SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and High Performance Computing
Siena College
Fall 2021

Topic Notes: Introduction and Overview

Welcome to CSIS 335!

Why Parallel Computing?

The basic idea of parallel computing is simple enough – when one computer isn't powerful enough to solve the problem you want to solve in the timeframe in which you want that solution, use more than one!

Before we start to think about how to use parallelism on a computer, let's think about a parallel approach to solving a "real-world" problem.

- Adding up a collection of numbers (*e.g.*, sum of an array)

  We'll experiment with this one in class.

- Taking a census of Loudonville.

  One person doing this would visit each house, count the people, and ask whatever questions are supposed to be asked. This person would keep running counts. At the end, this person has gathered everything.

  If there are two people, they can work concurrently. Each visits some houses, and they need to "report in" along the way or at the end to combine their information. But how to split up the work?

  - Each person could do what the individual was originally doing, but would check to make sure each house along the way had not yet been counted.
  - Each person could start at the town hall, get an address that has not yet been visited, go visit it, then go back to the town hall to report the result and get another address to visit. Someone at town hall keeps track of the cumulative totals. This is nice because neither person will be left without work to do until the whole thing is done. This is the *master-worker* or *bag of tasks* method of breaking up the work.
  - The town could be split up beforehand. Each could get a randomly selected collection of addresses to visit. Maybe one person takes all houses with even street numbers and the other all houses with odd street numbers. Or perhaps one person would take everything west of Route 9 and the other everything east of Route 9. The choice of how to divide up the town may have a big effect on the total cost. There could be excessive travel if one person walks right past a house that has not yet been visited. Also, one person could finish completely while the other still has a lot of work to do. This is a *domain decomposition* approach.

- Grading a stack of exams. Suppose each has several questions. Again, assume two graders to start.
  - Each person could take half of the stack. Simple enough. But we still have the potential of one person finishing before the other.
  - Each person could take a paper from the "ungraded" stack, grade it, then put it into the "graded" stack.
  - Perhaps it makes more sense to have each person grade half of the *questions* instead of half of the exams, maybe because it would be unfair to have the same question graded by different people. Here, we could use variations on the approaches above. Each takes half the stack, grades his own questions, then they swap stacks.
  - Or we form a *pipeline*, where each exam goes from one grader to the next to the finished pile. Some time is needed to start up the pipeline and drain it out, especially if we add more graders. These models could be applied to the census example, if different census takers each went to every house to ask different questions.
  - Suppose we also add in a "grade totaler and recorder" person. Does that make any of the approaches better or worse?
- Adding two 1,000,000 $\times$ 1,000,000 matrices.
  - Each matrix entry in the sum can be computed independently, so we can break this up any way we like. Could use the bag of tasks approach, though a domain decomposition would probably make more sense. Depending on how many processes we have, we might break it down by individual entries, or maybe by rows or columns.

In each of these cases, we have taken what we might normally think of as a *sequential* process, and taken advantage of the availability of *concurrent processing* to make use of multiple workers (cores or processing units).

Parallelism adds complexity (as we will see in great detail), so why bother?

- we want to solve the same problem but in a shorter time than possible on one processor – goal: *speedup*
  - think: a weather model that takes 2 days to produce tomorrow's weather forecast isn't so good
- we want to solve larger problems than can currently be solved at all on a single processor – goal: *scale-up*
  - think: again back to the weather model, the choice might be the level of resolution, what level of detail can we compute, and more paralellism could mean more accurate forecasts for more places in time for them to be meaningful

- some algorithms are more naturally expressed or organized concurrently
- and now: that's where performance gains come from in modern processors!
  See the chart on page 28 of Saving the Future of Moore's Law

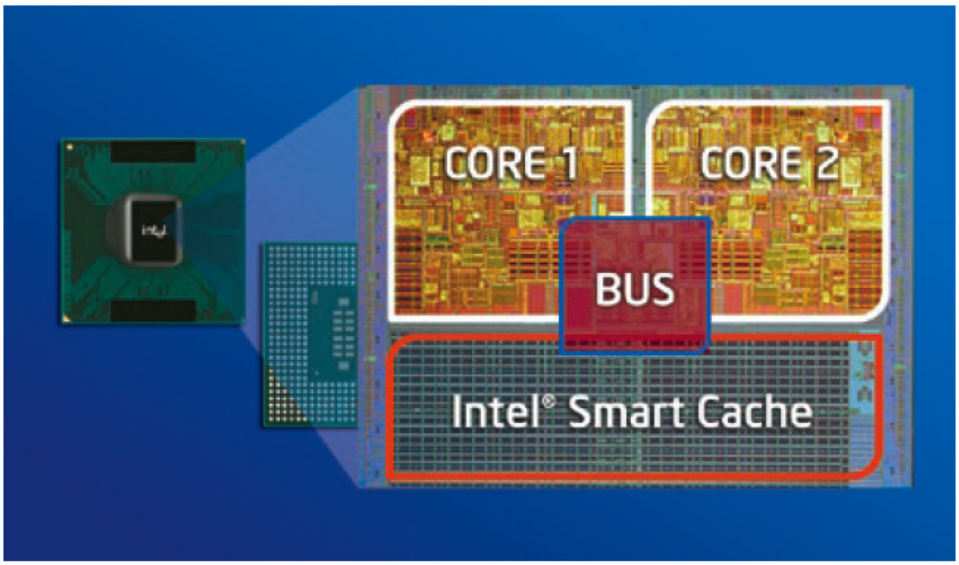

Image from Intel Core Duo Processor product brief.

Some Basics

Sequential Program: sequence of actions that produce a result (statements + variables), called a process, task, or thread (of control). The state of the program is determined by the code, data, and a *single* program counter.

Concurrent Program: two or more processes that work together. Big difference: *multiple* program counters.

To cooperate, the processes need *communication* and *synchronization*, which can be achieved through *shared variables*, or *message passing*.

We will consider all of these

Hardware to run concurrent processes

- single processor – logical concurrency (see Operating System course)
- multiprocessor – shared memory
- multicomputer – separate memories

- network – slower communication

Computers may be classified as:

- *single instruction, single data (SISD)* – one processor doing one thing at a time to one piece of data at a time.
- *single instruction, multiple data (SIMD)* – multiple processors all doing the same thing at the same time, but operating on different data. Also known as: vector computers. Program operates in "lock step" on each processor.
- *multiple instruction, multiple data (MIMD)* – multiple processors each doing their own thing.
- *single program, multiple data (SPMD)* – not really a classification of the computer, but of a model used to program a MIMD computer. Multiple processors run the same program, but do not operate in lock step. Also known as the *interacting peers* model. This is the model we will use most in this class.

Some examples:

- SISD: Pre-"multi-core" desktops and laptops.
- SIMD: graphics cards that apply a single operation to an array of data points at the same time.
- MIMD: desktops/laptops/mobile devices with multiprocessors or multi-core chips – each processor can be executing any instruction and operating on any data.
- MIMD: `noreaster.teresco.org`: Dual Intel Xeon Processor E5-2630 v4 (10 Cores, 2.2GHz, 3.1GHz Turbo)
- MIMD: ASCI Red, Sandia National Labs: 4600+ nodes, each with 2 Intel Pentium II Xeon processors, first TeraOp machine in 1997.
- MIMD: Stampede2: Cluster with 5940 nodes and a total of 368,280 cores, Intel Xeon Phi Knights Landing, Intel Xeon Skylake, peak performance 12800 TFlops, installed 2017

See `https://www.top500.org/`.

Moral: from the your phones and tablets to desktop and laptop computers to the most powerful supercomputers, it's a world of parallel processing out there!

How to Achieve Parallelism

- We need to determine where concurrency is possible, then *partition* the work accordingly. There are two major approaches to the partitioning problem.

  - *task parallelism*: different tasks are partitioned among the processing units
  - *data parallelism*: processing units all do the same tasks, but on different parts of the data

- This is easiest if a compiler can do this for you – take your sequential program and extract the concurrency automatically. This is sometimes possible, especially with fixed-size array computations.
- If the compiler can't do it, it is possible to give "hints" to the compiler to tell it what is safe to parallelize.
- But often, the parallelization must be done explicitly: the programmer has to create the threads or processes, assign work to them, and manage necessary communication.

We will consider all of these types of parallelism here.