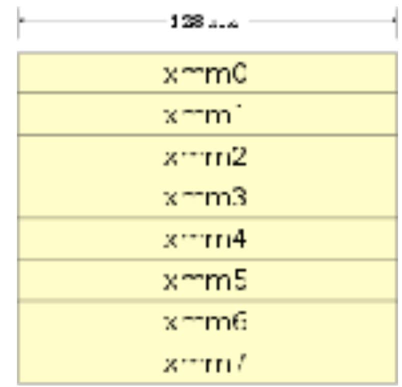
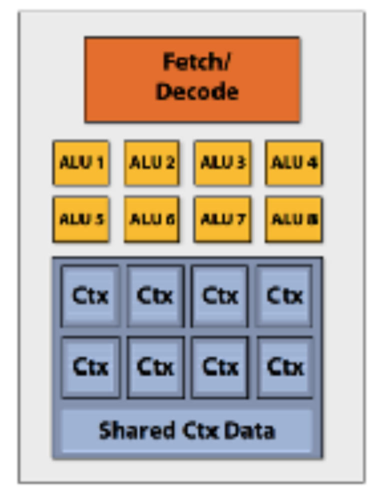
Lect. 2: Types of Parallelism

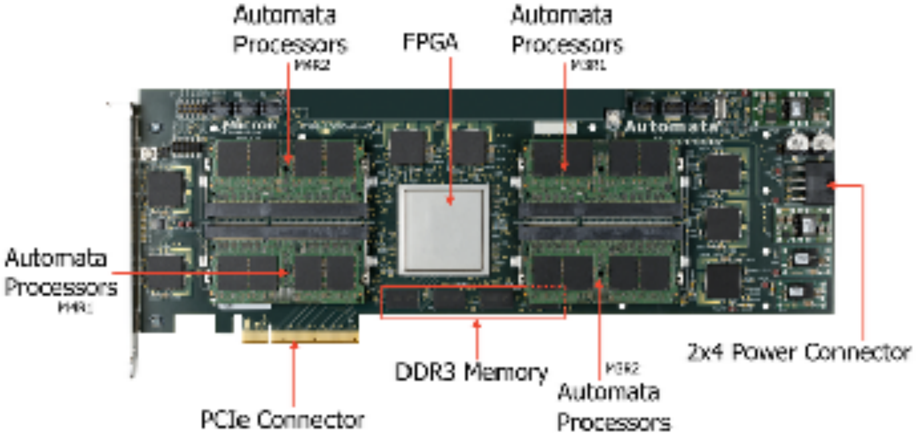
- Parallelism in Hardware (Uniprocessor)
  - Parallelism in a Uniprocessor
    - Pipelining
    - Superscalar, VLIW etc.
  - SIMD instructions, Vector processors, GPUs
  - Multiprocessor
    - Symmetric shared-memory multiprocessors
    - Distributed-memory multiprocessors
    - Chip-multiprocessors a.k.a. Multi-cores
  - Multicomputers a.k.a. clusters
- Parallelism in Software
  - Instruction level parallelism
  - Task-level parallelism
  - Data parallelism
  - Transaction level parallelism





Taxonomy of Parallel Computers

- According to instruction and data streams (Flynn):
  - Single instruction single data (**SISD**): this is the standard uniprocessor
  - Single instruction, multiple data streams (**SIMD**):
    - Same instruction is executed in all processors with different data
    - E.g., Vector processors, SIMD instructions, GPUs
  - Multiple instruction, single data streams (**MISD**):
    - Different instructions on the same data
    - Fault-tolerant computers, Near memory computing (Micron Automata processor).

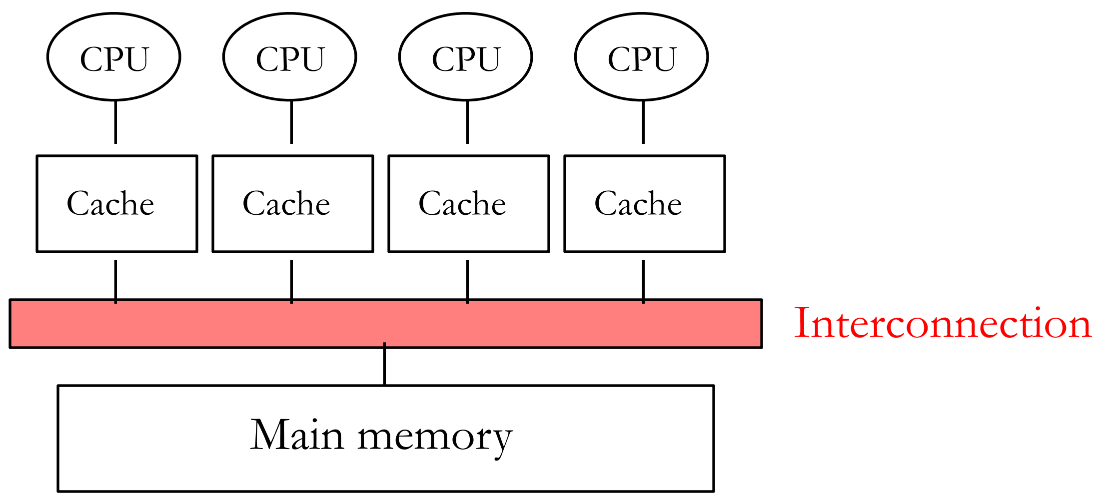


Taxonomy of Parallel Computers

- According to instruction and data streams (Flynn):
  - Single instruction single data (**SISD**): this is the standard uniprocessor
  - Single instruction, multiple data streams (**SIMD**):
    - Same instruction is executed in all processors with different data
    - E.g., Vector processors, SIMD instructions, GPUs
  - Multiple instruction, single data streams (**MISD**):
    - Different instructions on the same data
    - Fault-tolerant computers, Near memory computing (Micron Automata processor).
  - Multiple instruction, multiple data streams (**MIMD**): the "common" multiprocessor
    - Each processor uses it own data and executes its own program
    - Most flexible approach
    - Easier/cheaper to build by putting together "off-the-shelf" processors

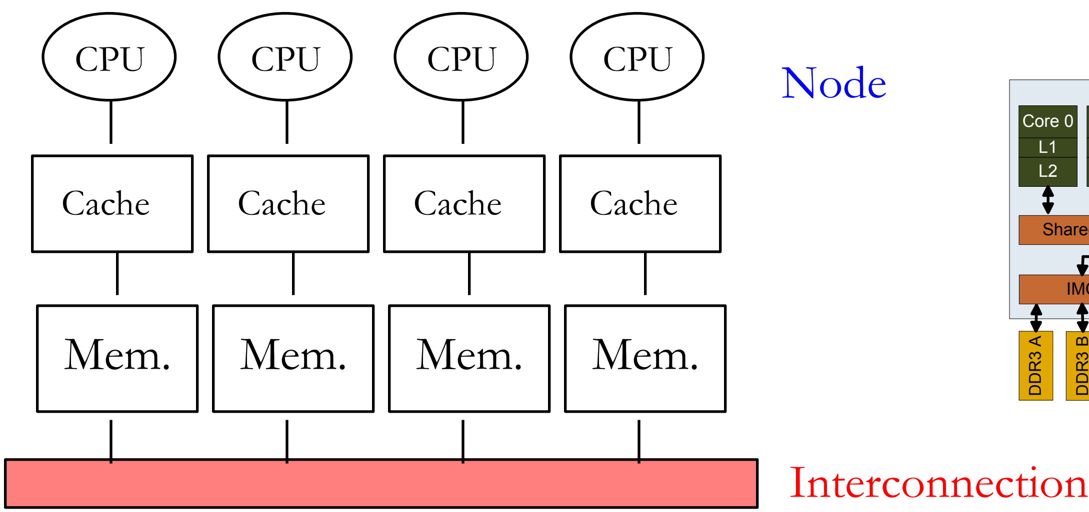
Taxonomy of Parallel Computers

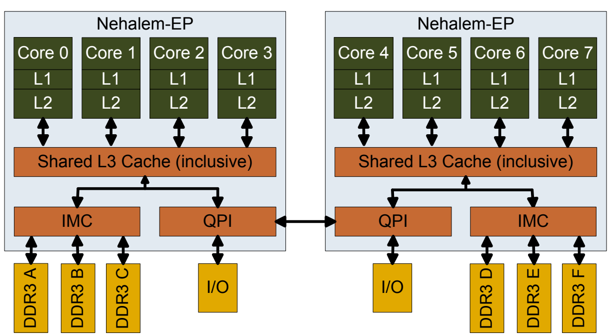
- According to physical organization of processors and memory:
  - Physically centralized memory, uniform memory access (UMA)
    - All memory is allocated at same distance from all processors
    - Also called symmetric multiprocessors (SMP)
    - Memory bandwidth is fixed and must accommodate all processors → does not scale to large number of processors
    - Used in CMPs today (single-socket ones)



Taxonomy of Parallel Computers

- According to physical organization of processors and memory:
  - Physically distributed memory, non-uniform memory access (NUMA)
    - A portion of memory is allocated with each processor (node)
    - Accessing local memory is much faster than remote memory
    - If most accesses are to local memory than overall memory bandwidth increases linearly with the number of processors
    - Used in multi-socket CMPs E.g Intel Nehalem





Taxonomy of Parallel Computers

- According to memory communication model
  - Shared address or shared memory
    - Processes in different processors can use the same virtual address space
    - Any processor can directly access memory in another processor node
    - Communication is done through shared memory variables
    - Explicit synchronization with locks and critical sections
    - Arguably easier to program??
  - Distributed address or message passing
    - Processes in different processors use different virtual address spaces
    - Each processor can only directly access memory in its own node
    - Communication is done through explicit messages
    - Synchronization is implicit in the messages
    - Arguably harder to program??
    - Some standard message passing libraries (e.g., MPI)

Shared Memory vs. Message Passing

- Shared memory

Producer (p1)

```
flag = 0;
…
a = 10;
flag = 1;
```

Consumer (p2)

```
flag = 0;
…
while (!flag) {}
x = a * y;
```

- Message passing

Producer (p1)

```
…
a = 10;
send(p2, a, label);
```

Consumer (p2)

```
…
receive(p1, b, label);
x = b * y;
```

Types of Parallelism in Applications

- Instruction-level parallelism (ILP)
  - Multiple instructions from the same instruction stream can be executed concurrently
  - Generated and managed by hardware (superscalar) or by compiler (VLIW)
  - Limited in practice by data and control dependences

- Thread-level or task-level parallelism (TLP)
  - Multiple threads or instruction sequences from the same application can be executed concurrently
  - Generated by compiler/user and managed by compiler and hardware
  - Limited in practice by communication/synchronization overheads and by algorithm characteristics

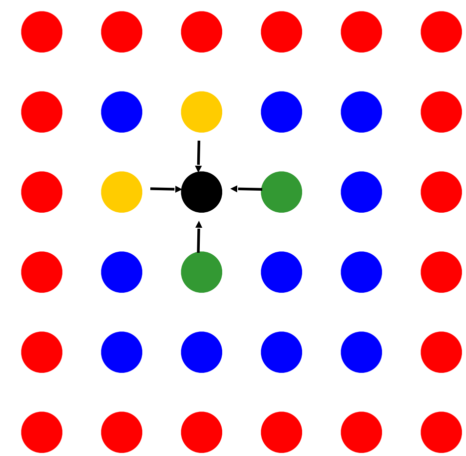
Types of Parallelism in Applications

- Data-level parallelism (DLP)
  - Instructions from a single stream operate concurrently on several data
  - Limited by non-regular data manipulation patterns and by memory bandwidth

- Transaction-level parallelism
  - Multiple threads/processes from different transactions can be executed concurrently
  - Limited by concurrency overheads

Example: Equation Solver Kernel

- The problem:
  - Operate on a (n+2)x(n+2) matrix
  - Points on the rim have fixed value
  - Inner points are updated as:

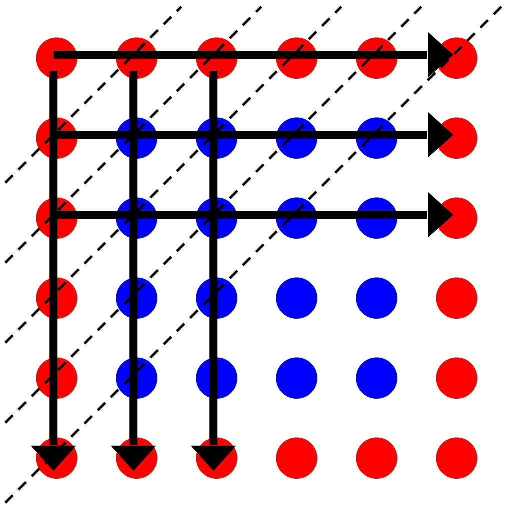


$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

  - Updates are in-place, so top and left are new values and bottom and right are old ones
  - Updates occur at multiple sweeps
  - Keep difference between old and new values and stop when difference for all points is small enough

Example: Equation Solver Kernel

- Dependences:
  - Computing the new value of a given point requires the new value of the point directly above and to the left
  - By transitivity, it requires all points in the sub-matrix in the upper-left corner
  - Points along the top-right to bottom-left diagonals can be computed independently



Example: Equation Solver Kernel

- ILP version (from sequential code):
  - Some machine instructions from each j iteration can occur in parallel
  - Branch prediction allows overlap of multiple iterations of j loop
  - Some of the instructions from multiple j iterations can occur in parallel

```
while (!done) {
  diff = 0;
  for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
      temp = A[i,j];
      A[i,j] = 0.2*(A[i,j]+A[i,j-1]+A[i-1,j] +
                A[i,j+1]+A[i+1,j]);
      diff += abs(A[i,j] – temp);
    }
  }
  if (diff/(n*n) < TOL) done=1;
}
```

Example: Equation Solver Kernel

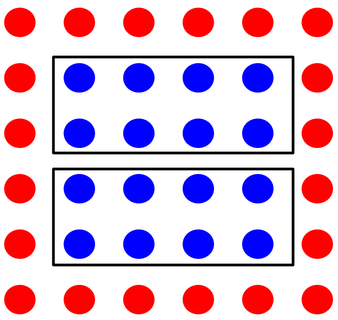
- TLP version (shared-memory):

```
int mymin = 1+(pid * n/P);
int mymax = mymin + n/P – 1;

while (!done) {
  diff = 0; mydiff = 0;
  for (i=mymin; i<=mymax; i++) {
    for (j=1; j<=n; j++) {
      temp = A[i,j];
      A[i,j] = 0.2*(A[i,j]+A[i,j-1]+A[i-1,j] +
                  A[i,j+1]+A[i+1,j]);
      mydiff += abs(A[i,j] – temp);
    }
  }
  lock(diff_lock); diff += mydiff; unlock(diff_lock);
  barrier(bar, P);
  if (diff/(n*n) < TOL) done=1;
  barrier(bar, P);
}
```

Example: Equation Solver Kernel

- TLP version (shared-memory) (for 2 processors):
  - Each processor gets a chunk of rows
    - E.g., processor 0 gets: mymin=1 and mymax=2 and processor 1 gets: mymin=3 and mymax=4



```
int mymin = 1+(pid * n/P);
int mymax = mymin + n/P – 1;
```

```
while (!done) {
  diff = 0; mydiff = 0;
  for (i=mymin; i<=mymax; i++) {
    for (j=1; j<=n; j++) {
      temp = A[i,j];
      A[i,j] = 0.2*(A[i,j]+A[i,j-1]+A[i-1,j] +
               A[i,j+1]+A[i+1,j]);
      mydiff += abs(A[i,j] – temp);
    }
  ...
```

Example: Equation Solver Kernel

- TLP version (shared-memory):
  - All processors can access freely the same data structure A
  - Access to diff, however, must be in turns
  - All processors update together their own done variable

```
...
  for (i=mymin; i<=mymax; i++) {
    for (j=1; j<=n; j++) {
      temp = A[i,j];
      A[i,j] = 0.2*(A[i,j]+A[i,j-1]+A[i-1,j] +
                A[i,j+1]+A[i+1,j]);
      mydiff += abs(A[i,j] – temp);
    }
  }
  lock(diff_lock); diff += mydiff; unlock(diff_lock);
  barrier(bar, P);
  if (diff/(n*n) < TOL) done=1;
  barrier(bar, P);
}
```

Types of Speedups and Scaling

- Scalability: adding x times more resources to the machine yields close to x times better "performance"
  - Usually resources are processors (but can also be memory size or interconnect bandwidth)
  - Usually means that with x times more processors we can get $\sim x$ times speedup for the same problem
  - In other words: How does efficiency (see Lecture 1) hold as the number of processors increases?

- In reality we have different scalability models:
  - Problem constrained
  - Time constrained

- Most appropriate scalability model depends on the user interests

Types of Speedups and Scaling

- Problem constrained (PC) scaling:
  - Problem size is kept fixed
  - Wall-clock execution time reduction is the goal
  - Number of processors and memory size are increased
  - "Speedup" is then defined as:

$$S_{PC} = \frac{\text{Time(1 processor)}}{\text{Time(p processors)}}$$

  - Example: Weather simulation that does not complete in reasonable time

Types of Speedups and Scaling

- Time constrained (TC) scaling:
  - Maximum allowable execution time is kept fixed
  - Problem size increase is the goal
  - Number of processors and memory size are increased
  - "Speedup" is then defined as:

$$S_{TC} = \frac{\text{Work(p processors)}}{\text{Work(1 processor)}}$$

  - Example: weather simulation with refined grid