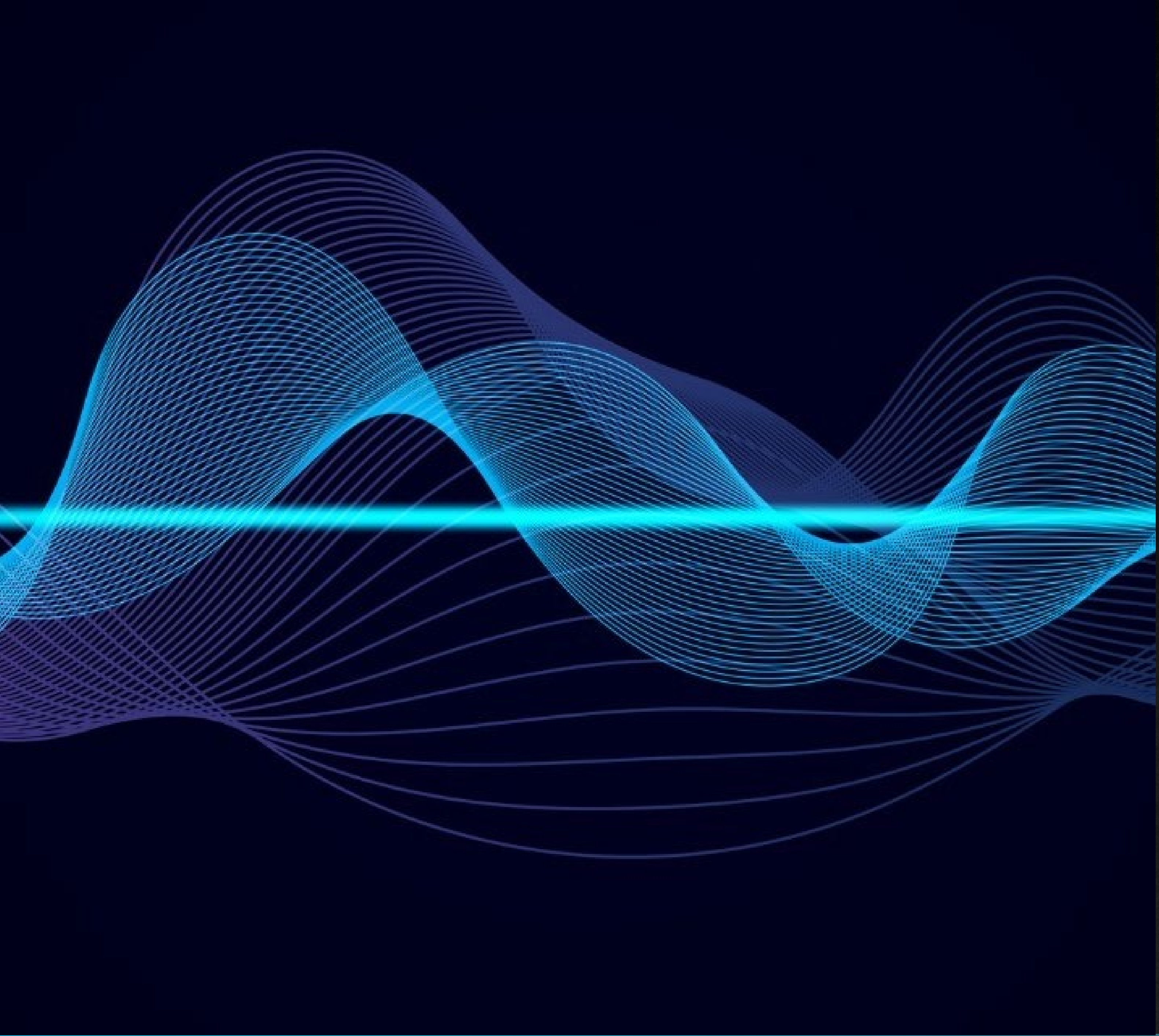


Computational Physics

PHYS 6260

Parallel Programming OpenMP

Announcements:

- HW6: Due Friday 2/28
- Project proposal due Friday 3/7

Lecture Outline

We will cover these topics

- Types of parallelism
- OpenMP

Optimization

- Getting performance out of your code means
  - Picking the right algorithm
  - Implementing the algorithm efficiently
- We previously covered about picking the most appropriate algorithm and saw some examples of speed-ups
- For performance in the implementation
  - We need to understand how the computer’s CPU/GPU and architecture works
  - To exceed a single core’s performance, we must go parallel

CPU + Memory System

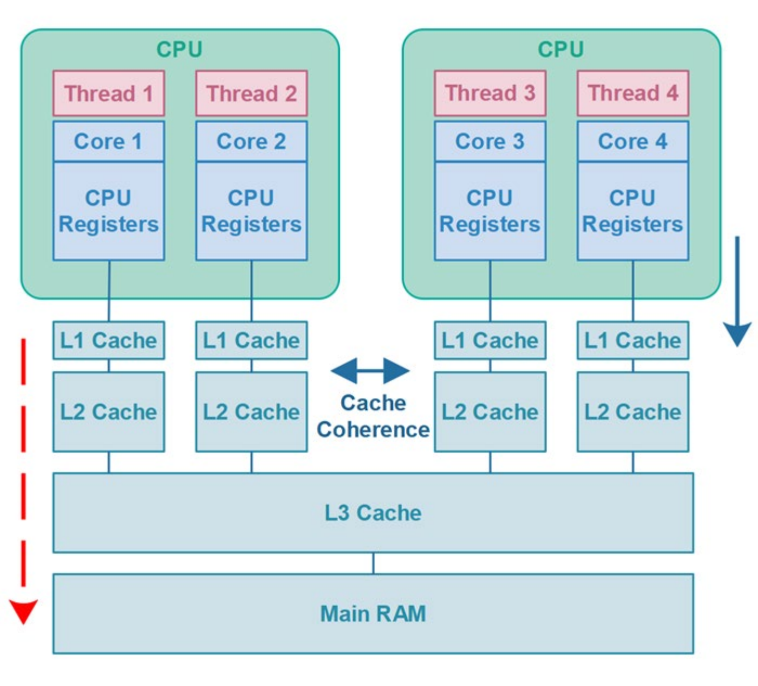
- Memory hierarchy
  - Data is stored in main memory (RAM)
  - Below this, there are multiple levels of cache on the CPU (L3, L2, L1)
  - A line of memory is moved into cache – you amortize the costs if you put all the data in the line
  - Data is moved to the registers in the CPU – this is where the computation occurs

- It is expensive to move data from main memory to the registers
  - We need to exploit the cache
  - For arrays, loop over data such that you operate on elements that are adjacent in memory
  - In Python, you don’t need to worry about this
  - But for compiled languages, you need to organize your loops optimally

CPU + Memory System

Computer Action	Avg Latency	Normalized Human Time
3GhzCPU Clock cycle 3Ghz	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
RAM access	70 - 100ns	3.5 to 5.5 min
NVMe SSD I/O	7-150 µs	2 hrs to 2 days
Rotational disk I/O	1-10 ms	11 days to 4 mos
Internet: SF to NYC	40 ms	1.2 years
Internet: SF to Australia	183 ms	6 years
OS virtualization reboot	4 s	127 years
Virtualization reboot	40 s	1200 years
Physical system reboot	90 s	3 Millenia

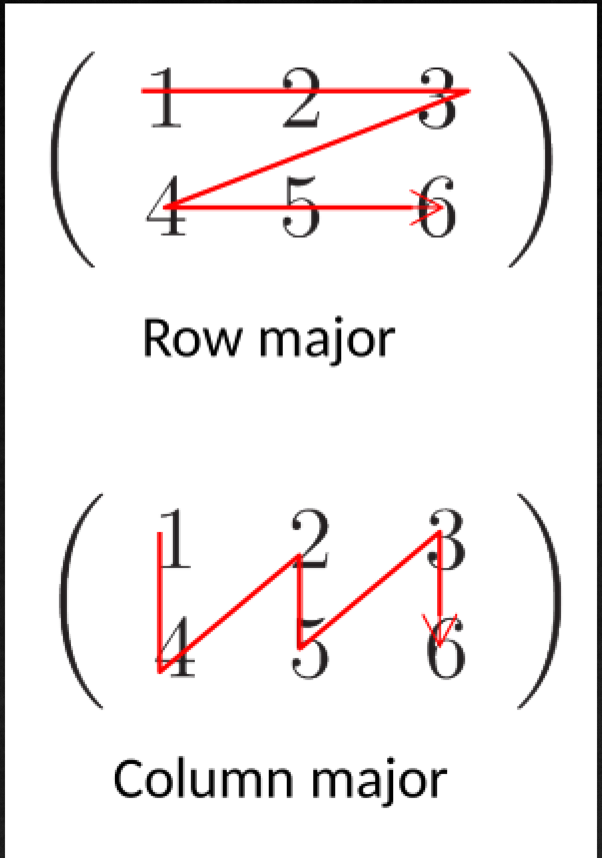
Table 1: Computer Time in Human Terms [i]



Arrays

- Row vs. Column major: A(m,n)
  - First index is called the row
  - Second index is called the column
  - Multi-dimensional arrays are flattened into one-dimensional sequence for storage
- Row-major (C, python): rows are stored one after the other
- Column-major (Fortran, matlab): columns are stored one after the other

- Ordering matters for:
  - Passing arrays between languages
  - Deciding which index to loop over first



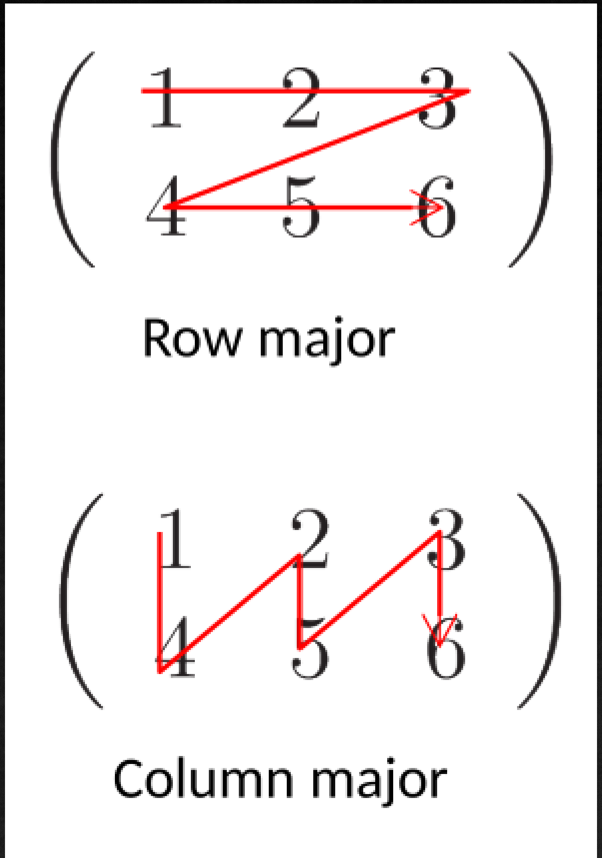
Arrays

- This is why in C, you'd want to loop as

```
double A[M][N];
for (i = 0; i < M; i++) {
   for (j = 0; j < N; j++) {
       A[i][j] = …;
   }
}
```

- And in Fortran / matlab

```
double precision :: A(M,N)
do j = 1, N
    do i = 1, M
         A(i,j) = …
    enddo
enddo
```



Types of parallelism

- Flynn's taxonomy classifies computer architectures
- 4 classifications: single or multiple data; single or multiple instruction

- Single instruction, single data (SISD)
  - Typical application on your computer – no parallelism
- Single instruction, multiple data (SIMD)
  - The same instruction set is done to multiple pieces of data all at once
  - Compile-time vectorization optimization; GPUs
- Multiple instructions, single data (MISD)
  - Not very interesting or useful
- Multiple instructions, multiple data (MIMD)
  - This is what we classify as parallel computing

Types of parallelism

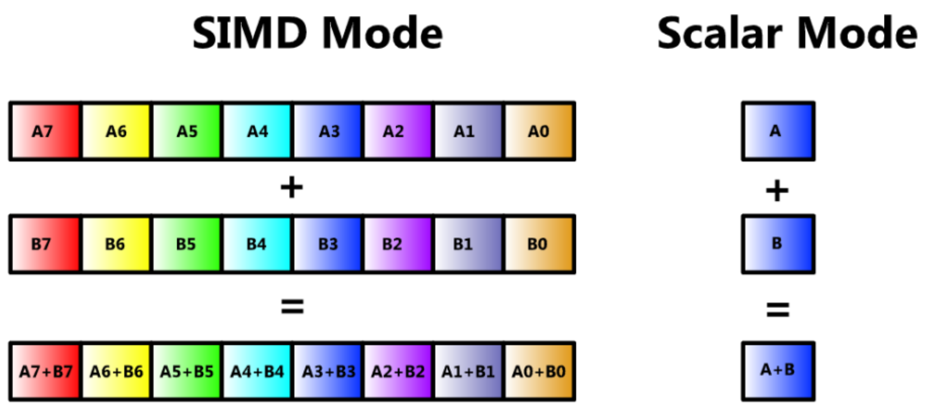


Figure 3. *SIMD versus scalar operations*

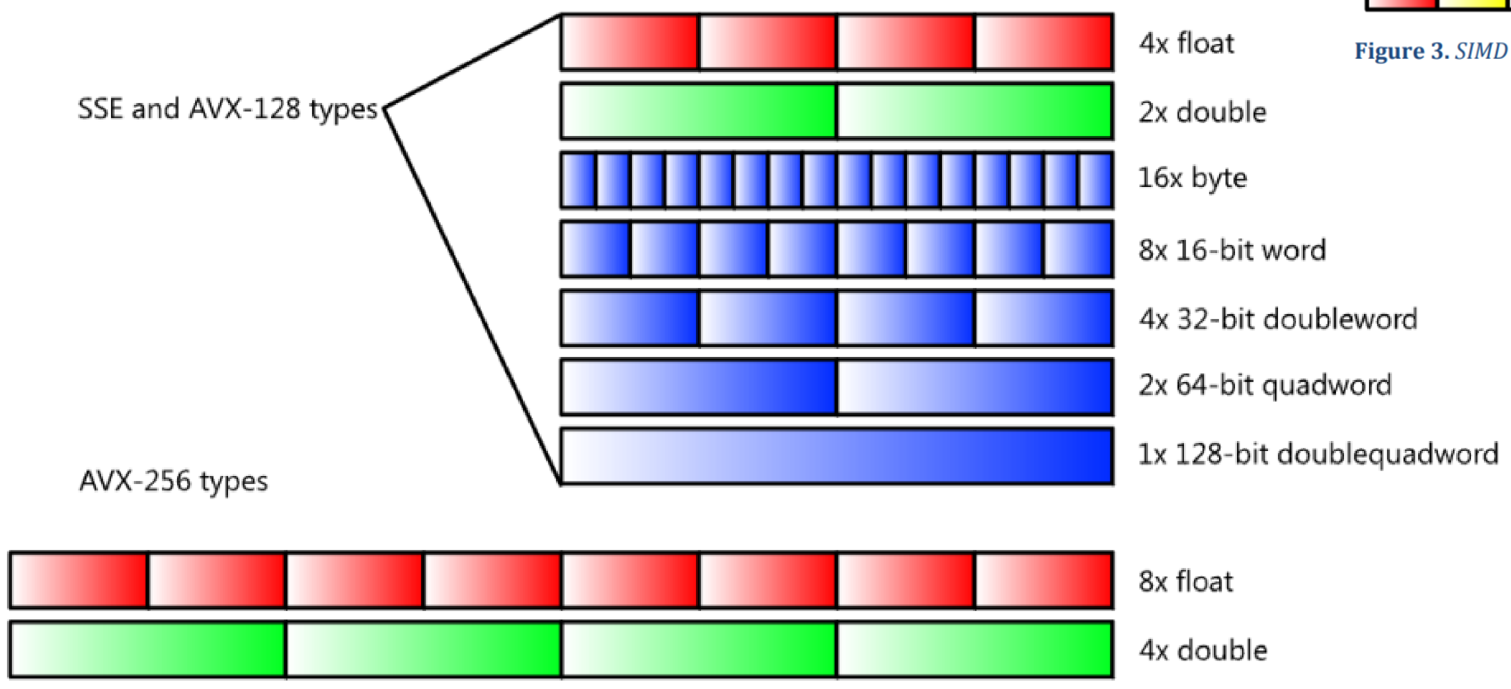


Figure 2. *Intel® AVX and Intel® SSE data types*

source

Trivially parallel

- Sometimes our tasks are trivially parallel
  - No communication is needed between processes
- Examples: ray tracing, Monte Carlo, parameter sweeps
  - Each realization can do its work independently
  - At the end, we may need to do some simple processing of all of the results
- Large data analysis
  - Imagine that you have many datasets and a reduction pipeline to work on them
  - Use multiple processors to work on the different data files as resources become available
  - Each file is processed on a single core

Trivially parallel with shell scripts

- Example: data analysis – launch independent jobs
- This can be done through a shell script – no libraries necessary
- Loop over files
  - Run jobs until all of the processors are full
  - Use lock files to indicate a job is running
  - When resources become free, start up the next job
  - Let’s look at some code

- Also see GNU parallel – very useful to run multiple commands at once
- Example: sorting all files named out*list by the 8th column in reverse order

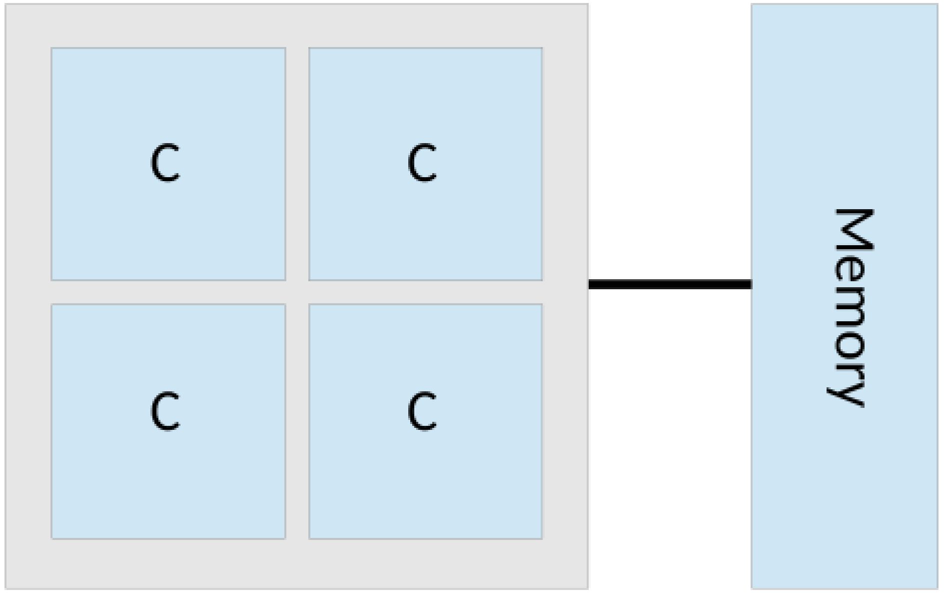
```
$ ls out*list | parallel sort -gr -k 8 -o sorted_{} {}
```

How do we make our code parallel?

- There is no simple compiler flag "—make-this-parallel"
- We have to understand the algorithm and determine what parts are amenable to parallelism
- However, if the bulk of your work is in one specific piece (say, solving a linear system), you may get all that you need by using a library that is already parallel
  - Example: hypre for solving linear systems; fftw for FFTs
  - This will require minimal changes to your code

Shared Memory / OpenMP

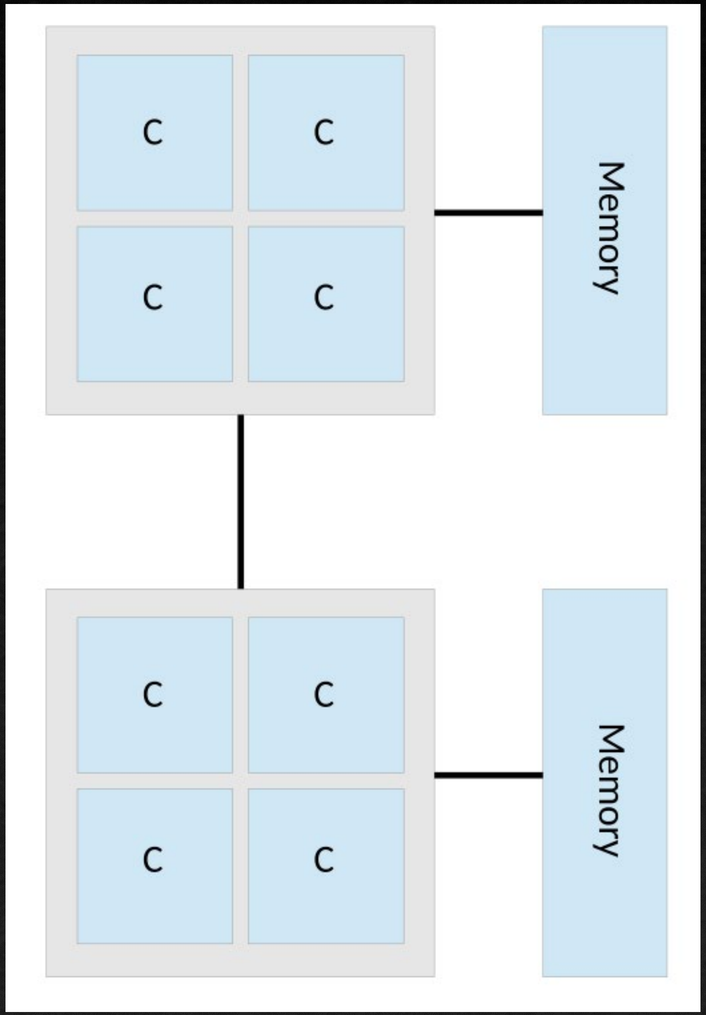
- Nodes consist of one or more multi-core (2-128) CPUs
- Everything can access the same pool of memory



Single 4-core chip and its pool of memory

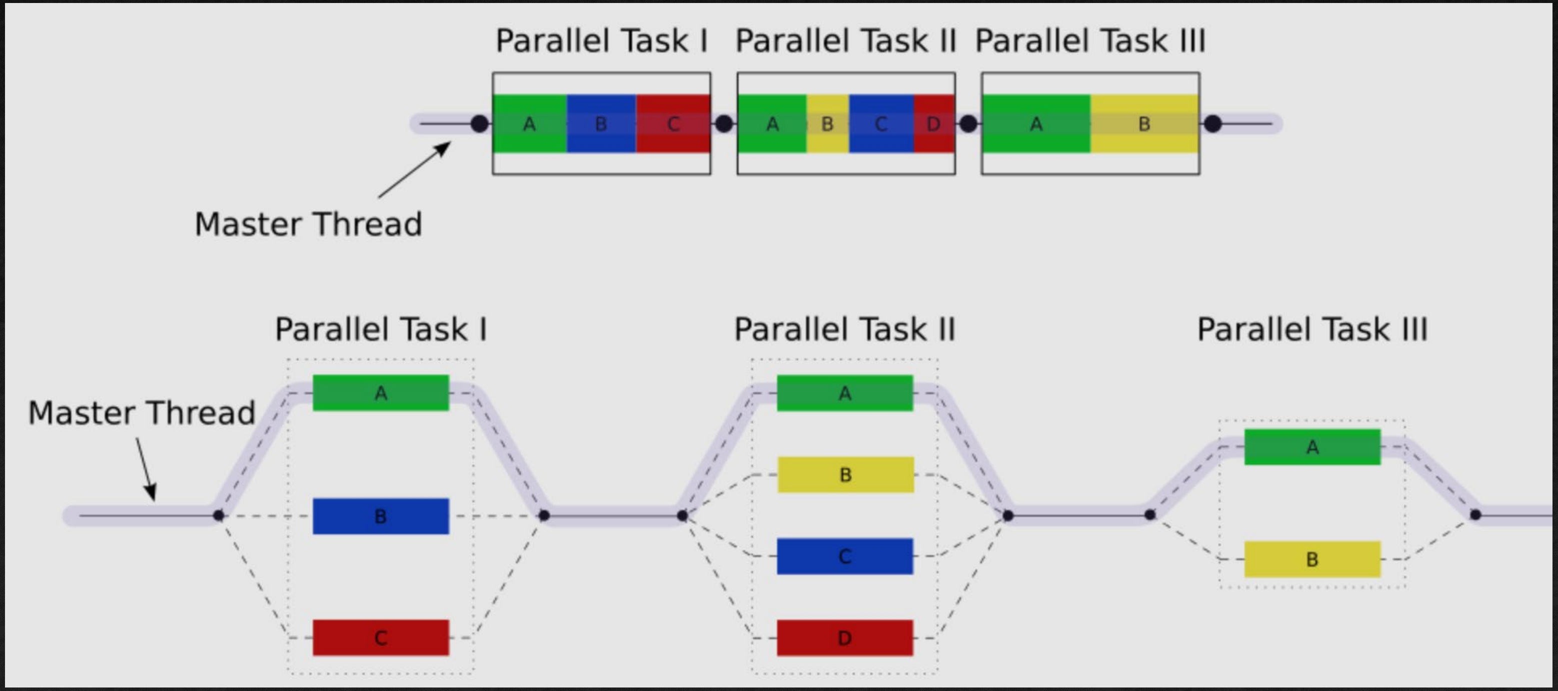
Shared Memory / OpenMP

- Some machines are more complex – multiple chips each with their own pool of local memory
- Latency may be higher when going “off-chip”
- Best performance will require you to know the machine’s architecture and optimizing the run-time parameters for it



Shared Memory / OpenMP

- When using OpenMP, threads are spawned as needed
- When you run the program, there is one thread – the main thread
- When you enter a parallel region, multiple threads run concurrently



Shared Memory / OpenMP

- OpenMP is initiated through directives or pragmas
- Look like comments unless you tell the compiler to interpret them
- Environment variable OMP_NUM_THREADS sets the number of threads
- Support for C, C++, Fortran
- Hello world

```
#include <stdio.h>
void main() {
    #pragma omp parallel
    printf(“Hello world\n”);
}
```

- Compile with gcc –o hello –fopenmp hello.c

Shared Memory / OpenMP

- In addition to using pragmas, there are a few functions that OpenMP provides to get the number of threads, the current thread, etc.

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

void main() {

  int nthreads, thread_num;

#ifdef _OPENMP
  nthreads = omp_get_num_threads();
  thread_num = omp_get_thread_num();
  printf("[thread %d] Outside of parallel region = %d\n", thread_num, nthreads);
#pragma omp parallel
{
  nthreads = omp_get_num_threads();
  thread_num = omp_get_thread_num();
  printf("[thread %d] Inside of parallel region = %d\n", thread_num, nthreads);
}
#else
  printf("OpenMP not enabled.\n");
#endif
}
```

Shared Memory / OpenMP

- Most (all?) modern compilers support OpenMP
- However, the performance across them can vary
- GCC does a reasonable job. Intel compilers are the fastest (this also includes SIMD vectorization)

- There is an overhead associated with spawning threads
- You will have to experiment to see if parallelism is worth it
- Some regions of code may not have enough work to offset the overhead

Shared Memory / OpenMP

- There will be a systemwide default for OMP_NUM_THREADS

- Things will still run if you use more threads than cores available
- But don't do this!
- This is called oversubscription, and will cause work to queue up
- It's better to use a maximum of 1 OpenMP thread per core

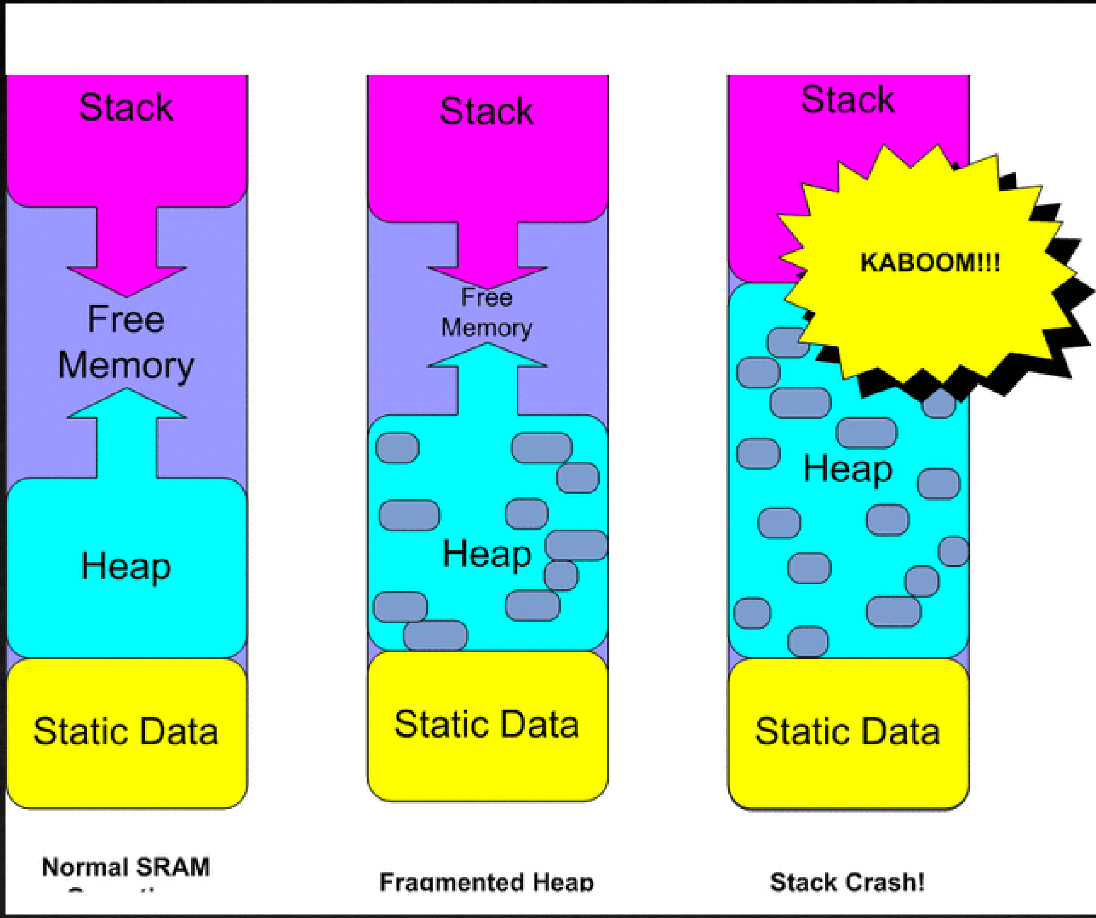
- Scaling: if you double the number of cores, does the code take ½ the time?

Aside about memory: Stack vs. Heap

- Memory allocated at compile time is put on the stack, e.g. double a[1000];

- Stack memory has a fixed and somewhat small size
  - Managed by the operating system
  - You don't need to clean up / free this memory

- Dynamic allocation puts the memory on the heap
  - Much bigger pool
  - You are responsible for deallocating / freeing the memory
  - Otherwise, you will have memory leaks and the memory footprint will continuously increase



Shared Memory / OpenMP: Loops

- Parallel example of matrix multiplication
- Notice the private() calls
- Inside the loop, all threads will have access to all of the variables declared in the main program
- For some variables, we want a private copy on each thread, like the loop counters
  - These are put in the private() clause

```
#pragma omp parallel private(i,j,n)
 {
 n = 0;
 #pragma omp for
 for (j = 0; j < N; j++) {
   for (i = 0; i < N; i++, n++) {
     a[n] = i + j;
   }
   x[j] = j;
   b[j] = 0.0;
 }

// Multiply
 n = 0;
 #pragma omp for
 for (j = 0; j < N; j++) {
   for (i = 0; i < N; i++, n++) {
     b[i] = b[i] + a[n] * x[j];
   }
 }
```

Shared Memory / OpenMP: Reduction

- Suppose that you are finding the minimum value of something or summing
- The loop over the data are spread across threads
- How do we get the data from each thread back to a single variable that all threads see?
- reduction() clause
  - Has both shared and private behaviors
  - Compiler ensures that the data is synchronized at the end

Shared Memory / OpenMP: Reduction

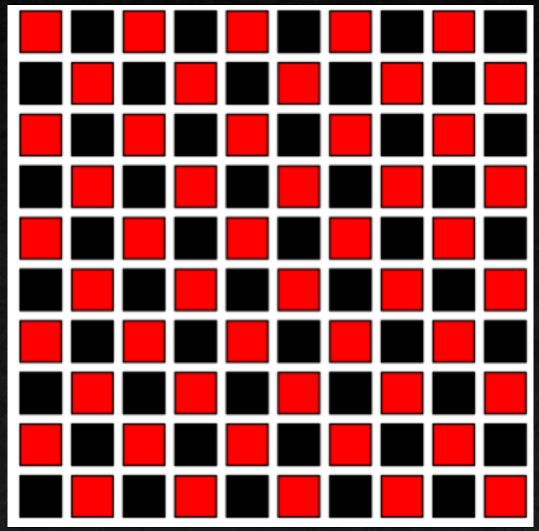
```
sum = 0;
#pragma omp parallel for reduction(+:sa, sb) reduction(max:m)
for (i = 0; i < N; i++){
 sa += A[i];
 sb += B[i];
 if (A[i]>m) m = A[i];
 m = B[i]>m?B[i]:m;  // same result as: if (B[i]>m) m = B[i];
}
```

sa will be the sum over array A
sb will be the sum over array B
m will be the max over both arrays

OpenMP Example: Relaxation

- Remember in the relaxation method to solve an elliptic PDE, we have

$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - (\Delta x)^2 f_{i,j}\right)$$



- Red-black Gauss-Seidel:
  - Update in place
  - First update the red cells with the black cells unchanged
  - Then update the black cells with the red cells unchanged

- Let's inspect a relaxation solver (in C). Remember that OpenMP is only supported for compiled programs.