Kinds of Parallel Programming

Parallel programming has many uses, but a fully parallel program is not always feasible. Find out more about this in the lesson below.

We'll cover the following

There are many flavours of parallel programming, some that are general and can be run on any hardware, and others that are specific to particular hardware architectures.

Two main paradigms we can talk about here are **shared memory** versus **distributed memory** models. In shared memory models, multiple processing units all have access to the same, shared memory space. This is the case on your desktop or laptop with multiple CPU cores. In a distributed memory model, multiple processing units each have their own memory store, and information is passed between them. This is the model that a networked **cluster** of computers operates with. A computer cluster is a collection of standalone computers that are connected to each other over a network, and are used together as a single system. We won't be talking about clusters here, but some of the tools we'll talk about (e.g. MPI) are easily used with clusters.

Types of parallel tasks

Broadly speaking we can separate a computation into two camps depending on how it can be parallelized. A so-called embarrassingly parallel problem is one for which it is dead easy to separate it into some number of **independent** tasks that then may be run in parallel.

Embarrassingly parallel problems

Embarrassingly parallel computational problems are the easiest to parallelize and you can achieve impressive speedups if you have a computer with many cores.

Even if you have just two cores, you can get close to a two-times speedup. An example of an embarrassingly parallel problem is when you need to run a preprocessing pipeline on datasets collected for 15 subjects. Each subject's data can be processed **independently** of the others. In other words, the computations involved in processing one subject's data do not in any way depend on the results of the computations for processing some other subject's data.

As an example, a grad student in my lab (Heather) figured out how to distribute her FSL preprocessing pipeline for 24 fMRI subjects across multiple cores on her Mac Pro desktop (it has 8) and as a result what used to take about 48 hours to run, now takes "just" over 6 hours.

Serial problems

In contrast to embarrassingly parallel problems, there is a class of problems that cannot be split into independent sub-problems, we can call them **inherently sequential** or **serial** problems. For these types of problems, the computation at one stage **does** depend on the results of a computation at an earlier stage, and so it is not so easy to parallelize across independent processing units. In these kinds of problems, there is a need for some communication or coordination between sub-tasks.

An example of a serial problem is a simulation of an arm movement. We run simulations of arm movements like reaching, that use detailed mathematical models of muscle mechanics, activation dynamics, musculoskeletal dynamics and spinal reflexes. Differential equations govern the relationship between muscle stimulation (the input) and the resulting arm movement (the output). These equations are "unwrapped" in time by a differential equation integrator, that takes small steps (like 1 millisecond at a time) to generate a simulation of a whole movement (e.g. 1 second of simulated time). On each step the current state of the system depends on both the current input (muscle command) **and** on the previous state of the system. With a 1 ms step, it takes (at least) 1000 computations to simulate a 1 sec arm movement ... but we cannot simply split up those 1000 computations and distribute them to a set of independent processing units. This is an inherently serial problem where the current computation cannot be carried out without the results of the previous computation.

As an aside, the way to take advantage of parallelism even with a serial problem, is to parallelize meta-computations. So for example, we typically want to run

"sensitivity analyses" where we vary some parameter(s) of the arm model and re-run the simulation. If we have 100 such simulations to run, even though each one of them is a serial problem on its own, they are independent of each other, and so we can parallelize the sensitivity analysis by distributing those 100 simulations to multiple processing units.

Mixtures

A good example of a problem that has both embarrassingly parallel properties as well as serial dependency properties, is the computations involved in training and running an artificial neural network (ANN). An ANN is made up of several layers of neuron-like processing units, each layer having many (even hundreds or thousands) of these units. If the ANN is a pure feedforward architecture, then computations within each layer are embarrassingly parallel, while computations between layers are serial.

The rest of this section will elaborate on the different tools and APIs which serve our purpose of parallel programming.