



LUND
UNIVERSITY



HPC2N, UmU

Parallel programming Shared and distributed Memory

Joachim Hein

LUNARC & Centre of Mathematical Sciences

Lund University

Pedro Ojeda May

HPC2N

Umeå University





LUND
UNIVERSITY

INTRODUCTION TO PARALLEL COMPUTING



Why parallel programming?

- Improve computational speed
 - Get faster time to solution
 - Larger, hopefully more realistic problem
- Divide problem into subtasks
 - Assign subtasks to different CPUs → parallel computing
- Parallel computing is presently the **only** game in town to get more performance
 - CPUs do not get faster (stuck around 3 GHz for years)



Example

- Calculate large sum:

$$\sum_{n=1}^{40000} a_n$$

- Split into four sums, use a different processor for each:

$$\sum_{n=1}^{10000} a_n$$

$$\sum_{n=10001}^{20000} a_n$$

$$\sum_{n=20001}^{30000} a_n$$

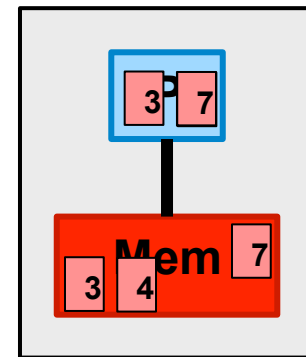
$$\sum_{n=30001}^{40000} a_n$$

- In the end: four partial results and need to make into one!



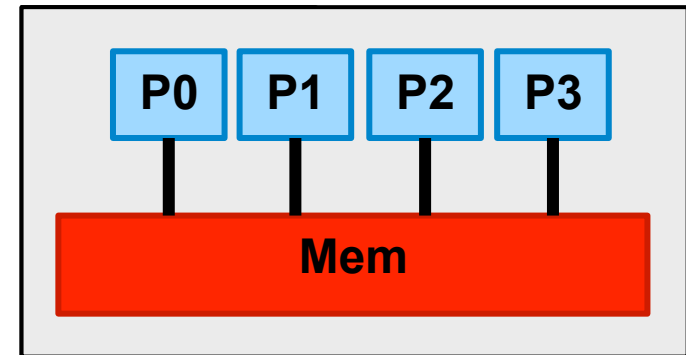
Serial programming

- We want to do a long calculation
- Pencil & Paper to slow ☹
- Take a computer
 - Write a program in C, Fortran, ...
 - Run the program
- What happens when the program runs?
 - Processor reads data from memory
 - Processes these data
 - Writes result back to memory
 - Write final result to disk
- What can we do if that is still too slow ???



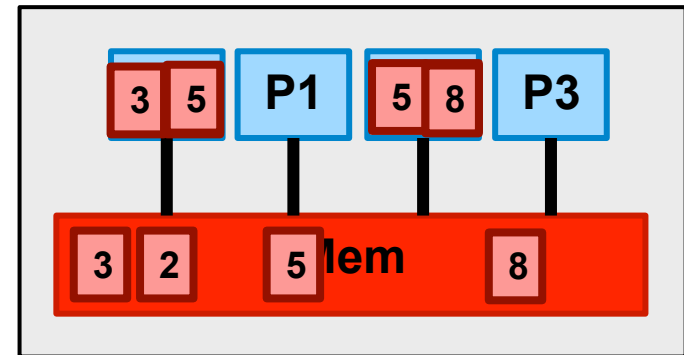
Parallel programming

- Take more than one processor
- Partition the entire calculation into independent parts
- Assign one or more of these parts to each processor
- Should be faster now!
- But normally the parts are **not** fully independent
- Dealing with these data dependencies is a key challenge in parallel computing



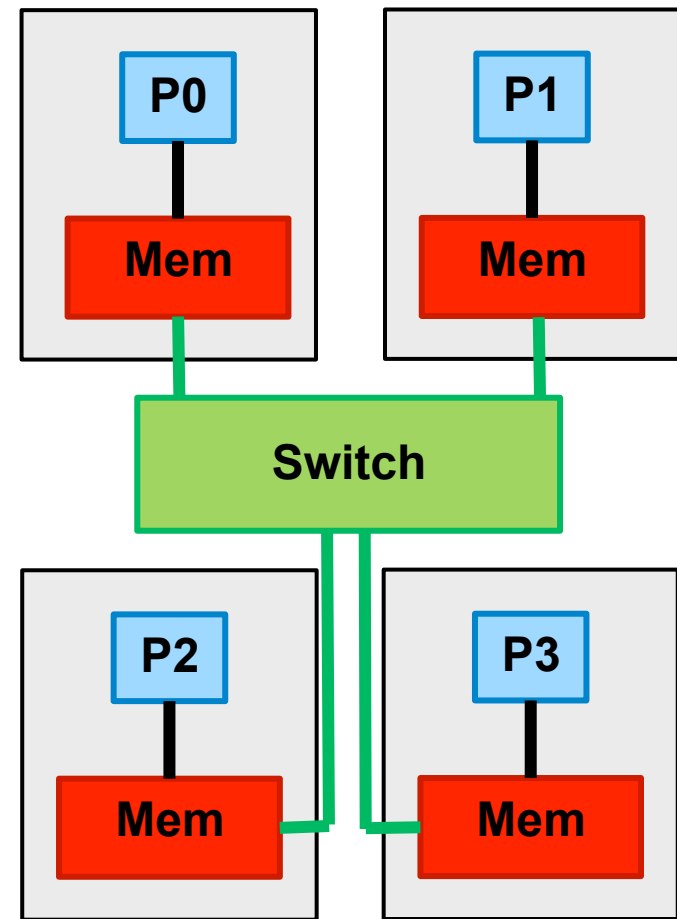
Shared Memory Architecture

- Several processing elements manipulate the same, shared memory space
- Easy to move data between processors
 - Write result to shared memory
 - Read on different processor
- Care is needed regarding order:
 - P0 needs to write before P2 can read
- Read/write to shared memory has typically higher cost than manipulating registers/cache



Distributed memory machine

- Several independent computers (typically referred to as: nodes)
- With interconnect network
- Separate program, called *task*, on each processor
- Each has its private data
- Typically explicit message passing
- Processes:
 - Send data into the network
 - Receive data from the network
 - They can not just use it



Double sided point-to-point communication

- Most basic form of communication in message passing:

Program on P0:

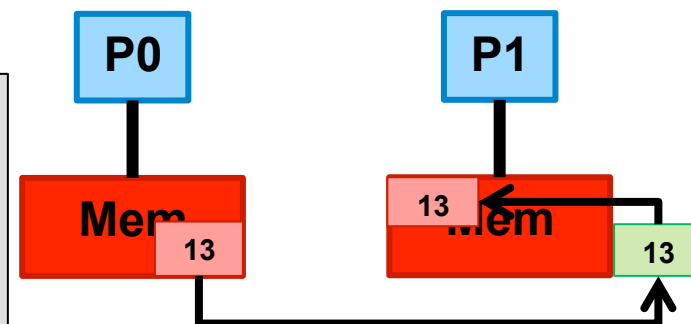
```
a = 13
```

```
send(a, 1)
```

Program on P1:

```
recv(b, 0)
```

```
c = c + b
```



Discussion

MESSAGE PASSING SHARED MEMORY



Message passing

- Programmer explicitly distributes and exchanges data
- Data transfer by “**value**” – explicit data copy
- Entire program typically needs updating in “one go”
- Suitable for large variety of architectures
 - Includes shared memory
 - Required for largest super-computers
- Work by programmer typically benefits performance
 - Good memory locality (Cache & CC-NUMA)
- Access to vast amounts of memory (TB of RAM)
- Typical problems encountered:
 - Segmentation faults
 - Dead lock



Shared memory programming

- Typically less demanding on the programmer
- Possible to work “incremental”
- Allows for more algorithms to be implemented
 - Replicated data situations
 - Unstructured data distributions
- Requires shared memory architecture
 - e.g.: Multicore, SMP, CC-Numa
- Limited in core count
- Typical problems encountered:
 - Data races → wrong results, often non-deterministic
 - Memory locality problems → bad performance



Hybrid: Message passing & shared memory

- All contemporary distributed memory machines use shared memory machines as basics building blocks (nodes):
 - Can ignore: use message passing between **and** within the nodes
 - Can use shared memory techniques inside the node and use message passing
- Danger of picking up the disadvantages of both:
 - Complexity of message passing programming
 - Performance penalty of shared memory programming
- Advantages in situations with:
 - Large memory required by each message passing task
 - Expensive/complex data redistributions
 - Can help with network congestion and latency



A simple performance model

AMDAHLS LAW OF PARALLEL COMPUTING



Parallel speed-up and efficiency

- Faster time to solution is the key aim of parallel computing

t_1 : time to solution using 1 processor

t_N : time to solution using N processors

- Parallel speed-up:

$$S(N) = \frac{t_1}{t_N}$$

- Parallel efficiency:

$$E(N) = \frac{t_1}{N \cdot t_N}$$

- Naïve performance expectation: $E(N) = 100\%$



Simple performance model: Amdahl's law for parallel programs

- Consider a program takes time t_1 in serial (on 1 processor)
- A fraction $f < 1$ can be parallelised efficiently on N processors
 - Time spend on the parallel part: $f t_1 / N$
 - Time spend on the serial, remaining part: $(1-f) t_1$
 - Total time on parallel code: $t_N = (1-f + f/N)t_1$

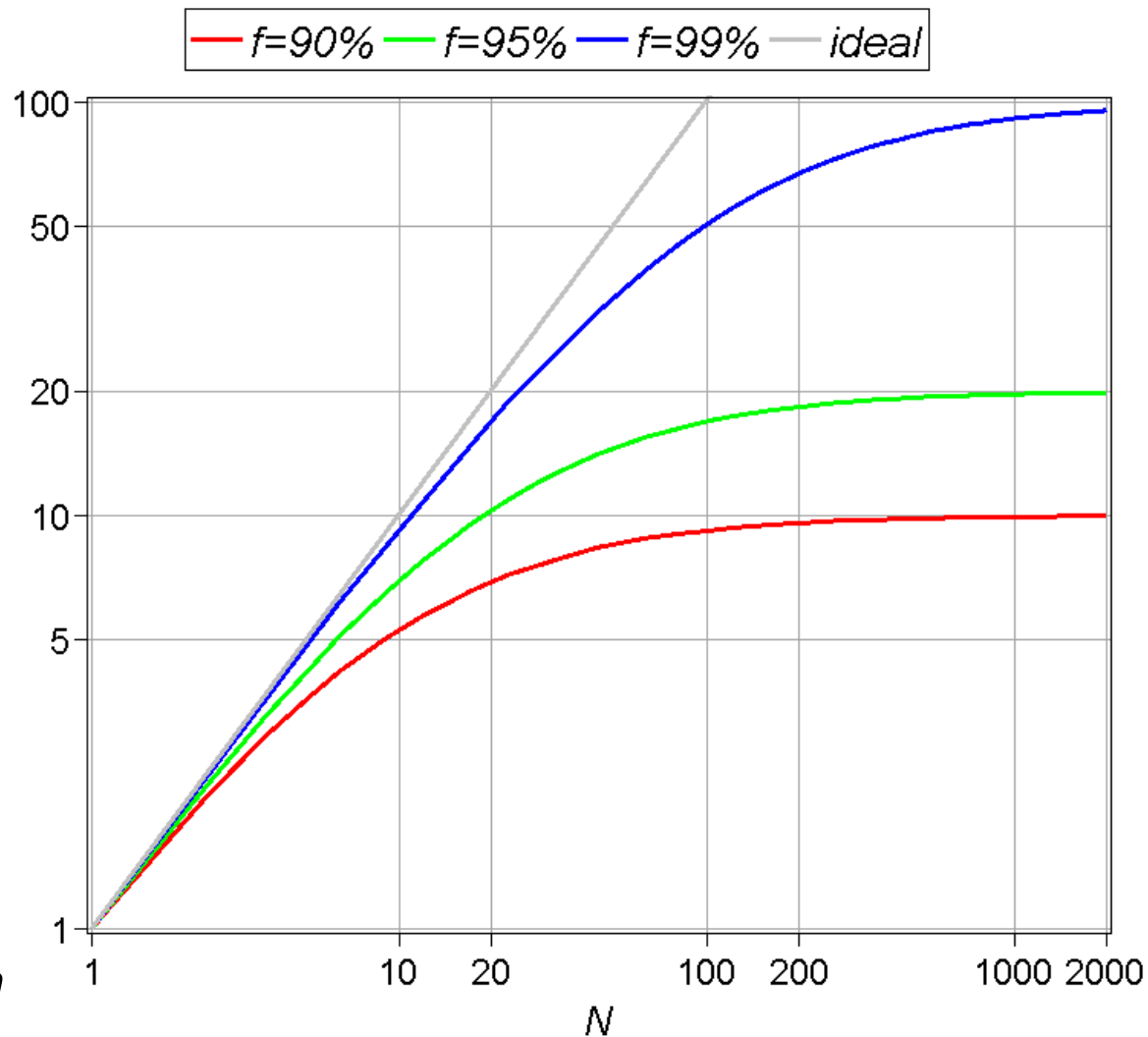
- Speed up:

$$\frac{t_1}{t_N} = \frac{1}{1 - f \left(1 - \frac{1}{N} \right)}$$



Amdahl's law

- Speed-up limited to:
 $1/(1 - f)$
- 50% of max speed-up:
 $N_{50} \approx 1/(1 - f)$
- Not much point in running beyond $\approx 2N_{50}$
- **Fall-out:** Serial parts in computational kernel can not be tolerated



Summary

- Parallel computing offers faster time to solution
- Basic ideas behind
 - Shared memory programming
 - Message passing programming
 - Hybrid programming
- Discussed the impact of remaining serial parts

