CS1645 and CS2045: Introduction to High Performance Computing

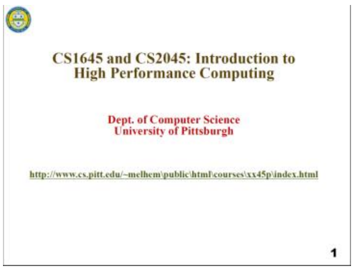
Dept. of Computer Science
University of Pittsburgh

http://www.cs.pitt.edu/~melhem/courses/xx45p/index.html

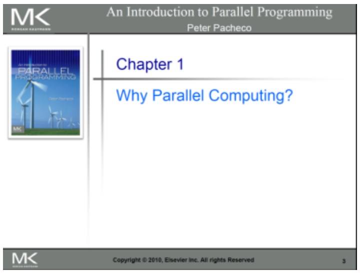
1

Disclaimer

In my presentations, I will use two types of slides:



My own slides



Slides provided by the publisher (may be modified)

2

Evolution of parallel hardware

- I/O channels and DMA (direct memory access)
- Pipelined functional units

- Vector processors (ILLIAV IV was built in 1974)
- Multiprocessors (cm* and c.mmp were built in the 70's)
- Instruction pipelining and superscalers

- 80's and 90's supercomputers - Massively Parallel Processors (Connection machine, Cray T3D, Intel Hypercube, …)
- Symmetric Multiprocessors (sequent, Encore, Pyramid, quad-cores, ….)

- Distributed computing (Clusters, server farms, grids, the cloud)

- Multi-core processors and Chip Multiprocessors
- Graphics Processor Units (GPU) as accelerators

Parallel processing used to be a luxury --- now it is a necessity

Exploring Instruction Level Parallelism (ILP) through Pipelining

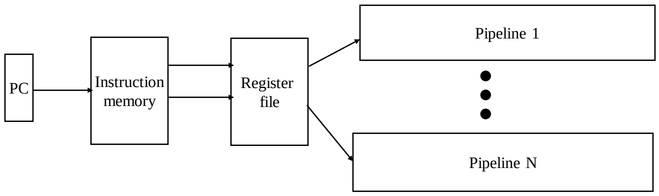
- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.

- Pipelining, however, has several limitations.
  - The speed of a pipeline is limited by the slowest stage.
  - Data and structural dependencies
  - Control dependencies - in typical programs, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.

- One simple way of alleviating these limitations is to use multiple pipelines.

Superscalar architecture

- Use multiple pipelines and start more than one instruction in the same cycle



- Scheduling of instructions is determined by a number of factors:
  - Resource, branch and data dependencies
  - The scheduler, a piece of hardware, looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently
  - In-order or out-of-order execution
  - The complexity of the hardware is a constraint on superscalar processors.

Industry trend

- Up to 2002, performance increases have been due to increasing VLSI density
  - More complex logic
  - Faster clocks

- Faster clocks → increased power/energy consumption
  → increased heat dissipation
  → increased cooling

- Increased heat → lower reliability

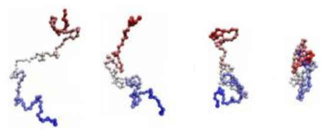
Solution: Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.

Why do we need ever-increasing performance

- Computational power is increasing, but so are our computation problems and needs.
- Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.
- More complex problems are still waiting to be solved.

Example problems

Climate modeling



Protein folding

Drug discovery

Energy research

Data analysis

Why we need to write parallel programs

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.

- What you really want is for it to run faster.

MK

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

MK

More problems

- Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
- However, it's likely that the result will be a very inefficient program.
- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

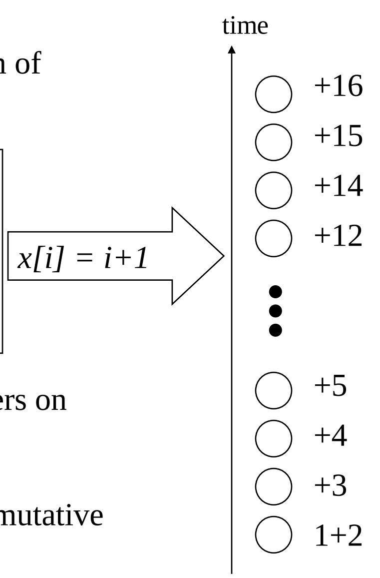
MORGAN KAUFMANN

11

Thinking parallel: The sum algorithm

- The following computes the sum of *x[0]+…+x[15]* serially:

```
sum = 0
For (i = 0 ; i < 16  ;  i++)
  {
     sum +=  x[i]
  }
```

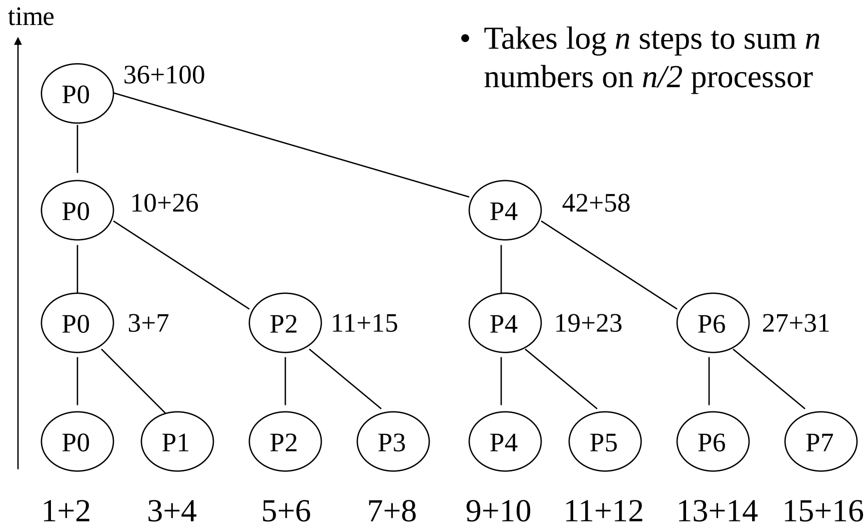


- Takes *n-1* steps to sum *n* numbers on one processor
- Applies to associative and commutative operations (+, *, min, max, …)

12

Parallel sum algorithm (on 8 processors)

- Takes log n steps to sum n numbers on $n/2$ processor



13

Speedup and efficiency (page 58)

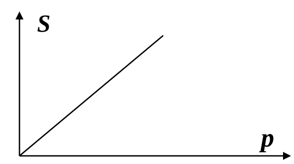
- For a given problem A, of size n, let $T_p(n)$ be the execution time on p processors, and $T_s(n)$ be the execution time on one processor. Then,

  Speedup $S_p(n) = T_s(n) / T_p(n)$

  Efficiency $E_p(n) = S_p(n) / p$

  Speedup is between 0 and p, and efficiency is between 0 and 1.

- Linear Speedup means that S is linear with p (linearly scalable system)
- If speedup is independent of n, then the algorithm is said to be perfectly scalable.

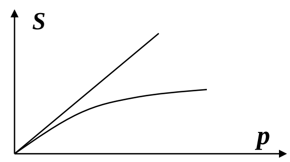


14

Speedup and efficiency

Minsky's conjecture:

Speedup is logarithmic in p



For parallel sum, $S_{n/2}(n) = (n\text{-}1) / \log n \approx n / \log n$

$E_{n/2}(n) = (n\text{-}1) / n \log n \approx 1 / \log n$

Example: sum 1024 numbers on 512 processors:

Speedup ≈ 1024 / 10 = 10.24

Efficiency ≈ 10.24 / 512 = 2%

15

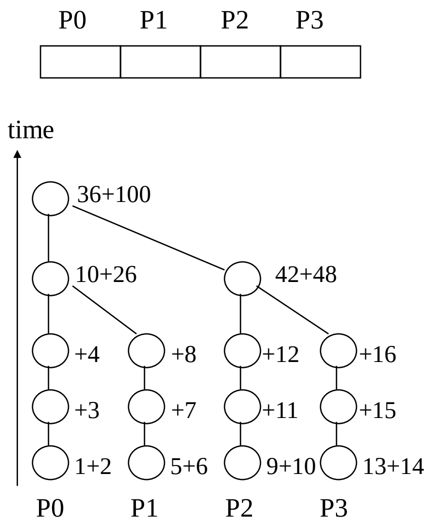
An efficient parallel sum (on 4 processors)

- Divide the array to be summed into 4 parts and assign one part to each processor

- Takes *5* steps to sum *16* numbers on 4 processor
- Takes 255+2 steps to sum 1024 numbers on 4 processors
- Speedup = 1023/257 = 3.9

- How long does it take to sum *n* numbers on *p* processors?

- What is the speedup?



16

Example

- Problem: compute and add n values
- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

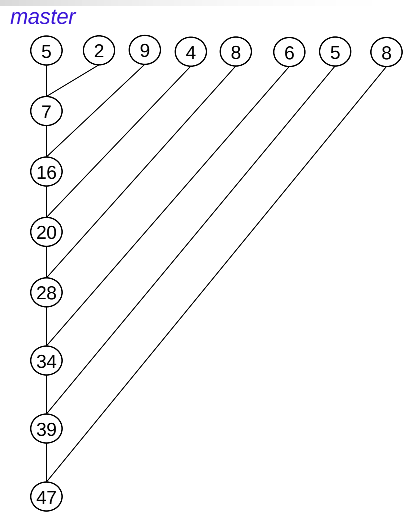
Example (cont.)

- After each core completes execution of the code, is a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.
- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

Example (cont.)

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Uses the linear algorithm.
Less efficient than the tree structured algorithm described earlier



The tree structured algorithm

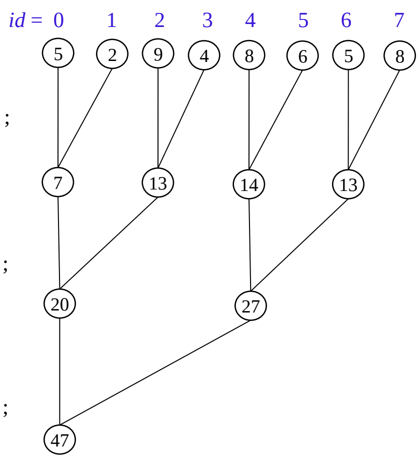
log n steps

n = 8

If (id mod 2 == 1)
 send *sum* to core id-1 ;
 else { receive *value* from core id+1 ;
 sum = *sum* + *value* };

If (id mod 4 == 2)
 send *sum* to core id-2 ;
 elseif (id mod 4 == 0)
 { receive *value* from core id+2 ;
 sum = *sum* + *value* };

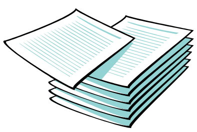
If (id mod 8 == 4)
 send *sum* to core id-4 ;
 elseif (id mod 8 == 0)
 { receive *value* from core id+4 ;
 sum = *sum* + *value* };



How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out solving the problem among the cores.
- Data parallelism
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it's part of the data.

Professor P



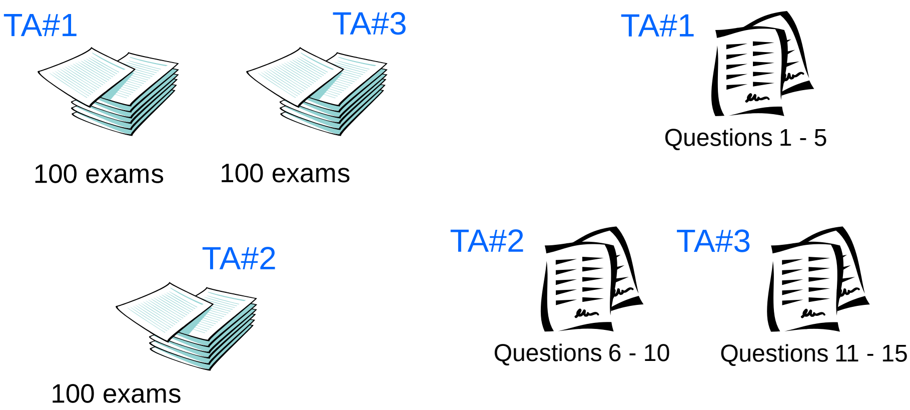
15 questions

300 exams

Division of work

Data parallelism

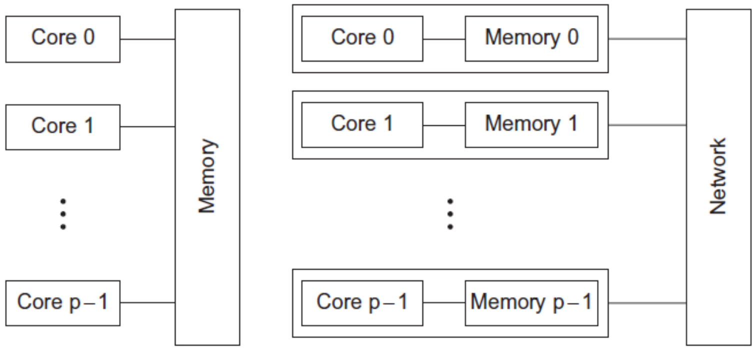
Task parallelism



Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

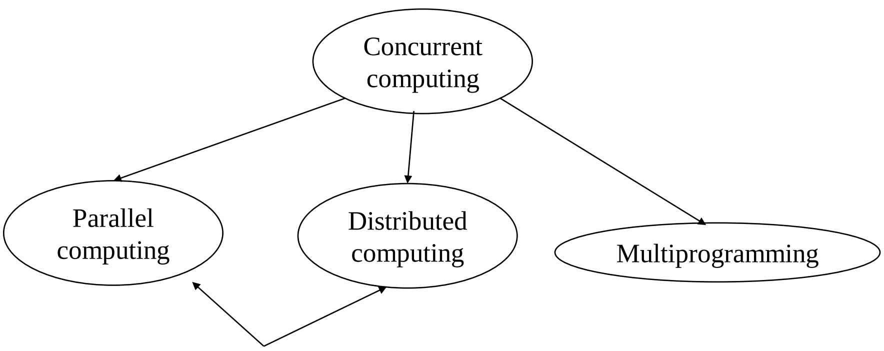
Type of parallel systems



(a) (b)

Shared-memory Distributed-memory

Terminology



The difference is how tight is the synchronizations and how fast is the communication?

Concluding remarks

- The laws of physics have forced multicores and parallel computing.
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.
- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually complex and therefore, require sound program techniques and development.