

# CS 610: OpenMP

Swarnendu Biswas

Semester 2022-2023-I

CSE, IIT Kanpur

---

Content influenced by many excellent references, see References slide for acknowledgements.

Material adapted from

- Several tutorials by Tim Mattson et al.
- P. Sadayappan, Ohio State, CS 5441
- Blaise Barney, OpenMP Tutorial, LLNL

# What is OpenMP?

- OpenMP (Open Multi-Processing) is a popular directive-based parallel programming model for shared-memory systems
  - OpenMP program is essentially a sequential program augmented with compiler directives to specify parallelism, eases conversion of existing sequential programs
- Standardizes established SMP practice + vectorization and heterogeneous device programming
- OpenMP supports C/C++ and Fortran on a wide variety of architectures
  - Supported by popular C/C++ compilers, for e.g., LLVM/Clang, GNU GCC, Intel ICC, and IBM XLC

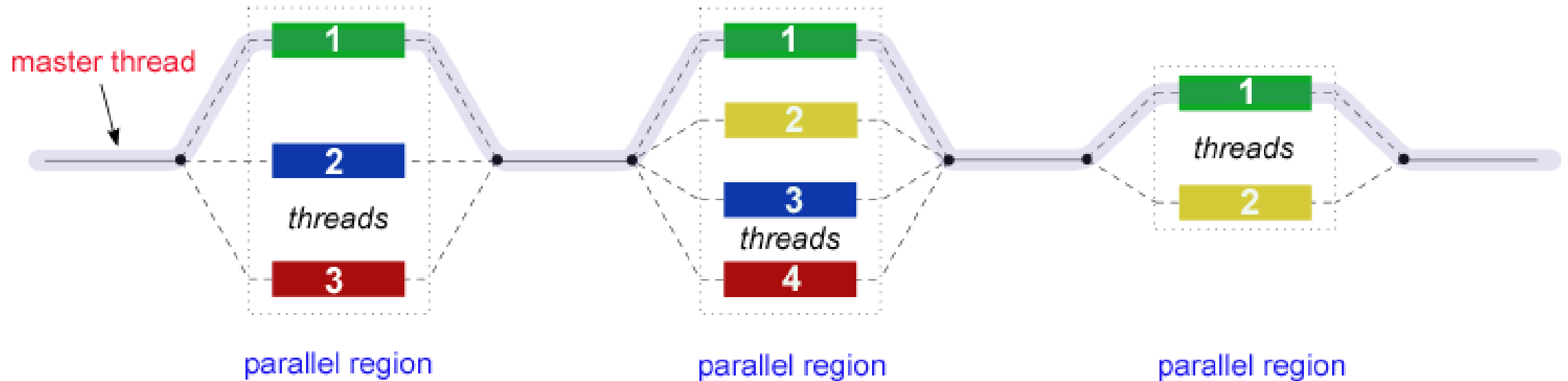
# Key Concepts in OpenMP

- **Parallel regions** where parallel execution occurs via multiple concurrently-executing threads
  - Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution
- **Shared and private data:** shared variables are the means of communicating data between threads
- **Synchronization:** fundamental means of coordinating execution of concurrent threads
- Mechanism for **automated work distribution** across threads

# Goals of OpenMP

- Standardization
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- Ease of use
  - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all-or-nothing approach
  - Provide the capability to implement both **coarse-grain and fine-grain parallelism**
- Portability
  - Most major platforms and compilers support OpenMP

# Fork-Join Model of Parallel Execution



# Other Key Features

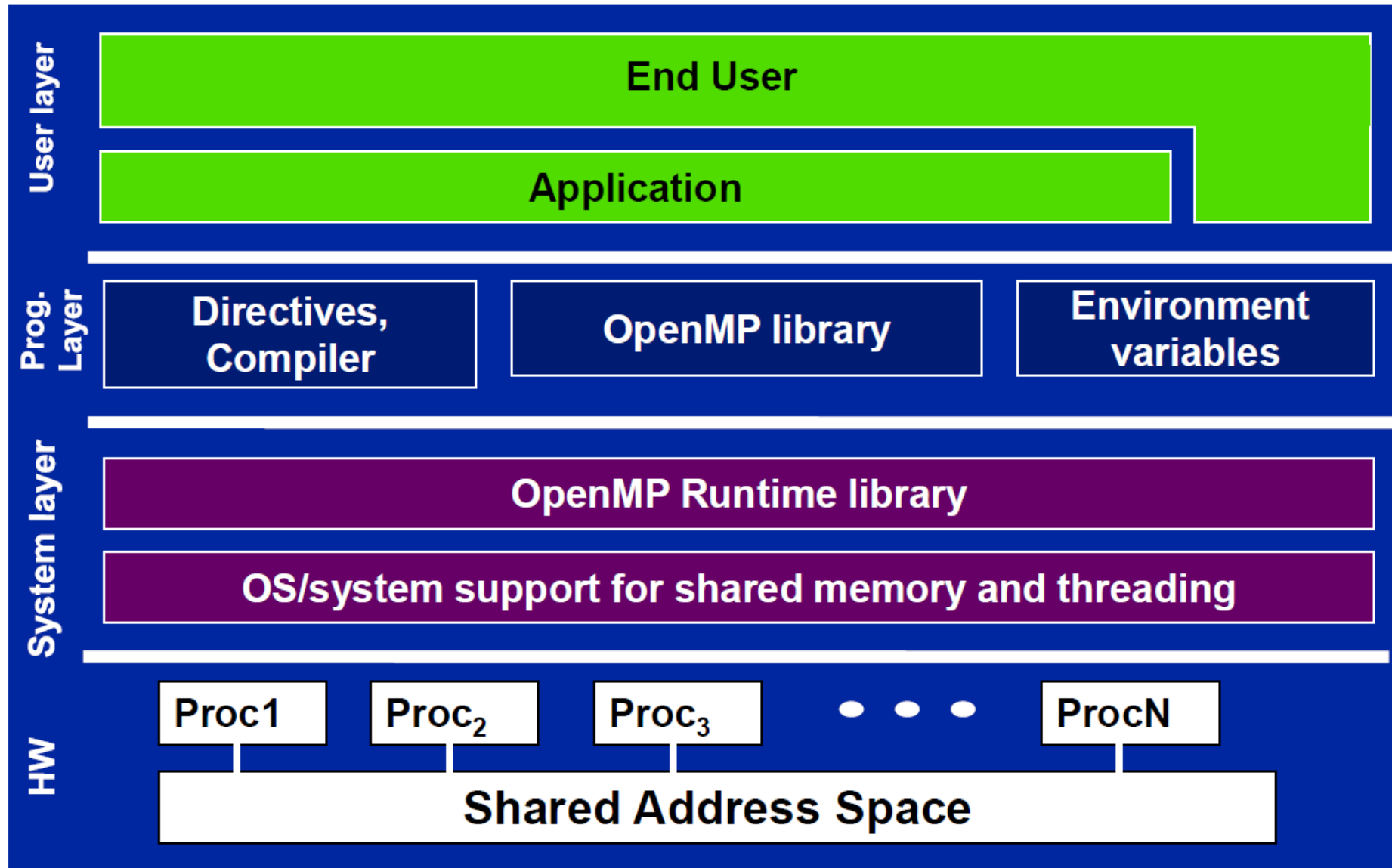
- **Scope of data**
  - Most data within a parallel region is shared by default
  - Programmers can control scope of access to data by different threads
- **Nested parallelism**
  - You can invoke a parallel region within another parallel region
- **Dynamic threads and scheduling**
  - The runtime can dynamically alter the number of parallel threads and also control the scheduling of work among threads

# The OpenMP API

- Compiler directives: `#pragma omp parallel`
  - Comments, ignored unless instructed not to
- Runtime library routines: `int omp_get_num_threads(void);`
- Environment variables: `export OMP_NUM_THREADS=8`
- Most common constructs in OpenMP are compiler directives
  - `#pragma omp directive [clause [clause]...] newline`
  - Example: `#pragma omp parallel num_threads(4)`
- Function prototypes and types are defined in the header `omp.h`



# OpenMP Solution Stack



# General Code Structure

```
#include <omp.h>
...
int main() {
    ...
    // serial code, master thread
    ...
    // begin parallel section,
    // fork a team of threads
    #pragma omp parallel ...
    {
```

```
        // parallel region executed by
        // all threads

        // other logic
        ...
        // all parallel threads join
        // master thread
    }
    // resume serial code
    ...
}
```

# Structured Block

- Most OpenMP constructs apply to a **structured block**
- Structured block is a block of one or more statements surrounded by “{ }”, with one point of entry at the top and one point of exit at the bottom
  - It is okay to have an exit within the structured block
  - Disallows code that branches in to or out of the middle of the structured block

# Format of Compiler Directives

- `#pragma omp`
  - Required for all OpenMP C/C++ directives
- `directive-name`
  - A valid OpenMP directive. Must appear after the pragma and before any clauses
  - Scope extends to the the structured block following a directive, does not span multiple routines or code files
- `[clause, ...]`
  - Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted
- `newline`
  - Required. Precedes the structured block which is enclosed by this directive.

# Compiling an OpenMP Program

- Linux and GNU GCC
  - `g++ -fopenmp hello-world.cpp`
- Linux and Clang/LLVM
  - `clang++ -fopenmp hello-world.cpp`
- Can use the preprocessor macro `_OPENMP` to check for compiler support

# Hello World with OpenMP!

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

using namespace std;

int main() {
    cout << "This is serial code\n";
    #pragma omp parallel
    {
        int num_threads = omp_get_num_threads();
        int tid = omp_get_thread_num();
        if (tid == 0) {
            cout << num_threads << "\n";
        }
        cout << "Hello World: " << tid << "\n";
    }
}
```

```
    cout << "This is serial code\n";
    #pragma omp parallel num_threads(2)
    {
        int tid = omp_get_thread_num();
        cout << "Hello World: " << tid << "\n";
    }
    cout << "This is serial code\n";
    omp_set_num_threads(3);
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        cout << "Hello World: " << tid << "\n";
    }
}
```

# Hello World with OpenMP!

- Each thread has a unique integer “id”; master thread has “id” 0, and other threads have “id” 1, 2, ...
- OpenMP runtime function `omp_get_thread_num()` returns a thread’s unique “id”
- The function `omp_get_num_threads()` returns the total number of executing threads
- The function `omp_set_num_threads(x)` asks for “x” threads to execute in the next parallel region (must be set outside region)

# Types of Parallelism with OpenMP

## Coarse-grained

- Task parallelism
  - Split the work among threads that execute in parallel
  - Implicit join at the end of the segment, or explicit synchronization points

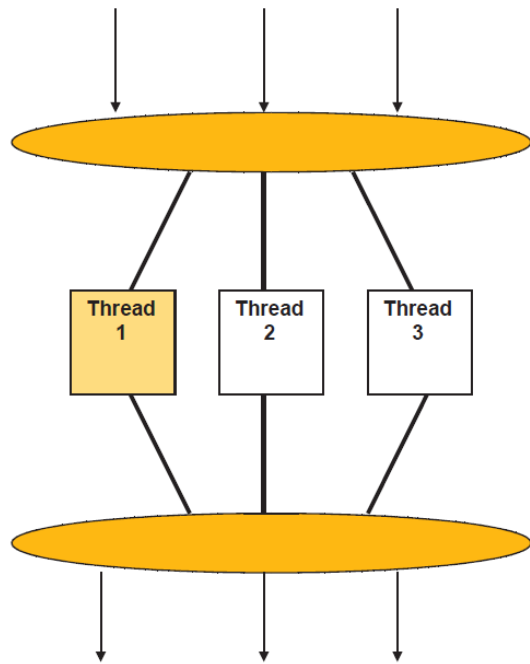
## Fine-grained

- Loop parallelism
  - Execute independent iterations of for-loops in parallel
  - Several choices in splitting the work

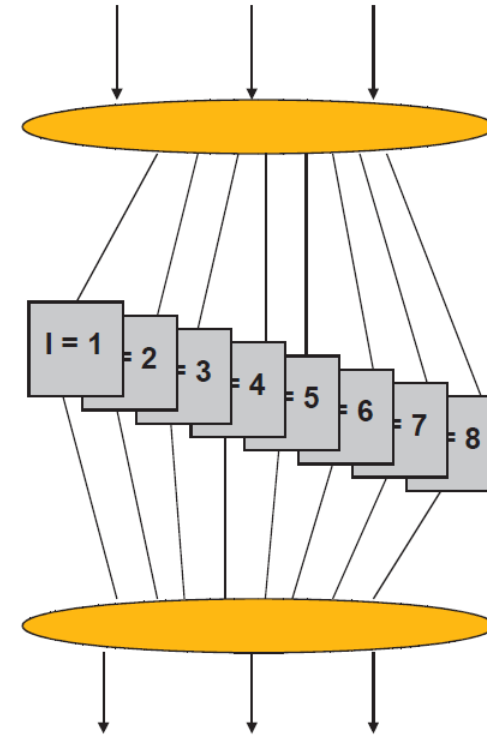


# Types of Parallelism with OpenMP

## Task parallelism



## Loop parallelism



# The Essence of OpenMP

- **Create threads that execute in a shared address space**
  - The only way to create threads is with the `parallel` construct
  - Once created, all threads execute the code inside the construct
- **Split up the work between threads by one of two means**
  - SPMD (Single Program Multiple Data) – all threads execute the same code and you use the thread ID to assign work to a thread
  - Workshare constructs split up loops and tasks between threads
- **Manage data environment to avoid data access conflicts**
  - Synchronization so correct results are produced regardless of how threads are scheduled
  - Carefully manage which data can be private (local to each thread) and shared

# OpenMP Constructs

Lexical or  
static extent

- Construct consists of an executable directive and the associated loop, statement, or structured block

```
#pragma omp parallel
{
    // inside parallel construct
    subroutine();
}

void subroutine () {
    // outside parallel construct
}
```

# OpenMP Regions

Run time or  
dynamic  
extent

- Region consists of all code encountered during a specific instance of the execution of a given construct
  - Also includes implicit code introduced by the OpenMP implementation

```
#pragma omp parallel
{
    // inside parallel region
    subroutine( );
}

void subroutine (void) {
    // inside parallel region
}
```

# Parallel Region Construct

- Block of code that will be executed by multiple threads
- `#pragma omp parallel [clause ...]`  
    `structured_block`
- Example of clauses
  - `private (list)`
  - `shared (list)`
  - `default (shared | none)`
  - `firstprivate (list)`
  - `reduction (operator: list)`
  - `num_threads (integer-expression)`
  - ...

# Parallel Region Construct

- When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team
  - By default OpenMP creates as many thread as many cores available in the system
- The master is a member of that team and has thread number 0 within that team
- The code is duplicated and all threads will execute that code
- There is an implied barrier at the end of a parallel section
- Only the master thread continues execution past this point

# Threading in OpenMP

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

- OpenMP implementations use a **thread pool** so full cost of threads creation and destruction is not incurred for reach parallel region
- Only three threads are created excluding the parent thread

```
void thunk () {
    foobar ();
}

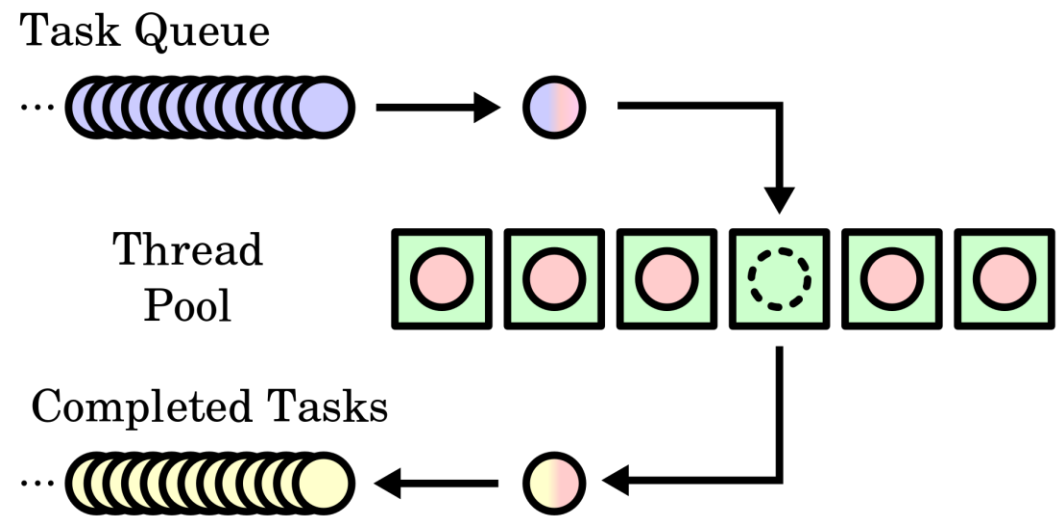
pthread_t tid[4];

for (int i = 1; i < 4; ++i)
    pthread_create (&tid[i], 0, thunk, 0);

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

# Thread Pool

- Software design pattern
- Maintains a pool of threads waiting for work
- Advantageous when work is short-lived
  - Avoid the overhead of frequent thread creation and destruction
- Excess threads can degrade performance
  - Memory, context-switch, and other resource overhead





# Specifying Number of Threads

- Desired number of threads can be specified in many ways
  - Setting environmental variable `OMP_NUM_THREADS`
  - Runtime OpenMP function `omp_set_num_threads(4)`
  - Clause in `#pragma` for parallel region

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int t_id = omp_get_thread_num();
    int nthrs = omp_get_num_threads();
    for (int i = t_id; i < 1000; i += nthrs)
    {
        A[i] = foo(i);
    }
}
```

# Specifying Number of Threads

- Three ways
  - `OMP_NUM_THREADS`
  - `omp_set_num_threads(...)`
  - `#pragma omp parallel num_threads(...)`
- `OMP_NUM_THREADS` (if present) specifies initially the number of threads
- Calls to `omp_set_num_threads()` override the value of `OMP_NUM_THREADS`
- Presence of the `num_threads` clause overrides both other values

# Distributing Work

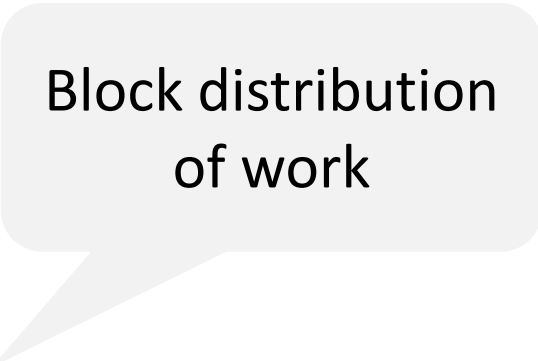
- Threads can perform disjoint work division using their thread ids and knowledge of total # threads

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    for (int i = t_id; i < 1000; i += omp_get_num_threads()) {
        A[i]= foo(i);
    }
}
```

Cyclic distribution  
of work

# Distributing Work

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    int num_thrs = omp_get_num_threads();
    int b_size = 1000 / num_thrs;
    for (int i = t_id*b_size; i < (t_id+1)*b_size; i += num_thrs) {
        A[i]= foo(i);
    }
}
```

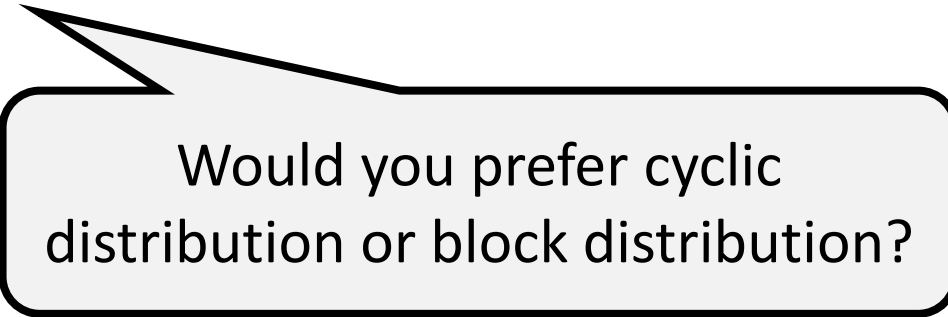


Block distribution  
of work

# Cyclic vs Block Distribution of Work

- Say I have a computation like

```
for (int i = 0; i < N; i++) {  
    A[i] = B[i] + C[i];  
}
```



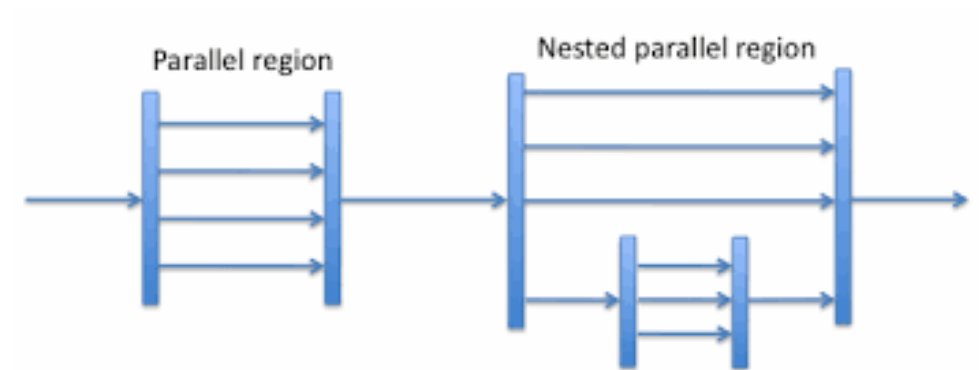
Would you prefer cyclic distribution or block distribution?

# Other Subtle Issues about parallel Construct

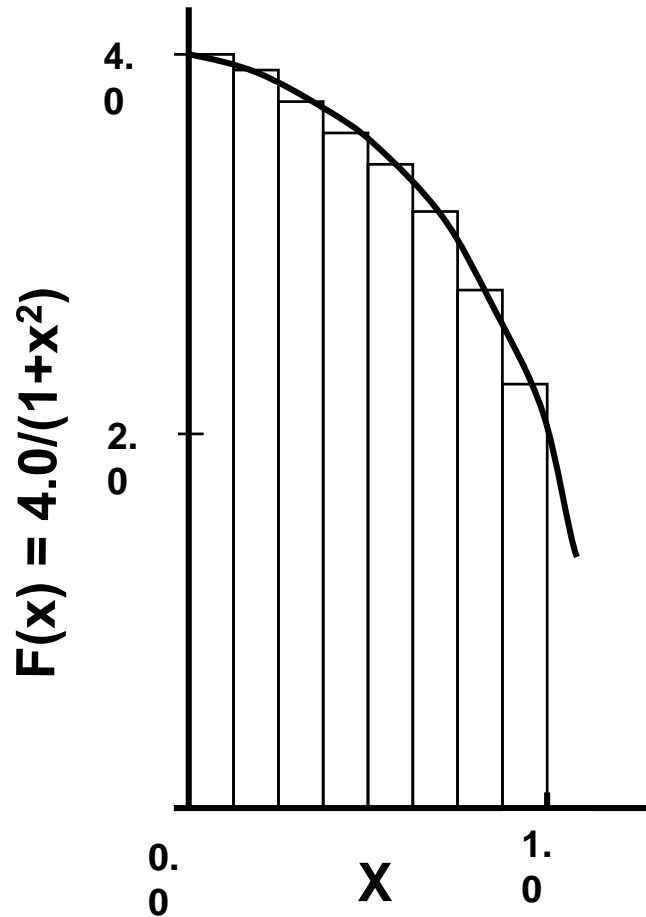
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team
- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates
  - The order of termination of threads is unspecified

# Nested Parallelism

- Allows to create parallel region within a parallel region itself
- Nested parallelism can help scale to large parallel computations
- Usually turned off by default
  - Can lead to oversubscription by creating lots of threads
- Set `OMP_NESTED` as `TRUE` or call `omp_set_nested()`



# Recurring Example of Numerical Integration



- Mathematically

$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$

- We can approximate the integral as the sum of the rectangles

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

where each rectangle has width  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$



# Serial Pi Program

```
double seq_pi() {
    int i;
    double x, pi, sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    for (i = 0; i < NUM_STEPS; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    return pi;
}
```

```
$ g++ -fopenmp compute-pi.cpp
$ ./a.out
3.14159
```

# Computing Pi with OpenMP

```
double omp_pi_with_fs() {
    omp_set_num_threads(NUM_THRS);
    double sum[NUM_THRS] = {0.0};
    double pi = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;

    #pragma omp parallel
    {
        // Parallel region with worker threads
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
```

```
        if (tid == 0) {
            num_thrs = nthrds;
        }
        double x;
        for (int i = tid; i < NUM_STEPS; i+=nthrds) {
            x = (i + 0.5) * step;
            sum[tid] += 4.0 / (1.0 + x * x);
        }
    } // end #pragma omp parallel
    for (int i = 0; i < num_thrs; i++) {
        pi += (sum[i] * step);
    }
    return pi;
}
```

# Computing Pi with OpenMP

```
double omp_pi_with_fs() {
    omp_set_num_threads(NUM_THRS);
    double sum[NUM_THRS] = {0.0};
    double pi = 0.0;
    double step = 1.0 / NUM_THRS;
    uint16_t num_threads = NUM_THRS;

    #pragma omp parallel
    {
        // Parallel region with worker threads
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
```

```
        if (tid == 0) {
            num_thrs = nthrds;
        }
        double x = 0.0;
        for (int i = 0; i < num_thrs; i++) {
            x = (i * step);
            sum[i] = x * x;
        }
    } // end #pragma omp parallel

    for (int i = 0; i < num_thrs; i++) {
        pi += (sum[i] * step);
    }
    return pi;
}
```

Great! This is a correct implementation, but...

Is there any problem with this code?

# Avoid False Sharing

- Array `sum[ ]` is a shared array, with each thread accessing exactly one element
- Cache line holding multiple elements of `sum` will be locally cached by each processor in its private L1 cache
- When a thread writes into an index in `sum`, the entire cache line becomes “dirty” and causes invalidation of that line in all other processor’s caches
- Cache thrashing due to this “false sharing” causes performance degradation

# Optimize the Pi Program: Avoid False Sharing

```
double omp_pi_without_fs1() {
    omp_set_num_threads(NUM_THRS);
    double sum[NUM_THRS][8];
    double pi = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;
#pragma omp parallel
    {
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
        if (tid == 0) {
            num_thrs = nthrds;
        }
    }
}
```

```
double x;
for (int i = tid; i < NUM_STEPS; i+=nthrds) {
    x = (i + 0.5) * step;
    sum[tid][0] += 4.0 / (1.0 + x * x);
}
} // end #pragma omp parallel

for (int i = 0; i < num_thrs; i++) {
    pi += (sum[i][0] * step);
}
return pi;
}
```

# Optimize the Pi Program: Avoid False Sharing

```
double omp_pi_without_fs1() {
    omp_set_num_threads(NUM_THRS);
    double sum = 0;
    double pi = 0;
    double step = 1;
    uint16_t tid = 0;
#pragma omp parallel
    {
        uint16_t nthreads = omp_get_num_threads();
        uint16_t nthrds = omp_get_num_threads();
        if (tid == 0) {
            num_thrs = nthrds;
        }
        double x;
        for (int i = tid; i < NUM_STEPS; i+=nthrds) {
            sum += 4.0 / (1 + step * x);
        }
        pi += sum;
        return pi;
    }
}
```

How did we decide that PADDING has to be 8?

- Depends on the cache line size and the data type
- This is not portable, since it may not work across different cache configurations, architectures, and data types

# Optimize the Pi Program: Avoid False Sharing

```
double omp_pi_without_fs2() {
    omp_set_num_threads(NUM_THRS);
    double pi = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;
#pragma omp parallel
    {
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
        if (tid == 0) {
            num_thrs = nthrds;
        }
    }
}
```

```
double x, sum;
for (int i = tid; i < NUM_STEPS; i+=nthrds) {
    x = (i + 0.5) * step;
    // Scalar variable sum is
    // thread-private, so no false sharing
    sum += 4.0 / (1.0 + x * x);
}

pi += (sum * step);
} // end #pragma omp parallel

return pi;
}
```

# Optimize the Pi Program: Avoid False Sharing

```
double omp_pi_without_fs2() {
    omp_set_num_threads(NUM_THRS);
    double pi = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;
#pragma omp parallel
    {
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
        if (tid == 0) {
            num_thrs = nthrds;
        }
    }
}
```

```
double x, sum;
for (int i = tid; i < NUM_STEPS; i+=nthrds) {
    x = (i + 0.5) * step;
    // Scalar variable sum is
    // so no false sharing
    sum += x * x);

    pi += (sum * step);
} // end #pragma omp parallel

return pi;
}
```

This program is now wrong! Why?



# Synchronization Constructs

# critical Construct

- Only one thread can enter critical section at a time; others are held at entry to critical section
- Prevents any race conditions in updating res

```
float res;
#pragma omp parallel
{
    float B;
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    for (int i = id; i < MAX; i += nthrds) {
        B = big_job(i);
        #pragma omp critical
            consume (B, res);
    }
}
```

# critical Construct

- Works by acquiring a lock
- If your code has multiple `critical` sections, they are all mutually exclusive
- You can avoid this by naming `critical` sections
  - `#pragma omp critical (optional_name)`

# Correct Pi Program: Fix the Data Race

```
double omp_pi_without_fs2() {
    omp_set_num_threads(NUM_THRS);
    double pi = 0.0, step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;
    #pragma omp parallel
    {
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
        if (tid == 0) {
            num_thrs = nthrds;
        }
    }
}
```

```
double x, sum;
for (int i = tid; i < NUM_STEPS; i += nthrds)
{
    x = (i + 0.5) * step;
    // Scalar variable sum is
    // thread-private, so no false sharing
    sum += 4.0 / (1.0 + x * x);
}
#pragma omp critical // Mutual exclusion
    pi += (sum * step);
} // end #pragma omp parallel

return pi;
}
```

# Evaluate the Pi Program Variants

- Sequential computation of pi
- Parallel computation with false sharing
- Parallel computation with padding
- Parallel computation with thread-local sum

```
Tilix: Default
Default x Default
l: swarnendu@cse-BM1AF-BP1AF-BM6AF: ~/iitk-workspace/parallel-computing/src/openmp
warnendu:~/iitk-workspace/parallel-computing/src/openmp$ ./a.out
alue of PI computed sequentially: 3.14159 in 0.0165884 seconds
alue of PI computed in parallel (with false sharing): 3.14159 in 0.0119428 seconds
alue of PI computed in parallel (without false sharing via padding): 3.14159 in 0.00325493 seconds
alue of PI computed in parallel (without false sharing via thread-private variables, sync out of loop): 3.14159 in 0.00326013 seconds
alue of PI computed in parallel with task sharing construct: 3.14159 in 0.00325819 seconds
warnendu:~/iitk-workspace/parallel-computing/src/openmp$
```

# atomic Construct

- Atomic is an efficient critical section for simple reduction operations
- Applies only to the update of a memory location
- Uses hardware atomic instructions for implementation; much lower overhead than using critical section

```
float res;
#pragma omp parallel
{
    float B;
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    for (int i = id; i < MAX; i += nthrds) {
        B = big_job(i);
#pragma omp atomic
        res += B;
    }
}
```

# atomic Construct

- Expression operation can be of type
  - `x binop= expr`
    - X is a scalar type
    - binop can be `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`
  - `x++`
  - `++x`
  - `x--`
  - `--x`

```
float res;
#pragma omp parallel
{
    float B;
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    for (int i = id; i < MAX; i += nthrds) {
        B = big_job(i);
        #pragma omp atomic
        res += B;
    }
}
```

# critical vs atomic

## critical

- Locks code segments
- Serializes all unnamed critical sections
- Less efficient than `atomic`
- More general

## atomic

- Locks data variables
- Serializes operations on the same shared data
- Makes use of hardware instructions to provide atomicity
- Less general



# Is `atomic` always the way to go?

```
int sum = 0, m_val = 0;

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

int getVal() {
  return ++m_val;
}
```

# Barrier Synchronization

```
#pragma omp parallel private(id)
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
```

```
#pragma omp barrier
```

explicit barrier

```
    B[id] = big_calc2(id);
}
```

- Each thread waits until all threads arrive

# Barrier Synchronization

```
#pragma omp parallel private(id)
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
```

```
#pragma omp barrier
```

explicit barrier

```
#pragma omp for
for (i=0;i<N;i++) {
    B[i]=big_calc2(i,A);
}
```

implicit barrier

```
#pragma omp for nowait
for (i=0;i<N;i++) {
    C[i]=big_calc2(B, i);
}
```

no implicit barrier, nowait  
cancels barrier creation

```
A[id] = big_calc4(id);
}
```

# Use of `nowait` clause

Can be useful if the two loops are independent

```
# pragma omp for nowait
for ( /* ... */ ) {
    // .. first loop ..
}
```

```
# pragma omp for
for ( /* ... */ ) {
    // .. second loop ..
}
```

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

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

# Clause `ordered`

- Specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a **serial** processor
- It must appear within the extent of `omp for` or `omp parallel for`
- **Should be used in two stages**

```
omp_set_num_threads(4);
#pragma omp parallel
{
#pragma omp for ordered
    for (int i=0; i<N; i++) {
        tmp = func1(i);
#pragma omp ordered
        cout << tmp << "\n";
    }
}
```

# Synchronization Constructs

## High-level

- `critical`
- `atomic`
- `barrier`
- `ordered`

## Low-level

- `locks`
- `flush`

# Synchronization with Locks

- More flexible than critical sections (can use multiple locks)
- `critical` locks a code segment, while locks lock data
- More error-prone
  - For example, deadlock if a thread does not unset a lock after acquiring it

```
omp_lock_t lck;  
omp_init_lock(&lck);  
#pragma omp parallel  
{  
    do_many_things();  
    omp_set_lock(&lck);  
    // critical section  
    omp_unset_lock(&lck);  
    do_many_other_things ();  
}  
omp_destroy_lock(&lck);
```

# Synchronization with Locks

- Nested locks can be acquired if it is available or owned by the same thread

- `omp_init_lock()`
- `omp_set_lock()`
- `omp_unset_lock()`
- `omp_test_lock()`
- `omp_destroy_lock()`
  
- `omp_init_nest_lock()`
- `omp_set_nest_lock()`
- `omp_unset_nest_lock()`
- `omp_test_nest_lock()`
- `omp_destroy_nest_lock()`



# Example Program

```
int find_total_mass() {
    omp_lock_t lock;
    omp_init_lock(&lock);
#pragma omp parallel for
    for(int i=0;i<n;i++) {
        omp_set_lock(&lock);
        total_mass += mass[i];
        omp_unset_lock(&lock);

        while(!omp_test_lock(&lock)) {
            find_active_neighbors(list);
        }
    }
}
```

```
        total_mass = sum_neigh(list);
        omp_unset_lock(&lock);
    }

    omp_destroy_lock(&lock);
    return total_mass;
}
```

# Synchronization Construct: `flush`

- `#pragma omp flush (list)`
- Identifies a point at which a thread is guaranteed to see a consistent view of memory with respect to the variables in “`list`”
  - Flush forces data to be updated in memory so other threads see the most recent value
- In the absence of a list, **all** shared objects are synchronized

# Synchronization Construct: `flush`

- If `list` contains a pointer, the pointer is flushed, not the object referred to by the pointer
- It is recommended not to use flushes, excepting certain cases where you want to implement say your own spin lock
- Flushes are expensive, since they require compilers to generate memory fences

# Clause master

```
#pragma omp parallel  
{  
    do_many_things();  
#pragma omp master  
{  
    reset_boundaries();  
}  
do_many_other_things();  
}
```

multiple threads  
of control

only master thread executes this  
region, other threads just skip it,  
no barrier is implied

multiple threads  
of control

# Clause `single`

```
#pragma omp parallel  
{
```

multiple threads  
of control

```
  do_many_things();
```

```
#pragma omp single
```

```
{
```

```
  reset_boundaries();
```

a single thread executes  
this region, may not be  
the master thread

```
}
```

implicit barrier, all other threads wait;  
can remove with `nowait` clause

```
  do_many_other_things();
```

multiple threads  
of control

```
}
```

# Simplify Control Flow: Use `single`

```
double omp_pi_without_fs2() {
    omp_set_num_threads(NUM_THRS);
    double pi = 0.0, step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;
    #pragma omp parallel
    {
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();

        #pragma omp single
            num_thrs = nthrds;
    }
}
```

```
double x, sum;
for (int i = tid; i < NUM_STEPS; i+=nthrds) {
    x = (i + 0.5) * step;
    // Scalar variable sum is
    // thread-private, so no false sharing
    sum += 4.0 / (1.0 + x * x);
}
#pragma omp critical // Mutual exclusion
    pi += (sum * step);
}
return pi;
}
```

# Reductions in OpenMP

- Reductions are common patterns
  - True dependence that cannot be removed
- OpenMP provides special support via `reduction` clause
  - OpenMP compiler automatically creates local variables for each thread, and divides work to form partial reductions, and code to combine the partial reductions
- Predefined set of **associative** operators can be used with reduction clause,
  - For e.g., `+`, `*`, `-`, `min`, `max`

```
double sum = 0.0;

omp_set_num_threads(N);
#pragma omp parallel
{
    double my_sum = 0.0;
    my_sum = func(omp_get_thread_num());
#pragma omp critical
    sum += my_sum;
}
```

# Reductions in OpenMP

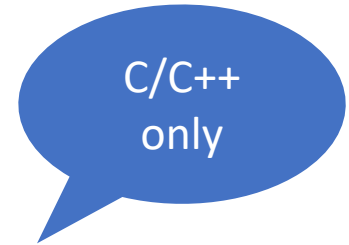
- Reductions clause specifies an operator and a list of reduction variables (must be **shared** variables)
- OpenMP compiler creates a local copy for each reduction variable, initialized to operator's identity (e.g., 0 for +; 1 for \*)
- After work-shared loop completes, contents of local variables are combined with the "entry" value of the shared variable
- Final result is placed in shared variable

```
double sum = 0.0;

omp_set_num_threads(N);
#pragma omp parallel reduction(+ : sum)
    sum += func(omp_get_thread_num());
```



# Reduction Operators and Initial Values



Operator	Initial value
+	0
*	1
-	0
Min	Largest positive number
Max	Smallest negative number

Operator	Initial value
&	~0
	0
^	0
&&	1
	0

# Computing Pi with OpenMP

```
double omp_pi_with_fs() {
    omp_set_num_threads(NUM_THRS);
    double sum[NUM_THRS] = {0.0};
    double pi = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    uint16_t num_thrs;

    #pragma omp parallel
    {
        // Parallel region with worker threads
        uint16_t tid = omp_get_thread_num();
        uint16_t nthrds = omp_get_num_threads();
```

```
#pragma omp single
    num_thrs = nthrds;
    double x;
    for (int i = tid; i < NUM_STEPS; i+=nthrds) {
        x = (i + 0.5) * step;
        sum[tid] += 4.0 / (1.0 + x * x);
    }
} // end #pragma omp parallel

#pragma omp parallel for reduction(+ : pi)
for (int i = 0; i < num_thrs; i++) {
    pi += (sum[i] * step);
}
return pi;
}
```

# Data Sharing

# Understanding Scope of Shared Data

- As with any shared-memory programming model, it is important to identify shared data
  - Multiple child threads may read and update the shared data
  - Need to coordinate communication among the team by proper initialization and assignment to variables
- Scope of a variable refers to the set of threads that can access the thread in a `parallel` block
- Variables (declared outside the scope of a parallel region) are **shared** among threads unless explicitly made private
- A variable in a parallel region can be either shared or private
  - Variables **declared** within parallel region scope are **private**
  - Stack variables declared in functions called from within a parallel region are private

# Implicit Rules

```
int n = 10, a = 7;

#pragma omp parallel
{
    ...
    int b = a + n;
    b++;
    ...
}
```

- n and a are shared variables
- b is a private variable

# Data Sharing: `shared` and `private` Clause

- `#pragma omp parallel shared(x)`
  - `shared (varlist)` - Shared by all threads, all threads access the same storage area for shared variables
  - Responsibility for synchronizing accesses is on the programmer
- `#pragma omp parallel private(x)`
  - `private (varlist)`
    - A new object is declared for each thread in the team
    - Variables declared `private` should be assumed to be **uninitialized** for each thread
    - Each thread receives its own uninitialized variable `x`
    - Variable `x` falls out-of-scope after the parallel region
  - A global variable with the same name is unaffected (v3.0 and later)

# Understanding the `private` clause

```
int p = 0;

#pragma omp parallel private(p)
{
    // value of p is undefined
    p = omp_get_thread_num();
    // value of p is defined
    ...
}
// value of p is undefined
```

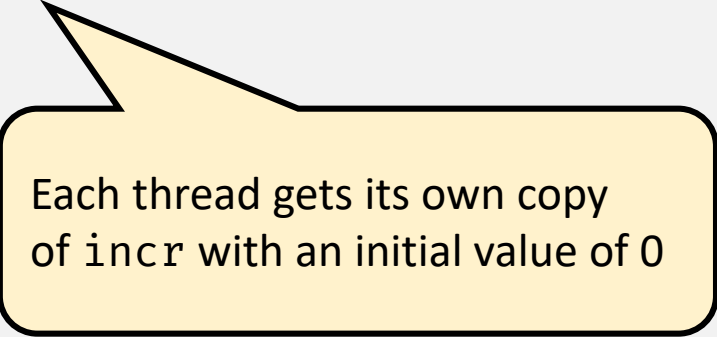
# Clause `firstprivate`

- `firstprivate (list)`
  - Variables in `list` are private, and are initialized according to the value of their original objects **prior** to entry into the parallel construct
- `#pragma omp parallel firstprivate(x)`
  - `x` must be a global-scope variable
  - Each thread receives a **by-value copy** of `x`
  - The local `x`'s fall out-of-scope after the parallel region
  - The base global variable with the same name is unaffected



# Clause `firstprivate`

```
incr = 0;
#pragma omp parallel firstprivate(incr)
{
    ...
    for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
    }
    ...
}
```



Each thread gets its own copy of `incr` with an initial value of 0

# Clause `lastprivate`

- `lastprivate (list)`
  - Variables in `list` are private, the values from the **last (sequential)** iteration or section is copied back to the original objects

```
void sq2(int n, double *lastterm) {  
    double x; int i;  
    #pragma omp parallel for  
    lastprivate(x)  
    for (i = 0; i < n; i++) {  
        x = a[i]*a[i] + b[i]*b[i];  
        b[i] = sqrt(x);  
    }  
    *lastterm = x;  
}
```

“x” has the value it held for the “last sequential” iteration, i.e., for  $i=(n-1)$

# Clause default

- default (shared | none)
  - Specify a default scope for all variables in the lexical extent of any parallel region

```
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
}
```

# Clause default

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

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



Is this snippet correct?

# Data Sharing Example

A = 1, B = 1, C = 1

```
#pragma omp parallel private(B) firstprivate(C)
```

- What can we say about the scope of A, B, and C, and their values?
- Inside the parallel region
  - “A” is shared by all threads; equals 1
  - “B” and “C” are local to each thread.
    - B’s initial value is undefined
    - C’s initial value equals 1
- Following the parallel region
  - B and C revert to their original values of 1
  - A is either 1 or the value it was set to inside the parallel region

# Data Sharing Example

```
double A[10];
int main() {
    int index[10];
#pragma omp parallel
    work(index);
    printf("%d\n", index[0]);
}

void work(int *index) {
    double temp[10];
    static int count;
    ...
}
```

A, index, and count are shared by all threads

temp is local to each thread

# Threadprivate Variables

- A threadprivate variable provides one instance of a variable for each thread
- The variable refers to a unique storage block in each thread
- Enables persistent private variables, not limited in lifetime to one parallel region

```
int a, b;  
# pragma omp threadprivate(a, b)  
// a and b are thread-private
```

# private vs threadprivate

## private

- Local to a parallel region
- Mostly allocated on the stack
- Value is assumed to be undefined on entry and exit from a parallel region

## threadprivate

- Persists across parallel regions
- Mostly allocated on the heap on thread-local storage
- Value is undefined on entry to the first parallel region



# Clause `copyin`

- Used to initialize threadprivate data upon entry to a parallel region
- Specifies that the master thread's value of a threadprivate variable should be copied to the corresponding variables in the other threads

```
int a, b;
...
# pragma omp threadprivate (a, b)
  // .. code ..
# pragma omp parallel copyin (a, b)
{
  // a and b copied from master thread
}
```

# Summary of Data Sharing Rules

- Variables are shared by default
- Variables declared within parallel blocks and subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise
- Default scoping rule can be changed with default clause
- **Recommendation**
  - Always use the default(none) clause
  - Declare private variables in the parallel region

# Runtime Routines and Environment Variables

# Runtime Library Routines and Environment Variables

- `omp_set_num_threads()`
- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_get_max_threads()`
- `omp_in_parallel()`
- `omp_set_dynamic()`
- `omp_get_dynamic()`
- `omp_num_procs()`
- ...

- Set the default number of threads to use
  - `OMP_NUM_THREADS` `int_literal`
- Control the size of child threads' stack
  - `OMP_STACKSIZE`
- Hint to runtime how to treat idle threads
  - `OMP_WAIT_POLICY`
  - `ACTIVE` keep threads alive at barriers/locks
  - `PASSIVE` try to release processor at barriers/locks
- Process binding is enabled if this variable is true, the runtime will not move threads around between processors
  - `OMP_PROC_BIND` `true | false`
- ...

# Worksharing Construct

Coarse-grained Parallelism

# Worksharing Construct

- Loop structure in parallel region is same as sequential code
- No explicit thread-id based work division; instead system automatically divides loop iterations among threads
- User can control work division: block, cyclic, block-cyclic, etc., via “schedule” clause in pragma

```
float res;
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < MAX; i++)
  {
    B = big_job(i);
  }
}
```

# Worksharing Construct

```
#pragma omp parallel  
{  
#pragma omp for  
  for (int i=0; i<N; i++) {  
    func1(i);  
  }  
}
```

If the team consists of only one thread then the worksharing region is not executed in parallel.

Variable `i` is made “private” to each thread by default. You could also do this explicitly with a “`private(i)`” clause.

data-parallel  
execution

# Worksharing Construct

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

sequential code

work sharing  
construct

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

OpenMP parallel  
region

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



# Combined Worksharing Construct

```
float res;
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < MAX; i++) {
    B = big_job(i);
  }
  #pragma omp critical
  consume (B, res);
}
```

```
float res;
#pragma omp parallel for
for (int i = 0; i < MAX; i++) {
  B = big_job(i);
  #pragma omp critical
  consume (B, res);
}
```

Often a parallel region has a single work-shared loop

# Limitations on the Loop Structure

- Loops need to be in the canonical form
  - Cannot use `while` or `do-while`
- Loop variable must have integer or pointer type
- Cannot use a loop where the trip count cannot be determined

```
for (init-expr; test-expr; incr-expr)  
    structured-block
```



canonical form

# Take Care when Sharing Data with the Worksharing Construct

OpenMP compiler will not check for dependences

```
#pragma omp parallel for
{
    for(i=0; i<n; i++) {
        tmp = 2.0*a[i];
        a[i] = tmp;
        b[i] = c[i]/tmp;
    }
}
```

```
#pragma omp parallel for
private(tmp)
{
    for(i=0; i<n; i++) {
        tmp = 2.0*a[i];
        a[i] = tmp;
        b[i] = c[i]/tmp;
    }
}
```

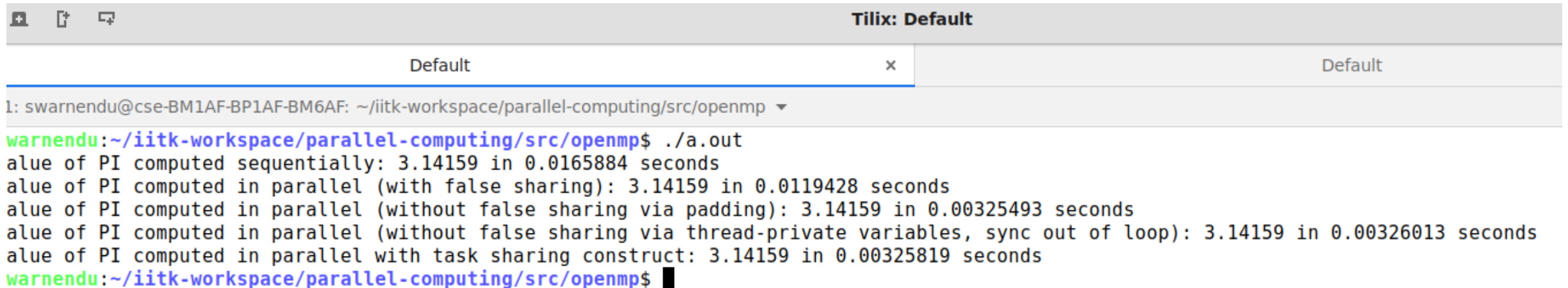
# Our Refined Pi Implementation

```
double omp_pi() {
    double x, pi, sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;

    #pragma omp parallel for private(x) reduction(+ : sum) num_threads(NUM_THRS)
    for (int i = 0; i < NUM_STEPS; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    return pi;
}
```

# Evaluate the Pi Program Variants

- Sequential computation of pi
- Parallel computation with false sharing
- Parallel computation with padding
- Parallel computation with thread-local sum
- Worksharing construct



```
l: swarnendu@cse-BM1AF-BP1AF-BM6AF: ~/iitk-workspace/parallel-computing/src/openmp ▾
warnendu:~/iitk-workspace/parallel-computing/src/openmp$ ./a.out
alue of PI computed sequentially: 3.14159 in 0.0165884 seconds
alue of PI computed in parallel (with false sharing): 3.14159 in 0.0119428 seconds
alue of PI computed in parallel (without false sharing via padding): 3.14159 in 0.00325493 seconds
alue of PI computed in parallel (without false sharing via thread-private variables, sync out of loop): 3.14159 in 0.00326013 seconds
alue of PI computed in parallel with task sharing construct: 3.14159 in 0.00325819 seconds
warnendu:~/iitk-workspace/parallel-computing/src/openmp$ █
```

# Finer Control on Work Distribution

- `#pragma omp parallel for schedule [..., <chunksize>]`
- The `schedule` clause determines how loop iterators are mapped onto threads
  - Most implementations use block partitioning
- Good assignment of iterations to threads can have a significant impact on performance

# Finer Control on Work Distribution

- `#pragma omp parallel for schedule(static[,chunk])`
  - Fixed-sized chunks (or as equal as possible) assigned (alternating) to `num_threads`
  - Typical default is: `chunk = iterations/num_threads`
  - Set `chunk = 1` for cyclic distribution
- `#pragma omp parallel for schedule(dynamic[,chunk] )`
  - Run-time scheduling (has overhead)
  - Each thread grabs “chunk” iterations off queue until all iterations have been scheduled, default is 1
  - Good load-balancing for uneven workloads

# Finer Control on Work Distribution

- `schedule(static)`
  - OpenMP guarantees that if you have two separate loops with the same number of iterations and execute them with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range(s) in both parallel regions
  - Beneficial for NUMA systems: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node.

---

<https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp>



# Finer Control on Work Distribution

- `#pragma omp parallel for schedule(guided[,chunk])`
  - Threads dynamically grab blocks of iterations
  - Chunk size starts relatively large, to get all threads busy with good amortization of overhead
  - Subsequently, chunk size is reduced to “chunk” to produce good workload balance
  - By default, initial size is  $\text{iterations}/\text{num\_threads}$

# Example of guided Schedule with Two Threads

Thread	Chunk	Chunk Size	Remaining Iterations
0	1-5000	5000	5000
1	5001-7500	2500	2500
1	7501-8750	1250	1250
1	8751-9375	625	625
0	9376-9688	313	312
1	9689-9844	156	156
0	9845-9922	78	78
1	9923-9961	39	39
0	9962-9981	20	19
1	9982-9991	10	9
0	9992-9996	5	4
0	9997-9998	2	2
0	9999	1	1
1	10000	1	0

# Finer Control on Work Distribution

- `#pragma omp parallel for schedule(runtime)`
  - Decision deferred till run-time
  - Schedule and chunk size taken from `OMP_SCHEDULE` environment variable or from runtime library routines
    - `$ export OMP_SCHEDULE="static,1"`
- `#pragma omp parallel for schedule(auto)`
  - Schedule is left to the compiler runtime to choose (need not be any of the above)
  - Any possible mapping of iterations to threads in the team can be chosen

# Understanding the `schedule` clause

Schedule clause	When to use?
<code>static</code>	Predetermined and predictable by the programmer; low overhead at run-time, scheduling is done at compile-time
<code>dynamic</code>	Unpredictable, highly variable work per iteration; greater overhead at run-time, more complex scheduling logic
<code>guided</code>	Special case of dynamic to reduce scheduling overhead
<code>auto</code>	When the runtime can learn from previous executions of the same loop

# Nested Loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
    }
  }
```

j is implicitly private with the collapse clause

- Will form a single loop of length NxM and then parallelize that
- Useful when there are more than N threads

# Nested Loops

- collapse works with square loops, not with triangular loops

```
int i, j;
#pragma omp parallel for num_threads(2)
collapse(2) private(j)
    for (i = 0; i < 4; i++)
        for (j = 0; j <= i; j++)
            cout << i << j <<
omp_get_thread_num()) << "\n";
```

Does not compile  
with GCC 7.4

```
int i, j;
#pragma omp parallel for num_threads(2)
collapse(2) private(j)
    for (i = 0; i < 4; i++)
        for (j = 0; j < 100; j++)
            cout << i << j <<
omp_get_thread_num()) << "\n";
```

# Data Sharing with Worksharing

What is going to happen?

```
#include <omp.h>

int main() {
    int i, j=5;
    double x=1.0, y=42.0;
    #pragma omp parallel for default(none)
    reduction(*:x)
    for (i=0;i<N;i++) {
        for (j=0;j<3;j++)
            x += foobar(i, j, y);
    }
    printf(" x is %f\n", (float)x);
}
```

```
#include <omp.h>

int main() {
    int i, j=5;
    double x=1.0, y=42.0;
    #pragma omp parallel for
    default(none) reduction(*:x)
    shared(y) collapse(2)
    for (i=0;i<N;i++) {
        for (j=0;j<3;j++)
            x += foobar(i, j, y);
    }
    printf(" x is %f\n", (float)x);
}
```

# OpenMP Sections

- Noniterative worksharing construct
- Worksharing for function-level parallelism; complementary to “omp for” loops
- The sections construct gives a different structured block to each thread

```
#pragma omp parallel
{
    ...
    #pragma omp sections
    {
        #pragma omp section
            x_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    } // implicit barrier
    ...
}
```



# Explicit Tasks

# Explicit Task Constructs in OpenMP

- OpenMP can only parallelize loops in a basic standard form with loop counts known at runtime
- Not all programs have canonical loops

- Consider a program to traverse a linked list

```
p=head;
while (p) {
    dowork(p);
    p = p->next;
}
```

A thought bubble with a light gray fill and a black outline. It contains the text "How can we modify the program to parallelize with OpenMP?". Three small circles of varying sizes are connected to the top of the bubble by thin lines, suggesting a thought process or a question being pondered.

How can we modify the program to parallelize with OpenMP?

# One Potential Solution to Using OpenMP

1

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}
```

2

```
p = head;  
for (i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}
```

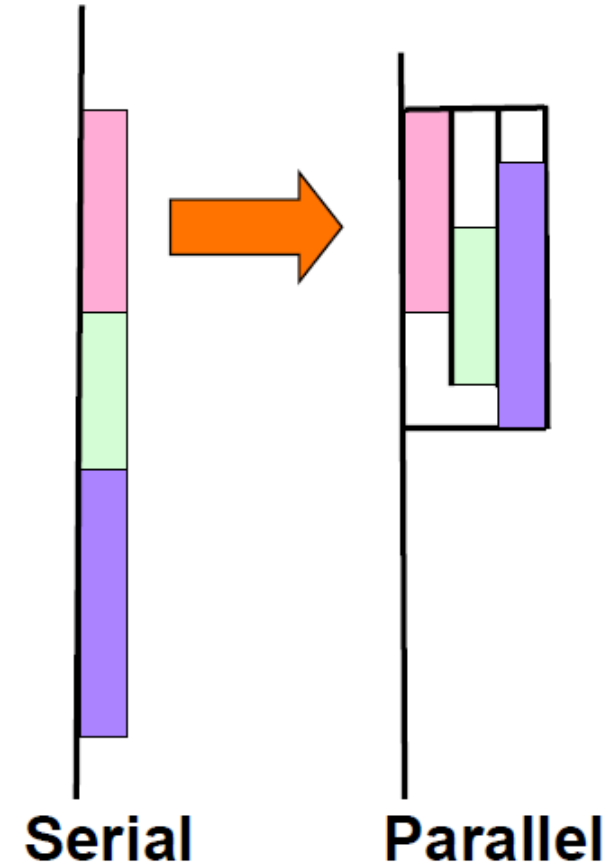
3

```
#pragma omp parallel  
{  
    #pragma omp for schedule (static,1)  
    for(i=0; i<count; i++)  
        dowork(parr[i]);  
}
```

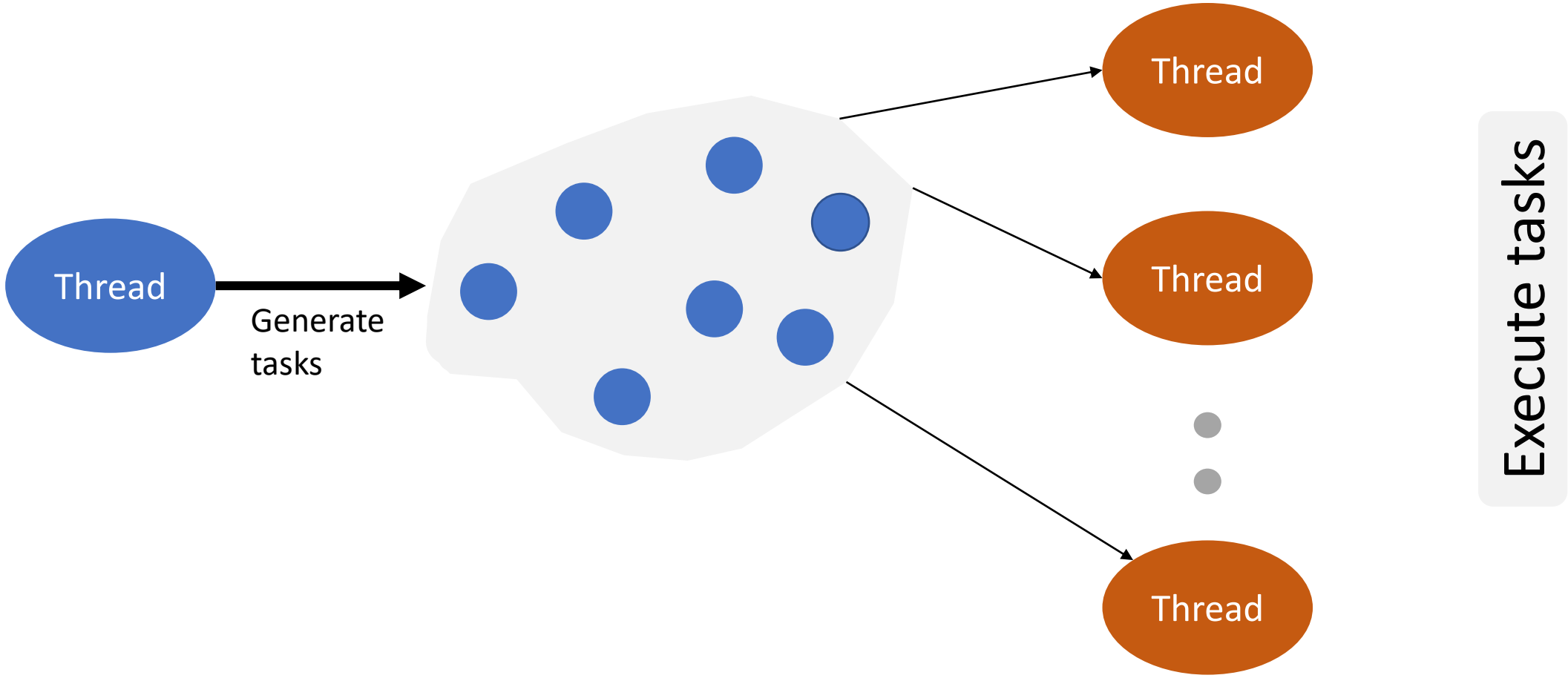
This works, but is inelegant (had to use a vector or array as an intermediate) and is inefficient (requires multiple passes over the data)

# Tasks in OpenMP

- Explicit tasks were introduced in OpenMP 3.0
- Tasks are independent units of work and are composed of
  - code to execute
  - data to compute with
  - control variables
- Threads are assigned to perform the work of each task
- The runtime system decides when tasks are executed
  - Tasks may be deferred or may be executed immediately

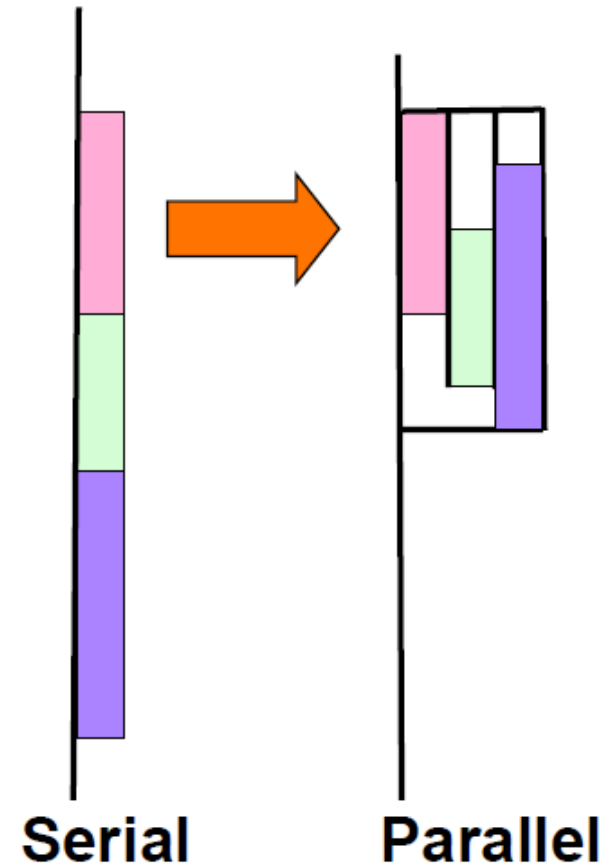


# The Tasking Concept in OpenMP



# Tasks in OpenMP

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested, i.e., a task may itself generate tasks



# Task Directive

```
#pragma omp parallel  
{  
  #pragma omp master  
  {  
    #pragma omp task  
    fred();  
    #pragma omp task  
    daisy();  
    #pragma omp task  
    billy();  
  }  
}
```

Task 0 packages data

Tasks executed by some thread in some order

All tasks complete before this barrier ends

- You can use a barrier for task completion
- #pragma omp taskwait
  - Wait for child tasks to complete

# Example of Tasks

```
#pragma omp parallel
{
#pragma omp single
    {
        cout << "A ";
#pragma omp task
        cout << "race ";
#pragma omp task
        cout << "car ";
        cout << "is fun to watch!";
    }
}
```

```
#pragma omp parallel
{
#pragma omp single
    {
        cout << "A ";
#pragma omp task
        cout << "race ";
#pragma omp task
        cout << "car ";
#pragma omp taskwait
        cout << "is fun to watch!";
    }
}
```



# Example with Tasks

```
void walk(node *p) {  
    if (p->left) {  
#pragma omp task ..  
        walk(p->left);  
    }  
    if (p->right) {  
#pragma omp task  
        walk(p->right);  
    }  
#pragma omp taskwait ..  
    sum_weight(p);  
}
```

Tasks are executed  
in any order

Be careful about  
dependences

```
int main() {  
    node *p = head;  
#pragma omp parallel  
    walk(p);  
}
```

# taskwait and taskgroup

```
void generate () {  
#pragma omp parallel  
#pragma omp single  
  {  
#pragma omp task  
  {  
    printf("task 1\n");  
#pragma omp task  
    printf("task 2\n");  
  }  
#pragma omp taskwait  
#pragma omp task  
  printf("task 3\n");  
  }  
}
```

task 2 is a child  
of task 1

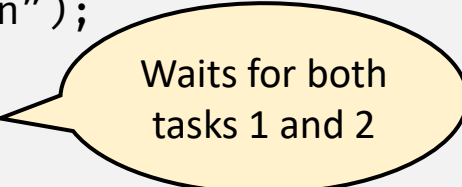
Waits only for task 1  
to complete before  
task 3 is scheduled

- taskwait suspends a thread till all the child tasks generated before the taskwait are completed

- With taskgroup, the thread waits till all the child tasks and their descendant tasks complete execution

# taskwait and taskgroup

```
#pragma omp parallel
#pragma omp single
{
#pragma omp taskgroup
  {
    #pragma omp task
    {
      printf("task 1\n");
    }
    #pragma omp task
    {
      printf("task 2\n");
    }
  } // end of taskgroup
  #pragma omp task
  {
    printf("task 3\n");
  }
}
```



Waits for both  
tasks 1 and 2

- `taskwait` suspends a thread till all the child tasks generated before the `taskwait` are completed
- With `taskgroup`, the thread waits till all the child tasks and their descendant tasks complete execution

# Generating Large Number of Tasks

```
void generate () {
    const int num_elem=1e7;
    int arr[num_elem];
#pragma omp parallel
    {
#pragma omp single
    {
        for(int i=0;i<num_elem;i++) {
#pragma omp task
            check(arr[i]);
        }
    }
}
}
```

- If the number of tasks reaches a limit, the task generator thread can be stopped from creating further tasks and starts executing unassigned tasks
- If the generator thread takes a long time to finish executing unassigned tasks, the other threads will idle till the generator thread is done
  - The tasks are “tied” to the generator thread
- The generator thread can start generating new tasks once the number of unassigned tasks becomes low

The untied clause will allow any thread to resume the task generating loop

# SIMD Programming

Fine-grained Parallelism

# SPMD Programming with OpenMP

- Single Program Multiple Data
  - Each thread runs the same program
  - Selection of data, or branching conditions, is based on thread id
- In OpenMP implementations
  1. Perform work division in parallel loops
  2. Query `thread_id` and `num_threads`
  3. Partition work among threads

# How about SIMD support?

- Support in older versions of OpenMP required vendor-specific extensions
  - Programming models (e.g., Intel Cilk Plus)
  - Compiler pragmas (e.g., `#pragma vector`)
  - Low-level constructs or intrinsics (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + 10;
}
```

# simd Construct

- `#pragma omp simd ...`
  - Can be applied to a loop to indicate that the loop can be transformed to a SIMD loop
  - Partition loop into chunks that fit a SIMD vector register
  - Does not parallelize the loop body

```
#pragma omp simd simdlen(16)
for (int i=0; i<n; i++)
    a[i] = b[i] + c[i]
```

process 16  
single-precision  
elements

hint about  
dependence  
distance

```
#pragma omp simd safelen(8)
for (int i=m; i<n; i++)
    a[i] = a[i-m] + b[i]
```

vectorize over  
n\*m iterations

```
#pragma omp simd collapse(2)
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a[i,j] = b[i,j] + c[i,j]
```



# simd Worksharing Construct

- `#pragma omp for simd ...`
- **Parallelize and vectorize** a loop nest
  - Distribute a loop's iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

```
#pragma omp simd for collapse(2)
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a[i,j] = b[i,j] + c[i,j]
```

---

[https://chryswoods.com/vector\\_c++/features.html](https://chryswoods.com/vector_c++/features.html)

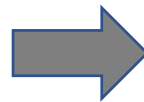
# SIMD Function Vectorization

```
#pragma omp declare simd ...  
function-definition-or-declaration
```

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop
- Enables creation of one or more versions to allow for SIMD processing

```
#pragma omp declare simd  
float min(float a, float b) {  
    return a < b ? a : b;  
}
```

arguments  
are scalar



```
// Vector version  
vec8 min_v(vec8 a, vec8 b) {  
    return a < b ? a : b;  
}
```

# declare simd Construct

```
#pragma omp simd private(temp) reduction(+:sum)
for (i=0; i<n; i++) {
    sum += add_values(a[i], b[i]);
}
```

```
#pragma omp declare simd
int add_values(int a, int b) {
    return a+b;
}
```

- #pragma omp simd alone may not be sufficient to vectorize the call to add\_values()
- Compiler can inline function add\_values() and vectorize it across the loop over n

# References

- Tim Mattson et al. [The OpenMP Common Core: A hands on exploration](#), SC 2018.
- Tim Mattson and Larry Meadows. [A “Hands-on” Introduction to OpenMP](#). SC 2008.
- Ruud van der Pas. [OpenMP Tasking Explained](#). SC 2013.
- Peter Pacheco. An Introduction to Parallel Programming.
- Blaise Barney. OpenMP. <https://hpc-tutorials.llnl.gov/openmp/>
- Nitya Hariharan. [CS 610: Advanced OpenMP](#). IIT Kanpur.