Parallel Computing
Why & How?

Xing Cai

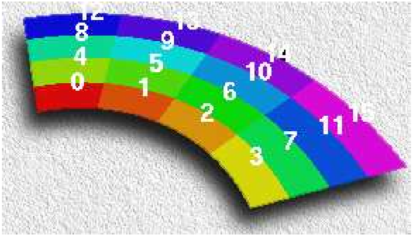
Simula Research Laboratory

Dept. of Informatics, University of Oslo

Winter School on Parallel Computing
Geilo
January 20–25, 2008

Outline

1. Motivation
2. Parallel hardware
3. Parallel programming
4. Important concepts

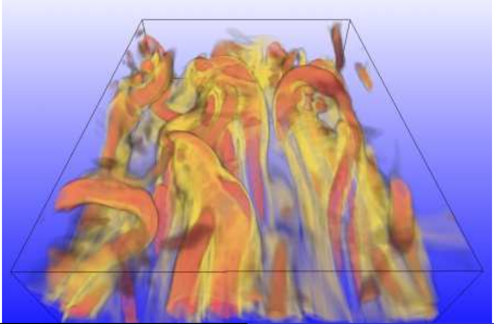


List of Topics

Background (1)

There's an everlasting pursuit of realism in computational sciences

- More sophisticated mathematical models
- Smaller Δx, Δy, Δz, Δt
- Longer computation time
- Larger memory requirement



Background (2)

Traditional serial computing (single processor) has limits

- Physical size of transistors
- Memory size and speed
- Instruction level parallelism is limited
- Power usage, heat problem

Moore´s law will not continue forever

Background (3)

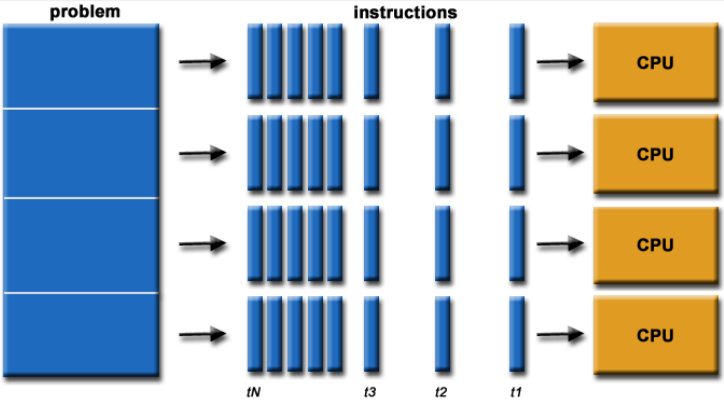
Parallel computing platforms are nowadays widely available

- Access to HPC centers
- Local Linux clusters
- Multiple CPUs and multi-core chips in laptops
- GPUs (graphics processing units)

What is parallel computing?

Parallel computing: simultaneous use of multiple processing units to solve one computational problem

Plot obtained from https://computing.llnl.gov/tutorials/parallel_comp/



Why parallel computing?

- Saving time
- Solving larger problems
  - access to more memory
  - better memory performance (when programmed correctly)
- Providing concurrency
- Saving cost

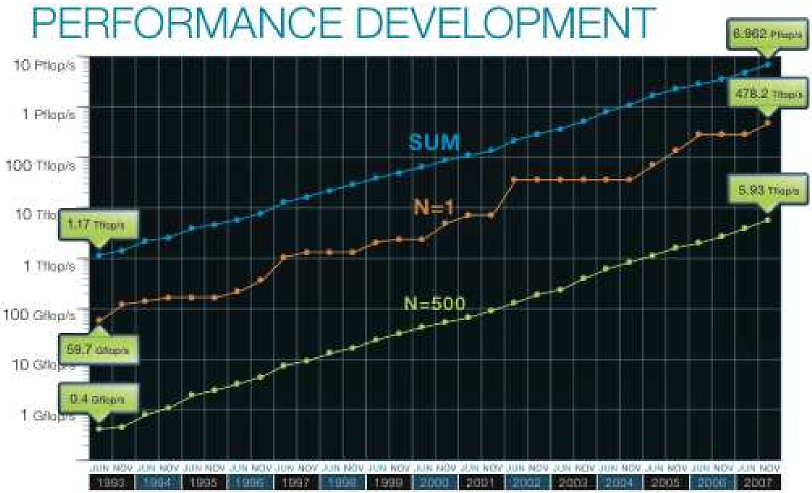
List of Topics

Today's most powerful computer

- IBM BlueGene/L system (Lawrence Livermore Lab)
- 106,496 dual-processor nodes (PowerPC 440 700 MHz)
- Peak performance: 596 teraFLOPS (596×10^{12})
- Linpack benchmark: 478.2 teraFLOPS

https://asc.llnl.gov/computing_resources/bluegenel/

Top500 list (Nov 2007)



http://www.top500.org

Flynn's taxonomy

Classification of computer architectures

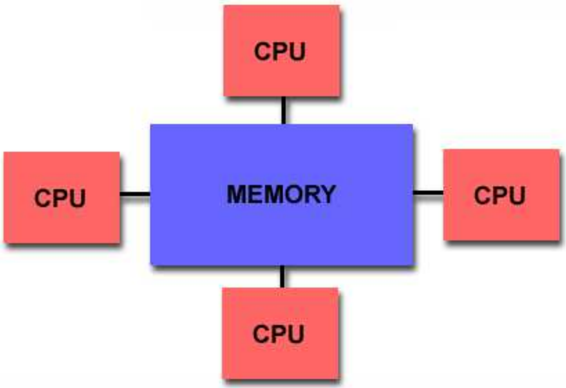
- SISD (single instruction, single data) – serial computers
- SIMD (single instruction, multiple data) – array computers, vector computers, GPUs
- MISD (mulitple instruction, single data) – systolic array (very rare)
- MIMD (mulitple instruction, multiple data) – mainstream parallel computers

Classification of parallel computers

Classification from the memory perspective

- Shared memory systems
  - A single global address space
  - SMP – (symmetric multiprocessing)
  - NUMA – (non-uniform memory access)
  - Multi-core processor – CMP (chip multi-processing)
- Distributed memory systems
  - Each node has its own physical memory
  - Massively parallel systems
  - Different types of clusters
- Hybrid distributed-shared memory systems

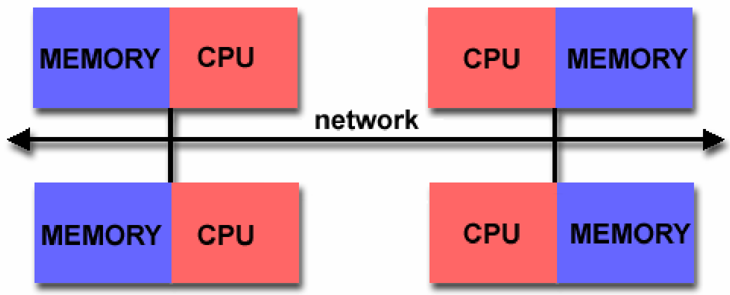
Shared memory



- Advantages: user-friendly
- Disadvantages: scalibility

Plot obtained from https://computing.llnl.gov/tutorials/parallel_comp/

Distributed memory



- Advantages: data locality (no interference), cost-effective
- Disadvantages: explicit communication, explicit decomposition of data or tasks

Plot obtained from https://computing.llnl.gov/tutorials/parallel_comp/

List of Topics

Parallel programming models

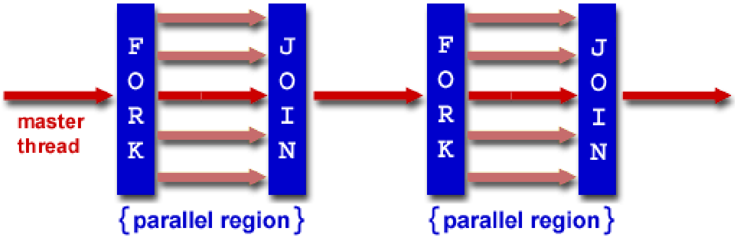
- Threads model
  - Easy to program (inserting a few OpenMP directives)
  - Parallelism "behind the scene" (little user control)
  - Difficult to scale to many CPUs (NUMA, cache coherence)
- Message passing model
  - Many programming details (MPI or PVM)
  - Better user control (data & work decomposition)
  - Larger systems and better performance
- Stream-based programming (for using GPUs)
- Some special parallel languages
  - Co-Array Fortran, Unified Parallel C, Titanium
- Hybrid parallel programming

OpenMP programming

OpenMP is a portable API for programming shared-memory computers

- Existence of multiple threads
- Use of compiler directives
- Fork-join model

Plot obtained from `https://computing.llnl.gov/tutorials/openMP/`



OpenMP example

Inner-product between two vectors: $c = \sum_{i=1}^{n} a(i)b(i)$

```
chunk = 10;
c = 0.0;

#pragma omp parallel for       \
  default(shared) private(i)   \
  schedule(static,chunk)       \
  reduction(+:c)

  for (i=0; i < n; i++)
    c = c + (a[i] * b[i]);
```

MPI programming

MPI (message passing interface) is a library standard

- Implementation(s) of MPI available on almost every major parallel platform
- Portability, good performance & functionality
- Each process has its local memory
- Explicit message passing enables information exchange and collaboration between processes

More info: `http://www-unix.mcs.anl.gov/mpi/`

MPI example

Inner-product between two vectors: $c = \sum_{i=1}^{n} a(i)b(i)$

```
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

my_start = n/num_procs*my_rank;
my_stop = n/num_procs*(my_rank+1);

my_c = 0.;
for (i=my_start; i<my_stop; i++)
  my_c = my_c + (a[i] * b[i]);

MPI_Allreduce (&my_c, &c, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
```

Standard way of parallelization

- Identify the parts of a serial code that have concurrency
- Be aware of inhibitors to parallelism (e.g. data dependency)
- When using OpenMP
  - insert directives to create parallel regions
- When using MPI
  - decide an explicit decomposition of tasks and/or data
  - insert MPI calls

Parallel programming requires a new way of thinking

List of Topics

Some useful concepts

- Cost model of sending a message

$$t_C(L) = \tau + \beta L$$

- Speed-up

$$S(P) = \frac{T(1)}{T(P)}$$

- Parallel efficiency

$$\eta(P) = \frac{S(P)}{P}$$

- Factors of parallel inefficiency
  - communication
  - load imbalance
  - additional calculations that are parallelization specific
  - synchronization
  - serial sections (Amdahl´s Law)

Amdahl´s Law

The upper limit of speedup

$$\frac{T(1)}{T(P)} \leq \frac{T(1)}{(f_s + \frac{f_p}{P})T(1)} = \frac{1}{f_s + \frac{1-f_s}{P}} < \frac{1}{f_s}$$

- f_s – fraction of code that is serial (not parallelizable)
- f_p – fraction of code parallelizable. $f_p = 1 - f_s$

Gustafson´s law

Things are normally not so bad as Amdahl´s law says

- Normalize the parallel execution time to be 1
- Scaled speed-up

$$S_s(P) = \frac{f_s + P f_p}{f_s + f_p} = f_s + P(1 - f_s) = P + (1 - P) f_s$$

- f_s is normally small
- f_s normally decreases as the problem size grows

Granularity

Granularity is a qualitative measure of the ratio of computation over communication

- Fine-grain parallelism
  - small amounts of computation between communication
  - load imbalance may be a less important issue
- Coarse-grain parallelism
  - large amounts of computation between communication
  - high ratio of computation over communication

Objective: Design coarse-grain parallel algorithms, if possible

Summary

- We´re already at the age of parallel computing
- Parallel computing relies on parallel hardware
- Parallel computing needs parallel software
- So parallel programming is very important
  - new way of thinking
  - identification of parallelism
  - design of parallel algorithm
  - implementation can be a challenge