Parallel Programs

[§2.1] Why should we care about the structure of programs in an architecture class?

- Knowing about them helps us make design decisions.
- It led to key advances in uniprocessor architecture
  - Caches.
  - Instruction-set design.

This is even more important in multiprocessors. Why?

In our discussion of parallel programs, we will proceed as follows.

- Introduce “motivating problems”—application case-studies.
- Describe the steps in creating a parallel program
- Show what a simple parallel program looks like in the three programming models,
  and consider what primitives a system must support.

We will study these parallel applications.

- Simulating ocean currents.
  - Discretize the problem on a set of regular grids, and solve an equation on those grids.
  - Common technique, common communication patterns.
  - Regular structure, scientific computing.
- Simulating the evolution of galaxies.
  - No discretization of domain.
  - Rather, the domain is represented as a large number of bodies interacting with each other—an $n$-body problem.
Irregular structure, unpredictable communication, scientific computing.

- Rendering scenes by ray tracing
  - Traverses a 3D scene with unpredictable access patterns and renders it into a 2-dimensional image for display.
  - Irregular structure, computer graphics

- Data mining
  - Irregular structure, information processing.
  - I/O intensive; parallelizing I/O important.
  - Not discussed here (read in book).

**Simulating ocean currents**

*Goal:* Simulate the motion of water currents in the ocean. Important to climate modeling.

Motion depends on atmospheric forces, friction with ocean floor, & “friction” with ocean walls.

Predicting the state of the ocean at any instant requires solving complex systems of equations.

The problem is *continuous* in both space and time, but to solve it, we discretize it over both dimensions.

Every important variable, e.g.,

- pressure
- velocity
- currents

has a value at each grid point.

This model uses a set of 2D horizontal cross-sections through the ocean basin.
Equations of motion are solved at all the grid points in one time-step. Then the state of the variables is updated, based on this solution. Then the equations of motion are solved for the next time-step.

Each time-step consists of several computational phases.

- Values are set up.
- A system of equations is solved.

All phases sweep through all points of the arrays and manipulate their values.

The more grid points we use to represent an ocean, the finer the spatial resolution, and the more accurate our simulation.

Simulating an ocean that is 7,000 km. across with 100 × 100 points => 70 km. between points.

Simulating 5 years with 5,000 time steps means updating the state every 8 1/2 hrs.

The need for parallel processing is clear.

**Simulating the evolution of galaxies**

What happens when galaxies collide? How does a random collection of stars become a galaxy?

This involves simulating the motion of a number of bodies moving under forces exerted by all the bodies.

In each time-step—

- Compute gravitational forces exerted on each star by all the others.
- Update the position, velocity, and other attributes of the star.
A brute-force approach to calculating interactions between stars would be $O(\ldots)$. However, smarter algorithms are able to reduce that to $O(\ldots)$, making it possible to simulate systems of millions of stars. They take advantage of the fact that the strength of gravitational attraction falls off with distance.

So the influences of stars far away don’t need to be computed with such great accuracy.

We can approximate a group of far-off stars by a single star at the center of the group.

The strength of many physical forces falls off with distance, so hierarchical methods are becoming increasingly popular.

Some galaxies are denser in some regions. These regions are more expensive to compute with.

- Stars in denser regions interact with more other stars.
- Ample concurrency exists across stars within a time-step, but it is irregular and constantly changing => hard to exploit.
Ray tracing

Ray tracing is a common technique for rendering complex scenes into images.

- Scene is represented as a set of objects in 3D space.
- Image is represented as a 2D array of pixels.
- Scene is rendered as seen from a specific viewpoint.
- Rays are shot from that viewpoint through every pixel into the scene.
- Follow their paths ...
  - They bounce around as they strike objects.
  - They generate new rays: ray tree per input ray
- Result is color and opacity for that pixel.

There is parallelism across rays.

The parallelization process

[§2.2] Sometimes, a serial algorithm can be easily translated to a parallel algorithm. Other times, to achieve efficiency, a completely different algorithm is required.

Pieces of the job:

- Identify work that can be done in parallel.
- Partition work and perhaps data among processes.
- Manage data access, communication and synchronization.
- Note: Work includes computation, data access and I/O.

Main goal: Speedup (plus low programming effort and low resource needs).
Tasks

The first step is to divide the work into tasks.

- A task is an arbitrarily defined portion of the work done by the program.
- It is the smallest unit of concurrency that the program can exploit.
- Example: In the ocean simulation, it can be computations on a single grid point, a row of grid points, or any arbitrary subset of the grid.
- Example: In the galaxy application, it may be a

Tasks are chosen to match some natural granularity in the work.

- If the grain is small, the decomposition is called
- If it is large, the decomposition is called

Processes

A process (or “thread”) is an abstract entity that performs tasks.

- A program is composed of cooperating processes.
- Tasks are assigned to processors.
Example: In the ocean simulation, an equal number of rows may be assigned to each processor.

Processes need not correspond 1-to-1 with processors!

Four steps in parallelizing a program:

- **Decomposition** of the computation into tasks.
- **Assignment** of tasks to processors.
- **Orchestration** of the necessary data access, communication, and synchronization among processes.
- **Mapping** of processes to processors.

Together, decomposition and assignment are called *partitioning*.

They break up the computation into tasks to be divided among processes.

- Tasks may become available dynamically.
- The number of available tasks may vary with time.

**Goal:** Enough tasks to keep processes busy, but not too many.

The number of tasks available at a time is an upper bound on the achievable

**Amdahl’s law**

If some portions of the problem don’t have much concurrency, the speedup on those portions will be low, lowering the average speedup of the whole program.

Suppose that a program is composed of a serial phase and a parallel phase.

- The whole program runs for 1 time unit.
- The serial phase runs for time \( s \), and the parallel phase for time \( 1 - s \).

Then regardless of how many processors \( p \) are used, the execution time of the program will be at least
and the speedup will be no more than ___. This is known as Amdahl’s law.

For example, if 25% of the program’s execution time is serial, then regardless of how many processors are used, we can achieve a speedup of no more than

*Example*: Consider a program with two phases.

- In the first phase, a single operation is performed on all points of a 2D $n$-by-$n$ grid, as in
- In the second phase, the sum of the $n^2$ grid-point values is computed.

*Assume*: In a sequential program, the first phase and second phase take the same amount of time (say, 1 unit per point, or ____ altogether).

If we have $p$ processors, we can assign $\frac{n^2}{p}$ points to each processor, and complete the first phase in parallel in time

In the second phase, each processor adds each of its $\frac{n^2}{p}$ values into a global-sum variable.

What’s wrong with this?

Thus, the second phase takes ____ time, regardless of $p$.

What is the best possible speedup in this decomposition? Well, ...

How much time does it take to execute the program sequentially?

How much time does it take to execute it in parallel, with the above decomposition?

The speedup is thus at most ______________, or $\frac{2p}{p + 1}$. 
What is the highest speedup we can get, regardless of the number of processors?

An easy way to improve this is to have each processor sum up its $\frac{n^2}{p}$ values into its own local sum.

This gives us a 3-phase program:

1. Operation performed on all $n^2$ points.
2. Processes add their values independently into private sum.
3. Processes add their private sums into global sum.

Phase 2 is now fully parallel (all $p$ processors operate independently). Phase 3 is sequential, but it is shorter; there are only _________ operations in it, not

The total time for the parallel program is now

The speedup can be as great as

What happens to this speedup as the number of processors $p$ increases?

Here is a diagram of what is happening in these decompositions.

This diagram, depicting how many tasks can be performed concurrently at each point in time, is called a concurrency profile.

The concurrency profile depends on the problem and the decomposition. It is independent of the of the number of processors (it is given in terms of the number of processors $p$).
It is also independent of the assignment or orchestration.

Concurrency profiles may be regular, as above, or they may be irregular, like the profile for this discrete-event logic simulator.

The area under the curve is the amount of work done.

The horizontal extent is a lower bound on the time it would take to run the best parallel program given that decomposition, assuming—

- an infinite number of processors
- are free.

The area divided by the horizontal extent therefore gives us a limit on the achievable speedup.

Amdahl’s law can be expressed as—

\[
\text{Speedup} \leq \frac{\text{Area under concurrency profile}}{\text{Horizontal extent of concurrency profile}}.
\]

It is easy to show (see p. 87 of CS&G) that the speedup with an infinite number of processors is limited by \(1/s\), and with \(p\) processors, it is limited by
\[
\frac{1}{s + \frac{1 - s}{p}}
\]