MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 39:

Parallelization: Introduction

Premise

Basic premise for these lectures:

The problem we want to solve is large.

It takes too much time and/or
too much memory.

We have exhausted algorithmic improvements.

Consequence: We need to make more efficient use of the available hardware.

Background

Observations:

- Since ~2005, processor clock rates no longer increase
- Operations can also not be made much faster any more

- To keep making processors more powerful, makers put
  – multiple processors into each machine
  – multiple “cores” onto single processors
  – multiple “hardware threads” into each core
- To allow bigger computations, supercomputers consist of “clusters” of machines

Consequence: Software today needs to be parallel to be efficient.
(Whether we like that or not.)

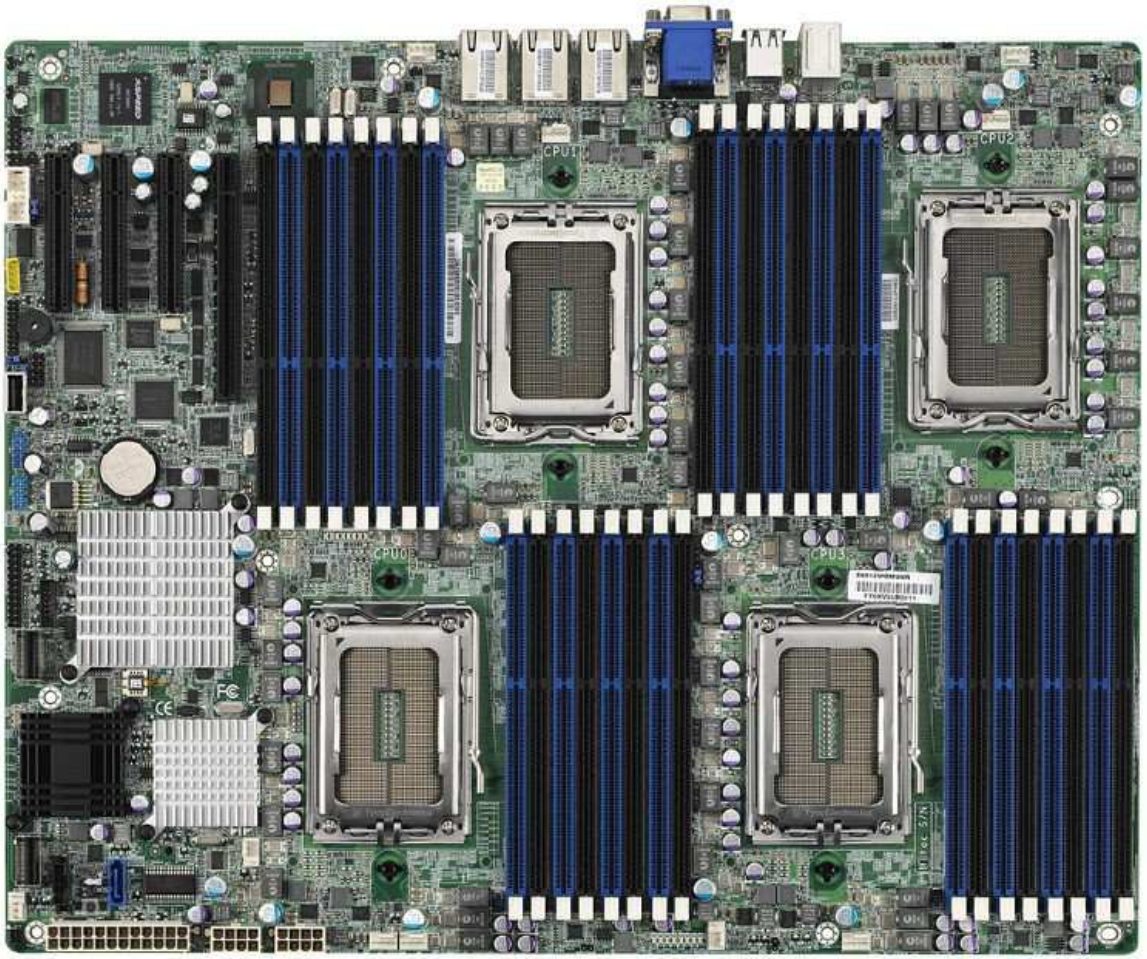
Hardware background

Hardware jargon: “Clusters” of machines

Credit: Lawrence Livermore National Lab

Hardware background

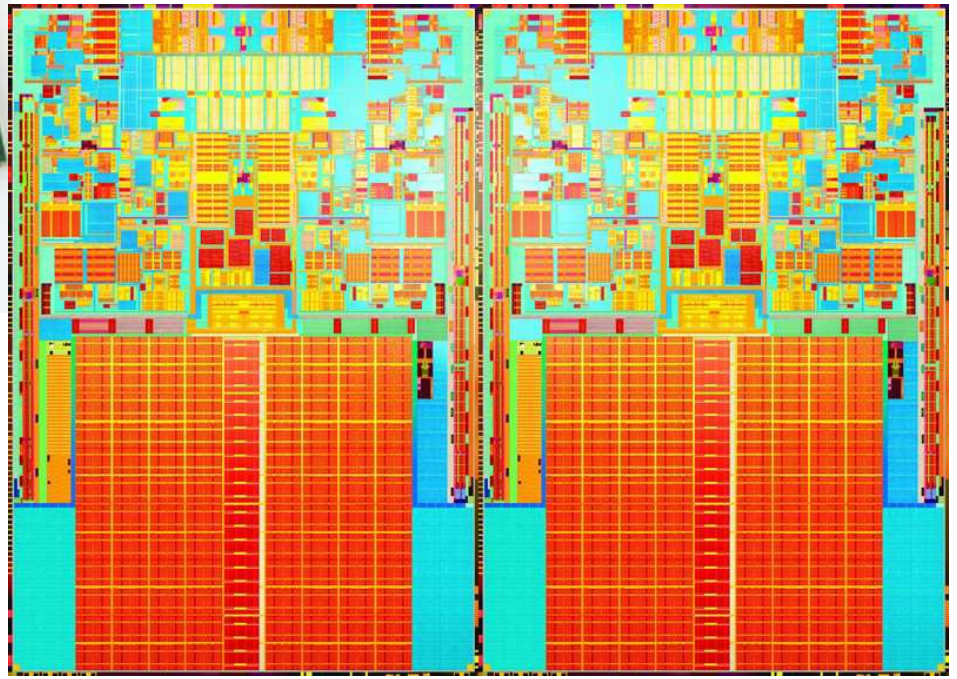
Hardware jargon: “Motherboard” of a single machine



Credit: Tyan

Hardware background

Hardware jargon: One single “processor”, 4 “cores”



Intel Core2 Quad

Credit: Intel

Hardware background

Hardware jargon: “Virtual cores”

- Today, it takes 60–100ns to fetch data from main memory
- Comparison:
  - 1 clock tick @ 4 GHz: 0.25ns
  - 1 integer addition: ~3 clock ticks = 0.75ns
  - 1 floating point multiplication: ~6 clock ticks = 1.5ns
- Matrix-vector product: 2 data loads for 1 mult + 1 add

Consequence:

- Processor units are *frequently idle*!
- Each physical core pretends to be *multiple virtual cores* to execute multiple threads of execution at once

Hardware background

Hardware jargon: “Virtual cores”

Note: “Virtual cores” are referred to by many names:

- Hardware threads
- Hyperthreading
- Simultaneous multithreading (SMT)
- “Logical cores”
- Chip-level multithreading

Current examples

My laptop:
- 1 processor
- 4 cores per processor
- 2 virtual cores (“hyperthreading”) per physical core

One of my compute servers (bottom.math.tamu.edu):
- 4 processors
- 8 cores per processor
- 2 virtual cores (“hyperthreading”) per physical core

Supercomputer (sequoia.llnl.gov):
- 98,304 machines
- 1 processor per machine
- 18 cores per processor
- 4 virtual cores (“hyperthreading”) per physical core

Software background

Software jargon: “Machine”

- Collection of processor and memory resources
- Managed by one “instance” of an operating system
- Connected to the rest of the world via a “network”

- Typically one “software machine” per physical machine
- Virtualization: multiple “software machines” per physical machine

Software background

Software jargon: “Process”

- Technical term for “program”
- Each process has its own, separate “memory space”
- Operating system manages processes:
  – provides memory
  – provides interfaces to get data to disk/screen/network
- Typical system has a 10s – 100s of processes at all times

Software background

Software jargon: “Thread”

- “Thread” is an execution unit:
  – a stream of CPU instructions
  – equivalent of a “von Neumann program”
- One or more threads per process
  – new process starts with one thread
  – threads can create new threads
  – threads can terminate
- The threads of each process share a memory space
- Operating system *scheduler* manages threads:
  – maps threads to virtual cores
  – interrupts threads to let other threads run

Software background

Software jargon: “Thread”

...let us see how this is done in practice...

Parallelization

Basic truths about efficient computing:

- On a machine with *N* virtual cores, we are only efficient if:
  – our software has at least *N* threads
  – each thread does something all the time
  – no two threads duplicate computations

- We call this “parallelization”: splitting all the work onto multiple threads

- This is *impractical* theory:
  – not all threads will be busy all the time
  – some duplication is unavoidable
  – there is overhead (e.g. communication)

Amdahl's law

More formally:

- T = amount of work to be done overall
  = time to solve problem on one processor
- P = number of threads participating

- s = fraction of the work that can be done in parallel
- $1-s$ = fraction of work that either
  - can only be done by one thread, or
  - needs to be replicated on every thread

Then:

The parallel run-time of the program is

$$\frac{sT}{P}+(1-s)T$$

Amdahl's law

Example: Computing the l_2 norm of a vector of length 10^6

- T = 10^6 floating point multiplications
  + sending "local" contributions to thread 1
  + P-1 additions
  + 1 square root
  + printing result to the screen

  = $6\ 10^6 + 10^4 + (P\text{-}1) + 20 + 10^4$ clock ticks
- s = $6\ 10^6/T$
- $1\text{-}s$= $(10^4 + (P\text{-}1) + 20 + 10^4)/T$

Then:

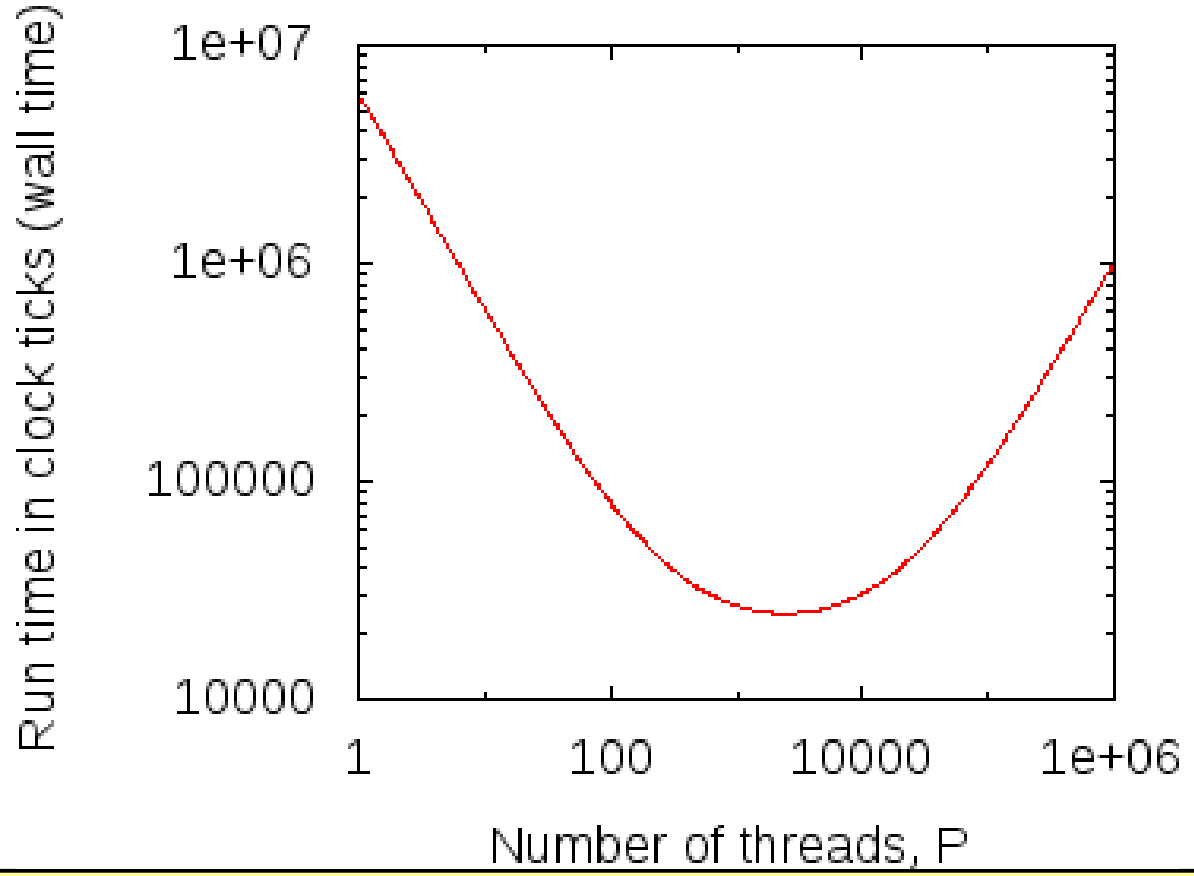
The parallel run-time of the program is

$$\frac{6 \cdot 10^6}{P} + (20020 + P - 1)$$

Amdahl's law

The parallel run-time of the program is

$$\frac{6 \cdot 10^6}{P} + (20020 + P - 1)$$



Amdahl's law

Definition: “Speedup”

- Ratio of “wall time” for 1 thread to “wall time” for P threads

- With Amdahl's law:

$$\text{speedup} = \frac{T}{\frac{sT}{P}+(1-s)T} = \frac{1}{\frac{s}{P}+(1-s)}$$

- Ideally: speedup = P

Parallelization

Two basic paradigms for parallelization:

- *Shared memory parallelization:*
  - Work is partitioned onto multiple threads within one process that share a memory space
  - Uses pthreads, OpenMP, TBB, Cilk, ...
  - $2 \ldots N_{cores}$ threads

- *Distributed memory parallelization:*
  - Work is partitioned onto different processes in separate memory spaces
  - Can reside on different machines
  - Uses the *Message Passing Interface (MPI)*
  - $2 \ldots N_{cores} \cdot N_{machines}$ processes

Note: *Hybrid parallelization* uses both.

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University