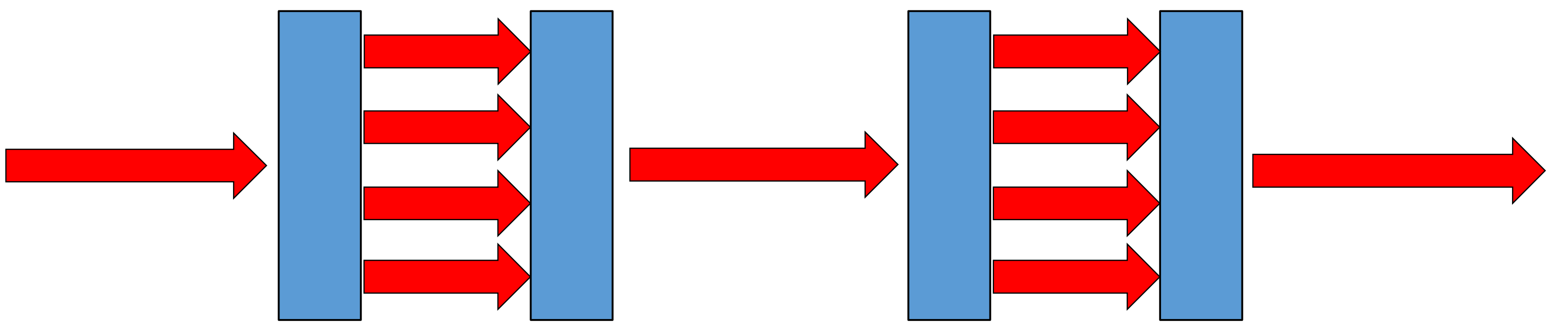
Introduction to Parallel Programming with OpenMP

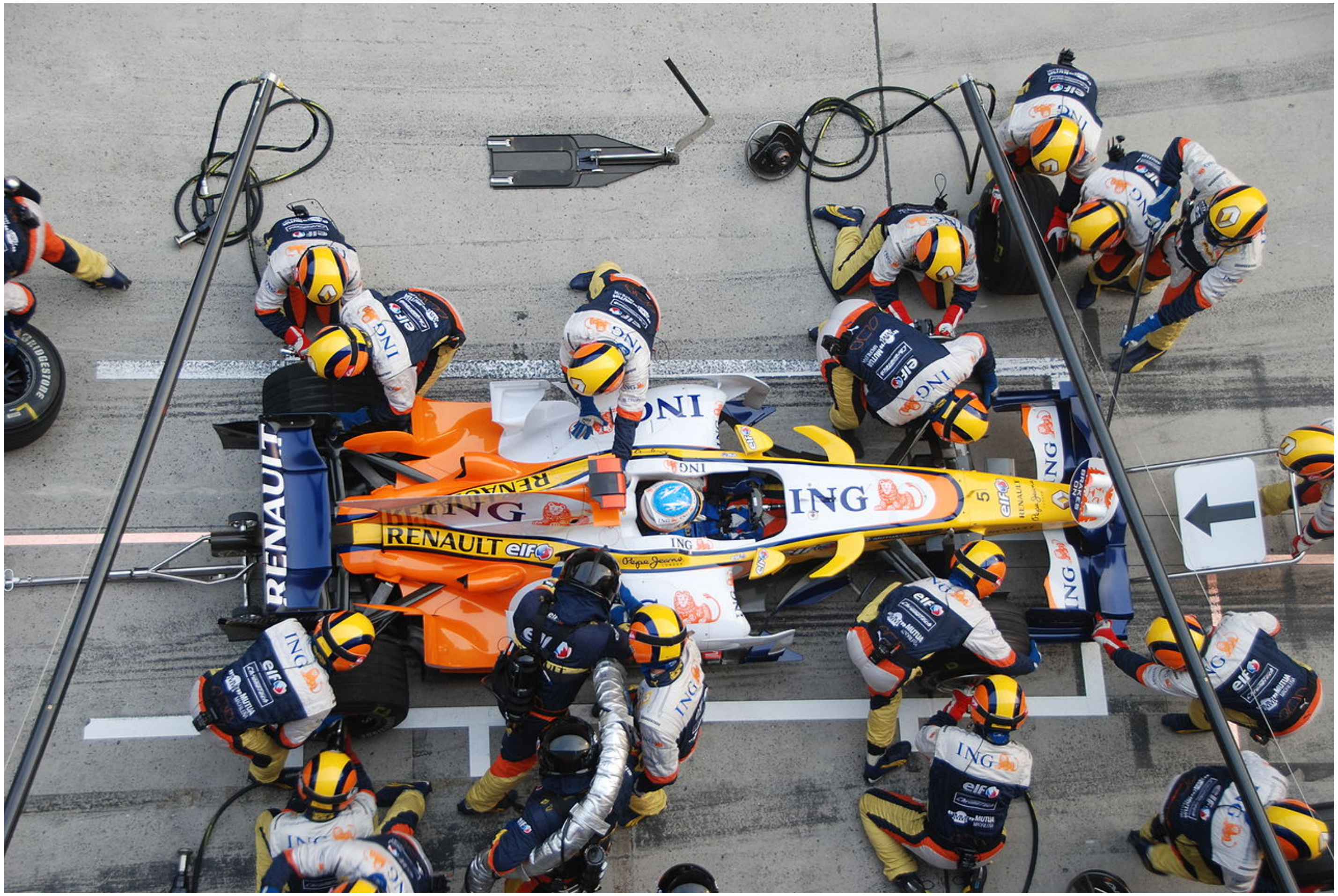


Reference: Charles Augustine
October 29, 2018

OFFICE of INFORMATION TECHNOLOGY

PICSciE

Parallel Programming Analogy



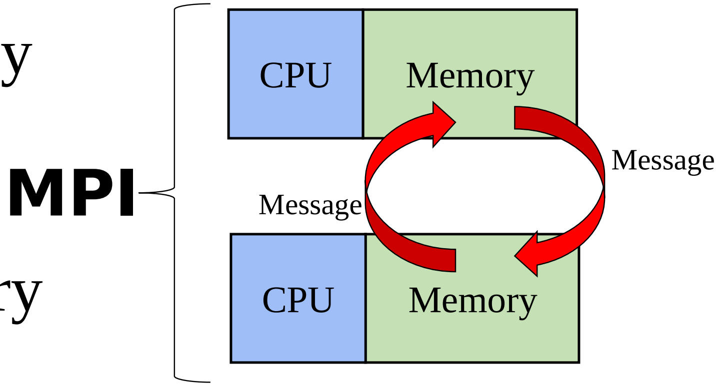
Source: Wikapedia.org

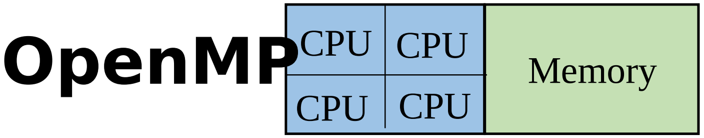
Computer Architecture

- As you consider parallel programming understanding the underlying architecture is important
- Performance is affected by hardware configuration
  - Memory or CPU architecture
  - Numbers of cores/processor
  - Network speed and architecture

MPI and OpenMP

- MPI – Designed for distributed memory
  - Multiple systems
  - Send/receive messages
- OpenMP – Designed for shared memory
  - Single system with multiple cores
  - One thread/core sharing memory
- C, C++, and Fortran
- There are other options
  - Interpreted languages with multithreading
    - Python, R, matlab (have OpenMP & MPI underneath)
  - CUDA, OpenACC (GPUs)
  - Pthreads, Intel Cilk Plus (multithreading)
  - OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)

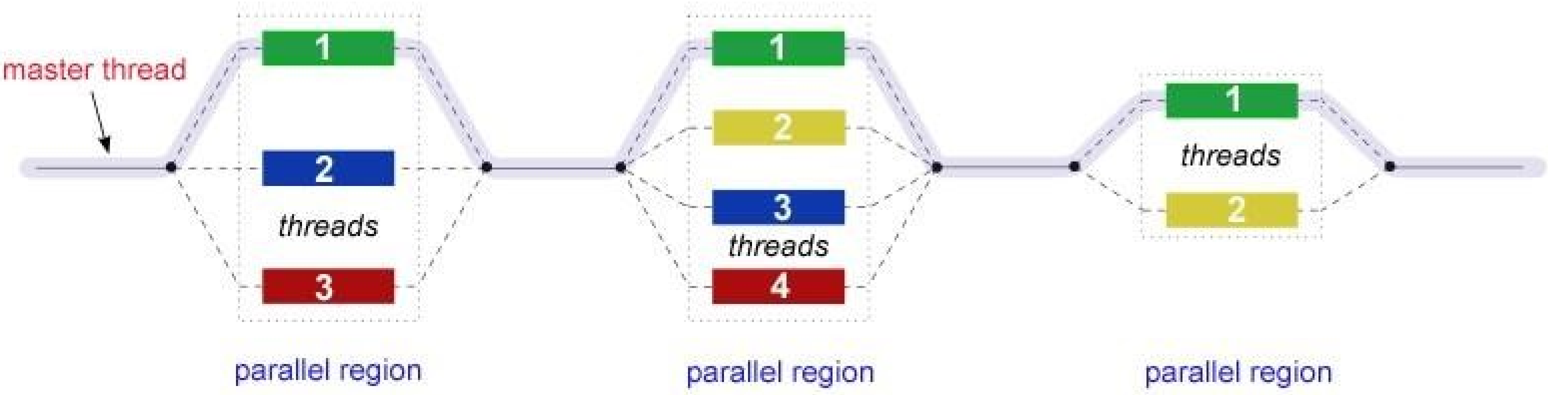




OpenMP

- What is it?
  - Open Multi-Processing
  - Completely independent from MPI
  - Multi-*threaded* parallelism
- Standard since 1997
  - Defined and endorsed by the major players
- Fortran, C, C++
- Requires compiler to support OpenMP
  - Nearly all do
- For shared memory machines
  - Limited by available memory
  - Some compilers support GPUs

Fork - Join Model:Three Components:



Preprocessor Directives

- Preprocessor directives tell the compiler what to do
- Always start with #
- You’ve already seen one:

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

- OpenMP directives tell the compiler to add machine code for parallel execution of the following block

```
#pragma omp parallel
```

- “Run this next set of instructions in parallel”

OpenMP compiler directives Always start with # are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Some OpenMP Subroutines

```
int omp_get_max_threads()
```

- Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

- Returns number of threads in current team\\

```
int omp_get_thread_num()
```

- Returns thread id of calling thread
- Between 0 and omp_get_num_threads-1

Control the Number of Threads

- Parallel region
  #pragma omp parallel **num_threads**(*integer*)
- Run-time function
  omp_set_num_threads()
- Environment variable
  export OMP_NUM_THREADS=n

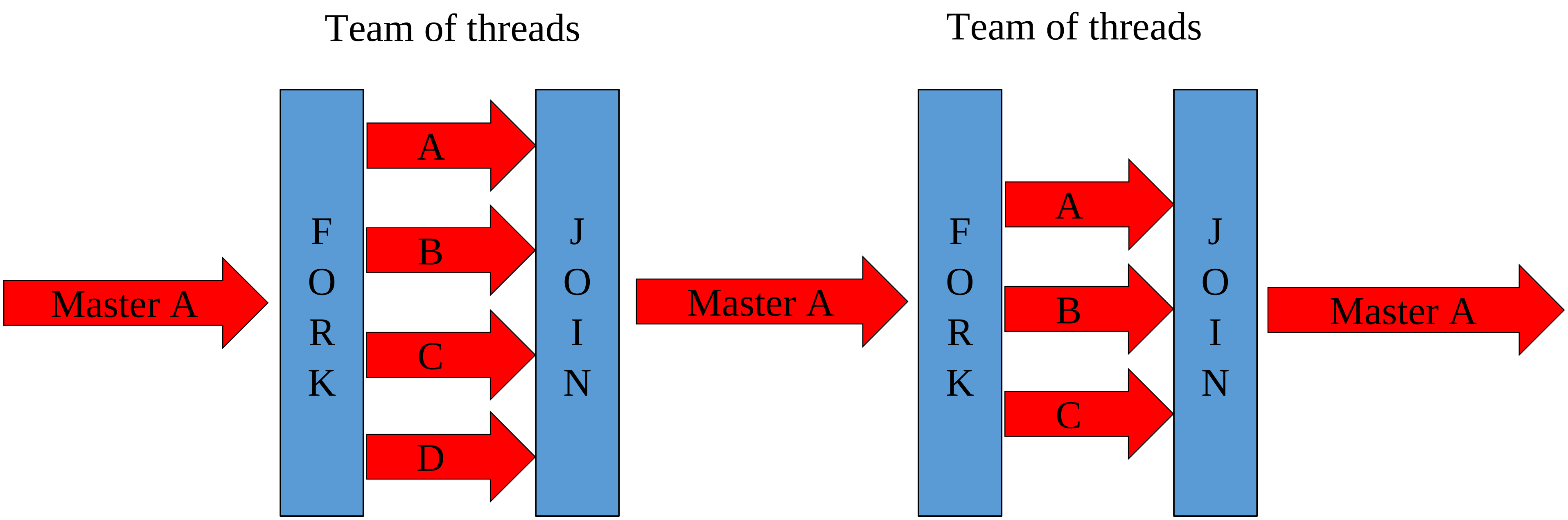
higher priority

Process vs. Thread

- MPI = Process, OpenMP = Thread
- Program starts with a single process
- Processes have their own (private) memory space
- A process can create one or more threads
- Threads created by a process share its memory space
  - Read and write to same memory addresses
  - Share same process ids and file descriptors
- Each thread has a unique instruction counter and stack pointer
  - A thread can have private storage on the stack

OpenMP Fork-Join Model

- Automatically distributes work
- Fork-Join Model



OpenMP Code Structure

```
#include<omp.h>
int main ()  {

int var1, var2, var3;
  Serial code
        .
        .
        .
Beginning of parallel region. Fork a team of threads.
   Specify variable scoping
   #pragma omp parallel private(var1, var2) shared(var3)
      {
      Parallel region executed by all threads          .
      Other OpenMP directives
      Run-time Library calls
      All threads join master thread and disband
      }
Resume Serial Code
     .
     .
     .
}
```

OpenMP Constructs

- Parallel region
  - Thread creates team, and becomes master (id 0)
  - All threads run code after
  - Barrier at end of parallel section

```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     num_threads (integer)


   structured_block
```

(not a complete list)

OpenMP Hello World

```
#include <omp.h> //<-- necessary header file for OpenMP API
#include <stdio.h>

int main(int argc, char *argv[]){

   printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
   {
      //Code here will be executed by all threads
      printf("Hello World from thread %d\n", omp_get_thread_num());
   }

   return 0;
}
```

Running OpenMP Hello World

```
[user@adroit4]$ gcc -o hello_world_omp hello_world_omp.c -fopenmp
```

Compiler flag to enable OpenMP

(-fopenmp for gcc)
(-qopenmp-stubs for icc serial)

Environment variable defining max threads

```
[user@adroit4]$ export OMP_NUM_THREADS=4
[user@adroit4]$ ./hello_world_omp
OpenMP running with 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 2
Hello World from thread 3
```

- OMP_NUM_THREADS defines run time number of threads can be set in code as well using: omp_set_num_threads()
- OpenMP may try to use all available cpus if not set (On cluster–Always set it!)

OMP Parallel Clauses 1

```
#pragma omp parallel if (scalar_expression)
```

- Only execute in parallel if true
- Otherwise serial

```
#pragma omp parallel private (list)
```

- Data local to thread
- Values are not guaranteed to be defined on exit (even if defined before)
- No storage associated with original object

OMP Parallel Clause 3

```
#pragma omp shared (list)
```

- Data is accessible by all threads in team
- All threads access same address space

- Improperly scoped variables are big source of OMP bugs
  - Shared when should be private
  - Race condition

```
#pragma omp default (shared | none)
```

- Tip: Safest is to use default(none) and declare by hand

Shared and Private Variables

- Take home message:
  - Be careful with the scope of your variables
  - Results must be independent of thread count
  - Test & debug thoroughly!

- Important note about compilers
  - C (before C99) does not allow variables declared in for loop syntax
    - Compiler will make loop variables private
    - Still recommend explicit

C

```
#pragma omp parallel private(i)
   for (i=0; i<N; i++) {
      b = a + i;
   }
```

C++

```
#pragma omp parallel
   for (int i=0; i<N; i++) {
      b = a + i;
   }
```

Automatically private

Private Variables 1

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(shared)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_1.c -o omp_private_1
[user@adroit3]$ export OMP_NUM_THREADS=1
[user@adroit3]$ ./omp_private_1
a=50 b=1049 (expected a=50 b=1049)
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_1
a=50 b=799 (expected a=50 b=1049)
```

Private Variables 2

```
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(none) private(i) private(a) private(b)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_2.c -o omp_private_2
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_2
a=50 b=0 (expected a=50 b=1049)
```

OMP parallel for example 1

Example 1: Table Computation

Below code prints table using OpenMP for loop Parallelization

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

int main() {
        int num;
        int i;
        printf("Table [PROMPT] Enter Your Number: "); scanf("%d",&num);

        #pragma omp parallel num_threads(10)
        #pragma omp for
        for(i=0;i<10000;i++) {
                printf("Table [INFO] Thread ID: %d | %d X %d = %d \n", omp_get_thread_num(), i, num,
i*num );
        }

        return 0;
}
```

Step 1: Copy this code into a .c File.
Step 2: Compile the using the following command

```
gcc -o table table.c -fopenmp
```

Step 3 (optional) set environmental variables
Step 4: Run the code and observe the processor using system monitor.

Example 2: Setting Runtime Variables with For Loops
Below code will demonstrate how we can set runtime library routines with OpenMP

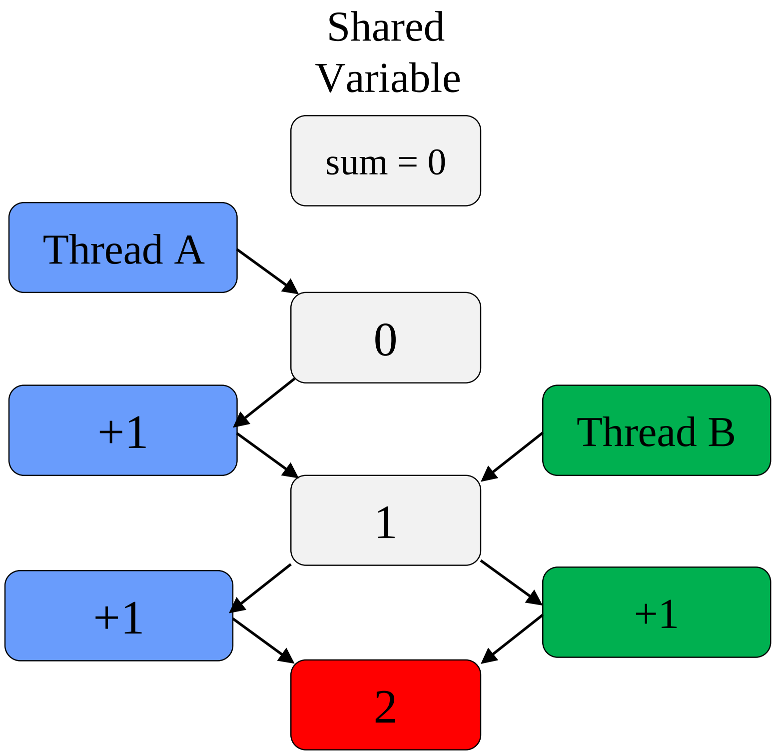
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main (int argc, char *argv[]) {

int i, tid, nthreads, n = 10, N = 100000000;
double *A, *B, tResult, fResult;
time_t start, stop;
clock_t ticks;
long count;
A = (double *) malloc(N*sizeof(double));
B = (double *) malloc(N*sizeof(double));
for (i=0; i<N; i++) {
        A[i] = (double)(i+1);
        B[i] = (double)(i+1);
      }
    time(&start);
    //this block use single process
for (i=0; i<N; i++)
    {
            fResult = fResult + A[i] + B[i];
     }
```

Example 2: Setting Runtime Variables with For Loops (continued)

```
    }
    //begin of parallel section
    #pragma omp parallel private(tid, i,tResult) shared(n,A,B,fResult)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
          }
    #pragma omp for schedule (static, n)
        for (i=0; i < N; i++) {
            tResult = tResult + A[i] + B[i];
}
    #pragma omp for nowait
        for (i=0; i < n; i++)
        {
          printf("Thread %d does iteration %d\n", tid, i);
}
    #pragma omp critical
        fResult = fResult + tResult;
}
    //end of parallel section
time(&stop);
printf("%f\n",fResult);
printf("Finished in about %.0f seconds. \n", difftime(stop, start));
exit(0);
}
```

Caution: Race Condition

- When multiple threads simultaneously read/write shared variable
- Multiple OMP solutions
  - Reduction
  - Atomic
  - Critical




```
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```

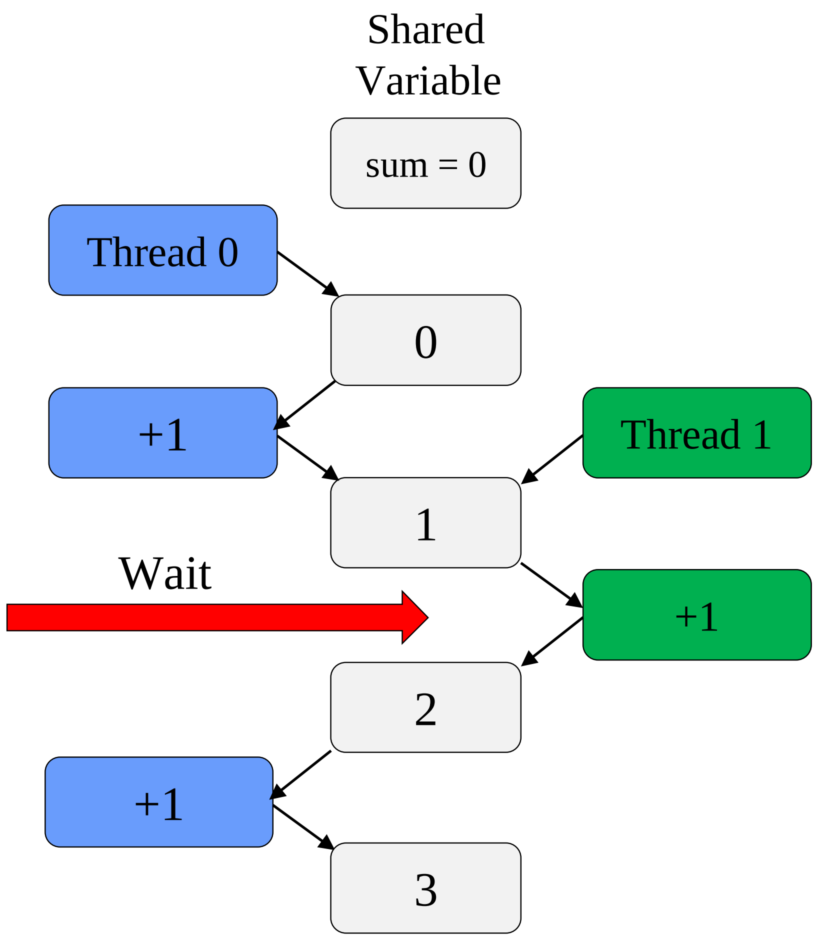
Should be 3!

Critical Section

- One solution: use critical
- Only one thread at a time can execute a critical section

```
#pragma omp critical
    {
      sum += i;
    }
```

- Downside?
  - SLOOOOWWW
  - Overhead & serialization

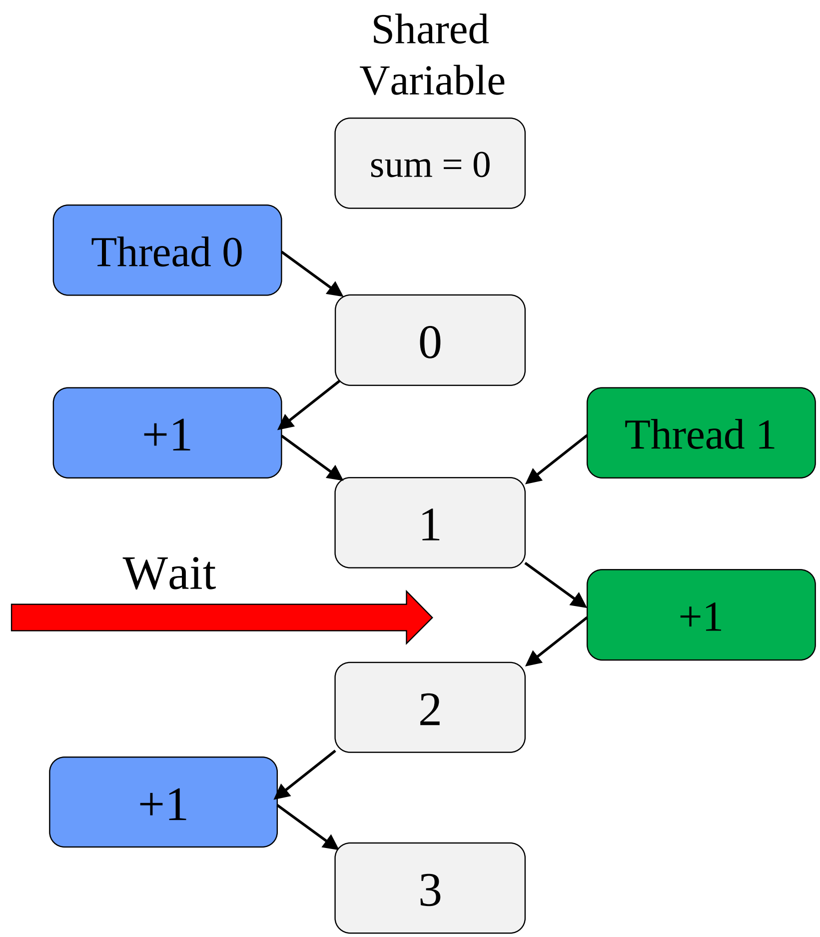


OMP Atomic

- Atomic like “mini” critical
- Only one line
  - Certain limitations

```
#pragma omp atomic
     sum += i;
```

- Hardware controlled
  - Less overhead than critical



OMP Reduction

```
#pragma omp reduction (operator:variable)
```

- Avoids race condition
- Reduction variable must be shared
- Makes variable private, then performs operator at end of loop
- Operator cannot be overloaded (c++)
  - One of: +, *, -, / (and &, ^, |, &&, ||)
  - OpenMP 3.1: added min and max for c/c++

Reduction Example

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

int main() {

   int i;
   const int N = 1000;
   int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
   for (i=0; i<N; i++) {
      sum += i;
   }

   printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

```
[user@adroit3]$ gcc -fopenmp omp_race.c -o omp_race.out
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_race.out
reduction sum=499500 (expected 499500)
```

Where not to use OpenMP

What could go wrong here?

```
.. .
   const int N = 1000;
   int A[N], B[N], C[N];

... // arrays initialized etc.

#pragma omp parallel for shared(A,B,C) private(i)
   for (i=1; i<(N-1); i++) {
      B[i] = A[i-1] + 2*A[i] + A[i+1];
      C[i] = B[i-1] + 2*B[i] + B[i+1];
   }

...
```

`B[i-1]` and `B[i+1]` are not guaranteed to be available/correct

OpenMP Performance Tips

- Avoid serialization!
- Avoid using `#pragma omp parallel for` before each loop
  - Can have significant overhead
    - Thread creation and scheduling is NOT free!!
  - Try for broader parallelism
    - One `#pragma omp parallel`, multiple `#pragma omp for`
  - Always try to parallelize the outer most loop
- Use reduction whenever possible
- Minimize I/O
- Minimize `critical`
  - Use `atomic` instead of `critical` where possible

Timing with MPI and OpenMP APIs

- OpenMP

```
double t1, t2;
t1=omp_get_wtime();
//do something expensive...
t2=omp_get_wtime();
   printf("Total Runtime = %g\n", t2-t1);
```

Work to do:

1) Run the codes given to you on google classroom
2) Calculate the following series with and without openmp and calculate the time in each case . Also append the time in a log file logg.txt

   $$1 + 1/2 + 1/3 + 1/4 + .. + 1/n$$

3) Calculate Addition of two matrices with and without openmp and calculate the time