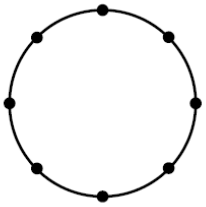
(a)



(b)

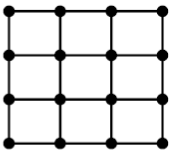


(c)

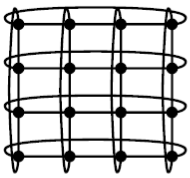


(d)

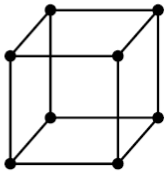
Interconnection Network Topologies II



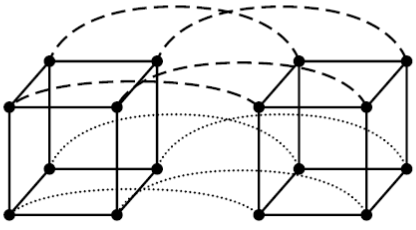
(e)



(f)

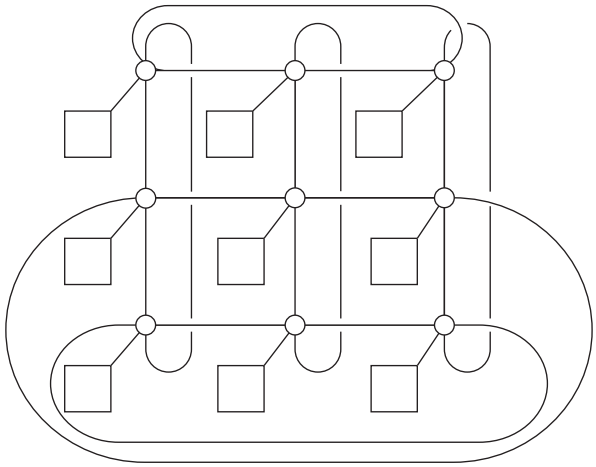


(g)



(h)

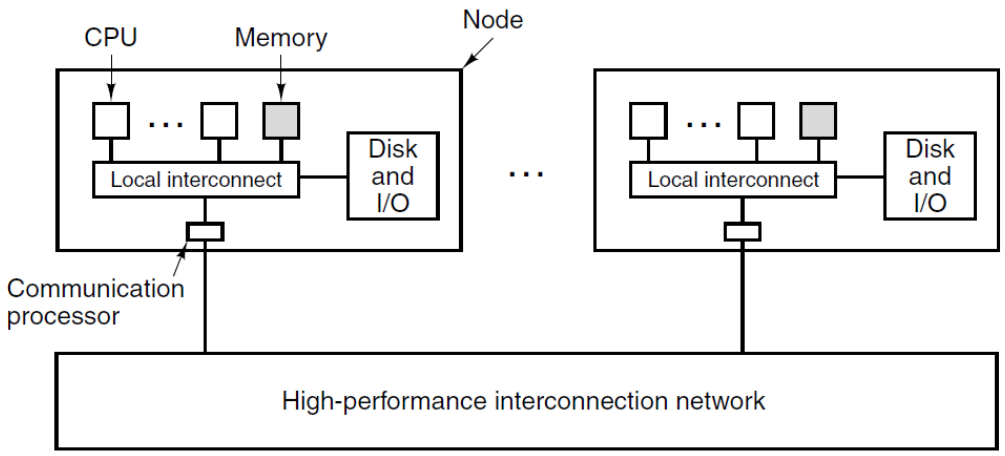
On-chip Interconnect: Toroidal Mesh



(b)

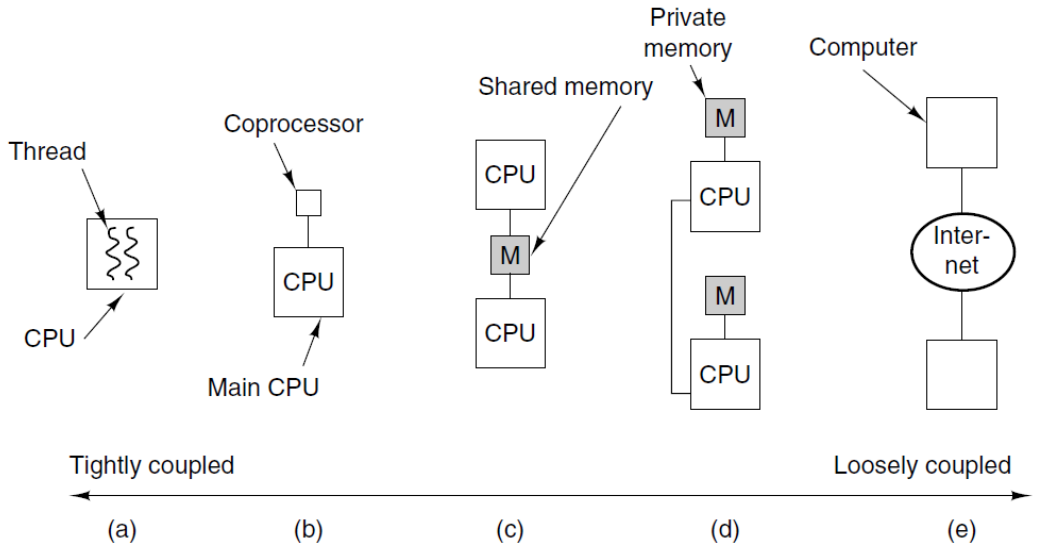
Hybrid Architectures

- Cluster



- DTU HPC clusters: Eg. 30 nodes with 12+12 cores

Perspective: Parallel Architectures



Parallel Programming

- The art of utilizing machine architectures with many parallel processors
- Program each processor individually?
  No — utilize OS scheduling of processes/threads

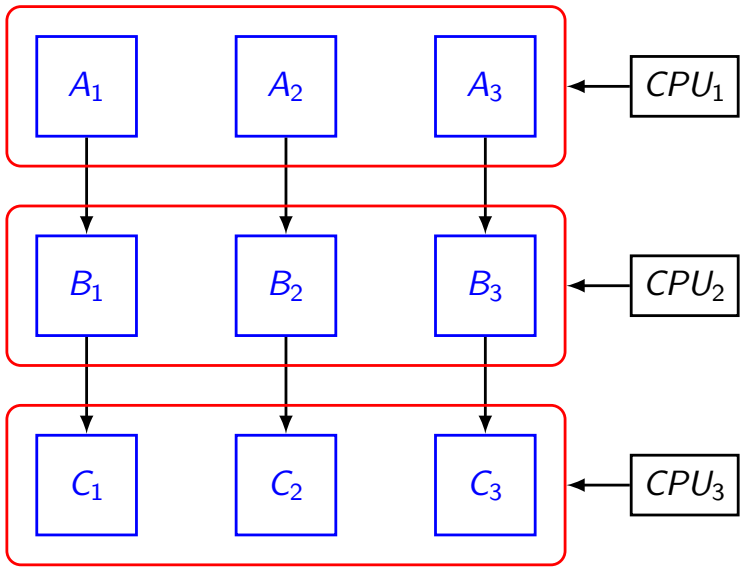
Principles

- *Data parallelism:* Dividing the data into sections which can processed in parallel using the same procedure
- *Task parallelism:* Dividing the computation into phases which can be processed in parallel using dedicated procedures

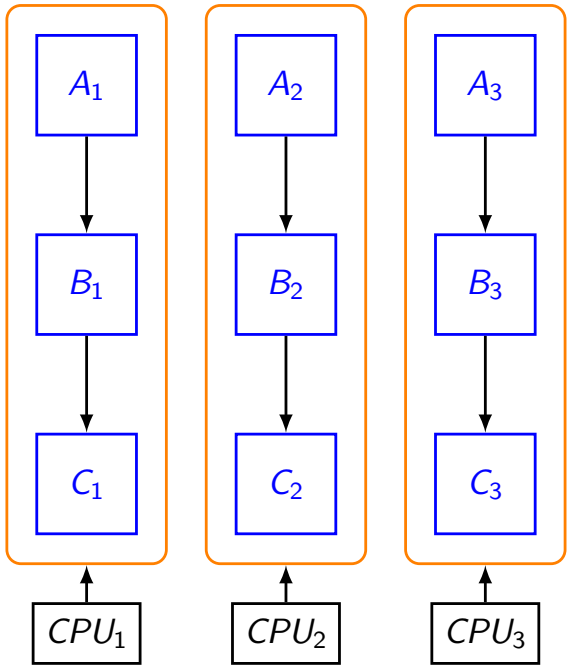
Approaches

- Manual reconstruction of program using *explicit parallelism*
- Manual parallelization *annotation* of sequential programs
- Decomposition into many small computation parts (tasks)
  — to be submitted to a *parallel computation engine*
- *Automatic parallelization* of sequential programs

Work Decomposition — Task Parallelism

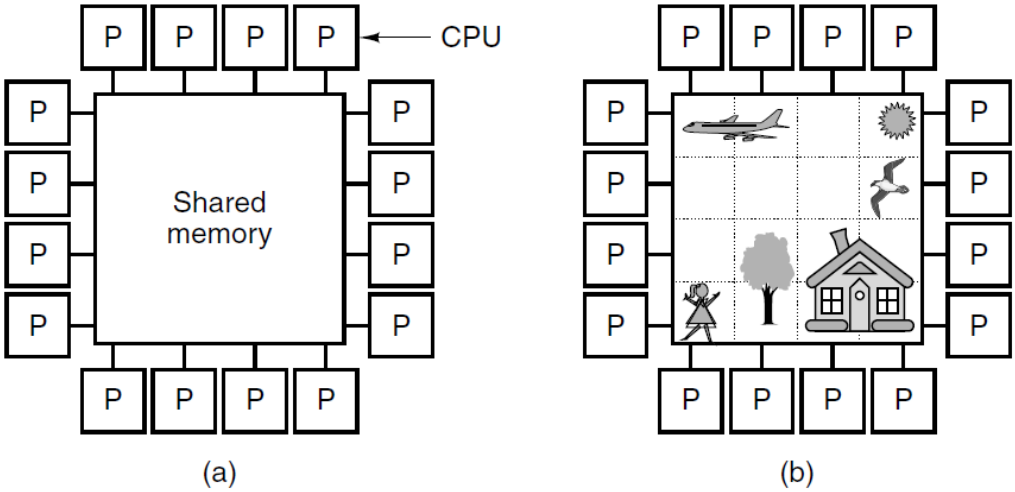


Work Decomposition — Data Parallelism



Multiprocessors — work distribution

- Each processor is given a fair share of the data in memory



- Basis for OpenMP

OpenMP

Background

- Thread-programming on multi-processors: Much *boiler-plate code*
- 1990'ies: HW and SW vendors formed *OpenMP ARB* (Arch. Rev. Brd.)
- First OpenMP standard for Fortran in 1997 and C/C++ in 1998.
- Now at version 6.0 (November 2024) [964 pages!]

Idea

- To parallelize sequential programs by annotations (known as *pragmas*)
- Integrated within compilers (e.g. gcc)
- Supported by run-time library

OpenMP Directives

- In general, a *pragma* is an instruction or hint to the compiler
- C pragmas have the same form as preprocessor commands:

```
#pragma ...
```

- An *OpenMP directive* is a C-pragma of the form

```
#pragma omp ...
```

- Examples:

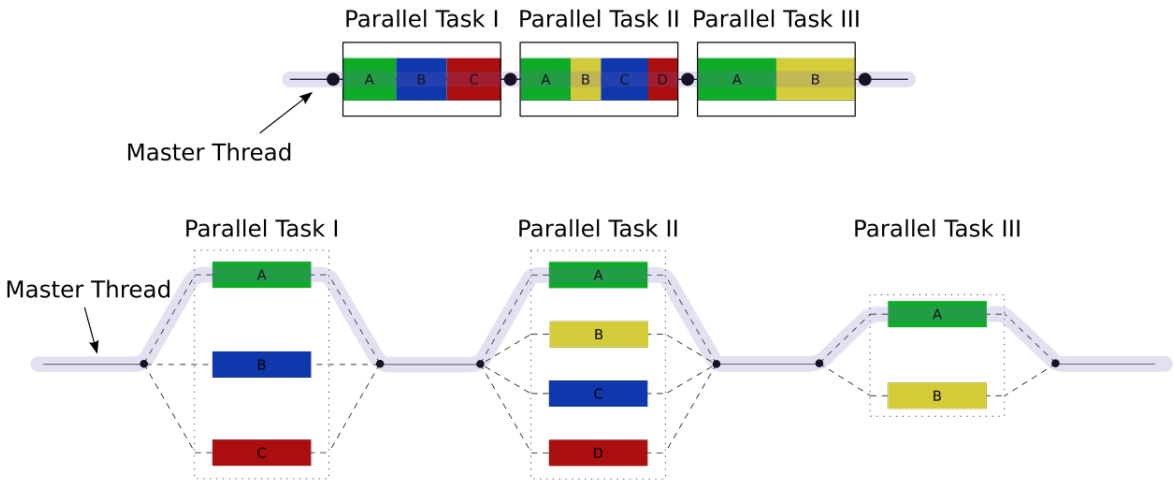
```
#pragma omp parallel
#pragma omp critical
#pragma omp parallel for
```

- Applies to succeeding statement/structured block
- A *clause* is a modification to a directive, e.g.

```
#pragma omp parallel num_threads(8)
```

OpenMP Threads

- Main thread forks into a *team of threads* which join again



OpenMP Skeleton

```
#include <omp.h>

int main() {

#pragma omp parallel
  { /* block to be executed by a thread team */
    int procs = omp_get_num_threads();
    int id = omp_get_thread_num();
    ...
  }

  /* gather result */
}
```

- No. of threads determined by environment variable `OMP_NUM_THREADS`
- May be overridden by parallel clause: `...num_threads(`*n*`)`

Example: Hello OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello␣from␣thread␣%d␣of␣%d\n", my_rank, thread_count);
}

int main(int argc, char* argv[]) {

# pragma omp parallel
   Hello();

}
```

Scoping

- Global variables are normally *shared* in a `parallel` block
- Parallel clause `...private(x, y)` introduces local copies

Example

- ```
  int i, x, y;

  x = 17;
  y = 42;

  # pragma omp parallel private(i, x);
    for (i = 0; i < N; i++ } {
      int z ;
      x = x + 1;
      z = x + y;
    }
  ```
- Here `i, x`, and `z` are private to each thread

Critical region

- Critical pragma: `#pragma omp critical (name)`
- Effect: Succeeding statement is executed within critical region *name*
- If name omitted: Use global, default anonymous region

Example

- ```
  #pragma omp parallel
    { ...
  # pragma omp critical (xreg)
      x = x + 1;
  # pragma omp critical (yreg)
      y = y + 1;
      ...
    }
  ```
- `x` and `y` are incremented atomically — but concurrently

Reduction

- Often results are collected in global variables protected by `critical` pragmas
- Parallel clause `...reduction(`$\oplus$`:`x`)`:
- $\oplus$ is the *reduction operator*: `+, *, &, |, &&, ||, max, min`
- x is the *reduction variable*

Effect

1. Introduces a local copy x_i of x for each thread i
2. Initializes $x_i =$ *< neutral element >*
3. At join: $x = x \oplus x_0 \oplus x_1 \oplus \cdots \oplus x_{p-1}$

Parallel Loop Construct

- Iterations in for loops are often fairly independent
- They may be divided into blocks for *data parallelism*
- Parallel loop: `#pragma omp parallel for`
- Next statement must be a **for** loop

Effect

- Creates a team of threads (as would a `parallel` construct)
- Distributes the iterations of the for statement among the threads
- By default, the iterations are evenly divided into large blocks distributed among the threads
- No explicit work division is needed
- Loop variable automatically becomes `private`
- Shared variables must still be protected or used for reduction

Example: Calculation of π

Given

$$\pi = 4 \cdot \sum_{i=0}^{\infty} (-1)^i \frac{1}{2i+1} == 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right)$$

Sequential program

```
double pi
double factor = -1.0;
double sum = 0.0;
int i;


for (i = 0; i< n; i++) {
  factor = -factor;
  sum += factor/(2*i + 1);
}
pi = 4.0 * sum;
```

Example: Calculation of π (ERRONEOUS)

Given

$$\pi = 4 \cdot \sum_{i=0}^{\infty} (-1)^i \frac{1}{2i+1} == 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right)$$

Parallel program

```
double pi
double factor = -1.0;
double sum = 0.0;
int i;

#pragma omp parallel for reduction(+: sum)
for (i = 0; i< n; i++) {
  factor = -factor;
  sum += factor/(2*i + 1);
}
pi = 4.0 * sum;
```

Example: Calculation of π (CORRECT)

Given

$$\pi = 4 \cdot \sum_{i=0}^{\infty} (-1)^i \frac{1}{2i+1} == 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right)$$

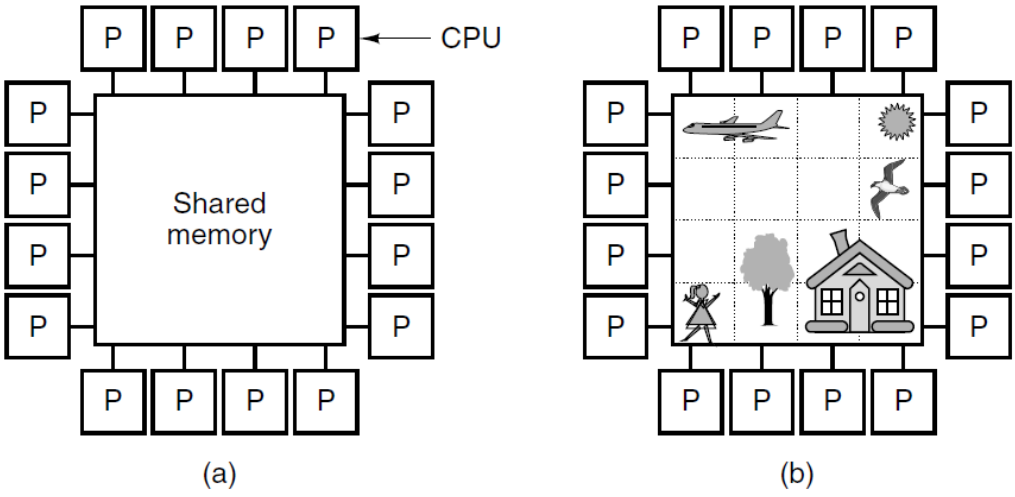
Parallel program

```
double pi
double factor = -1.0;
double sum = 0.0;
int i;

#pragma omp parallel for reduction(+: sum) private(factor)
for (i = 0; i< n; i++) {
  factor = (i%2 ==0) ? 1.0 : -1.0;
  sum += factor/(2*i + 1);
}
pi = 4.0 * sum;
```

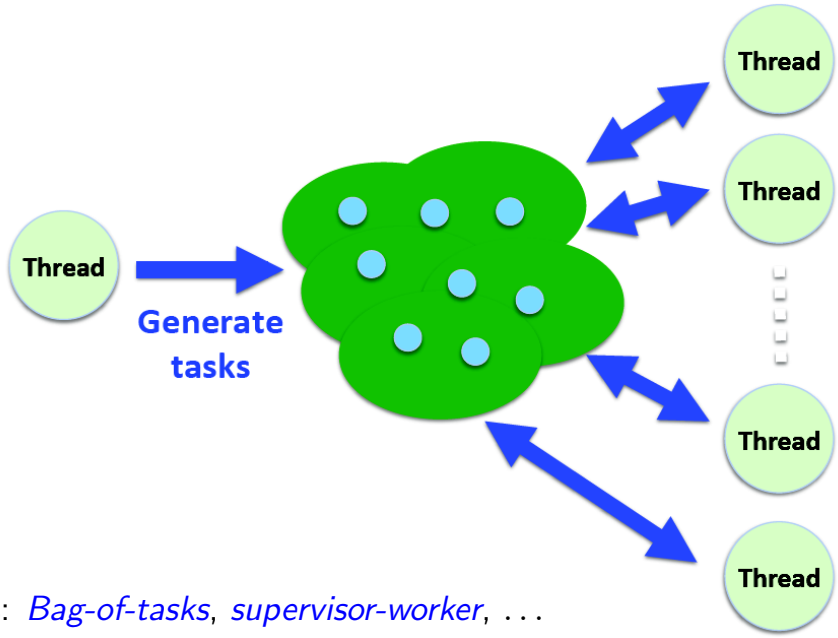
Work Load Balancing

- Goal: Keep all processors busy all the time



- Load not known in advance: Use *dynamic distribution* of smaller *tasks*

Bag-of-Tasks

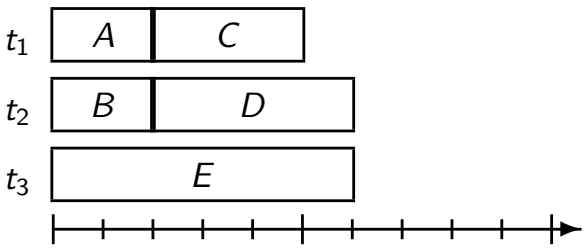


- Many names: *Bag-of-tasks*, *supervisor-worker*, ...
- Many frameworks: OpenMP, Java fork/join , .NET TPL, ...

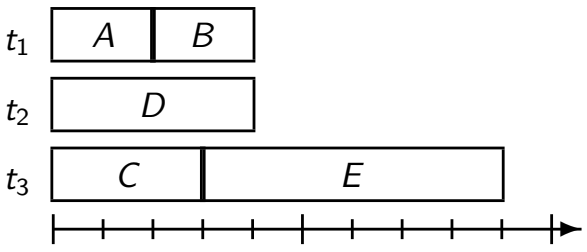
Graphics by Ruud van der Pas

Task Scheduling

- Given task sizes: A: 2, B 2, C: 3, D: 4, E: 6
- Fastest (3 worker threads)



- Slowest



OpenMP Work Scheduling

- Default work division of `parallel for` not always adequate
- Loop clause `...schedule(`*kind*`[,` *k*`])` controls division

Kinds of schudulings

`static`	Default. One (big) chunk of iterations per thread.
`static,` *k*	Fixed round-robin distribution of *k*-iteration chunks
`dynamic,` *k*	Dynamic distribution of *k*-iteration chunks
`guided,` *k*	Proportional share of remaining iterations (at least *k*)
`auto`	System controlled
`runtime`	Determined by `OMP_SCHEDULE` environment variable

OpenMP Tasks

- Introduced in OpenMP 3.0 — extended in 4.0 and 4.5
- A means for dynamic distribution of work on *irregular structures*:
  - Unbounded while loops
  - Lists and trees
  - Producer/consumer schemes
- Tasks are generated (created, spawned) from other tasks

```
#pragma omp task
```

- Execution of tasks is distributed among current team of threads

Example: List Processing

- Given a linked list of elements pointed to by *p*

- ```
  while (p != null) {
    compute(p);
    p = p->next;
  }
  ```

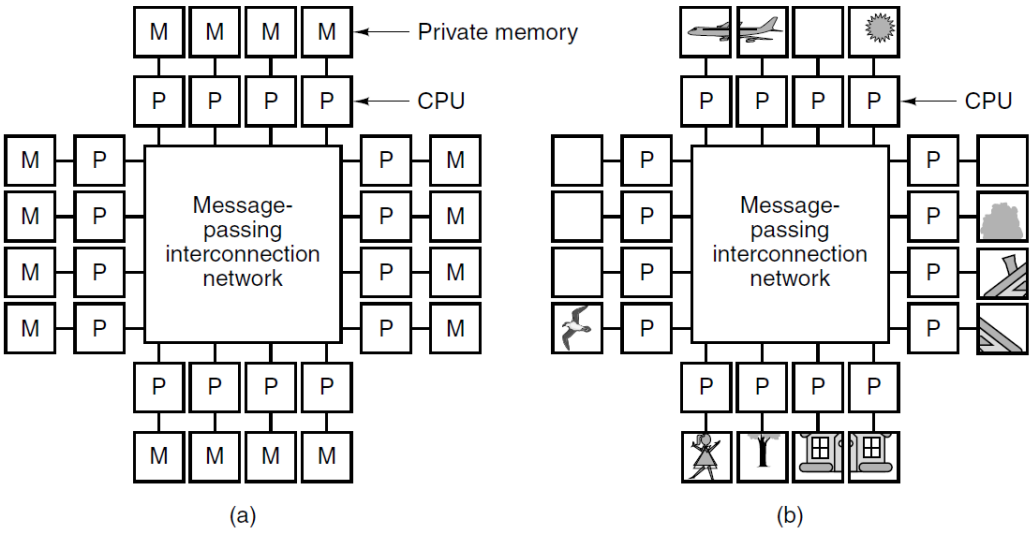
- ```
  #pragma omp parallel
    {
      #pragma omp single
        while (p != null) {
          #pragma omp task firstprivate(p)
            compute(p);
          p = p->next;
        }
    }
  ```

OpenMP Summary

- OpenMP attemps to make parallel programming *declarative*
- Still calls for *explicit* parallelism
- Supports both *explicit* and *implicit* work distribution
- Requires *compiler integraton* and a *runtime*
- Works only with C/C++ (and Fortran)
- Generalizes loop parallelism to *task-based parallelism*

Multicomputers — work distribution

- Each computer is given a part of the data in its own memory



- Calls for message passing — typically MPI

Message Passing Interface (MPI) — History

Background $\sim$ 1990

- Many vendors of *"supercomputers"* with similar software
- A group of 80 people (vendors and researchers) formed *MPI Forum*
- Goal: A uniform way of programming *distributed memory systems*
- June 1994: Version 1.0
- API and protocols for interaction (ca. 130 functions)

Status

- *De facto standard* for programming distributed memory systems
- Official bindings for: C, C++, FORTRAN
- Unofficial bindings for: .NET, Java, Python, ...
- OpenMPI is a very common, open source implementation
- Current major versions: 3.1 (2015), 4.1 (2021), 5.0 (2023) ...

MPI — Key Notions

Processes

- An MPI application consists of a set of communicating *(OS) processes*
- Processes are (usually) *single-threaded*

  — *Single Program Multiple Data (SPMD)*

Communicators

- A *communicator* is communication universe with a set of processes
- Each process within a communicator has a unique *rank* (0...)
- The full set of started processes belong to `MPI_COMM_WORLD`
- Processes within a communicator may communicate using:
  - Point-to-point communication
  - Collective communication

MPI — Point-to-Point Communication

Operations

- In s: `MPI_Send`($bufp_s, count_s, type_s, dest, tag_s, comm_s$)
- In r: `MPI_Recv`($bufp_r, count_r, type_r, source, tag_r, comm_r, statusp$)
- Type codes: `MPI_INT`, `MPI_CHAR`, `MPI_BYTE`, ...
- *source* and tag_r may be *wildcards*: `MPI_ANY_TAG`, `MPI_ANY_SOURCE` ($*$)

Communication

- If operations *match*: $comm_r = comm_s, dest = r,$
  $source = s$ **or** $source = *,$
  $tag_r = tag_s,$ **or** $tag_r = *,$
  $type_s = type_r$
- Then: $count_s$ elements are copied from $bufp_s$ to $bufp_r$.
- If $count_s > count_r$ an *error* occurs.
- From *statusp*, the source s, tag_s, and $count_s$ may be retrieved.

MPI — Point-to-Point Semantics

General

- Communication is *reliable*
- Communication between a given sender and a given receiver is *ordered*
- *No fairness* guarantee — starvation may occur

Standard mode

- MPI may buffer the message (or not).
- Both `MPI_Send` and `MPI_Recv` are *blocking*
- When `MPI_Send` returns, its buffer may be reused

Alternatives

- Other modes: Synchronous, buffered, non-blocking, ...
- Many auxiliary operations, e.g. `MPI_Probe()`

Example: Hello MPI World

```
int main(void) {
   char greeting[MAX_STRING]; /* String storing message */
   int comm_sz; /* Number of processes */
   int my_rank; /* My process rank */

   MPI_Init(NULL, NULL);

   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

     ⋮

   MPI_Finalize();
}
```

Example: Hello MPI World

```
if (my_rank != 0) {
   sprintf(greeting, "Greetings␣from␣process␣%d␣of␣%d!",
         my_rank, comm_sz);
   /* Send message to process 0 */
   MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
         MPI_COMM_WORLD);
} else {
   printf("Greetings␣from␣process␣%d␣of␣%d!\n", my_rank, comm_sz);
   for (int q = 1; q < comm_sz; q++) {
      /* Receive message from process q */
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
         0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("%s\n", greeting);
   }
}
```

Collective Communication

Basis

- Barrier:

⋮	⋮	⋮	⋮
MPI_Barrier(*c*);	*MPI_Barrier*(*c*);	*MPI_Barrier*(*c*);	*MPI_Barrier*(*c*);
⋮	⋮	⋮	⋮

- To be called by **all** in communicator *c*

Extensions

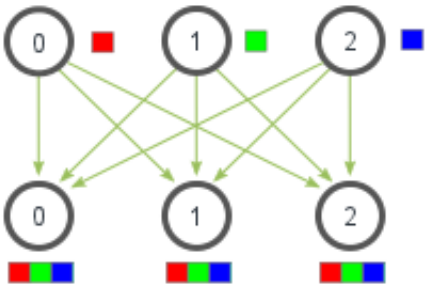
- Broadcast
- Scattering
- Gathering
- Reduction

Collective Communication — Allgather

Operation

- `MPI_Allgather`($bufp_s, count_s, type_s, bufp_r, count_r, type_r, comm$)
- To be called by **all** in communicator — senders/receivers
- Only $bufp_r$ acts as output (for all)— all others as input
- Must all agree on $type_s = type_r, count_s = count_r, comm$

Effect

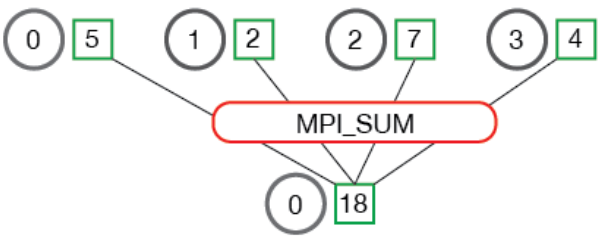


Collective Communication — Gather + processing = Reduce

Operation

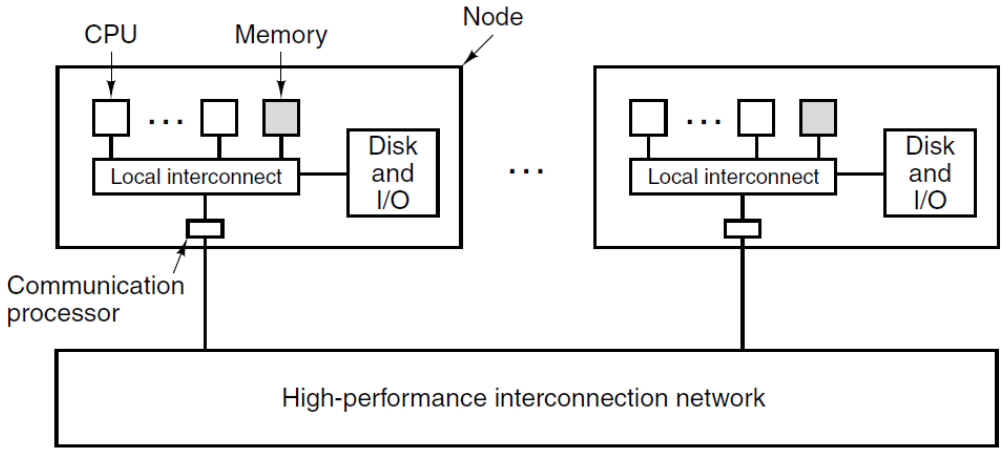
- `MPI_Reduce`($bufp_s, bufp_r, count, type, op, dest, comm$)
- To be called by **all** in communicator — senders and receiver
- Only $bufp_r$ acts as output (for $dest$)— all others as input
- Op codes: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`, ...
- Must all agree on $type, count, op, dest, comm$

Effect



MPI on Hybrid Architectures

- Cluster



- DTU HPC clusters: Eg. 30 nodes with 12+12 cores

Parallel Programming Summary

- Processors are programmed *indirectly* through *threads* or *OS processes*
- May be done through a *bag-of-task* (*thread pool*)
- The number of worker threads/processes should match the number of processors
- *Speedup* and *efficiency* are important measures
- There are limits to what can be achieved (*Amdahl's Law*)