OpenMP

Swarnendu Biswas

An Introductory Course on High-Performance Computing in Engineering

Indian Institute of Technology Kanpur

30th Sep 2019

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

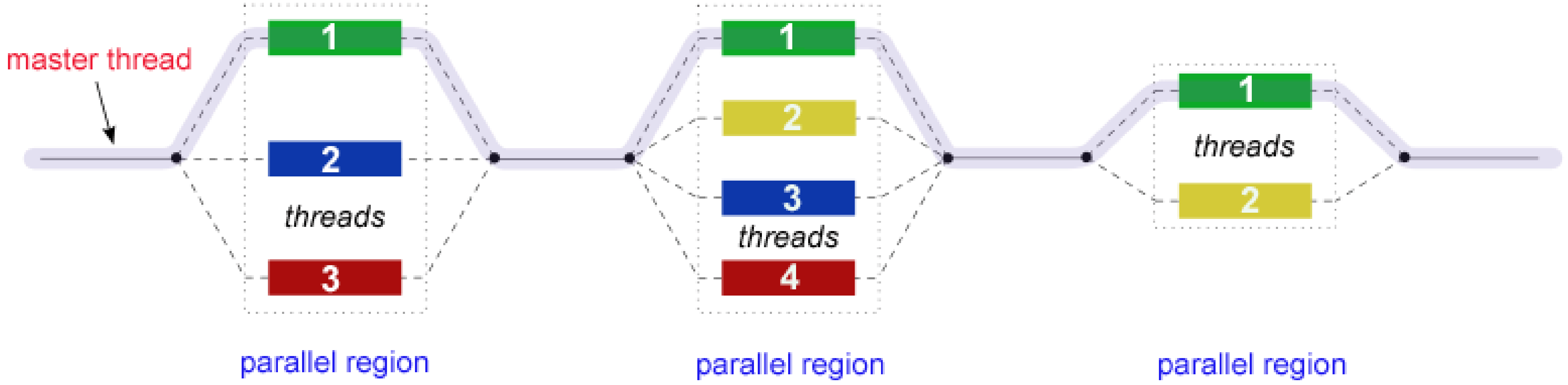
What is OpenMP?

- OpenMP (Open Multi-Processing) is a popular **shared-memory** programming API
  - A directive based parallel programming model that helps standardize practices established in SMP, vectorization and heterogeneous device programming
  - OpenMP program is essentially a sequential program augmented with **compiler directives** to specify parallelism
  - Eases conversion of existing sequential programs
- OpenMP supports C/C++ and Fortran on a wide variety of architectures
- OpenMP is 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

Fork-Join Model of Parallel Execution



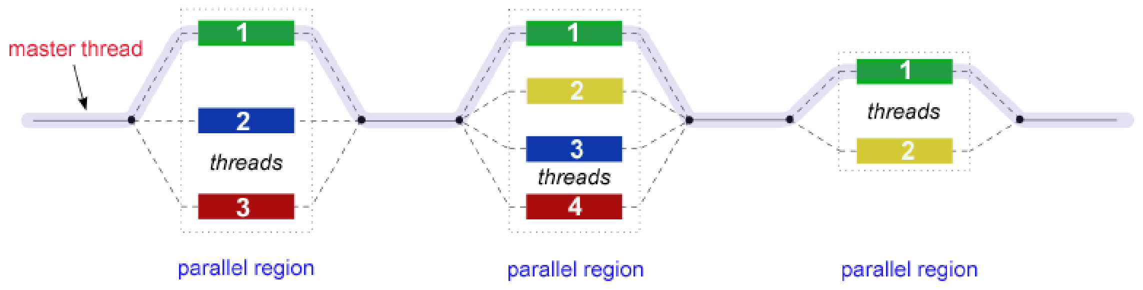
Goals of OpenMP

- 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 have OpenMP support

The OpenMP API

- Compiler directives
  - `#pragma omp parallel`
  - Treated as comments with no/disabled OpenMP support
- Runtime library routines
  - `int omp_get_num_threads(void)`
- Environment variables
  - `export OMP_NUM_THREADS=8`

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
  …
}
```

OpenMP Core Syntax

- Most common constructs in OpenMP are compiler directives
  - **`#pragma omp`** `directive [clause [clause]…] newline`
  - Example: `#pragma omp parallel num_threads(4)`
- `directive`
  - 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.
- Function prototypes and types are defined in `#include <omp.h>`

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 into or out of the middle of the structured block

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 in a **team** 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)

OpenMP Constructs

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

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

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

OpenMP Regions

- A region consists of all code encountered during a specific instance of the execution of a given construct
  - 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]);
```

Specifying Number of Threads

- Desired number of threads can be specified in many ways
    1. Setting environmental variable `OMP_NUM_THREADS`
    2. Runtime OpenMP function `omp_set_num_threads(4)`
    3. 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
  1. `OMP_NUM_THREADS`
  2. `omp_set_num_threads(…)`
  3. `#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

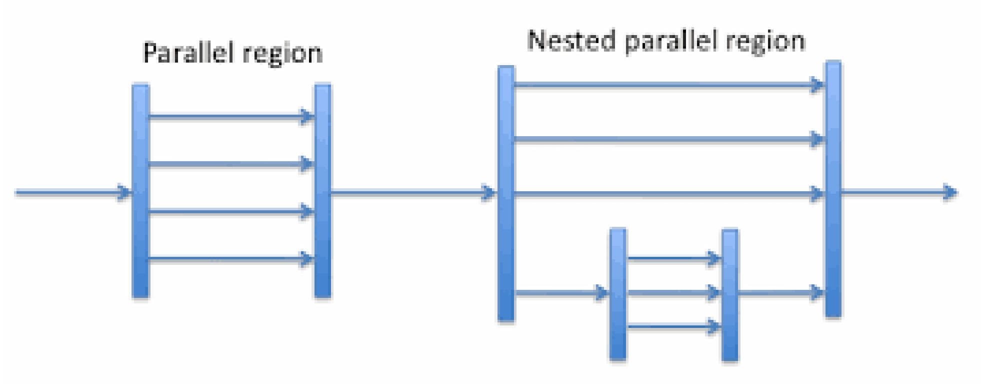
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int t_id = omp_get_thread_num();
  int num_thrs = omp_get_num_threads();
  int b_size = 1000 / num_thrs;
  for (int i = t_id*b_size; i < (t_id+1)*b_size; i += num_thrs) {
    A[i]= foo(i);
  }
}
```

Block distribution of work

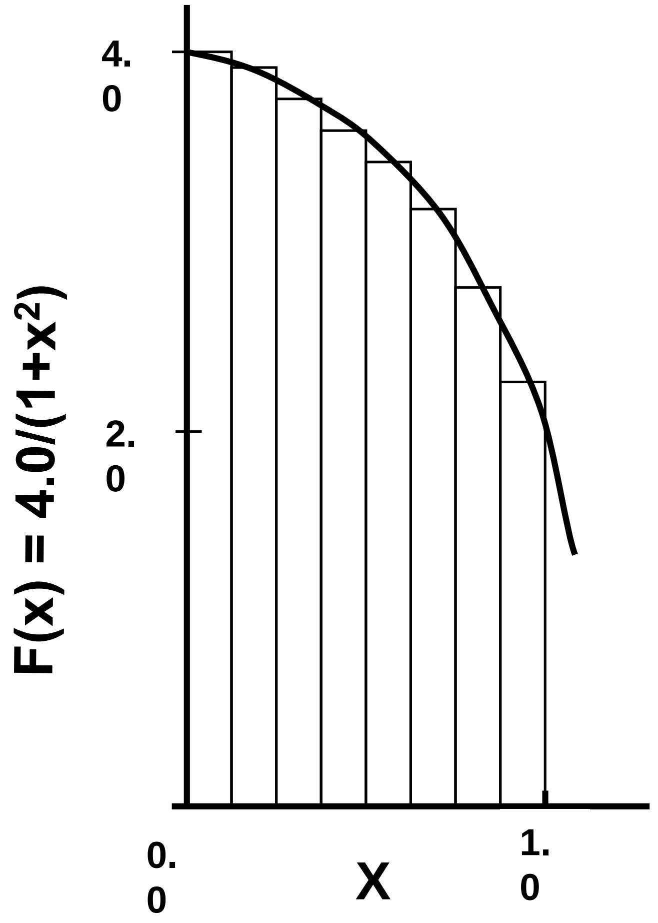
Nested Parallelism

- 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 Δx 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;
}
```

Optimize the Pi Program

```
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 += nt
hrds) {
      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

```
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
                          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_ST
EPS;
  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;
}
```

`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`	**`atomic`**
• Locks code segments	• Locks data variables
• Serializes all unnamed critical sections	• Serializes operations on the same shared data
• Less efficient than `atomic`	• Makes use of hardware instructions to provide atomicity
• More general	• Less general

Barrier Synchronization

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

#pragma omp barrier

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

explicit barrier

- Each thread waits until all threads arrive

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”;
  }
}
```

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
{
  do_many_things();
#pragma omp single
  {
    reset_boundaries();
  }

  do_many_other_things();
}
```

multiple threads of control

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

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

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_ST
EPS;
  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

C/C++ only

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

Data Scope

- 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` Clause

- `shared (list)`
  - Shared by all threads, all threads access the same storage area for shared variables
- `#pragma omp parallel shared(x)`

- Responsibility for synchronizing accesses is on the programmer

Data Sharing: `private` Clause

- `private (list)`
  - A new object is declared for each thread in the team
  - Variables declared private should be assumed to be uninitialized for each thread

- `#pragma omp parallel private(x)`
  - 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 `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?

Worksharing Construct

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.

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 (index = start; index < end; index++)`
- `for (index = start; index >= end; index = index - incr)`

Take Care with the Worksharing Construct

OpenMP compiler will not check for dependences

Take Care when Sharing Data

```
#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;
  }
}
```

Take Care when Sharing Data

```
int i = 0, n = 10, a = 7;

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

- n and a are shared variables
- b is a private variable
- A loop iteration variable is private by default
  - So i is private

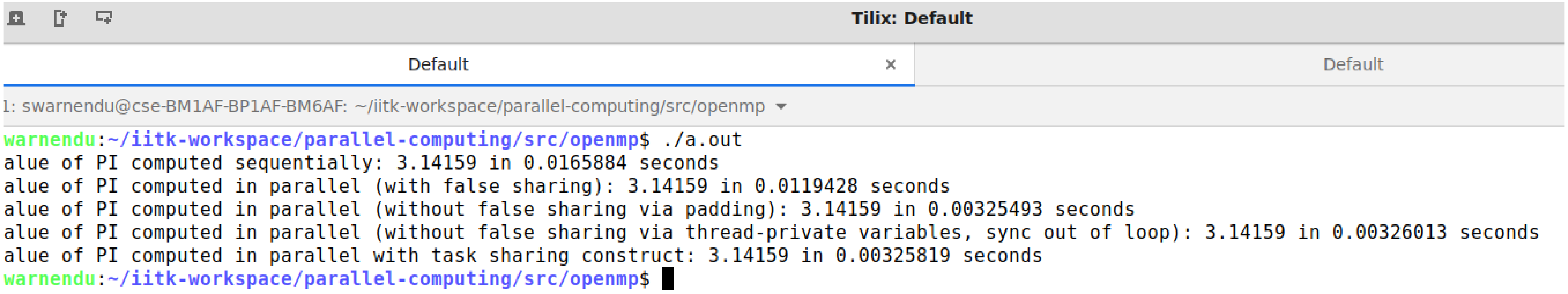
Our Refined Pi Implementation

```
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 thread-local sum
- Worksharing construct


Tilix: Default
Default
Default
1: 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

- The `schedule` clause determines how loop iterators are mapped onto threads
  - Most implementations use block partitioning

- `#pragma omp parallel for schedule [, <chunksize>]`

- 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

- `#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 iterations/num_threads

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

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
  …
}
```

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

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://computing.llnl.gov/tutorials/openMP/