Processes

A compute ***process*** is an instance of a program or subprogram that can be run independently on physical processors, or other computing unit (i.e. cores).

- A process can be *spawned* or *forked.* Python processes on Unix/OSX are forked by default
  - *a* forked process is essentially identical to the parent process (no need to re-import Numpy, for example).
- A process *owns* resources, i.e. they have their own memory.
- Separate processes may or may not be able to run in *simultaneously*. If more processes are requested than available hardware computing units, the processes will have to share available hardware resources. This is managed by the OS.
- Processes cannot communicate with each other except through dedicated communication channels, or through use of shared memory space.
- The parent process can terminate before the child process is finished, or use join to wait for the child process to complete before terminating.
- A *processor* is a hardware unit, whereas a *process* is an abstraction of a subprogram. These are sublimes used interchangeably.

Parallel verses serial program

- If a program is written so that various sub-problems can be executed simultaneously, then the program is said to “run in parallel”.
- Parallel processes are often closely related sub-tasks whose results are communicated to each other and to a master node. to obtain a final result in less than than would be required to run the program in serial.
- As opposed to “concurrent” programming where tasks may have nothing to do with each other. One program may be producing simulation data, while concurrently, as second program is creating graphics output.
- Term usually reserved for large-scale scientific and engineering tasks.
- As opposed to “multi-tasking”.
- A parallel program is a program that is written to take advantage of the availability of multiple compute units (cores, processors, and so on).
- A well-designed parallel algorithm should have minimal overhead, but run faster than the equivalent serial code.

Communication

- Typically refers to the communication between processes.
- On modern hardware, communication costs can dominate the cost of running a program in parallel.
- If communication costs swamp the actual processing, the parallel program is said to *scale poorly*. On the other hand, if communication costs are minimal (computing pi, for example), then the program is said to be *pleasingly parallel* or *embarrassingly parallel*.
- Data “serialization” - turn data into a stream of bytes for passing between processes.

In Python

- Passing arguments in Python (only available to communicate from parent process to spawned child process.
- Using a Queue (collective communication)
- Using a Pipe (point-to-point communication)

Memory

- Shared memory vs. distributed memory
- Uniform access verses non-uniform access
- Cache levels
- Instruction registers
- Memory Hierarchy.

Parallel Tasks

- Blocking verses non-blocking
- Synchronous verses asynchronous
- Gather, scatter, broadcast

Types of Parallelism

Data Parallelism

- Examples : matrix-vector multiply, 1d array functions (mean, min, maximum, median, computing pi, etc.

Task Parallelism

- different tasks run on different processors (compute mean on one process, compute the median on another).

Load balancing

- Median problem.
- Pool of workers to manage tasks with different amounts of work to do.

Parallel Performance

- Strong scaling
- Weak scaling
- Perfect scaling
- Pleasingly parallel
- embarrassingly parallel

Arithmetic intensity

Goal in designing parallel algorithms is to spend as much time doing actual work and minimize time in a accessing data.

One way to accomplish this is to design algorithms with *high arithmetic intensity.*

$$\textbf{\textit{Arithmetic intensity}} = \frac{\textit{Time spent using CPU}}{\textit{Time spend reading, writing and communicating data}}$$

Typically measured in **flops/byte,** where flops = (floating point operations)

Example : z = x + y 2 reads (x,y); 1 flop (+); 1 write (z). So arithmetic intensity is 1/3

Example : z = sqrt(x.^2 + y.^2) 2 reads + 3 ops + 2 (elem func) = 5/2

Roofline model is often used to visually illustrate arithmetic intensity.

Performance modeling

Given some assumptions about how the machine works, we can develop models of performance to predict the behavior of algorithms.

Model 1: Assume that communication costs are negligible. Then if a job takes W_1 time on 1 processor, then it will take

$$W_p = \frac{W_1}{P}, \qquad \text{no communication costs}$$

time on P processors.

Model 2: Assume that communication takes S time. Then if a job takes W_1 time on 1 processor, then it will take

$$W_p = \frac{W_1}{P} + S, \qquad \text{communication costs included}$$

time on P processors.

Other ideas

- Processes verses threads
- Multiprocessing, parallel processing, serial multi-tasking and concurrency
- Asynchronous verses synchronous computing and latency hiding
- Parallel I/O
- Amdahl's Law
- Flynn's Taxonomy