12.010 Computational Methods of Scientific Programming 2021

Lecture 21: Parallel programming. Algorithms and scaling, MPI

Summary

- Parallel programs
  - Examples
  - Amdahl Law
  - Gustafson Law
- Implementation in MPI
  - MPI concepts and basics
  - Python and MPI example
  - Other tools
    - Threads
    - Interesting builtin Fortran parallelism!

Explicitly Parallel Programs

- All programs are parallel at some level
  - CPU does a lot of things at once but hides from application.
  - For example, in a single threaded program a CPU might execute multiple independent instructions in parallel. This is hidden from application.

- Explicitly parallel programs
  - Application is written so parallelism (multiple things potentially/actually happening at same time) is part of the program code
  - Parallelism is visible to the application and programmer.

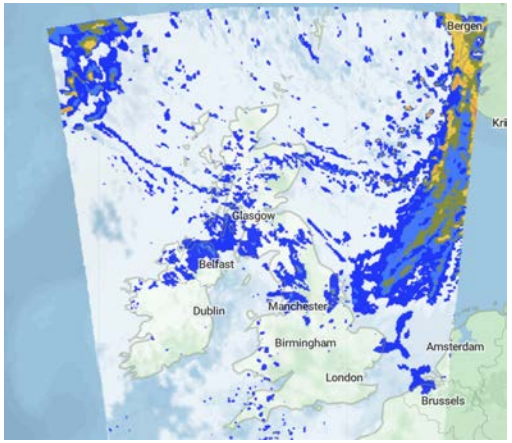
Examples

- Weather forecasts.

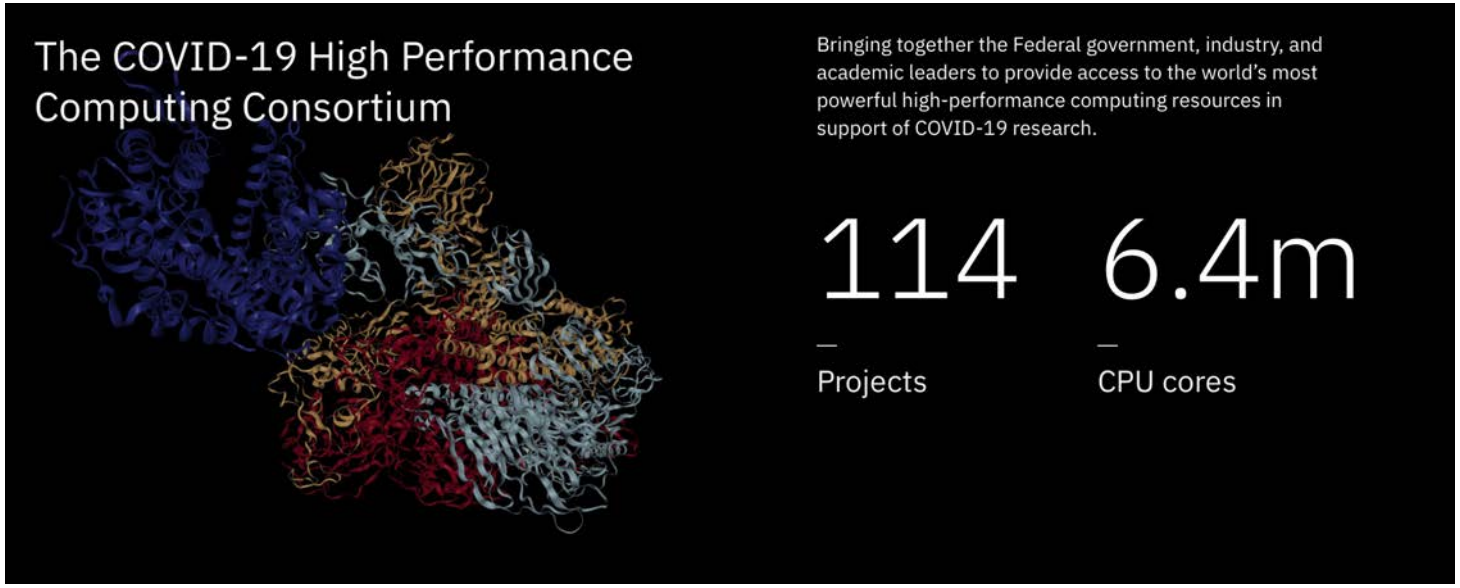
1922 proposal (L.F. Richardson)

~64,000 human computers

2021 actual ~50,000 CPU cores



Examples



Atomistic modeling, protein folding, pharmaceutical binding strength, aerosol dispersion, epidemiological modeling, genetic sequence correlation, phylogenetics, etc…

When is parallelism useful?

- Amdahls law
  - imagine program with fraction, f, that speeds up perfectly and fraction (1-f) that does not speed up at all when running on multiple processors
  - the execution time on N processors will be

$$T_N = \frac{fT_1}{N} + (1-f)T_1$$

where T_1 is time on one processor

  - define speedup as ratio of T_1 and T_N

$$S = \frac{T_1}{T_N}$$

e.g. if we run on 10 processors and time to solution is one tenth, then speedup is 10.

How does speedup vary as parallel fraction varies?

- Also define efficiency

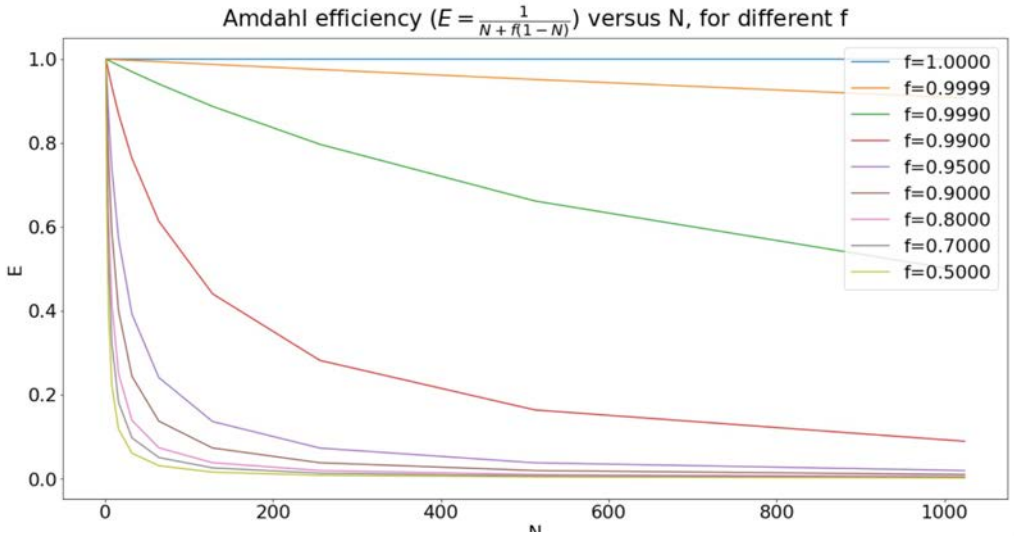
$$E = \frac{S}{N}$$

so that E=1 means (for example) speedup of 10 on 10 processors.

we can define efficiency, E, as a function of parallel fraction, f, for a processor count, N

$$E = \frac{1}{N + f(1 - N)}$$

Plotting efficiency for N and f



Notebook to plot Amdahl parallel efficiency curves

```
import numpy as np
import matplotlib.pyplot as plt
def eff(f,n):
    ϕ=n+f*(1-n)
    return 1/ϕ
```

```
N=np.array( (1,2,4,8,16,32,64,128,256,512,1024) )
F=np.array( (1,0.9999,0.999,0.99,0.95,0.90,0.80,0.70,0.5) )
```

```
plt.rcParams["figure.figsize"] = [20, 10]
plt.rcParams.update({'font.size': 22})
for f in F:
    evals=[eff(f,n) for n in N]
    fstr='f=%4.4f'%(f)
    plt.plot(N,evals,label=fstr)
leg = plt.legend(loc='upper right')
plt.ylabel('E')
plt.xlabel('N')
plt.title('Amdahl efficiency ($E=\\frac{1}{N+f(1-n)}$) versus N, for different f');
```

Amdahls characterizes “strong scaling” i.e. efficiency when problem size is held constant.
In reality, weather, molecular dynamics etc… run larger problems on larger computers such that f increases.

Gustafsons Law v Amdahl

Amdahls characterizes “strong scaling” i.e. efficiency when problem size is held constant.

In reality, weather, molecular dynamics etc… run larger problems on larger computers such that f increases. This is called weak scaling.

Gustafsons law expresses a weak scaling speedup relation

$$S_w = s + (1 - s)N = N + (1 - N)s$$

where s is the fraction of the program that runs serially.
In Gustafsons law speedup increases with N, because the serial fraction becomes a smaller part of the overall work.

In reality many real problems in scientific will exhibit some combination of strong and weak scaling.
i.e.
they are not completely fixed in size
and
they can not be sensibly grown forever

Handson

fall-2021-12.010/lec22/**amdahl.ipynb**

Writing parallel programs with MPI

Writing an explicitly parallel program involves using extra language syntax that expresses parallel operations.

There are multiple tools for doing this. The most common one in scientific computing is a library called the “Message Passing Interface” (MPI).

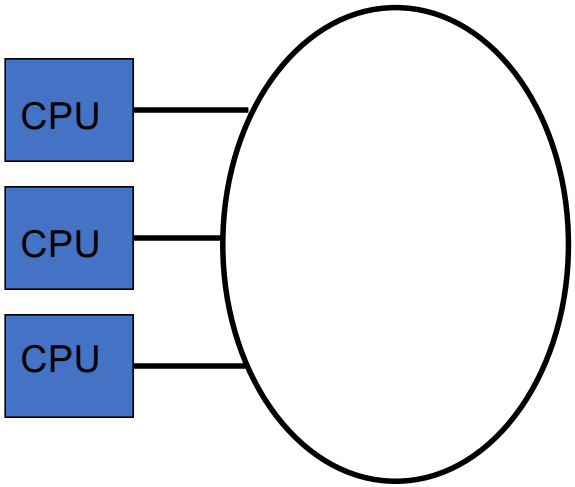
Programming with MPI

- MPI - “message passing interface” is a very popular tool for writing parallel programs in scientific and technical computing.
- Lots of information can be found at
  - https://www.mpi-forum.org/docs/
- Work on standard began in 1992.
- Three generations of standards MPI 1.2, MPI 2.0 and MPI 3.1
  - MPI 1.2 very widely available
  - MPI 2.0 widely available
  - MPI 3.1 generally available
  - MPI 4 under construction

Role of MPI

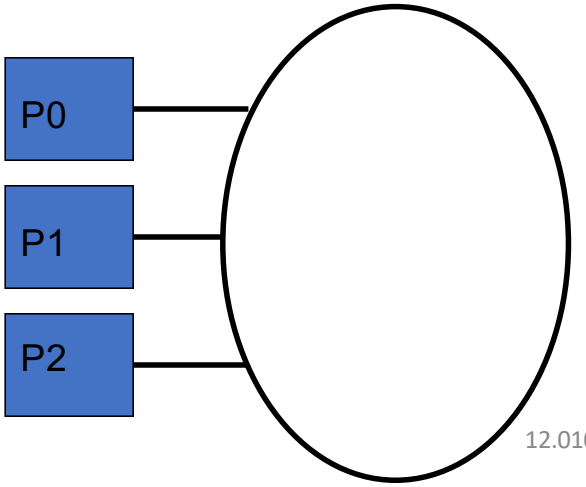
- Supercomputing
  - Most big supercomputing applications (weather/climate, materials/atomic, folding, massive ML) use MPI
    - HPC simulators often use MPI directly, or indirectly (through e.g. ScalaPack, MAGMA etc…)
    - ML applications often use MPI indirectly e.g. through Horovod et….
  - MPI can be used from C, C++, Fortran and from Python, Julia, Octave, Matlab and R.
  - It is the dominant tool for “tightly” coupled parallel codes on clusters and in cloud settings.
  - It can run on laptop, local clusters and on massive supercomputers.
  - It can work with CPU and/or GPU codes.
  - All the shared resources at MIT have MPI available.
  - MPI basics are minimalist and relatively easy to learn. Quite a bit of work is left to application developer to create an MPI program.

Basic hardware and programming model



Typical Hardware

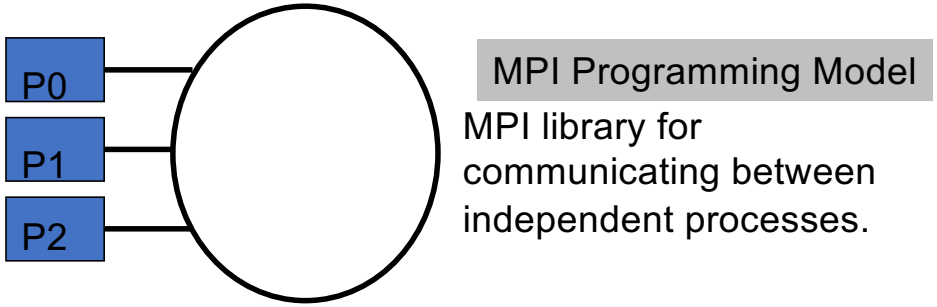
Some communication fabric e.g. network, shared memory etc…



MPI Programming Model

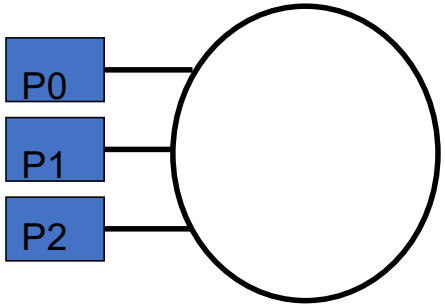
MPI library for communicating between independent processes.

Ingredients of MPI



- Some way to start up multiple processors
- Some way to figure out which process is which
- Some way to send information between and amongst processes
- Some way to synchronize between processes
- These ingredients are provided by library calls and by a special script for starting a program. Both are provided by MPI.

MPI Hello World



MPI Programming Model

MPI library for communicating between independent processes.

- Some way to start up multiple processors
  - **`MPI_Init()`**
- Some way to figure out which process is which
  - **`MPI_CommRank()`**
  - **`MPI_CommSize()`**

MPI_Init(), MPI_CommRank and MPI_CommSize are MPI library calls. Each has online documentation.

MPI Hello World – "classic MPI"

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world! I'm %d of %d\n",
        rank, size );
MPI_Finalize();
return 0;
}
```

Running in parallel after this, state before is undefined.

Each MPI process will get a different rank value. Ranks are in range 0 – (N_p1)

Everybody gets size = N_p

Hands on

fall-2021-12.010/lec22/mpi-hello.c

fall-2021-12.010/lec22/*py

Beyond Python MPI

- MPI also exists for C, C++, Fortran, Julia, R etc…
- Python Dask is an alternate for parallelism that tries to be more integrated with Python.
- Modern Fortran (2008) is a parallel language without any extensions using “co-arrays”.

Hands on

fall-2021-12.010/lec22/*py

Hands on

fall-2021-12.010/lec22/**coarray.F90**

MIT OpenCourseWare

https://ocw.mit.edu

12.010 Computational Methods of Scientific Programming, Fall 2024