Harvard Applied Mathematics 205

Group Activity: Introduction to multithreading\textsuperscript{1}

Yue Sun, Michael Emanuel, Chris Rycroft

September 14 & 16, 2021

\textsuperscript{1}Based off slides from (a) AM205 Fall 2020 by Nick Derr (b) AM225 Spring 2021 Unit 0.
Outline

1 Introduction to parallelization
   High-performance computing mini-glossary
   Why care about multithreading?
   Shared and distributed memory parallelization

2 Examples of shared memory parallelization
   Multithreading efficiency
   OpenMP in C++
   joblib in Python

3 Parallelization by default in Python libraries
   For-loops v.s. vectorization
   Compare the parallel results
   NumPy and JAX examples
High-performance computing mini-glossary

- **High performance computing**: the practice of aggregating computing power in a way that delivers much higher performance than typical setup to solve large problems.

- **CPU**: central processing unit, a collection of circuitry which performs instructions of a computer program, *i.e.* input/output, arithmetic/logical operations.

- **Core**: an individual processing unit in a “multicore” CPU.

- **Thread**: short for a thread of execution, a single sequential flow of instructions within a program; virtually divide the physical core of a CPU into multiple cores.

- **Hyperthreading**: a CPU managing two sets of tasks per core.
High-performance computing mini-glossary

Figure 1: Examples of CPU

Figure 2: A dual-core CPU
Exercise #1: Check the number of threads in your laptop

▶ Approach 1: Check the computer directly

▶ Mac OS: Apple Icon Menu > About This Mac > System Report > Hardware

▶ Windows: Open Task Manager, then select the Performance tab to see the number of cores and logical processors

▶ Approach 2: Command line wizardry with htop

▶ Mac OS: type `htop` in your terminal

▶ Windows: Microsoft Sysinternals Suite is an alternative to htop

---

2You need to install htop first. One way is through Homebrew via command `brew install htop`. 
Why care about multithreading?

Serial processors (CPU) are not getting faster. The emphasis on hardware improvements is on parallelism via multi-core processors.

- **microprocessor trend data**
- faster clock speeds $\Rightarrow$ higher power
- more cores + slower clock speeds can yield **higher** compute power with **lower** power consumption
- higher core counts are becoming more and more widely available
- important to understand how to adapt or write code to take advantage of parallel architecture
- **make full use of our laptops!**

---

*a* Moore’s law: “The transistor density of semiconductor chips will double roughly every 18 months.”

(Moore’s second law: “There is exponential growth in the cost of tools for chip manufacturing.”)

*b* Dennard scaling: “As transistors get smaller, their power density stay constant, so that the power use stays in proportion with area.” It stops mainly due to supply voltage limits, power densities rapidly increase on the chip.
Shared memory and distributed memory

**Shared memory model**
- program is a collection of control threads (dynamically created)
- each thread has private variables and shared variables
- threads can exchange data by reading/writing shared variables
- danger: more than one processor core reads/writes to a memory location—a race condition

**Distributed memory model**
- program is run as a collection of named processes; fixed at start-up
- local address space; no shared data
- logically shared data is distributed (e.g. every processor only has access to a part of rows of matrix)
- explicit communication through send/receive pairs
Shared memory parallelization

Programs apply “divide-and-conquer” to specific tasks, spawning multiple threads to divvy up large tasks when necessary.

A single thread loops sequentially through each spot in memory.
Shared memory parallelization

Programs apply “divide-and-conquer” to specific tasks, spawning multiple threads to divvy up large tasks when necessary.

A single thread loops sequentially through each spot in memory.

Multiple threads traverse different locations in memory concurrently.
Distributed memory parallelization (for comparison)

Programs run as several independent threads, pausing only to periodically communicate amongst each other.

One method of memory distribution is to divvy up the **simulation domain** across the various threads.
Distributed memory parallelization (for comparison)

Programs run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  \[\Rightarrow\] need **communication**

b, c, d send to a
Distributed memory parallelization (for comparison)

Programs run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains $\implies$ need **communication**
- requires “message passing interface” (e.g. OpenMPI)

**Discuss:** benefits? costs?
Multithreading efficiency

Parallelizing decreases calculation time, but there is cost–benefit due to e.g. spawning overhead, memory bandwidth saturation.

Use efficiency on $k$ threads to determine whether to multithread:

$$e_k = \frac{T_1}{kT_k} < 1$$

where $T_k$ is the time to perform a calculation across $k$ threads.

What do efficiency values imply?

- $e_k \approx 1$: savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$: savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$: savings $\ll$ cost (use fewer threads)

How do costs relate to job size $N$?

- overhead for $k$ threads $\propto k$
- as $k$ grows, $k > N$ (threading becomes a bigger job)
- as $N$ grows, problem may become memory bandwidth-limited (implementation-dependent)
OpenMP introduction

- built into all modern versions of GCC, and enabled with the compiler flag: `g++ -fopenmp -o ompexe yourcode.cc`

- Clang has OpenMP support, but not in Apple’s custom version

- on the Mac, you can obtain an OpenMP-capable compiler via the MacPorts and Homebrew (online tutorial about OpenMP)

- standard C++ but with additional `#pragma` commands to denote areas that need multithreading

- by default, OpenMP programs run with all available threads on the machine, but most times we don’t want all of them; we can specify the number of threads: `OMP_NUM_THREADS=4 ./ompexe`
OpenMP example #1

```c
#include <cstdio>

// OpenMP header file with specific thread-related functions
#include "omp.h"

int main() {

#pragma omp parallel
{
   // Variables declared within a parallel block are local
   int i=omp_get_thread_num(),
       j=omp_get_max_threads();

   printf("Hello world from thread %d of %d\n",i,j);
}
}
```
OpenMP example #2

```c
#include <cstdio>
#include "omp.h"

int main() {
    // Add two arrays of length N
    static const int N=100;
    double v1[N], v2[N], v3[N];

    // Assign values to v1 and v2
    // ...

    #pragma omp parallel for
    for(int i=0; i<N; i++) {
        v3[i]=v1[i]+v2[i];
    }
}
```

- **pragma omp** identifies preprocessor OpenMP directive: compiler will implement instructions if OMP is supported and appropriate compile flags (e.g. `-fopenmp`) are present
- parallel spawns multiple threads
- for distributes loop iterations among spawned threads, i.e. thread 1: \( i = [0, N/2) \), thread 2: \( i = [N/2, N) \)

**Discussion:** why can we parallelize this for-loop in this way? ➞ no one iteration depends on another! “embarrassingly parallel”
OpenMP example #3

```c
#include <cstdio>
#include "omp.h"

int main() {
    // Compute 1*1+...+1024*1024
    unsigned int c=0;
    #pragma omp parallel for
    for(unsigned int i=1;i<1025;i++) {
        c=i*i+c;
    }
    printf("Sum=\%u\n",c);
}
```

**Race condition**
- two threads access the same memory, leading to unpredictable behavior
- the code is legitimate if interpreted serially
- unpredictability is due to conflicts between the loading/storage of c, and multiple threads can write to the same memory space

**Discussion:** is this parallelization correct?
OpenMP example #3

```c
#include <cstdio>
#include "omp.h"

int main() {

  // Compute 1*1+...+1024*1024
  unsigned int c=0;
  #pragma omp parallel for shared(c)
  for(unsigned int i=1;i<1025;i++) {
    int d=i*i;
    #pragma omp atomic
    c+=d;
  }

  printf("Sum=%u\n",c);
}
```

- **solution 1: shared**
  - mark `c` as a shared variable among all threads (this is actually the default)

- **solution 1: atomic**
  - ensure the next line is executed as one unit
  - only one thread access `c` in memory to update its value
  - only work for very simple statements
  - fast, but not as fast as a regular operation
OpenMP example #3

```c
#include <cstdio>
#include "omp.h"

int main() {
    // Compute 1*1+...+1024*1024
    unsigned int c=0;
    #pragma omp parallel for
    for(unsigned int i=1;i<1025;i++) {
        int d=i*i;
        #pragma omp critical
        {
            printf("More complex code.");
            c+=d;
        }
    }
    printf("Sum=%u\n",c);
}
```

- solution 2: **critical**
  - mark a statement or block to only be processed by one thread at a time
  - unlike atomic, work for general blocks of code
  - come with a performance penalty—threads will stand idle waiting for the block to become free
OpenMP example #3

```c
#include <stdio.h>
#include "omp.h"

int main() {
    // Compute 1*1+...+1024*1024
    unsigned int c=0;

    #pragma omp parallel for reduction(+:c)
    for(unsigned int i=1;i<1025;i++) {
        c=i*i+c;
    }

    printf("Sum=%u\n",c);
}
```

- **solution 3:** reduction
  - mark a variable for accumulation across threads
  - give each thread a private \(c\) variable, at the end, set the global \(c\) variable equal the sum of the private variables
  - cleanest solution for this particular scenario

OMP is not smart enough to avoid multiple threads reading/writing to the same location at the same time. You have to help it!
OpenMP keywords

```c
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) schedule(static)
```

- specify number of threads
  - default: OMP_NUM_THREADS environment variable
- declare private variables
- set thread-spawning routine
  - static: assign set number of iterations to each thread at outset (low overhead: use if each iterate requires roughly equal work)
  - dynamic: assign iterates to threads as needed (high overhead: use if work per iterate can vary widely)
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared
- declare shared variables

Let's look at OpenMP in action!
**joblib example**

We can divide up embarrassingly parallel\(^3\) for-loops using joblib in Python, which is a wrapper for the multiprocessing module.

```python
import joblib as jl

# Function for sum of i-th components
def sum(i):
    global a1
    global a2
    return a1[i]+a2[i]

# Generator of functions and arguments
fgen=(jl.delayed(sum)(i)
    for i in range(len(a1)))

# Parallel sum
a3=jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

- require the joblib module
- function to produce requested value for each item in loop
- generator to produce tuple with pointer to function and args for each item in loop
- parallel object take in args (e.g. n_objs) and generator, return array with result of passed function at each spot in loop

---

Bonus: os.fork() example

When calling `os.fork()`, OS creates an **exact** clone of the process. There will be two process that are almost exactly the same.

```python
import os
import sys
tot=2001 # Total jobs
P=16 # Available slots
pid=[None]*(P+1)
t=1 # Set thread number
queue=P==1 # Set queue status

# Loop over jobs
for i in range(tot):
    print("Job %d to slot %d"%(i,t))
j=os.fork() # Fork a child process

    if j==0:
        ### Write your code here ###
        os._exit(0)

    pid[t]=j

    # If reach the max number of slots,
    # then wait for one job to finish

if queue:
    # Find available slots
    npid=os.wait()[0]
t=1
    while pid[t]!=npid:
        t+=1
        if t>P:
            print("PID return error")
            sys.exit()

    # Otherwise, move onto the next slot
else:
    t+=1
    queue=t>=P

# Wait for all remaining jobs to finish
if queue:
    for i in range(P-1):
        os.wait()
else:
    for i in range(t-1):
        os.wait()
```

---

4It is not strictly shared memory parallelization, but it is extremely useful in running Python programs parallel. Best to use `os.fork()` if the same program needs to be run multiple times, e.g. plot 100 frames of velocity field while the plotting script stays the same.
Exercise #2: Parallelize the code with OpenMP or joblib

Parallelize a matrix multiplication routine. Work in groups and complete the Python or the C++ code.
For-loops are more expensive in interpreted languages

The spawning of parallel threads take place at a higher level in Python, so there is a much higher overhead cost.

OpenMP: compiler $\rightarrow$ machine code

joblib: interpreter $\rightarrow$ system processes $\rightarrow$ precompiled libraries (machine code)

**Discuss: what properties make a job more (or less) appropriate for joblib?**

Often, modules such as NumPy already have precompiled code which is parallelized. We don’t have to do it! Write vectorized code.
Vectorization as code with “invisible” for-loops

For two NumPy arrays a and b, we can do the same calculation with an explicit for-loop in Python or vectorized NumPy functions that call precompiled C++ code.

- **sum**
  ```python
  for i in range(len(a)):
      c[i] = a[i] + b[i]
  c = a + b
  ```

- **dot product**
  ```python
d = 0
  for i in range(len(a)):
      d = d + a[i] * b[i]
  d = a.dot(b)
  ```

- **elementwise multiply**
  ```python
  for i in range(len(a)):
      c[i] = a[i] * b[i]
  c = a * b
  ```
Vectorized code is faster

First and foremost, because of precompiled library functions, but these libraries are often parallelized with OpenMP as well.

```
M.dot(v) #pragma omp parallel for
for (int r=0;r<N;r++) {
    for (int c=0;c<N;c++) {
        out[r] += M[r][c] * v[c];
    }
}
```

(NumPy code)     (interpreter call)     (machine code equivalent to above, compiled)

Easiest way to set thread number for vectorized NumPy? OMP_NUM_THREADS environment variable!
Compare the parallel results across C++ (OMP), Python (joblib), and Python (NumPy) for matrix multiplication.
NumPy and JAX examples

See the Colab notebook for examples.
Summary

- CPU → core → thread
- shared memory vs. distributed memory: choose appropriately
- multithreading efficiency: whether it is worth the effort
- compiled languages (e.g. C++)
  - it is often a good idea to multithread the (main) for-loops
  - speed up with just one line: #pragma omp parallel for
  - threads automatically divvy up jobs, but they are not smart enough to handle everything by themselves: use OMP keywords to help
  - compiler flexibility (multithreaded code can work on non-OMP compilers)
- interpreted languages (e.g. Python)
  - often done by default in calls to precompiled library functions
  - eschew for-loops: vectorize, vectorize, vectorize!
  - help your CPU out in determining number of threads v.s. cores

When in doubt, or suspect any problems, always run with one thread to compare.