A Primer on Parallel Programming

—

Garrett Wright
October 18, 2021

Audience

- Programmers of any level who are not familiar with parallel programming paradigms.
- Researchers with computational components in their projects and limited parallel programming exposure.

Objective

- Briefly review the classic parallel programming taxonomy. (Flynn 1966)
- Familiarize core concepts, using them to describe and differentiate parallel programming models at the application level.
- Provide additional resources.
- Demonstrate a few common changes for implementations (if time).

Essentially, this is a brief survey of a wonderfully enormous landscape.

Situational Analysis

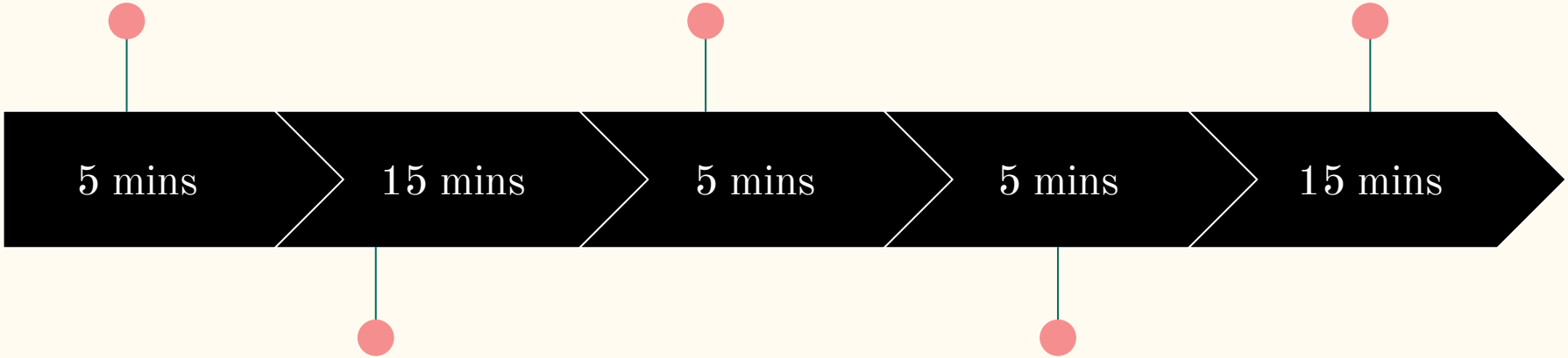
- A data science problem which computes the same stats models for many time series.
- It takes about four mins to run in serial on my laptop. I'd like it to take less so I can tweak my model, or play with hyperparameter optimization.
- I don't want to change code too much, or change original implementation.

- A physics simulation which models radiative transfer in atmosphere.
- It takes minutes to simulate a time step, but group intends to simulate tens of thousands of time steps.
- At meaningful scientific resolutions, requires a lot of RAM.

Background

Mapping Concepts To Technologies

Patterns, and What Common Approaches Feel Like



Key Concepts

Case Study and Additional Resources

If programming is hard,

parallel programming can be

rreaealllyly hardh.ard.
rreeaallllyy har had.rd.
rearlleya llhyard ha.rd.
reraellayl halrdy .hard.
realreallly hardy .hard.
realrley allhy haarrdd..
rreeallya hlardly .hard.

TODO That code might have a minor bug.

Parallel Programming is challenging because:

- Variety of solutions and technologies
- Code Changes
- Algorithmic modifications
  - These can range from minor to completely new algorithms.
- Sneaky bugs
  - Races
  - Non determinism
  - Numerical (floating point associativity...)
- Results limited by weakest link

This presentation should introduce you to the first topic.

Additional workshops can demonstrate common code changes, and the most common patterns (for something like OMP, MPI or CUDA). I'll just give you a peek...

Debugging and performance tuning come with "time on the tools". Try stuff.

Flynn's Taxonomy

Classic description of low level operations.

Comes from a time when people worked "closer to the machines".

SISD

Single Instruction Single Data

MISD

Multiple Instruction Single Data (Historical)

SIMD

Single Instruction Multiple Data (Vector)

MIMD

Multiple Instruction Multiple Data (Multi-processing)

Flynn's Taxonomy

Has been extended many ways over the years. Some stuck.

These are more commonly used today. Note, abstraction moves farther and farther from the machine...

SIMD

Single Instruction Multiple Data (Vector)

SIMT

Single Instruction Multiple *Thread* (CUDA/GPU)

SPMD

Single *Program* Multiple Data

MPMD

Multiple *Program* Multiple Data (Example later)

Concepts

- Processes vs Threads
- Execution Scheduling
- Granularity, Coupling, Synchronicity
- Memory Architecture
- Scaling

Processes Vs Threads

Processes have their own memory.

Threads belong to a process and share memory.

You should know there exists advanced techniques which fork this distinction, but this is a useful way to think about it.

Execution Scheduling

Serial

A sequence of tasks, over a sequence of data, using a single computational resource.

```
for x in ListOfDataFiles:
    download(x) # Network
    longComputation(x) # CPU
    writeToArchive(x) # Disk IO
```

Parallel

Still a sequence of tasks, but we'll perform the same tasks ***simultaneously*** using multiple computational resources.

	Time 0	Time 1	Time 2
Computer 1	download(x1)	longComputation(x1)	writeToArchive(x1)
Computer 2	download(x2)	longComputation(x2)	writeToArchive(x2)
Computer 3	download(x3)	longComputation(x3)	writeToArchive(x3)

Concurrent (Pipelined)

Compositions of tasks, *multiple* ***in progress*** (overlapping).

Time 0	Time 1	Time 2	Time 3	
download(x1)	download(x2)	download(x3)		
	longComputation(x1)	longComputation(x2)	longComputation(x3)	
		writeToArchive(x1)	writeToArchive(x2)	writeToArchive(x3)

- Note that concurrence can still be effective even with a single processor, because it can progress distinct components of a larger system. In this example, we might rely on the operating system or hardware to service network and disk in the background while the CPU is *mostly* busy computing....

Why even consider Concurrency?

In real systems with finite resources there are limits to parallelism, and we do a lot more than just CPU arithmetic.

Silly example: downloading three files from the same resource simultaneously might actually be slower than downloading sequentially.

We may be using limited cluster resources poorly, particularly when scaled up.

In many cases it can be practical to compose and overlap (pipeline) tasks, even purely computational ones, such as a Multiple Program model up next.

Concurrent memory/computation tasks is truly a critical performance consideration for GPU programming, and occasionally for large MPI programs as well.

A Simple Abstract Example

Let's say we have an Ocean model and Atmosphere model. They are both distinct parallel programs. They require asymmetric computational resources (1024 and 96 cores respectively) for similar time to solution (a step). We wish to run a time step of each, then exchange data, which we'll consider virtually free.

If we lease 1024 cores, then run a step of each model sequentially in this configuration *we are wasting roughly half of our compute cycles.*

Step1	Ocean1	….	….	….	Ocean1024
Step1	Atmos1	….	Atmos96	*Idle….*	*….Idle*
Step2	Ocean1	….	….	….	Ocean1024
Step2	Atmos1	….	Atmos96	*Idle….*	*….Idle*

Parallelism and Concurrency (MPMD)

Instead let's imagine the two distinct parallel programs both making progress…

This may require 1120 cores, 1024+96. However, if the Ocean and Atmosphere programs' time steps are balanced, we no longer have idle cores…

	Process1	….			Process1120
Step1	Ocean1	….	Ocean1024	Atmos1	Atmos96
Step2	Ocean1	….	Ocean1024	Atmos1	Atmos96

… and we now require about half the wall time to solution by leveraging both concurrency and parallelism. Roughly half the resources too…

Granularity, Coupling and Synchronicity

Granularity

Granularity is commonly used to describe the amount of arithmetic relative to non arithmetic work.

Examples of non arithmetic work would be system/network communication and IO.

Generally expressed as "*Coarse*" vs "*Fine*". Course would have a relatively high amount of computation relative to communication.

The term "*Embarrassingly Parallel*" is used to describe problems which can be parallelized in such a way they have minimal communication requirements. This is an extremal case of course granularity.

In practice when people are expressing their parallel programming ideas, I have found that Granularity generally can be expressed on a spectrum of "*Coupling*" and "*Synchronization*".

Coupling

Coupling is the functional strength of dependency in the *application*.

Another way to think of the axis of coupling is at what level in the *application* we are operating, or synchronizing. I have listed in terms of the computer, but more likely it would be in consideration how your problem subdivides.

Vectorization (SIMD)

Threading

Messaging

Distributed Systems

CPU Instructions

Cache/RAM

Processes/Network

Job/Task

Synchronization

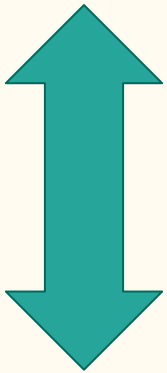
Synchronization is the temporal dependency of individual steps.

Another way to think of the axis of synchronization is the variety of time scales we can manage coupling in our application. These range from instruction level (SIMD or SIMT), frequently synchronized (common blocking MPI) code, to totally asynchronous (Distributed) systems.

SIMD

Threading

Messaging



Distributed Systems

Communication Synchronization

Synchronous

Process1
Send Msg to Process 2
Blocks; Wait for Ack
Blocks; Wait for Ack
Ack received.
…. continue

Process2
Block; Waiting For Data from Process1
Receives Msg
Send Ack
….continue

Fortunately, messaging libraries for parallel programming (MPI) wrap this up in a relatively simple way.

This crudely represents what happens behind the scenes of a single line *blocking MPI_SEND* - MPI_RECV pair.

Communication Asynchronization

One sided, Asynchronous:

Process1
Send Msg to Process 2
…. continues

Process2
Block; Waiting For Data from Process1
Receives Msg
…. continues

Again, messaging libraries for parallel programming (MPI) wrap this up in a simpler way.

This would represent a one sided pattern

non blocking MPI_ISEND with *blocking MPI_RECV* pair.

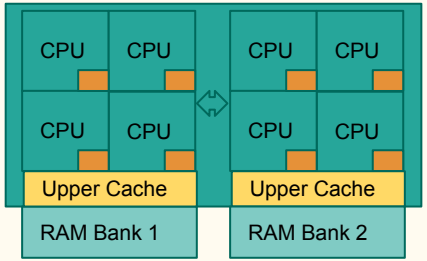
True async, send and forget, is much harder, and out of scope of this talk. More a systems topic...

Memory Architecture

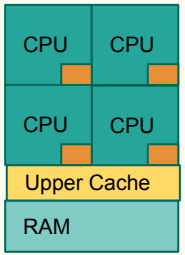
A very deep topic...

A single CPU eventually expanded to multiple single CPU chips in a server/cluster. Each having separate memory systems.

Also multicore (or multiple) chips, with asymmetric access to multiple memory subsystems.

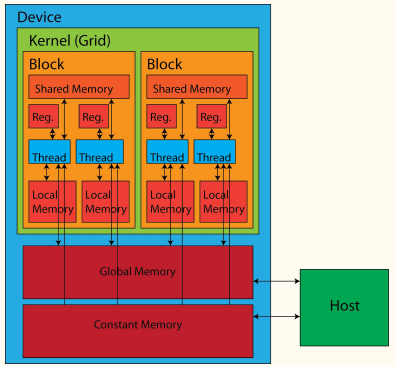


This has been replaced by multicore chips which *share* memory. This sharing of high speed memory facilitates threading for computation (a la OpenMP).



Study of modern memory systems and their relationship to the processors in modern system design would largely fall under "**NUMA**", non uniform memory architecture, and is a topic in Performance Tuning.

Add a GPU, and you have even more sophisticated memory hierarchy!



Scaling made Simple

No Formulas Necessary

Strong Scaling

Amdahl (1967)

For a fixed a problem size:

How does time to solution scale with core count?

Answers: Does my parallel algorithm perform as expected.

Serial code is the limiting factor.

Communication can get expensive.

Synchronization is never perfect.

Weak Scaling

Gustafson (1988)

Serial code is admitted as a constant factor, and ignored in the scaling formula.

For a scaling machine size:

How does time to solution scale with problem size?

Answers: How efficient is the *parallel* part of my program?

Downscaled, Linear, Superscaled?

Scaling is studied in Performance Tuning

Mapping Concepts

Differentiate parallel paradigms by how *memory* and *communication* is managed.

Shared Memory	**Distributed Memory**	**Hybrid, Distributed-Shared-Memory**
Posix Threads (pthread)	MPI	MPI+OpenMP
shmem	Spark/Hadoop	Multi-GPU
OpenMP	Dask	
SIMD or vector intrinsics (Intel MKL)	General Multiprocessing	
C++ Parallel STL (Intel TBB)		PGAS (Cray SHMEM)
GPU* (CUDA, OpenACC, HIP etc)		

* GPU's have their own non uniform memory architecture. Generally it is faster than RAM, with difference types of memory shared across various tiers of the GPU device.

Differentiate parallel paradigms by how *memory* and *communication* is managed.

Shared Memory	**Distributed Memory**	**Hybrid, Distributed-Shared-Memory**
Posix Threads (pthread)	MPI	MPI+OpenMP
shmem	Spark/Hadoop	Multi-GPU
OpenMP	Dask	
SIMD or vector intrinsics (Intel MKL)	General Multiprocessing	
C++ Parallel STL (Intel TBB)		PGAS (Cray SHMEM)
GPU* (CUDA, OpenACC, HIP etc)		

All of these are generally considered tightly coupled. While some are more tightly coupled than others, optimizing coupling and synchronization generally promotes performance in these systems.

Situational Analysis

- A data science problem which computes the same stats models for many time series.
  - Sounds *embarrassingly parallel*.
- It takes about four mins to run in serial on my laptop. I'd like it to take less so I can tweak my model, or play with hyperparameter optimization.
  - Problem isn't too large.
  - Probably wants to keep running on laptop,
    - Scaling out not important.
    - Loose/uncoupled
- I don't want to change code too much, or change original implementation.

Good approaches to this situation might be:

Multiprocessing (whole program in parallel)

OpenMP or MKL if the stats models are simple...

- A physics simulation which models radiative transfer in atmosphere.
  - Sounds *tightly coupled*.
  - Lots of computation.
- It takes minutes to simulate a time step, but we intend to simulate tens of thousands of time steps.
  - *Performance*, *scaling* and *synchronicity* will be important.
- At meaningful scientific resolutions, requires a lot of RAM.
  - Sounds like we need a special machine, or *Distributed Memory on a cluster*.

Good approaches here might be:

MPI (or MPI + OpenMP)

CUDA (or MPI+CUDA)

Misc Resources

A more in depth intro from LLNL (go at your own pace):

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

More Hands on Tutorials:

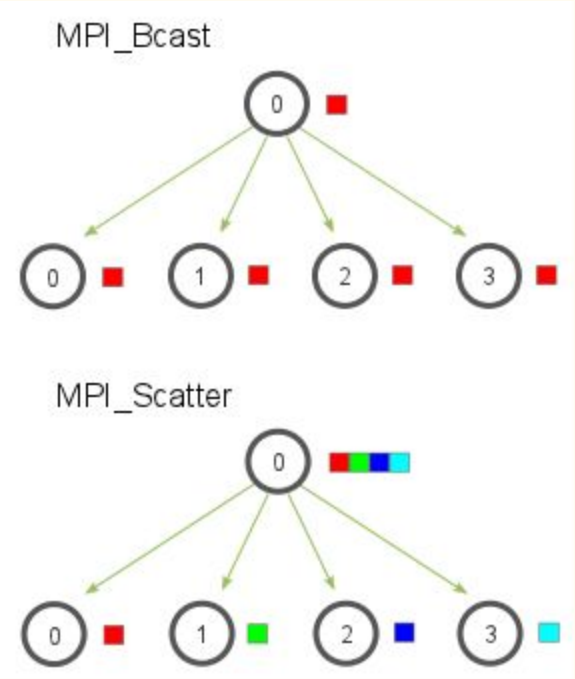
LLNL https://hpc.llnl.gov/training/tutorials
TACC https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html
MPI Tutorials (Love their graphics) https://mpitutorial.com/
And of course Princeton has lots of resources which you should learn about in our focused workshops.

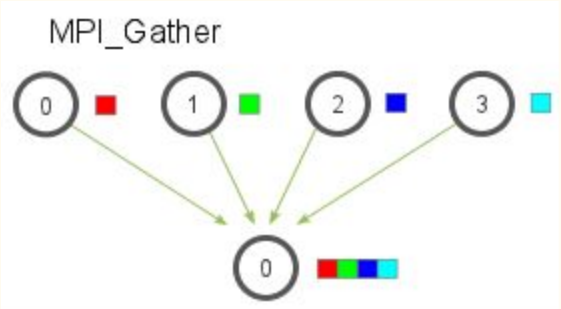
Links:

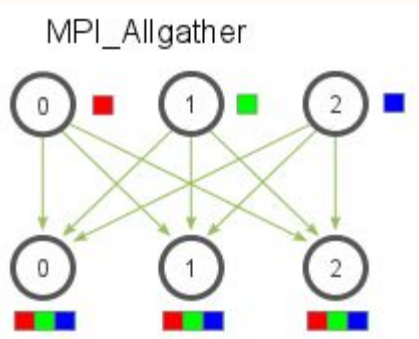
Go Blog (Concurrency Is not Parallelism)

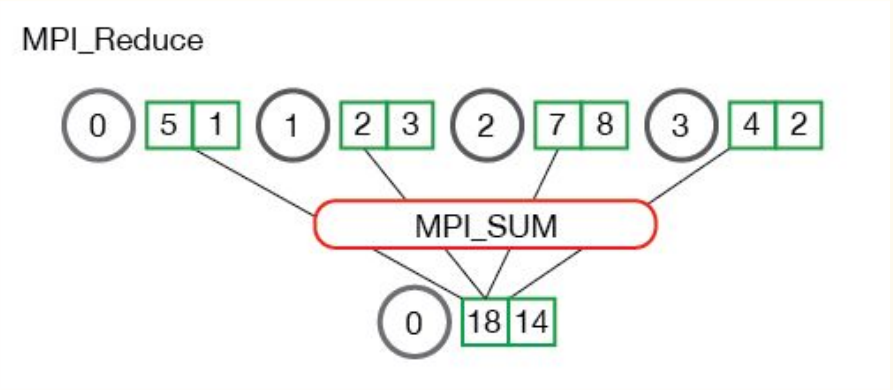
Books:

CUDA By Example https://developer.nvidia.com/cuda-example
The Linux Programming Interface (if you are interested in POSIX/Systems programming).









Common MPI Patterns - These should be covered in an MPI workshop, but the abstractions, like “Reduction” help to think about any parallel programming.

Images Courtesy
https://github.com/mpitutorial/mpitutorial/tree/gh-pages
(MIT License)

Examples

These are provided as a Github Repo:

`https://github.com/garrettwrong/parallel_primer`

We can briefly look at example changes to get a rough idea of modifications.

- Serial
  - Python
  - C
- OpenMP
- MPI
- CUDA (GPU)
- CUPY

OpenMP

- Low investment
- Usually maintains original (serial) program.
- Good performance
- Lowest barrier to entry among the classic HPC options.
- *Does not scale past multicore*

In this problem, to achieve a nice (nearly linear!) speedup required only six source code lines and a compiler flag.

Note that, forgetting any one of the sections does not yield the speedup. (Weakest link)

```
Makefile:
     For gcc, add -fopenmp to your C flags.

.c:
     #include <omp.h>
     ….
     #pragma omp parallel private(i, xi, j, xj, k)
     #pragma omp for reduction(+:tmp)
      for(i=0; i<n; i++){
         for(j=0; j<n; j++){
          ….
     #pragma omp parallel for private(i, j, val)
reduction(min:minD) reduction(max:maxD)
      for(i=0; i<n; i++){
         for(j=0; j<i; j++){
          …
     #pragma omp parallel private(i)
     #pragma omp for reduction(+:cnt)
     for(i=0; i<neps; i++){
          …
```

MPI

- Moderate amount of changes
  - Bounds checking
  - block/chunk sizing
- Required modification to algorithm to parallelize:
  - “Gather” Pattern
  - “All Gather” Pattern
  - “Reduction” Pattern
- Requires large problem size to overcome networking overhead.
  - Small problems actually slower.
- Once made MPI, it can be harder to port to other approaches (GPU).
- Potentially scales to large systems, and problems that would not fit on a single machine.

You’ll need some housekeeping to start...

```
#include <mpi.h>

   /* MPI Initialization */
   int mpi_pid, numprocs;
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank (MPI_COMM_WORLD, &mpi_pid);

     /* We’ll look at some common mods next.

   /* MPI shutdown */
   MPI_Finalize();
```

Common MPI Modifications

Frequently computing number of elements:

```
nelem = (n + numprocs - 1) / numprocs;
```

Branches for operations like printing, or serial code blocks:

```
if(mpi_pid == 0){...}
```

Computing/**Checking Bounds**:

```
ubnd = nelem*(mpi_pid+1);
if(n < ubnd){
  ubnd = n;
}
```

Changing loops:

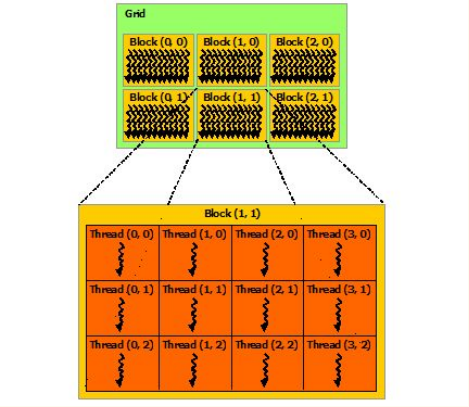
```
for(i=nelem*mpi_pid; i<ubnd; i++){...}
```

Generally speaking, this constitutes a lot of housekeeping. However boring that may be, it still must be done with great care. Parallel programming bugs are notoriously challenging.

CUDA

- Larger investment of time
- High performance (for some problems*)
- Reformulates algo components into kernels.
  - Something like Min/Max is not just a for loop anymore if you want to go parallel.
  - Kernels require “launching” code
- Requires memory management
  - There are ways around this, but if you want to go fast you still need to think about it.
- Can result in substantial amounts of bespoke code (not unlike MPI, just different)
- Hard to port CUDA code *back* to other platforms.

Mainly we need to map our problem to a “Grid” of “Blocks”, where each “Block” is an entry in a 1D, 2D, or 3D memory model.



I’ll give example of some housekeeping code.

Example Code Supporting CUDA

Memory management, allocations:

```
  double* X_dev;
  cudaMalloc(&X_dev, n * d * sizeof(double));
```

Management of transfers:

```
  cudaMemcpy(X_dev, X, n * d * sizeof(double),
cudaMemcpyHostToDevice);
```

Computing thread and block ids, usually in place of for loops:

```
  const int tidx = blockDim.x * blockIdx.x +
threadIdx.x;
  const int tidy = blockDim.y * blockIdx.y +
threadIdx.y;
```

Compute numbers of threads and blocks. Simplest 1D case:

```
  int blocksz = 1024;
  int nblocks = (n + blocksz -1 ) / blocksz;
```

Launching a kernel:

```
all_pairs_distances_kernel
<<<nblocks, blocksz>>>(X_dev, n, d, D);
```

Make sure you don't go out of bounds!:

```
  if( tidx >= n){
    return;
  }
  if( tidy >= neps){
    return;
  }
```

Thank you for attending!

—

Also a big thanks to everyone at PICSciE and OIT Research Computing for their efforts hosting these events!