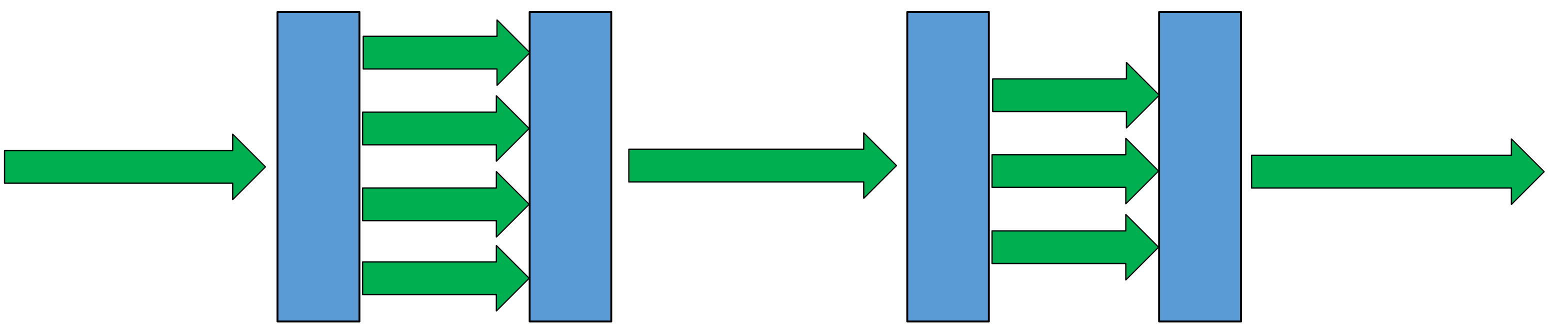
OPENMP

Open Multi-Processing

- Multi-*threaded* parallelism

https://www.openmp.org/



Source – Materials are from the Tutorials and Talks on OpenMP, Tim Mattson (Intel), Charles Augustine, Ruud Van der Paas (SUN), Ian Stoica

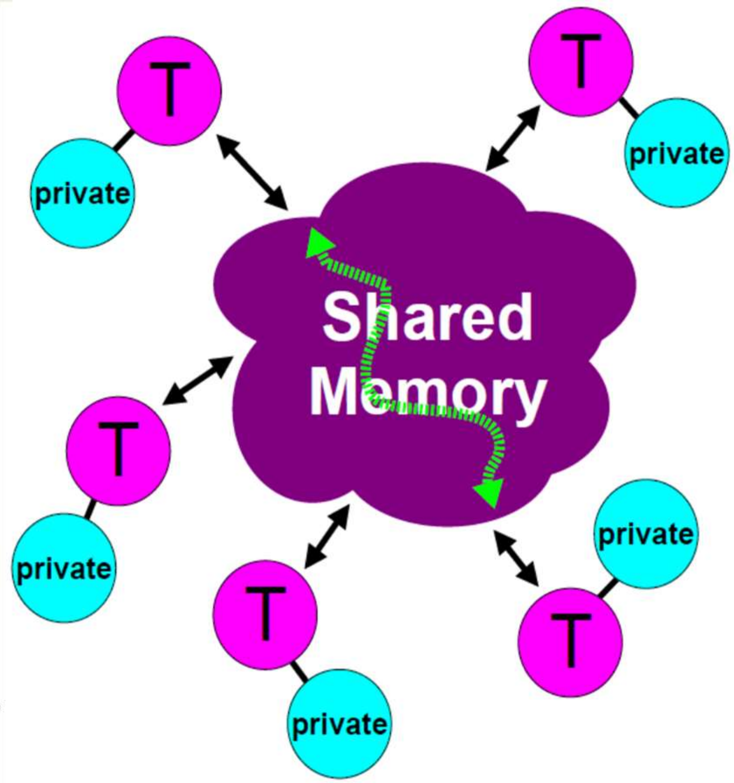
MOTIVATION

Multicore CPUs are everywhere:

- Servers with over 100 cores today
- Even smartphone CPUs have 8 cores

Multithreading, natural programming model

- All processors share the ***same memory***
- Threads in a process see ***same address space***
- OS scheduler decides when/which threads to run.
    - Typically threads are interleaved for fairness.



BUT…

Multithreading is hard

- Lots of expertise necessary
- Deadlocks and race conditions can occur.
- Non-deterministic behavior makes it hard to debug
- Synchronization is necessary to assure order and correct results.

Parallelize the following code using threads:

```
for (i=0; i<n; i++) {
    sum = sum + sqrt(data[i]);
}
```

Why hard?

- Need mutex to protect the accesses to sum
- Different code for serial and parallel version
- No built-in tuning (# of processors?)

OPENMP: MULTI-PROCESSING PROGRAMMING

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

- Standard since 1997: *Fortran, C, C++*
- Requires Native language compiler to support OpenMP. – *most do*
- For shared memory machines: *Limited by available memory*

OPENMP

A language extension with constructs for parallel programming:

- **D**irectives & ***C***lauses
- **E**nvironment Variables
- **F**unctions
  - Work-sharing, Critical sections, atomic access, barriers, private variables

Parallelization is orthogonal to functionality

- If the compiler does not recognize OpenMP directives, the code remains functional (albeit single-threaded).

Industry standard: supported by Intel, Microsoft, IBM, HP

INTRO TO OPENMP (C/C++)

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

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

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

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

```
#pragma omp parallel
```

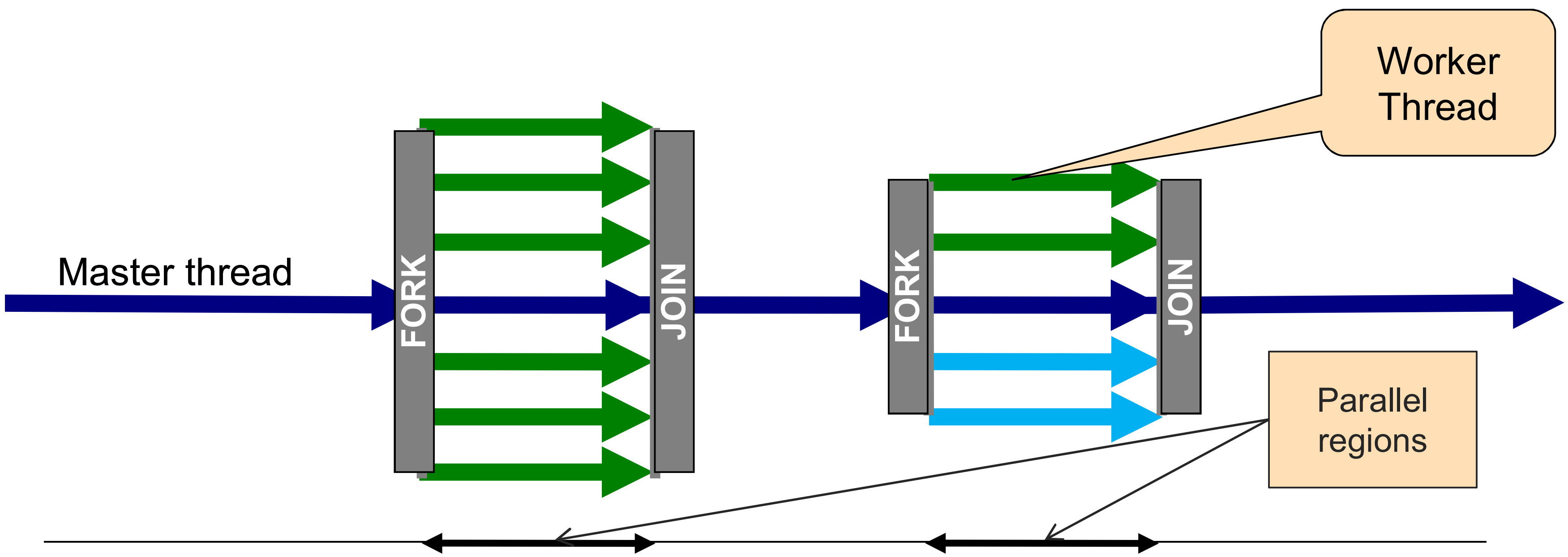
- ➔ "Consider/Run the next set of instructions in parallel"

HELLO WORLD IN OPENMP!

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

int main(void){

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

#pragma omp parallel //← Work Sharing Directives
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }
  return 0;
}
```

OPENMP EXECUTION MODEL

Fork and Join: Master thread spawns ***a team of threads*** as needed

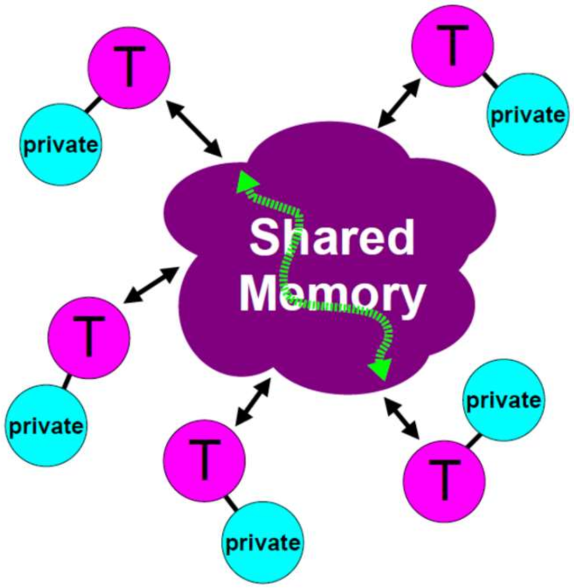


OpenMP memory model

Shared memory model
- Threads communicate by accessing shared variables

The sharing is defined syntactically
- Any variable that is seen by two or more threads is shared
- Any variable that is seen by one thread only is private

Race conditions possible
- Use synchronization to protect from conflicts
- Change how data is stored to minimize the synchronization



OpenMP – Key variables and Subroutines

```
$export OMP_NUM_THREADS=X
```

- Environment variable to define Number of threads.

```
int omp_get_max_threads()
```

- Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

- Returns number of threads in a current team

```
int omp_get_thread_num()
```

- Returns thread id of calling thread. (0 and omp_get_num_threads()]

```
int omp_get_max_procs()
```

- Returns the number of processors on the machine.

```
double omp_get_wtime()
```

- Returns the current reference time for the threat @current instruction.

OpenMP – Constructs

```
Master
```

- Only executed by the Master/Main thread

```
Single
```

- Executed by only one of the thread in a team

```
Section/sections
```

- Declare block of instructions for task parallelism

```
atomic: Execute instruction atomically
```

```
barrier: Synchronization point for team of threads
```

```
critical: Only one thread at a time
```

OpenMP – Clauses

```
Default
```

- Default policy for variable sharing

```
Shared/private
```

- variables can be shared or private

```
Firstprivate/lastprivate
```

- Declare block of instructions for task parallelism

```
Reduction
```

- Scatter and Gather data across multiple threads

OpenMP: Work sharing example – Data Decomposition

Sequential code

```
for (int i=0; i<N; i++) {sum = sum + a[i] + b[i];}
```

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

Semi-manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int i_start = id*N/nt, i_end = (id+1)*N/nt;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

- Launch ***nt*** threads
- Each thread uses ***id*** and ***nt*** variables to operate on a different segment of arrays

Automatic parallelization with **#pragma omp for**

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

OpenMP: Work sharing example – Functional Decomposition

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

How to parallelize?

```
#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { … }
```

OpenMP: Data Environment

Shared Memory programming model

- Most variables (including locals) are shared by threads

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

- Global variables are **shared**

Some variables can be private

- Variables inside the statement block
- Variables in the called functions
- Variables can be explicitly declared as private

QUICK CHECK!

```
int x = 5;
#pragma omp parallel num_threads(5)
    {
          x++;
    }
printf(“X = %d\n”, x);
```

```
int x = -1;
#pragma omp parallel num_threads(8)
{
    sleep(1);
    //Get thread number
    x = omp_get_thread_num();
}
printf("The value of x = %d\n", x);
```

- The same thing is done by all threads
- All data is shared between all threads
- Value of x at end of loop depends on .. (?)
- This code is non-deterministic and will produce different results on different runs

Overriding storage attributes

shared:
A single variable is shared across all threads
Correctness of updates is developer responsibility.

private:
A copy of the variable is created for each thread
There is no connection between original variable and private copies
Can achieve same using variables inside { }

firstprivate:
Same, but the initial value of the variable is **copied from** the main copy

lastprivate:
Same, but the last value of the variable is **copied to** the main copy

```
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

```
int idx=1;
int x = 10;
#pragma omp parallel for \
  firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i] == x)
       idx = i;
}
```

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
  - Use `firstprivate` and/or `lastprivate` clause to override

OMP PARALLEL CLAUSES 2

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

- Variables in list are private
- Initialized with the value the variable had *before* entering the construct

```
#pragma omp parallel for lastprivate (list)
```

- Only in for loops
- Variables in list are private
- The thread that executes the *sequentially last iteration* updates the value of the variables in the list

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

Beware of Race Conditions!

```
for (int i=0; i<N; i++) {sum = sum + i;}
```

How to parallelize this code?

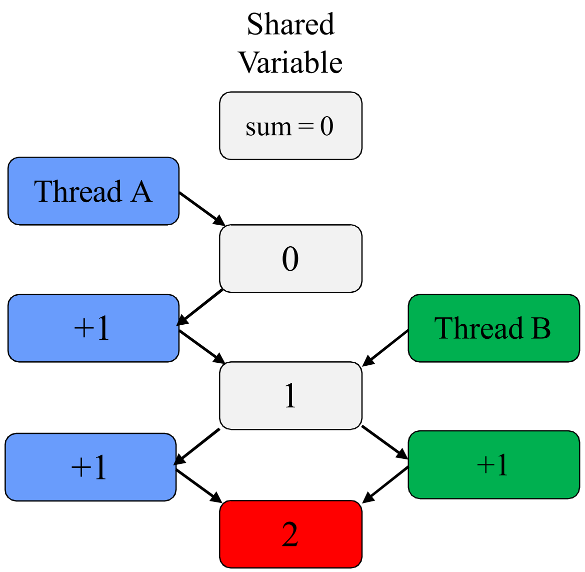
sum is not private!

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;
  }
```



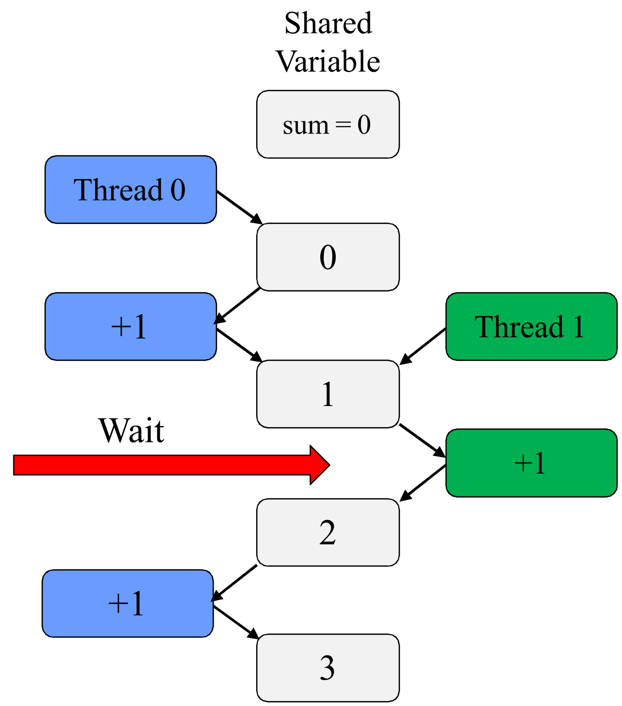
Should be 3!

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

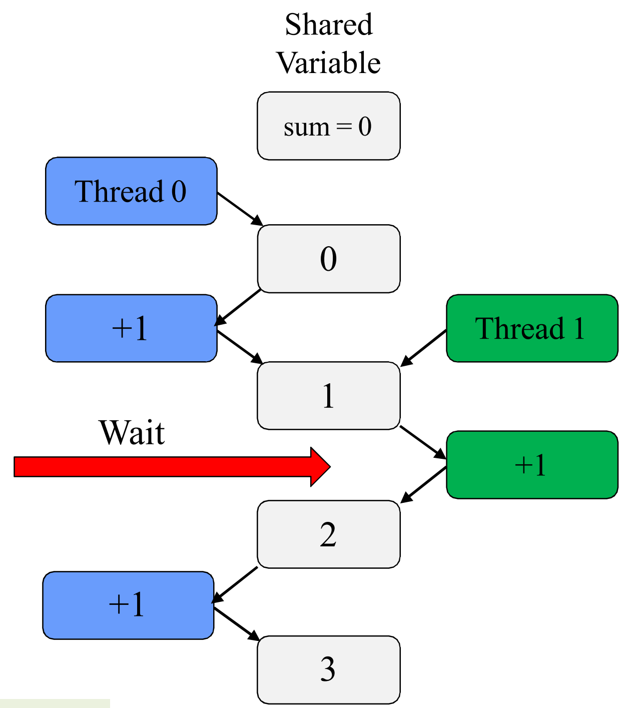


OMP Atomic

- Atomic like "mini" critical
- Only one line
  - Certain limitations

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

- Hardware controlled
  - Less overhead than critical



- Certain limitations…
- Bonus point for the student(s) who identify the limitations

#PRAGMA OMP REDUCTION- REDUCTION

```
for (int i=0; i<N; i++) {sum = sum + a[i] * b[i];}
```

How to parallelize this code?

- sum is not private, but accessing it atomically is too expensive
- Have a private copy of sum in each thread, then add them up

Use the reduction clause

```
#pragma omp parallel for reduction(+: sum)
```

- Any associative operator could be used: +, -, ||, |, *, etc
- The private value is initialized automatically (to 0, 1, …)

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for(int i = 0; i < N; i++) {sum += a[i] * b[i]; }
  return sum;
}
```

Reduction:

Avoids race condition

Reduction variable must be shared

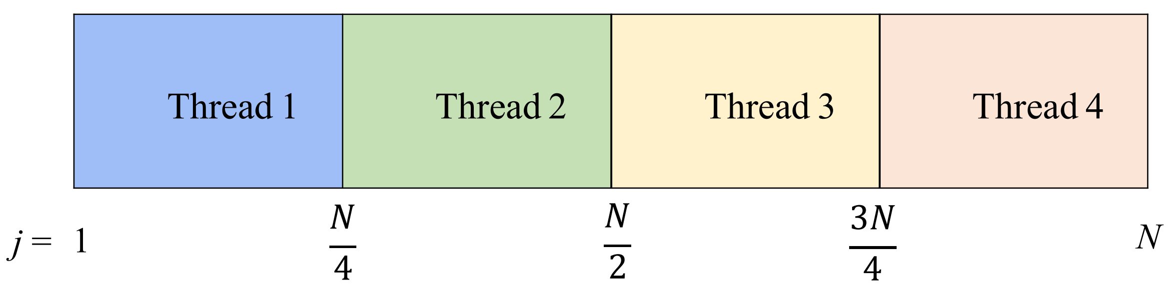
Makes variable private, then performs operator at end of loop

SCHEDULING `omp for`

- How does a loop get split up?
  - In MPI, we have to do it manually.
- If you don't tell it what to do, the compiler decides
- Usually compiler chooses "static" – chunks of N/p

```
#pragma omp parallel for default(shared) private(j)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

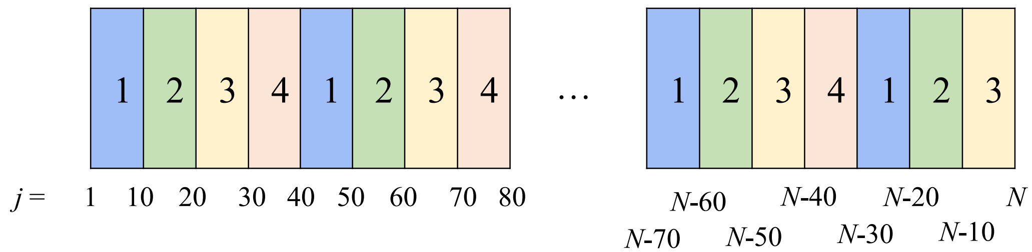
Unspecified schedule



STATIC SCHEDULING

- You can tell the compiler what size chunks to take

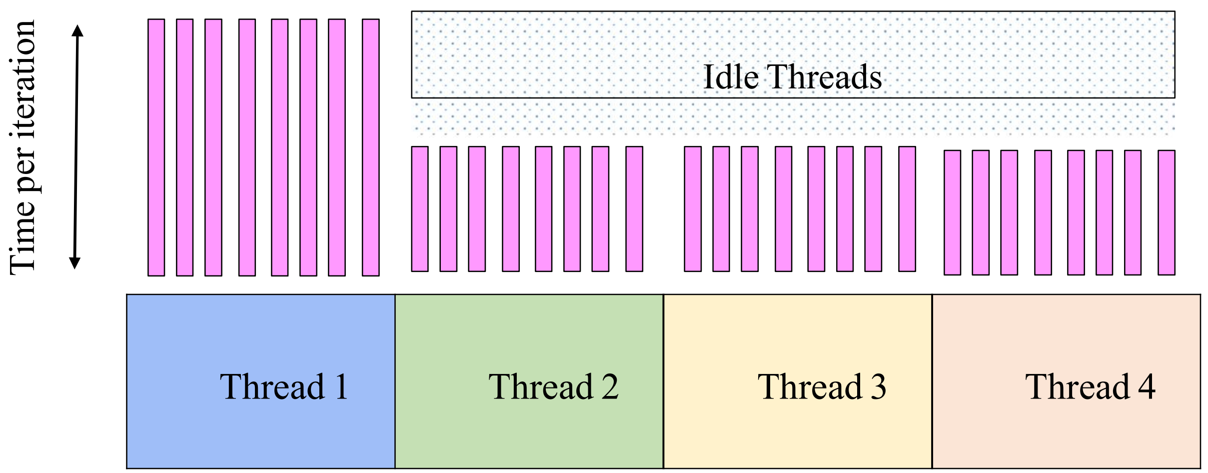
```
#pragma omp parallel for default(shared)                schedule(static,10)
  private(j) for (j=0; j<N; j++) {
            ... // some work here
}
```



- Keeps assigning chunks until done
- Chunk size that isn't a multiple of the loop will result in threads with uneven numbers

PROBLEM WITH STATIC SCHEDULING

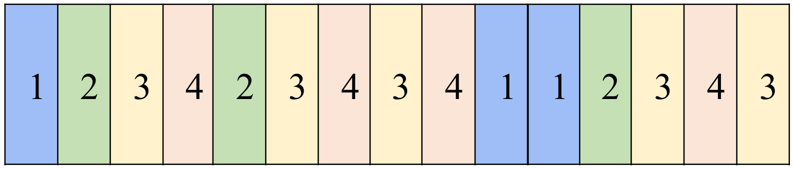
- What happens if loop iterations do not take the same amount of time?
  - Load imbalance



Dynamic Scheduling

- Chunks are assigned on the fly, as threads become available
  - When a thread finishes one chunk, it is assigned another

```
#pragma omp parallel for default(shared)                schedule(dynamic,1)
private(j)
   for (j=0; j<N; j++) {
            ... // some work here
}
```



Note: It will have higher overhead than static!

OMP FOR SCHEDULING RECAP

```
#pragma omp parallel for schedule(type [,size])
```

- Scheduling types
  - Static
    - Chucks of specified size assigned round-robin
  - Dynamic
    - Chunks of specified size are assigned when thread finishes previous chunk
  - Guided
    - Like dynamic, but chunks are exponentially decreasing
    - Chunk will not be smaller than specified size
  - Runtime
    - Type and chunk determined at runtime via environment variables

Summary

OpenMP: A framework for code parallelization

- Available for C/C++ and FORTRAN
- Based on a standard implementations from a wide selection of vendors

Relatively easy to use

- Write (and debug!) code first, parallelize later
- Parallelization can be incremental
- Parallelization can be turned off at runtime or compile time
- Code is still correct for a serial machine