OpenMP

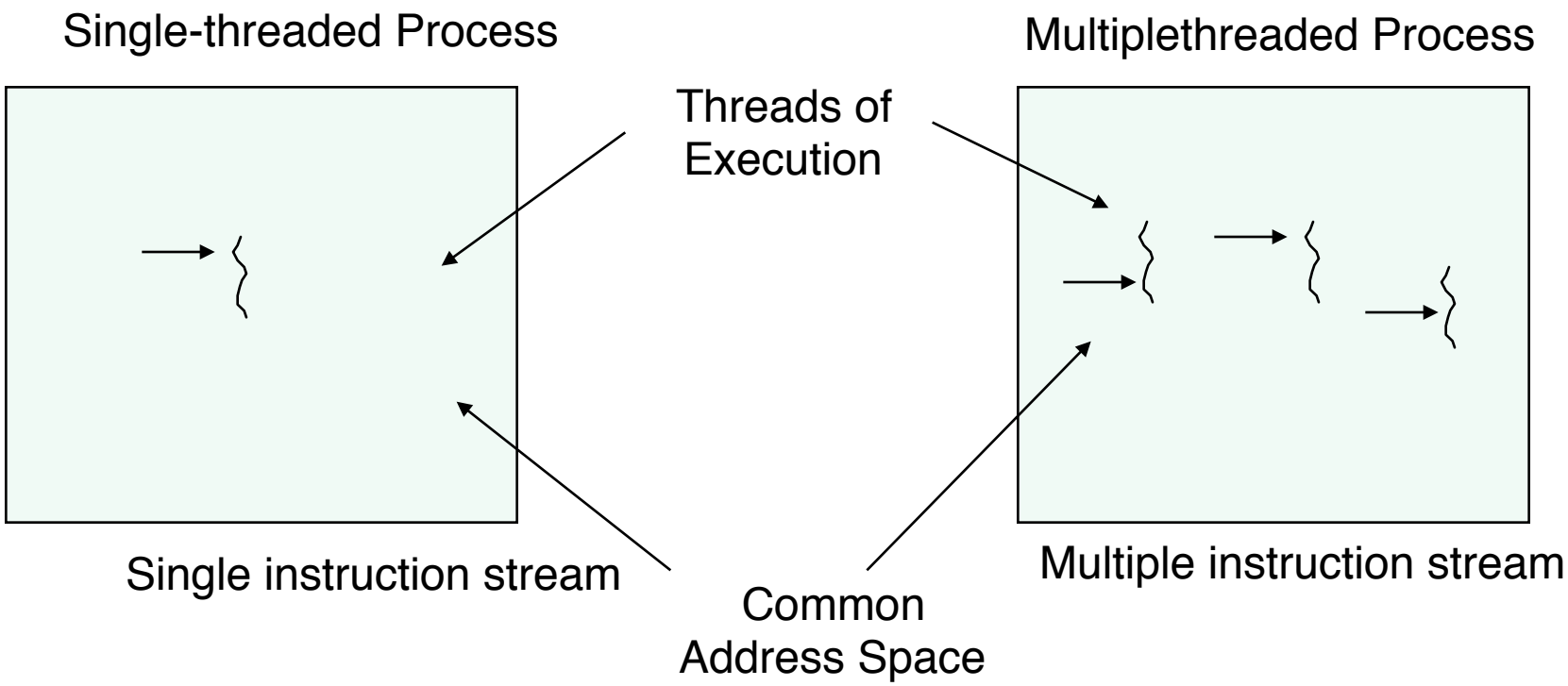
Thoai Nam

High Performance Computing Lab (HPC Lab)

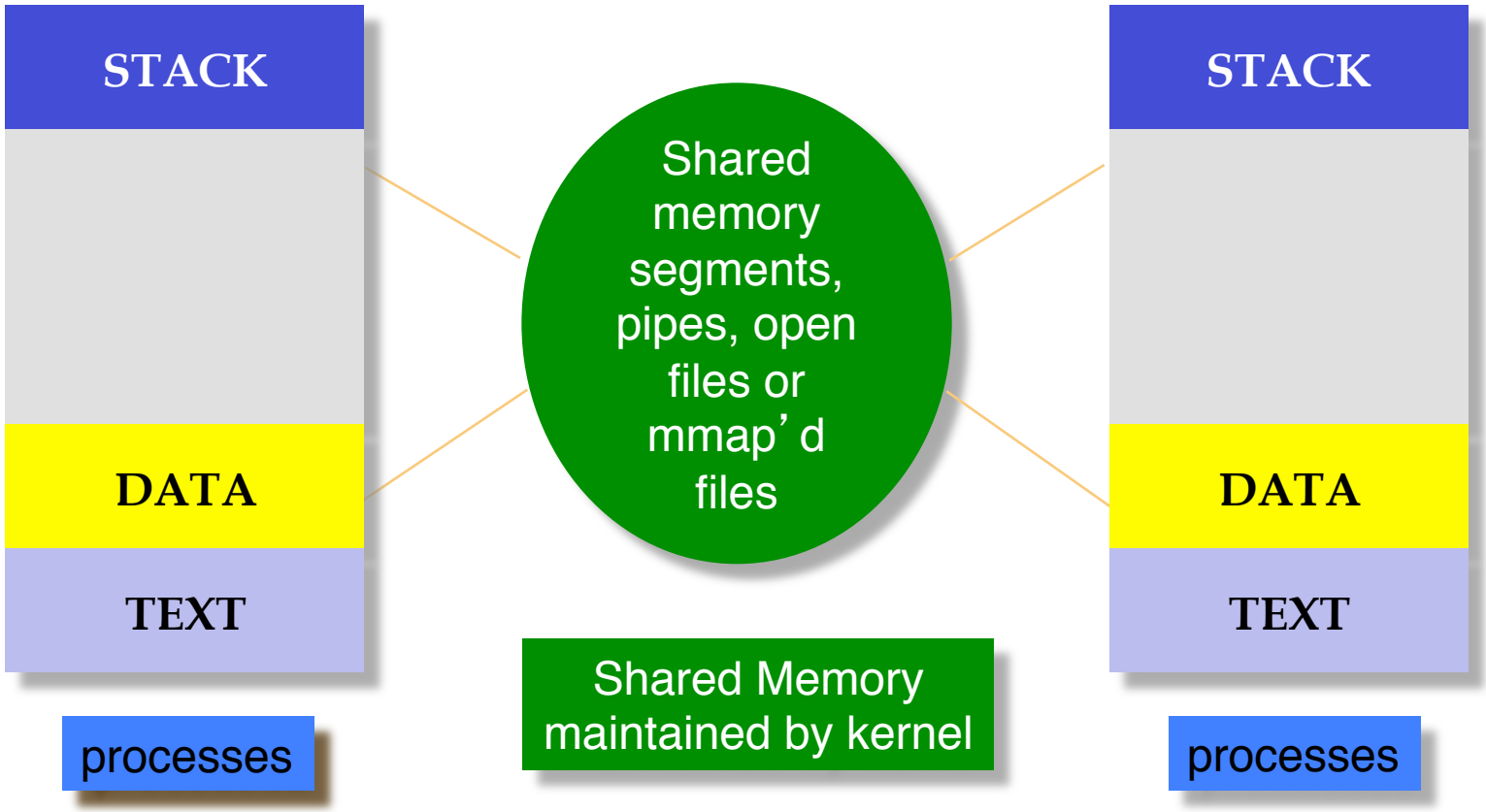
Faculty of Computer Science and Engineering

HCMC University of Technology

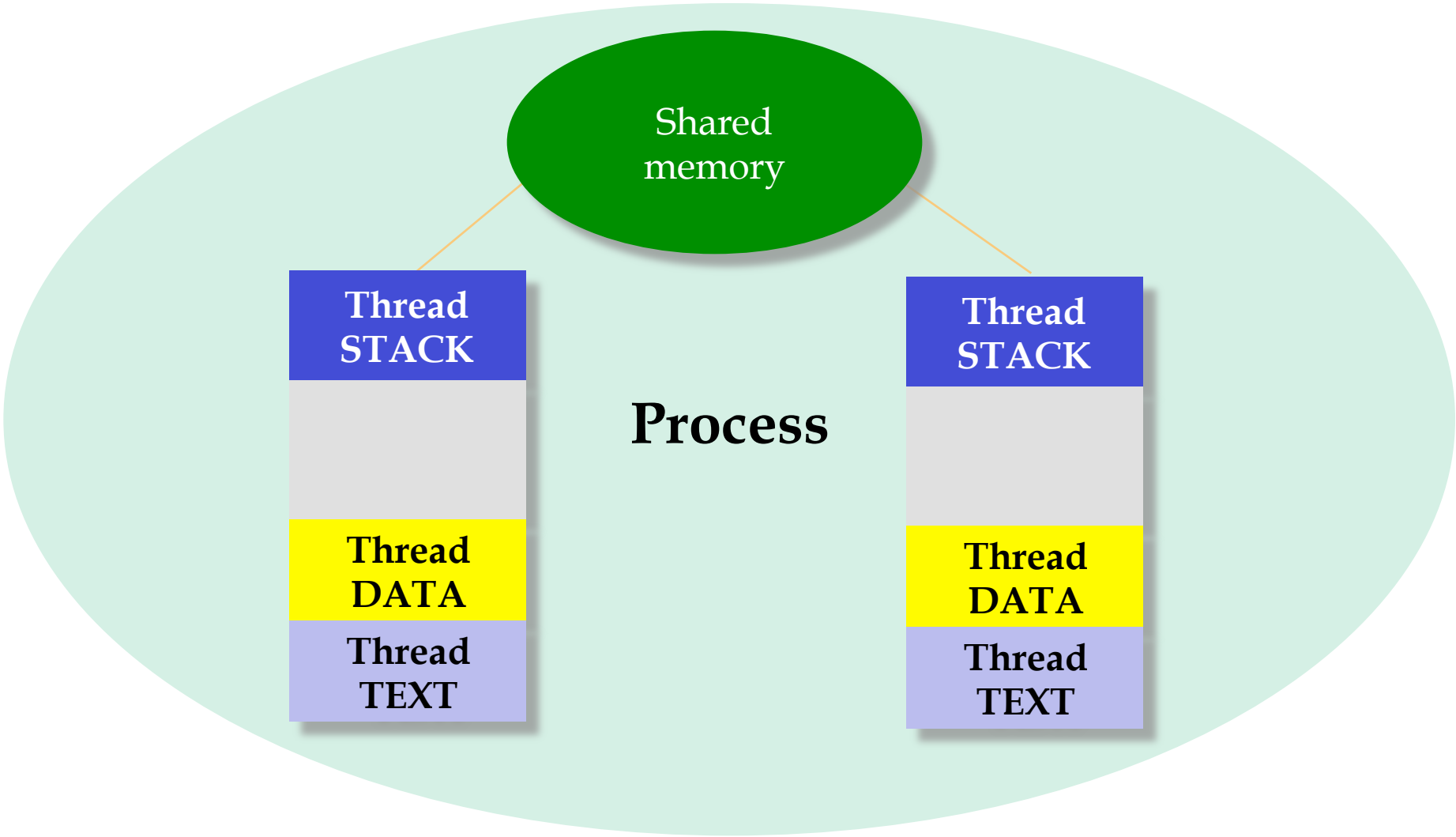
Process: single & multithreaded



Process Model



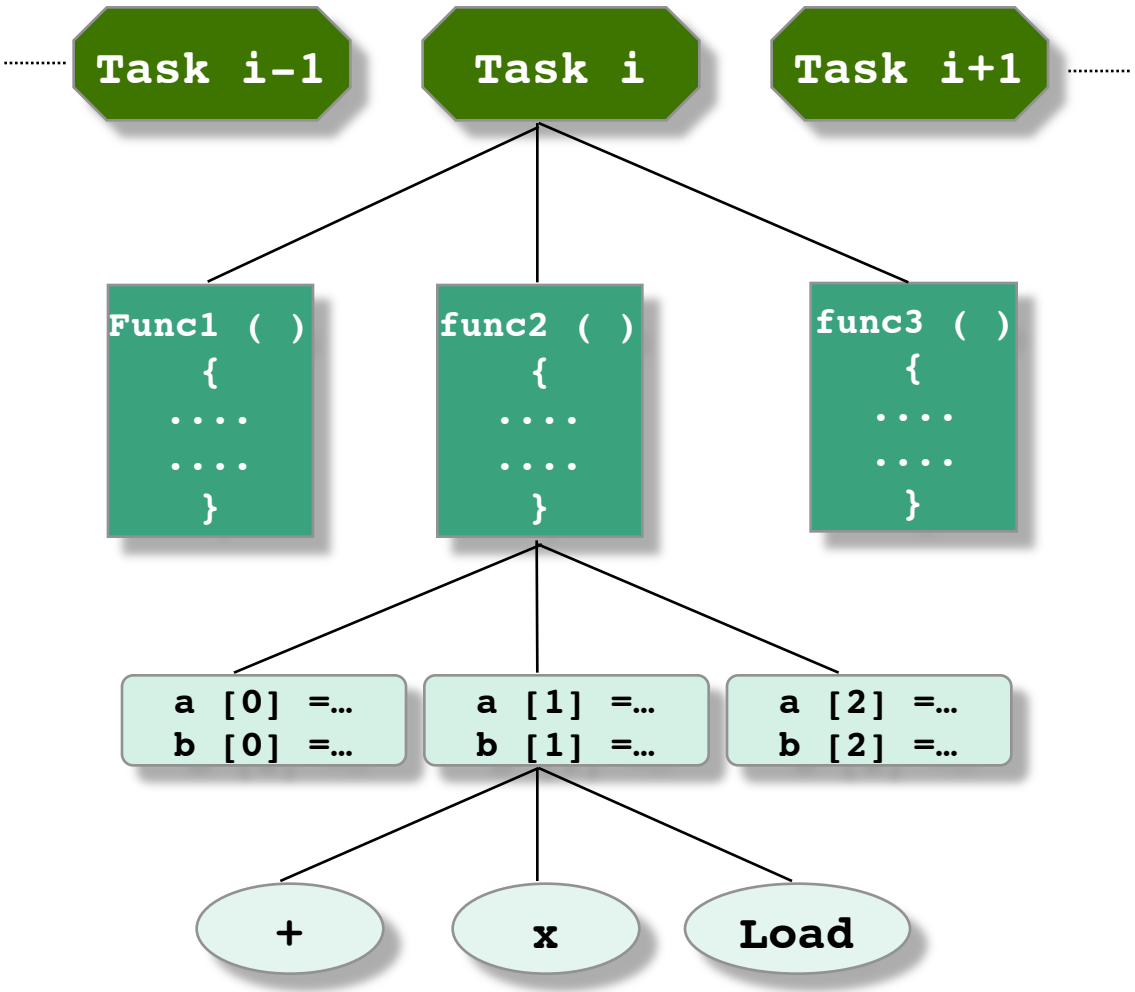
Threaded Process Model



What are Threads

- Thread is a piece of code that can execute in concurrence with other threads
- It is a schedule entity on a processor
  - **Local state**
  - **Global/shared state**
  - **PC**
  - **Hard/Software context**

Levels of Parallelism



Code-Granularity

Code Item

Large grain
(task level)

Program

Medium grain
(control level)

Function (thread)

Fine grain
(data level)

Loop

Very fine grain
(multiple issue)

With hardware

Thread Example

```
void *func ( )
{
    // define local data
    // function code
    thr_exit(exit_value);
}

main ( )
{
    thread_t tid;
    int exit_value;
    ...
    thread_create (0, 0, func(), NULL, &tid);
    ...
    thread_join (tid, 0, &exit_value);
    ...
}
```

Pthread problem

- Pthread is too tedious: explicit thread management is often unnecessary
  - Consider the matrix multiply example
    - We have a sequential code, we know which loop can be executed in parallel; the program conversion is quite mechanic: we should just say that the loop is to be executed in parallel and let the compiler do the rest.
    - OpenMP does exactly that!!!

OpenMP

- de fact standard model for programming shared memory machines
- C/C++/Fortran + parallel directives + APIs
  - by #pragma in C/C++
  - by comments in Fortran
- many free/vendor compilers, including GCC

What is OpenMP?

- What does OpenMP stands for?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia
- OpenMP is an Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism***
  - API components: Compiler Directives, Runtime Library Routines. Environment Variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for Fortran and pragmas for C/C++
- OpenMP website: *http://www.openmp.org*
  - Materials in this lecture are taken from various OpenMP tutorials in the website and other places

Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel and others
  - It is the de facto standard for writing shared memory programs
  - To become an ANSI standard?
- OpenMP can be implemented incrementally, one function or even one loop at a time
  - A nice way to get a parallel program from a sequential program

OpenMP reference

- Official home page: http://openmp.org/
- Specification: https://www.openmp.org/specifications/
- Version is 5.1 (Nov 2020)
- Version is 5.0 (Nov 2018)

- Compiler GCC
  - http://gcc.gnu.org/wiki/openmp
  - GCC 9 & 10 → OpenMP spec 5.0

OpenMP programs with GCC

- Compile with -fopenmp

```
$ gcc -Wall -fopenmp program.c
```

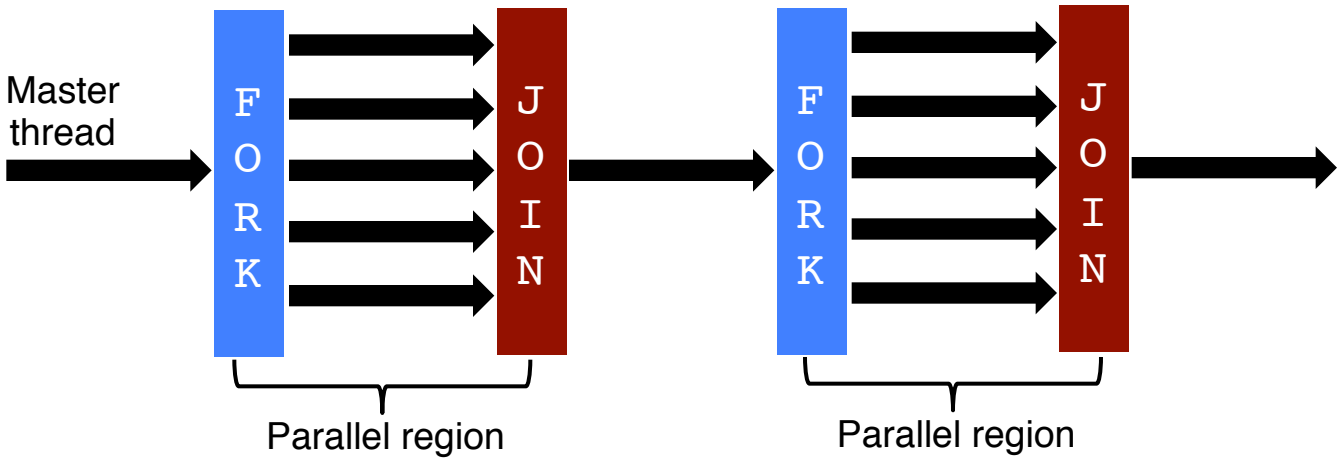
- Run the executable specifying the number of threads with OMP_NUM_THREADS environment variable

```
$ OMP_NUM_THREADS=1 ./a.out # use 1 threads

$ OMP_NUM_THREADS=4 ./a.out # use 4 threads
```

- See 2.6.1 in OpenMP 5.1 “Determining the Number of Threads for a parallel Region” for other ways to control the number of threads

OpenMP execution model



- OpenMP uses the fork-join model of parallel execution.
  - All OpenMP programs begin with a single master thread
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK)
  - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN)

OpenMP general code structure

```
#include <omp.h>
main () {
   int var1, var2, var3;
   Serial code
   . . .
  /* Beginning of parallel section. Fork a team of threads.
   Specify variable scoping*/
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      /* Parallel section executed by all threads */
     . . .
     /* All threads join master thread and disband*/
   }
   Resume serial code
   . . .
}
```

OpenMP directives

- Format:

```
#pragma omp directive-name [clause,...] newline
(use '\' for multiple lines)
```

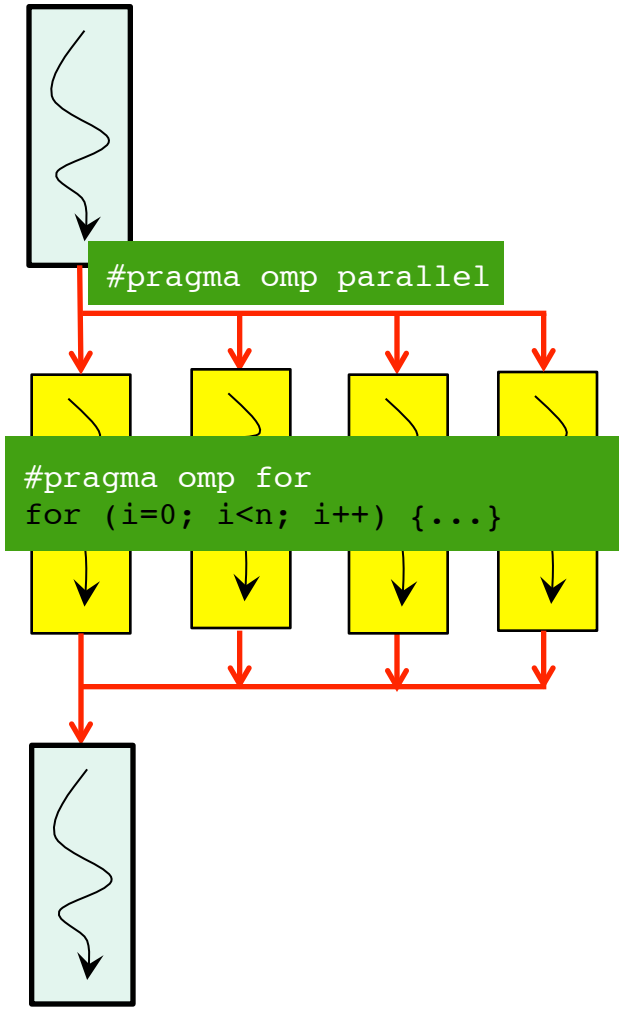
- Example:

```
#pragma omp parallel default(shared) private(beta, pi)
```

- Scope of a directive is one block of statements {…}

Two pragmas you must know first

- `#pragma omp parallel` to launch a team of threads
- then `#pragma omp for` to distribute iterations to threads
- Note: all OpenMP pragmas have the common format: `#pragma omp ...`

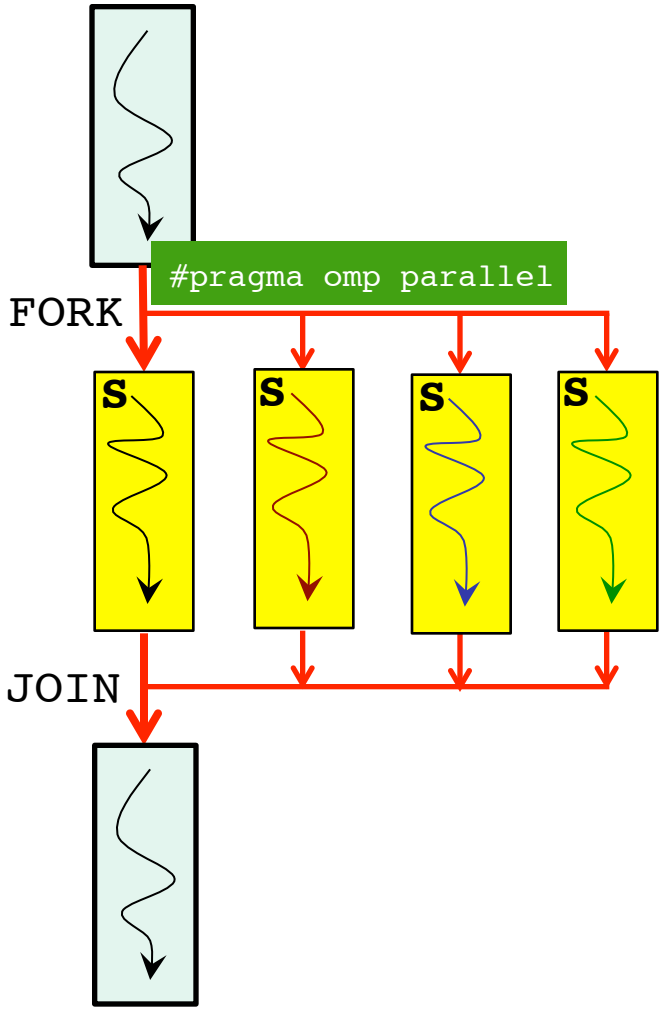


#pragma parallel

- Syntax:

```
...
#pragma omp parallel
  s
...
```

- Basic semantics:
  - create a team of OMP_NUM_THREADS threads
  - the current thread becomes the master of the team
  - *s will be executed by each member of the team*
  - the master thread waits for all to finish **s** and continue



Parallel pragma example

```
#include <stdio.h>
int main() {
   printf("hello\n");
   #pragma omp parallel
      printf("world\n");
   return 0;
}
```

```
$ OMP_NUM_THREADS=1     ./a.out
  hello
  world
$ OMP_NUM_THREADS=4     ./a.out
  hello
  world
  world
  world
  world
```

what does **parallel** do?

- You may assume an OpenMP thread ≈ OS-supported thread (e.g., Pthread)
- that is, if you write this program

```
int main() {
   #pragma omp parallel
       worker();
}
```

- and run it as follows,

```
$ OMP_NUM_THREADS=20   ./a.out
```

- You will get 20 OS-level threads, each doing worker()

How to distribute work among threads?

- `#pragma omp parallel` creates threads, all executing the same statement
- It's not a means to parallelize work, but just a means to create a number of similar threads (SPMD)
- So how to distribute (or partition) work among them?

  (1) do it yourself

  (2) use *work sharing* constructs

Parallel region construct (1)

- A block of code that will be executed by multiple threads

```
#pragma omp parallel [clause …]
{
   …
}   (implied barrier)
```

Clauses: *if (expression),* private *(list),* shared *(list),* default (shared | none), reduction *(operator: list),* firstprivate*(list),* lastprivate*(list)*

- if (expression): only in parallel if expression evaluates to true
- private(list): everything private and local (no relation with variables outside the block)
- shared(list): data accessed by all threads
- default (none | shared)

Parallel region construct (2)

- default (none I shared)
  - » default(none) clause forces a programmer to explicitly specify the data-sharing attributes of all variables, thus making it obvious which variables are referenced, and what is their data sharing attribute, thus increasing readability and possibly making errors easier to spot

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n, vector, a)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

Parallel region construct (3)

- default (none l shared)
  - » default(shared) clause sets the data-sharing attributes of all variables in the construct to shared

```
int a, b, c, n;
...
#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
    // using a, b, c, n are shared variables
}
```

```
int a, b, c, n;
...
#pragma omp parallel for default(shared) private(a, b)
for (int i = 0; i < n; i++)
{
    // a and b are private variables
    // c and n are shared variables
}
```

Parallel region construct (4)

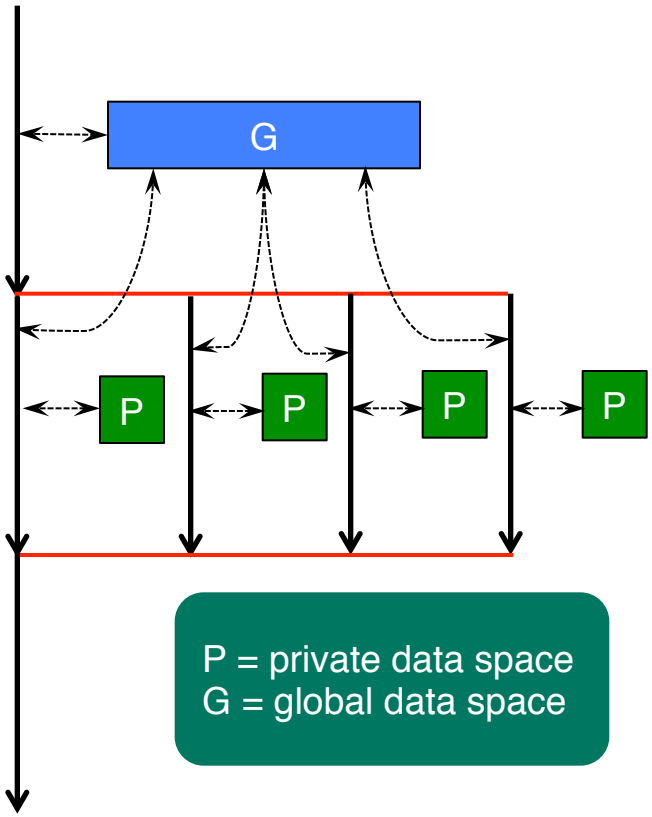
- The `reduction` clause:

```
Sum = 0.0;
#pragma parallel default(none) shared(n,x) private(i)
   reduction(+:sum)
{
   for(i=0; i<n; i++)
      sum = sum + x(i);
}
```

  - Updating *sum* must avoid racing condition
  - With the reduction clause, OpenMP generates code such that the race condition is avoided

- `firstprivate(list)`: variables are initialized with the value before entering the block
- `lastprivate(list)`: variables are updated going out of the block

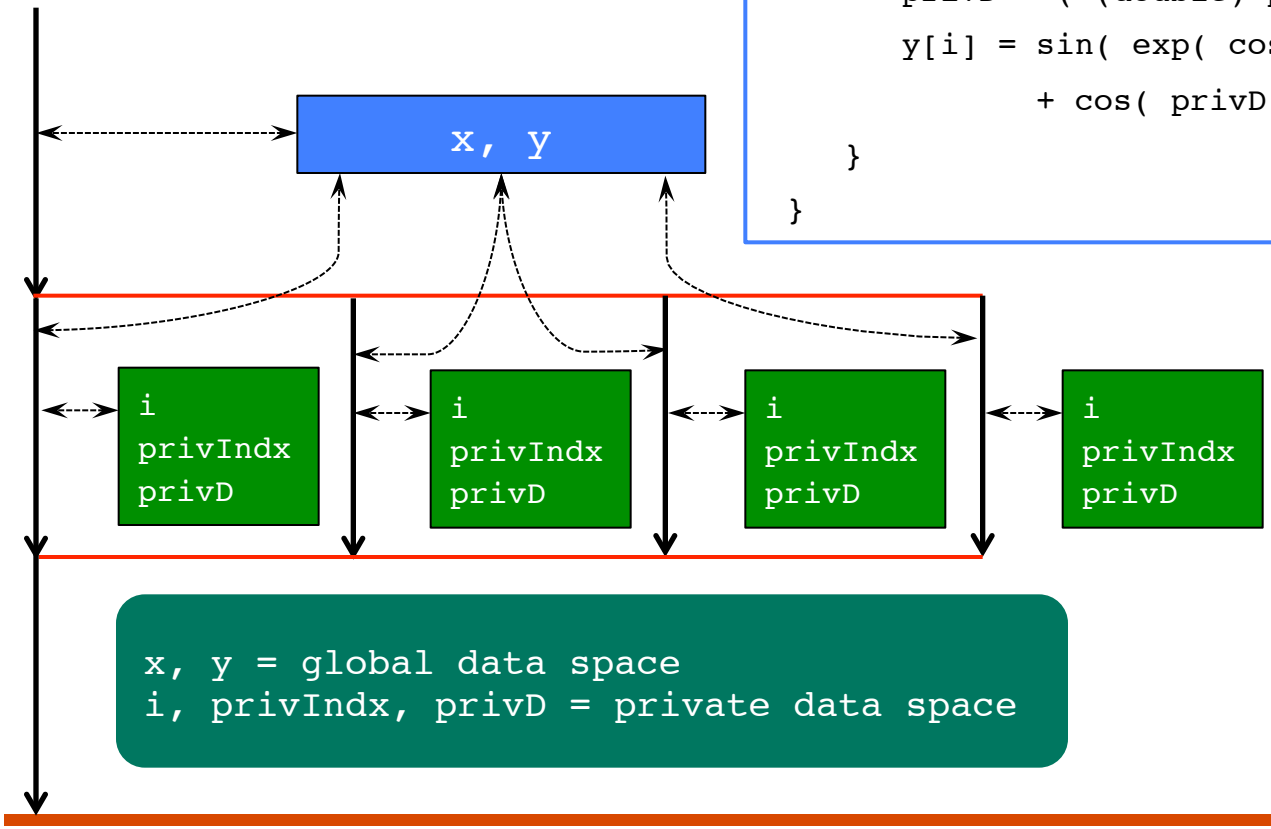
Data model



- Private and shared variables
  - Variables in the global data space are accessed by all parallel threads (shared variables)
    - The data-sharing attribute of variables, which are declared outside the parallel region, is usually shared
  - Variables in a thread's private space can only be accessed by the thread (private variables)
    - Several variations, depending on the initial values and whether the results are copied outside the region
    - The variables which are declared locally within the parallel region are private

Data in parallel loop

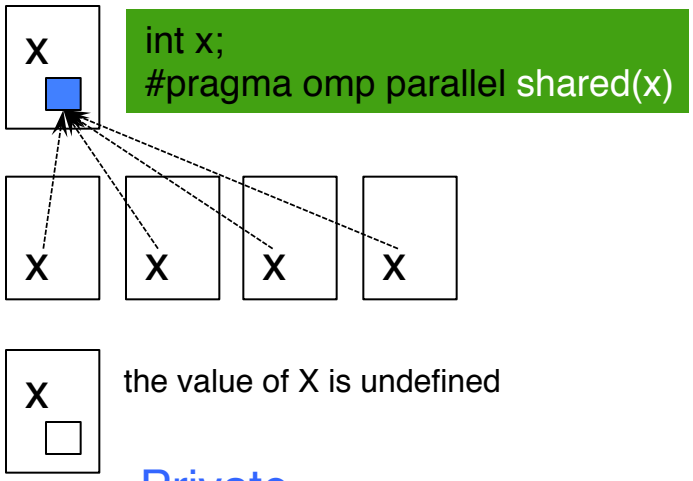
```
#pragma omp parallel for private( privIndx, privD)
  for (i = 0; i < arraySize; i++) {
     for (privIndx = 0; privIndx < 16; privIndx++) {
      privD = ( (double) privIndx ) / 16;
      y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) )
             + cos( privD );
   }
}
```



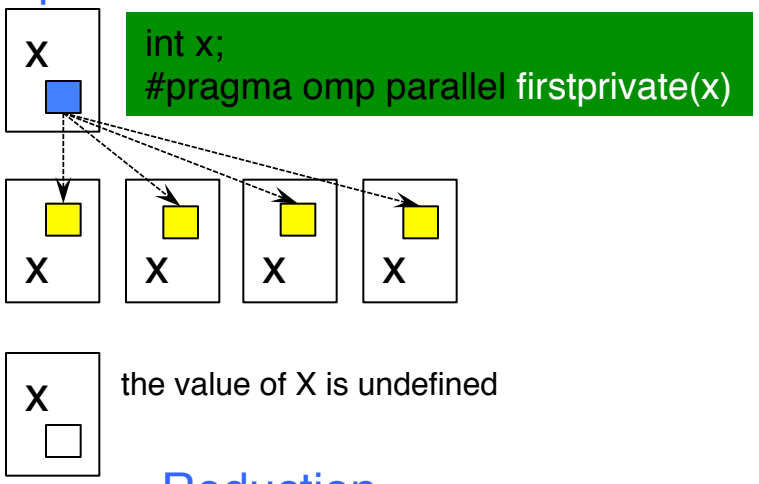
Parallel for loop index is Private by default: (i)

Data sharing behavior

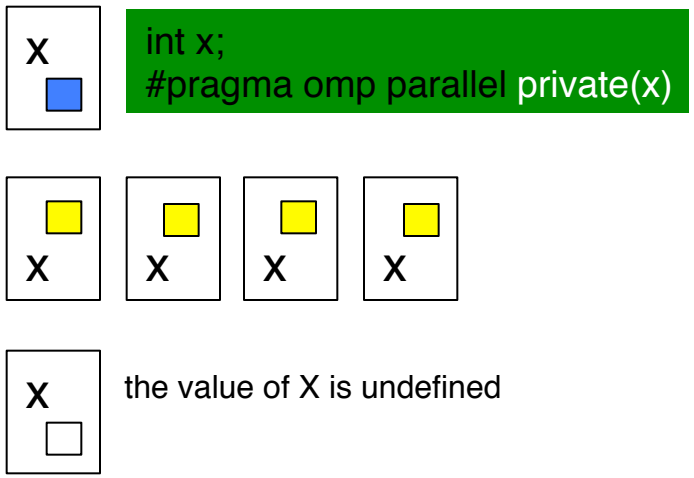
Shared



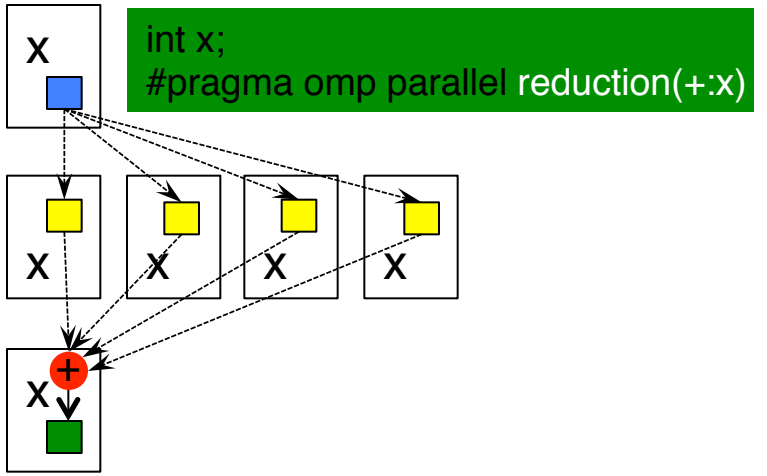
Firstprivate



Private



Reduction



Data sharing example

```
int main() {
int S; /* shared */
int P; /* made private below */
#pragma omp parallel private(P) shared(S)
 {
       int L; /* automatically private */
       printf("S at %p, P at %p, L at %p\n", &S, &P, &L);

 }
   return 0;
}
```

```
$ OMP_NUM_THREADS=2    ./a.out
S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

Functions to get the number/id of threads

- *omp_get_num_threads()*: the number of threads in the current team
- *omp_get_thread_num()*: the current thread's id (0, 1, ...) in the team
- They are primitives with which you may partition work yourself by whichever ways you prefer
- e.g.,

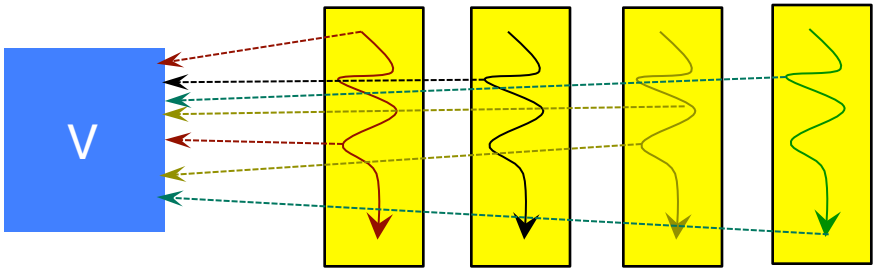
```
#pragma omp parallel
{
   int t = omp_get_thread_num();
   int nt = omp_get_num_threads();
   /* divide n iterations evenly among nt threads */
   for (i = t * n / nt; i < (t + 1) * n / nt; i++) { ... }
}
```

Reduction

- In general, “reduction” refers to an operation to combine many values into a single value. e.g.,
  - $v = v_1 + ... + v_n$
  - $v = max(v_1,\cdots,v_n)$
  - ...
- Simply sharing the variable (v) does not work (race condition)
- Even if you make updates atomic, it will be slow (by now you should know how slow it will be)

```
v = 0.0;
for (i = 0; i < n; i++) {
   v += f(a + i * dt) * dt;
}
```

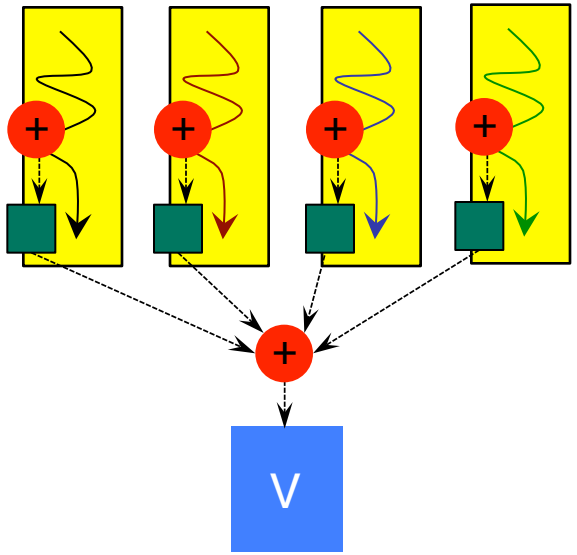
```
v = 0.0;
#pragma omp parallel shared(v)
#pragma omp for
for(i = 0; i < n; i++){
#pragma omp atomic
   v += f(a + i * dt) * dt;
}
```



Reduction

- A more efficient strategy is to let each thread reduce on its private variable, and when threads finish, combine their partial results into one
- `reduction` clause in OpenMP does just that

```
v = 0.0;
#pragma omp parallel shared(v)
#pragma omp for reduction(+:v)
for(i = 0; i < n; i++){
   v += f(a + i * dt) * dt;
}
```



Reduction & user-defined reduction

- Syntax

```
#pragma omp parallel reduction(op: var, var,...)
   s
```

  - Built in reductions
    - *op* is one of +, -, *, &, ^, |, &&, and ||
    - (Since 3.1) min or max
  - (Since 4.0) a user-defined reduction name

- User-defined reduction syntax:

```
#pragma omp declare reduction (name : type : combine statement)
```

Op in reduction

- OpenMP specifies which statements are supported in the definition of the reduction; and these statements occur in the body of the for loop:

No.	Format	Example	Note
(1)	x = x Δ expr	X = X - expr	
(2)	xΔ= expr	X+= expr	
(3)	x = expr Δ x	X = expr * X	Not support "-" op
(4)	xΔΔ ΔΔx	X++, X-- ++X, --X	Only support "+" and "-" op

  - Δ ~ *op* is one of +, -, *, &, ^, |, &&, || (min, max)
  - X is a reduction variable
  - expr is an expression which does not depend on X

Work sharing constructs

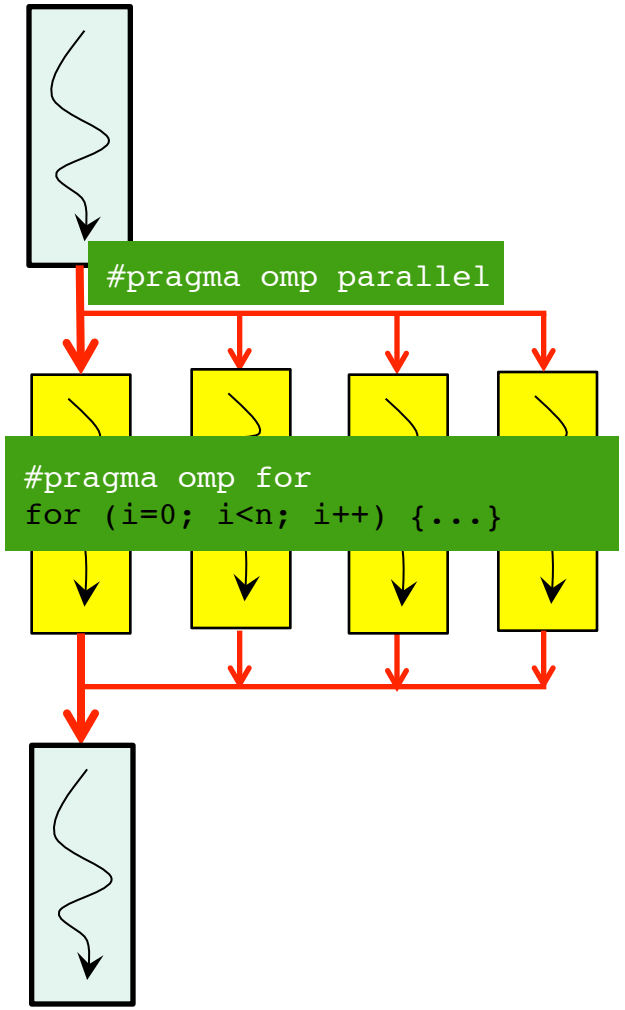
- In theory, `parallel` construct is all you need to do things in parallel
- But it's too inconvenient
- OpenMP defines ways to *partition* work among threads (*work sharing constructs*)
  - **`for`**
  - **`task`**
  - **`section`**

#pragma omp for (work-sharing for)

- Syntax:

```
...
#pragma omp for
  s
...
```

- basic semantics:
  - the threads in the team divide the iterations among them
- but how? ⇒ scheduling



#pragma omp for example

```
#pragma omp parallel default(none) shared(n, a, b, c, d) private(i)
{
    #pragma omp for nowait
    for (i=0; i < n-1; i++)
        b[i] = (a[i]+a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i < n; i++)
        d[i] = 1.0/c[i];

} {implicit barrier}
```

- An implicit barrier occurs at the end of a worksharing-loop region if a nowait clause is not specified

#pragma omp for restrictions

- not arbitrary for statement is allowed after a *for pragma*
- strong syntactic restrictions apply, so that the iteration counts can easily be identified at the beginning of the loop
- roughly, it must be of the form:

```
#pragma omp for
  for (i = init; i < limit; i+= incr)
    S
```

except < and += may be other similar operators

- *init, limit,* and *incr* must be loop invariant

Performance (1)

- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
  - Inversion does not significantly lower cache hit rate

Performance (2)

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The if clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

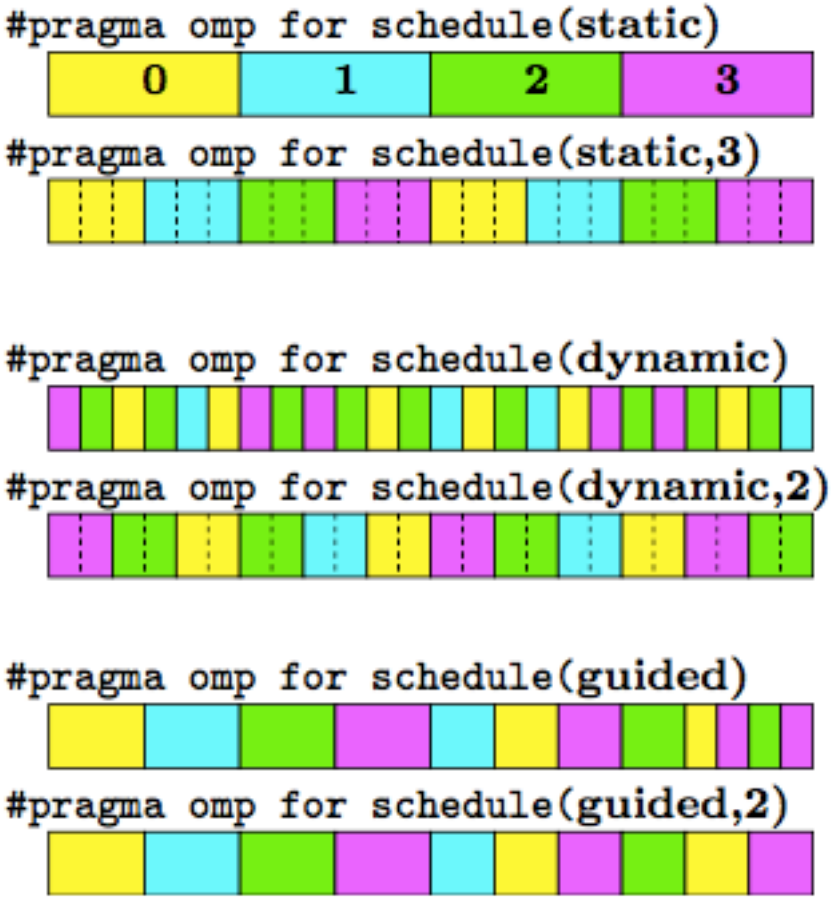
```
#pragma omp parallel for if (n > 2000)
```

Scheduling

- Schedule clause in work-sharing for loop determines how iterations are divided among threads
- There are three alternatives (*static, dynamic,* and *guided*)

static, dynamic, and guided

- schedule(**static** [,*chunk*]): predictable round-robin
- schedule(**dynamic** [,*chunk*]): each thread repeats fetching *chunk* iterations
- schedule(**guided** [,*chunk*]): threads grab many iterations in early stages; gradually reduce iterations to fetch at a time
- *chunk* specifies the minimum granularity (iteration counts)



Performance with scheduling

- We can use schedule clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations

Static & Dynamic

- Static scheduling
  - Low overhead
  - May exhibit high workload imbalance
- Dynamic scheduling
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads

Other scheduling options and notes

- Schedule(runtime) determines the schedule by OMP_SCHEDULE environment variable. e.g.,

```
$ OMP_SCHEDULE=dynamic,2      ./a.out
```

- Schedule(auto) or no schedule clause choose an implementation dependent default

Parallelizing loop nests by **collapse**

- *collapse(l)* can be used to partition nested loops. e.g.,

```
#pragma omp for collapse(2)
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    S
```

will partition n^2 iterations of the doubly-nested loop

- Schedule clause applies to nested loops as if the nested loop is an equivalent flat loop
- Restriction: the loop must be *"perfectly nested"* (the iteration space must be a rectangular and no intervening statement between different levels of the nest)

Task parallelism in OpenMP

- OpenMP's initial focus was simple parallel loops
- since 3.0, it supports task parallelism
- but why it's necessary?
- aren't `parallel` and `for` all we need?

Limitation of **parallel for**

- what if you have a parallel loop inside another

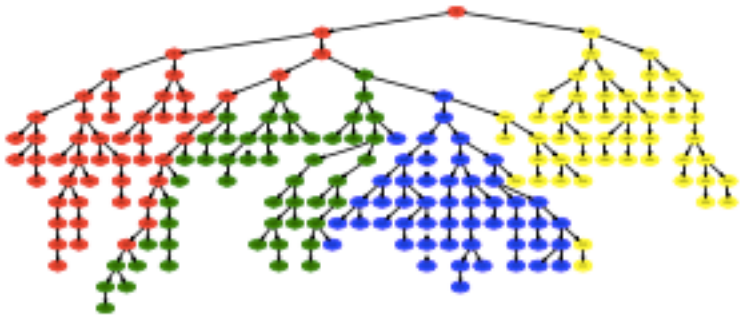
```
for (...) {
  ...
  for (...) ...
}
```

- perhaps inside a separate function

```
main() {
  for (...) {
     ...
     g();
  }
}
g() {
  for (...)...
}
```

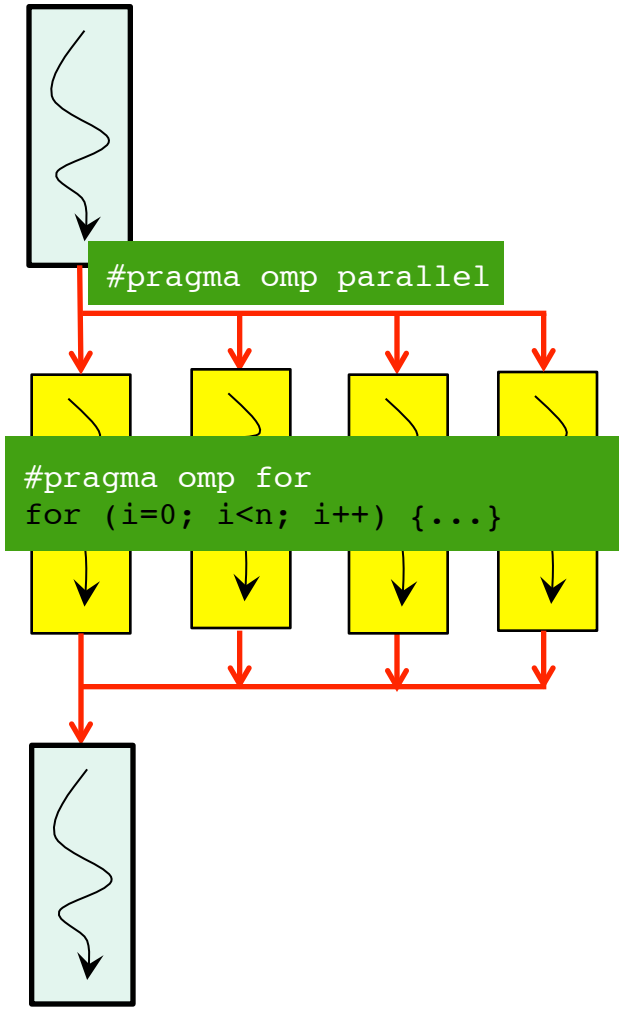
- what for parallel recursions?

```
qs() {
  if (...) ...
  else {
    qs();
    qs();
  }
}
```



parallel for can't handle nested parallelism

- OpenMP generally ignores nested parallel pragma when enough threads have been created by the outer parallel pragma, for good reasons
- The fundamental limitation is its simplistic work-sharing mechanism
- Tasks address these issues, by allowing tasks to be created at arbitrary points of execution (and a mechanism to distribute them across cores)



Task parallelism in OpenMP

- Syntax
  - create a task ≈ TBB's task_group::run

```
#pragma omp task
  s
```

  - wait for tasks ≈ TBB's task_group::wait

```
#pragma omp taskwait
```

Single/Master

```
#pragma omp single
  S
```

- Only one of the threads will execute the following block of code (S)
- The rest will wait for it to complete
- Good for non-thread-safe regions of code (such as I/O)
- Must be used in a parallel region
- Applicable to parallel for sections

```
#pragma omp master
  S
```

- The following block of code will be
- executed by the master thread
- No synchronization involved
- Applicable only to parallel sections

```
#pragma omp parallel
{
    do_preprocessing();
#pragma omp single
    read_input();
#pragma omp master
    notify_input_consumed();

    do_processing();
}
```

OpenMP task parallelism template

- don't forget to create a `parallel` region
- don't also forget to enter a `master` region, which says only the master executes the following statement and others "stand-by"

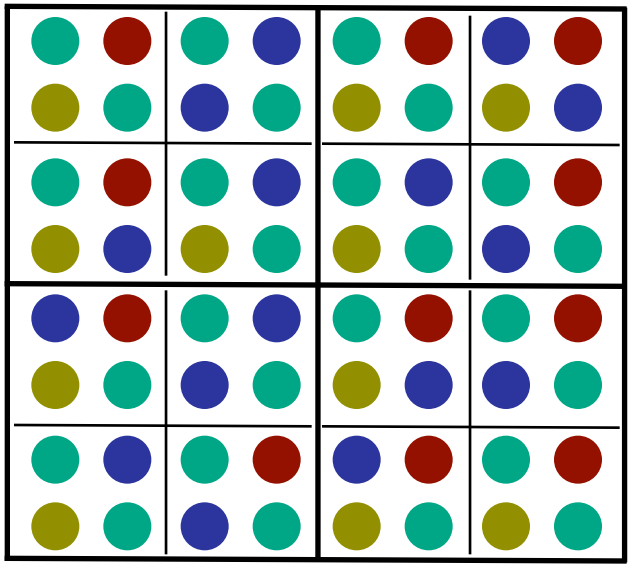
```
int main() {
#pragma omp parallel
#pragma omp master
// or #pragma omp single
   ms (a, a+n, t, 0);
}
```

- and create tasks in the master region

```
void ms(a, a_end, t, dest) {
   if (n == 1) {
    ...
   } else{
    ...
#pragma omp task
    ms(a, c, t, 1-dest);
#pragma omp task
    ms(c, a_end, t+nh, 1-dest);
#pragma omp taskwait
    ...
}
```

Task parallel example

```
#pragma omp for collapse(2) schedule(runtime)
  for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
       unit_work(i, j);
```



```
void work_rec(rectangle b) {
   if (small(b)) {
    ...
   } else {
       rectangle c[2][2];
       split(b, c); //split b into
                    // 2x2 sub-rectangles
       for (i = 0; i < 2; i++) {
          for (i = 0; i < 2; i++) {
#pragma omp task
             work_rec(b[i][j]);
          }
       }
#pragma omp taskwait
     }
}
```

Pros/cons of schedulers (1)

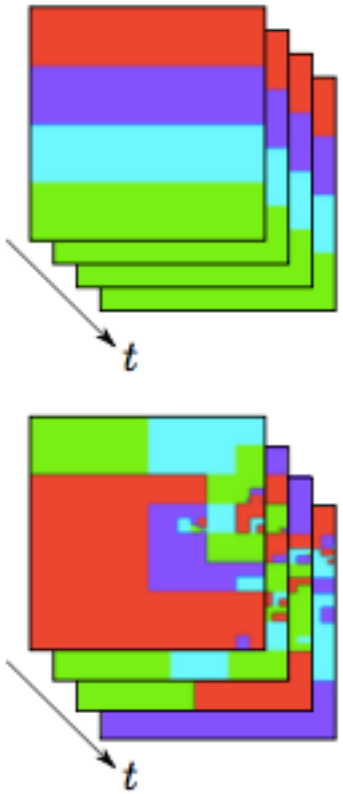
- static:
  - partitioning iterations is simple and does not require communication
  - may cause load imbalance (leave some threads idle, even when other threads have many work to do)
  - mapping between work ↔ thread is deterministic and predictable (why it's important?)
- dynamic:
  - no worry about load imbalance, if chunks are sufficiently small
  - partitioning iterations needs communication (no two threads execute the same iteration) and may become a bottleneck
  - mapping between iterations and threads is non-deterministic
  - OpenMP's dynamic scheduler is inflexible in partitioning loop nests

Pros/cons of schedulers (2)

- recursive (divide and conquer + tasks):
  - no worry about load imbalance, as in dynamic
  - distributing tasks needs communication, but efficient implementation techniques are known
  - mapping between work and thread is non-deterministic, as in dynamic
  - you can flexibly partition loop nests in various ways (e.g., keep the space to square-like)
  - need boilerplate coding efforts (easily circumvented by additional libraries; e.g., TBB's blocked range2d and parallel for)

Deterministic and predictable schedulers

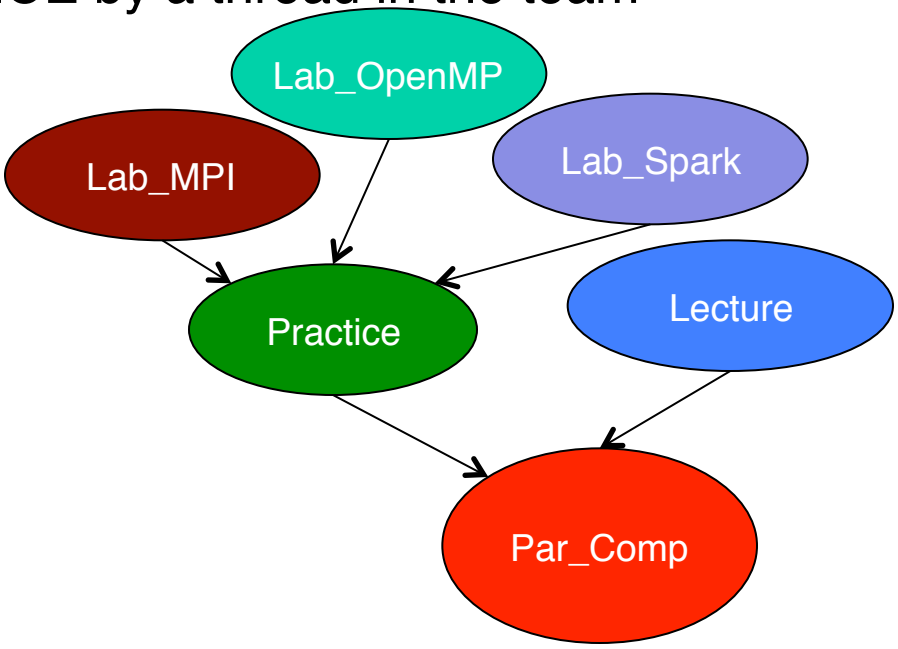
- programs often execute the same for loops many times, with the same trip counts, and with the same iteration touching a similar region
- such iterative applications may benefit from reusing data brought into cache in the previous execution of the same loop
- a deterministic scheduler achieves this benefit



Work sharing: sections (1)

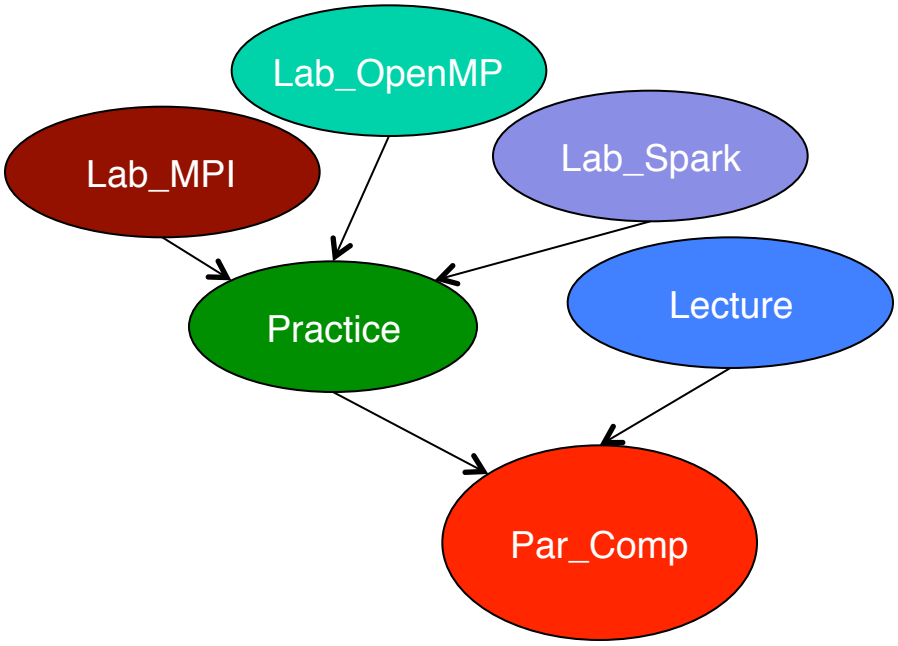
- SECTIONS directive is a non-iterative work-sharing construct
  - It specifies that the enclosed section(s) of code are to be divided among the threads in the team
  - Each SECTION is executed ONCE by a thread in the team

```
a = Lab_OpenMP();
b = Lab_MPI();
c = Lab_Spark();
P = Practice(a, b, c);
T = Lecture();
F = Par_Comp (T, P);
```



Work sharing: sections (2)

```
#pragma omp parallel sections
{
#pragma omp section
    a = Lab_OpenMP();
#pragma omp section
    b = Lab_MPI();
#pragma omp section
    c = Lab_Spark();
#pragma omp section
    T = Lecture();
}
P = Practice(a, b, c);
F = Par_Comp (T, P);
```



Synchronization

- Explicit barrier

```
#pragma omp barrier
```

- Implicit barrier at end of parallel region

```
#pragma omp parallel
{...
} // implicit barrier
```

```
#pragma omp for
{...
} // implicit barrier
```

```
#pragma omp parallel sections
{...
} // implicit barrier
```

```
#pragma omp single
{...
} // implicit barrier
```

- No barrier: nowait cancels barrier creation

```
#pragma omp parallel nowait
{...}
```

```
#pragma omp for nowait
{...}
```

```
#pragma omp parallel section nowait
{...}
```

```
#pragma omp single nowait
{...}
```

Synchronization: barrier

Both loops are in parallel region with no synchronization in between.

What is the problem?

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

for(i=0; i<N; i++)
    d[i] = a[i] + b[i]
```

fix

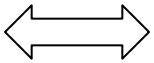
```
for(i=0; i<N; i++)
    a[i] = b[i] + c[i];

#pragma omp barrier

for(i=0; i<N; i++)
    d[i] = a[i] + b[i]
```

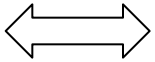
Serial & parallel code

```
#pragma omp parallel
#pragma omp for
    for (...)
```



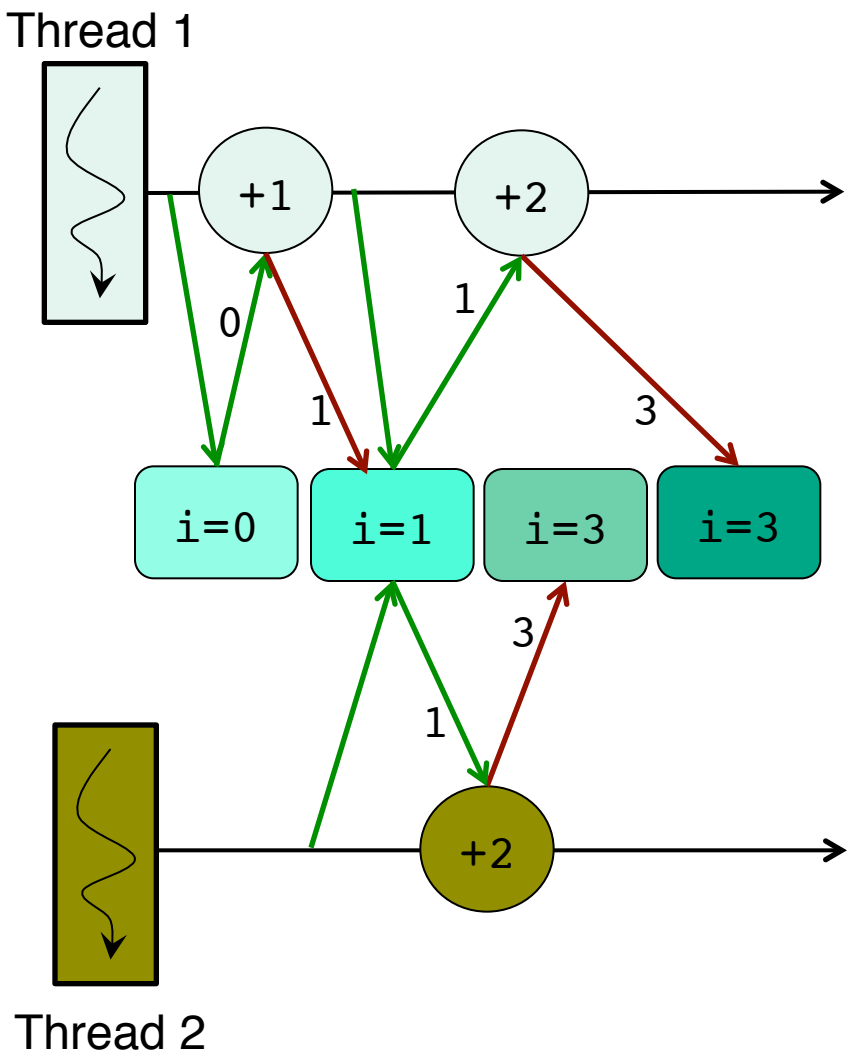
```
#pragma omp parallel for
for (...)
```

```
#pragma omp parallel
#pragma omp sections
{...}
```



```
#pragma omp parallel sections
{...}
```

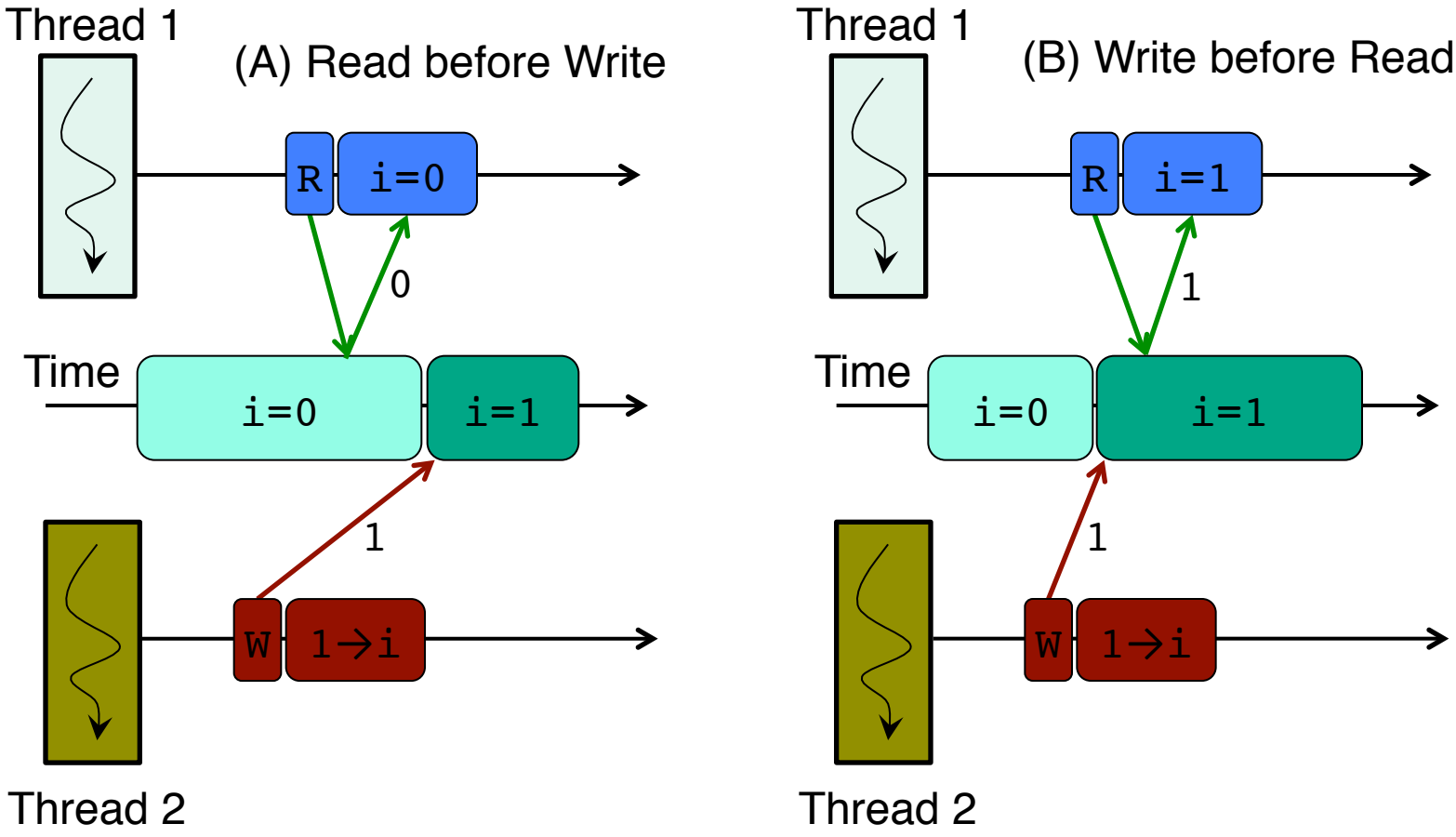
Race condition (1)



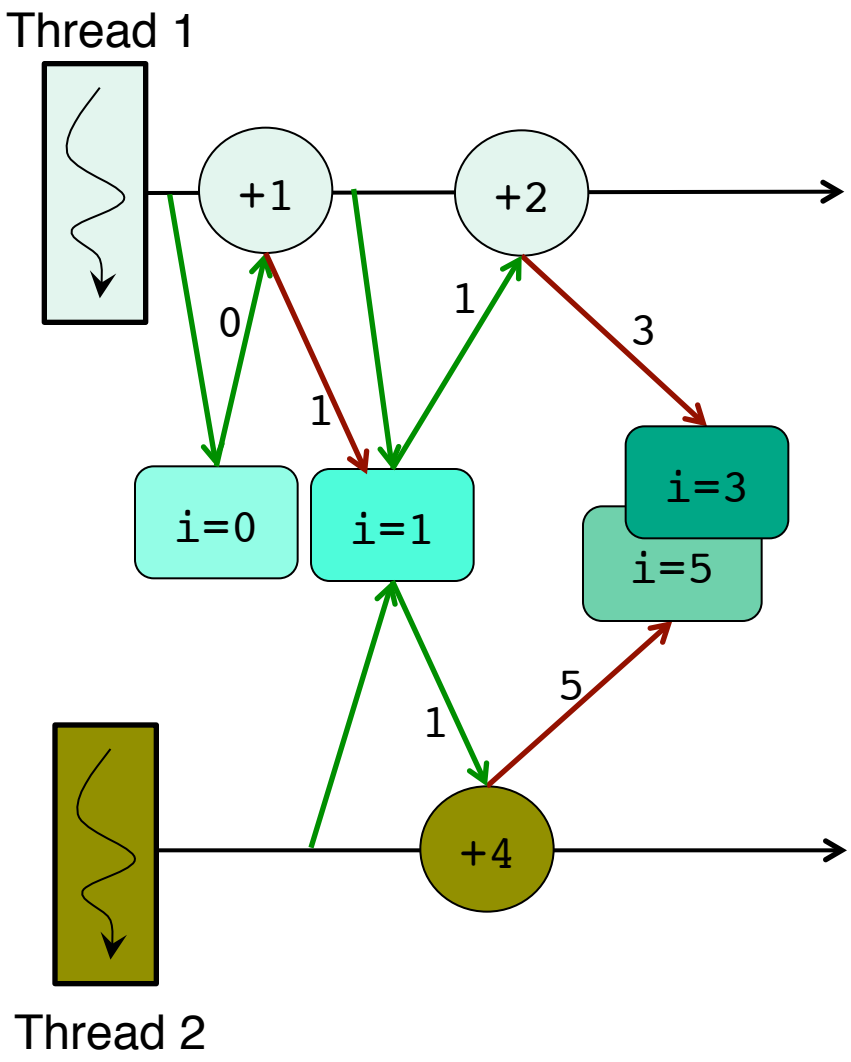
- A ***race condition*** or ***race hazard*** is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events
  - A race condition occurs when two threads can access (read or write) a data variable simultaneously, and at least one of the two accesses is a write

Race condition (2)

- A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations



Data race



- A ***data race*** occurs when two instructions from different threads access the same memory location, at least one of these accesses is a write

Critical section

```
#pragma omp critical [name]
    s
```

- Critical section (**s**): a portion of code that only thread at a time may execute
- A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name
- All unnamed critical sections are considered to have the same unspecified name

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access.

Computing Pi by method of Numerical Integration

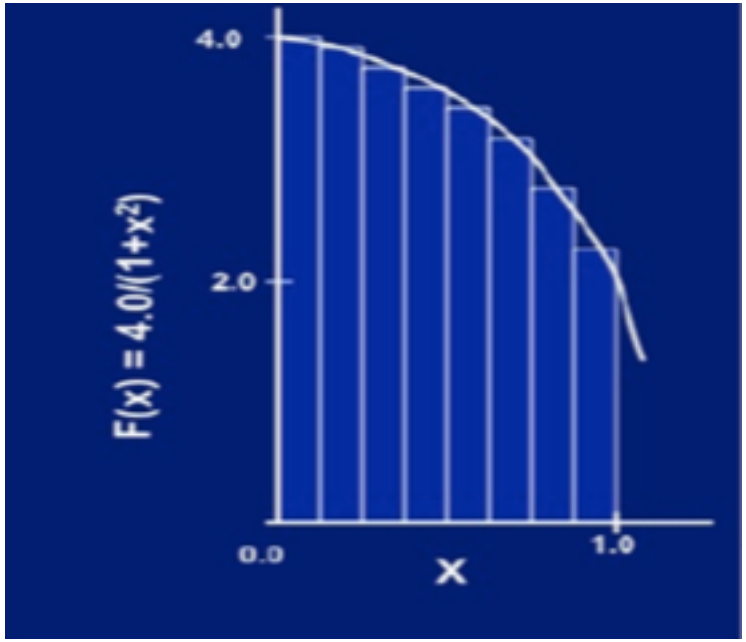
Mathematically, we know:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

And this can be approximated as a sum of the area of rectangles:

$$\sum_{i=1}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has a width of Δx and a height of $F(x_i)$ at the middle of interval i.



Serial code

```
static long num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0 / (double) num_steps;



    for (i = 0; i <= num_steps; i++)
    {
        x = (i + 0.5) * step;
       sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
}
```

Parallel code (1)

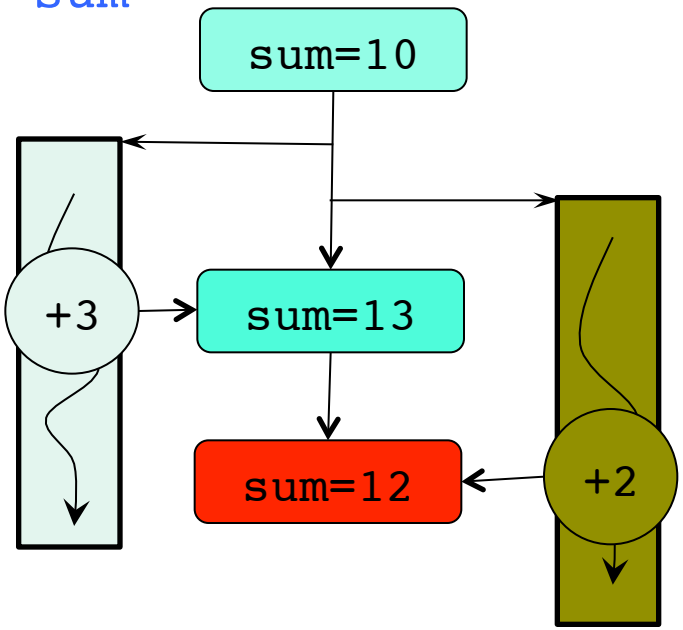
```
#include <omp.h>
#define NUM_THREADS 16

static long num_steps = 100000;
double step;

void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0 / (double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for private(x)
   for (i = 0; i <= num_steps; i++)
   {
      x = (i + 0.5) * step;
      sum = sum + 4.0 / (1.0 + x*x);
   }
   pi = step * sum;
}
```

- `sum` is a shared variable
- Race condition: one process may "race ahead" of another and not see its change to shared variable `sum`



Parallel code (2)

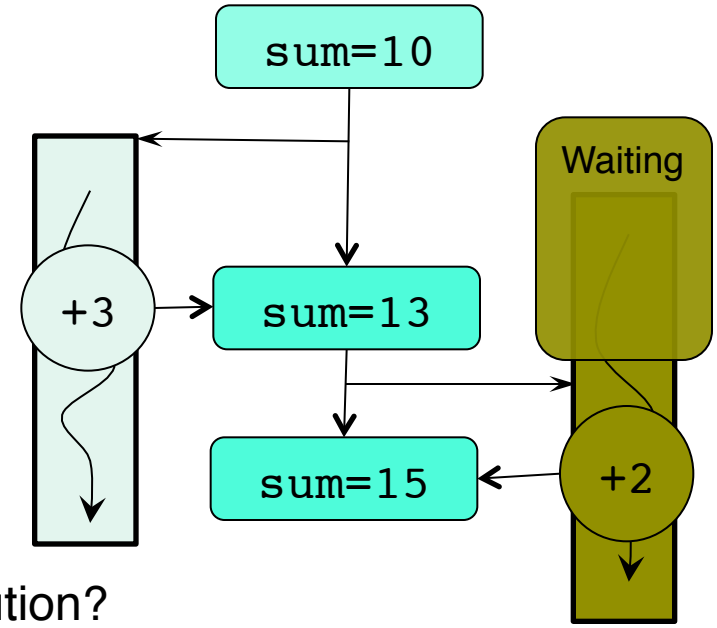
```
#include <omp.h>
#define NUM_THREADS 16

static long num_steps = 100000;
double step;

void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0 / (double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for private(x)
   for (i = 0; i <= num_steps; i++)
   {
      x = (i + 0.5) * step;
   #pragma omp critical
     sum = sum + 4.0 / (1.0 + x*x);
   }
   pi = step * sum;
}
```

- Update to sum inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code



What is a better solution?

Parallel code (3): Reduction

```
static long num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0 / (double) num_steps;



    for (i = 0; i <= num_steps; i++)
    {
        x = (i + 0.5) * step;
       sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
}
```

Serial code

```
#include <omp.h>
#define NUM_THREADS 16

static long num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum)
                                 private(x)
    for (i = 0; i <= num_steps; i++)
    {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
}
```

Parallel code

OpenMP discussion

- Exposing architecture features (performance):
  - Not much, similar to the pthread approach
    - Assumption: dividing job into threads = improved performance
    - How valid is this assumption in reality?
      - Overheads, contentions, synchronizations, etc
  - This is one weak point for OpenMP: the performance of an OpenMP program is somewhat hard to understand.

OpenMP final thoughts

- Main issues with OpenMP: performance
  - Is there any obvious way to solve this?
    - Exposing more architecture features?
  - Is the performance issue more related to the fundamental way that we write parallel program?
    - OpenMP programs begin with sequential programs
    - May need to find a new way to write efficient parallel programs in order to really solve the problem