CS 610: OpenMP

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-I

What is OpenMP?

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

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

OpenMP API

- Compiler directives: `#pragma omp directive [clause [clause]...] newline`
  - Most common constructs in OpenMP are compiler directives
  - For example, `#pragma omp parallel num_threads(4)`
  - Directives are treated as comments if OpenMP is not supported
- Runtime library routines: `int omp_get_num_threads(void);`
- Environment variables: `export OMP_NUM_THREADS=8`
- Function prototypes and types are defined in the header `omp.h`

OpenMP 6.0 API Syntax Reference Guide

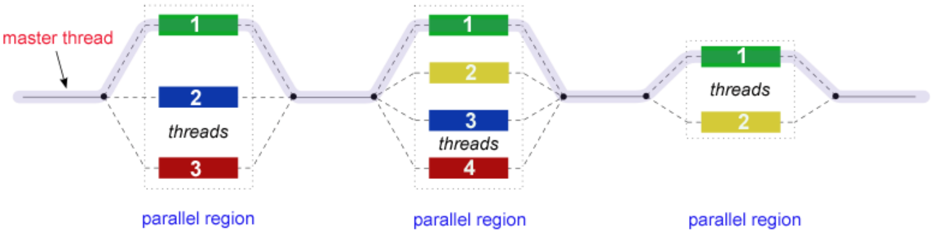
OpenMP Code Structure and Compilation

```
  #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
    ...
    // all parallel threads join
    // the master thread
}
    // resume serial code
    ...
  }
```

- 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

Fork-Join Model of Parallel Execution

```
  int main() {
    ... // serial code, master thread
    // begin parallel section, fork a team of threads
#pragma omp parallel ...
{ // parallel region executed by all threads
    ...
} // all parallel threads join the master thread
    ... // resume serial code
  }
```



Hello World with OpenMP

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

Makefile

Parallel Construct

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

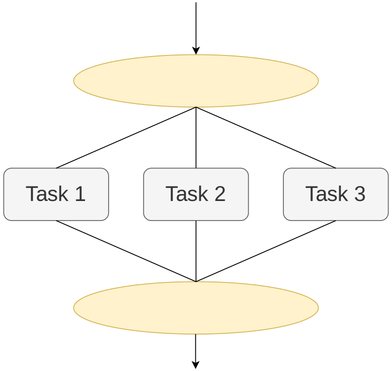
```
#pragma omp parallel [clause...]
  structured_block
```

Example of clauses

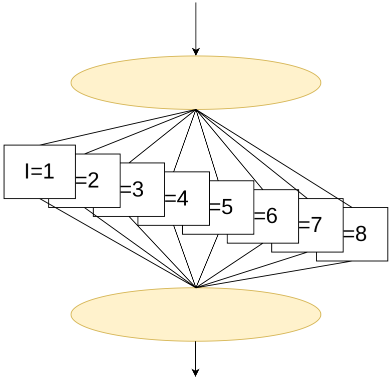
- `private (list)`
- `shared (list)`
- `default (shared | none)`
- `firstprivate (list)`
- `reduction (operator: list)`
- `num_threads (int_expr)`
- ...

Types of Parallelism with OpenMP

Task Parallelism



Loop Parallelism



The Essence of OpenMP

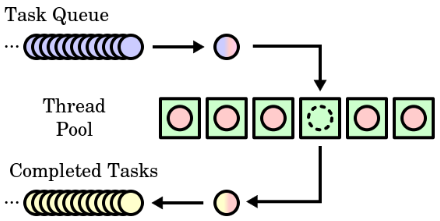
- Create threads that execute in a shared address space
  - The only way to create threads is with the parallel construct
  - Once created, all threads execute the code inside the construct
- Split up the work between threads by one of two means
  - SPMD (Single Program Multiple Data) — all threads execute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

Threading in OpenMP

- Thread pool is a software design pattern that maintains a pool of threads waiting for work
  - Advantageous when work is short-lived
  - Avoid the overhead of frequent thread creation and destruction
- OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region

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

Only three threads are created excluding the parent thread



pthread.cpp
Makefile

Specifying Number of Threads

Desired number of threads can be specified in many ways

(i) Setting environmental variable `OMP_NUM_THREADS`
(ii) Runtime OpenMP function `omp_set_num_threads()`
(iii) Clause in `#pragma omp parallel` region

- `OMP_NUM_THREADS` (if present) specifies the initial 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

set-num-threads.cpp
Makefile

Distributing Work

Cyclic distribution

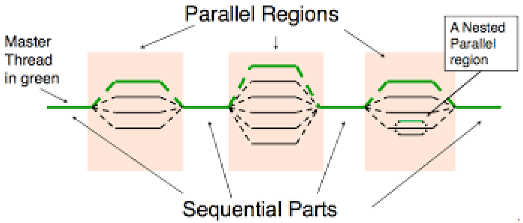
```
  double A[1000];
  omp_set_num_threads(4);
#pragma omp parallel
{
  int t_id = omp_get_thread_num();
  for (int i = t_id; i < 1000; i += omp_get_num_threads())
    A[i]= foo(i);
}
```

Block distribution

```
  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 += 1)
    A[i]= foo(i);
}
```

Nested Parallelism

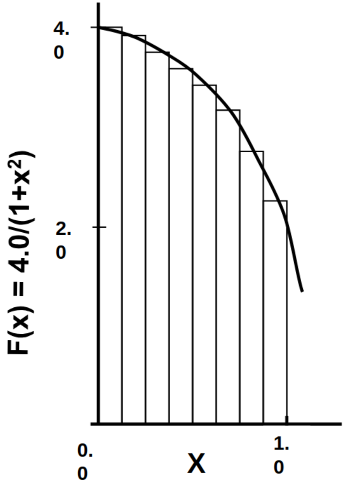
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team
- Nested parallelism creates parallel regions within a parallel region to handle large parallel computations
  - Can lead to oversubscription by creating lots of threads, hence turned off by default
  - Set `OMP_NESTED` as TRUE or call `omp_set_nested()`
- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates (order of termination is unspecified)



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 $\Sigma_{i=0}^{N} F(x_i)\Delta x \approx \pi$, where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i



Serial Program to Compute Pi

```
static const uint64_t NUM_STEPS = 10000000;
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
```

compute-pi.cpp

Makefile

Computing Pi with OpenMP

```
double omp_pi_with_fs() {
  omp_set_num_threads(NUM_THRS);
  double sum[NUM_THRS] = {0.0}, pi = 0.0, 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}, pi = 0.0, 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;
 }
```

This is a correct implementation, but ...
Is there a problem with the code?

Avoid False Sharing

- Array `sum[]` is shared, 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 line thrashing due to “false sharing” hurts performance

Computing Pi: Avoid False Sharing

```c
 double omp_pi_without_fs1() {
   omp_set_num_threads(NUM_THRS);
   double sum[NUM_THRS][8], 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;
   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;
 }
```

Computing Pi: Avoid False Sharing

```
 double omp_pi_without_fs1() {
   omp_set_num_threads(NUM_THRS);
   double sum[NUM_THRS][8], 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;
   for (int i=tid; i<NUM
     x = (i + 0.5) * ste
     sum[tid][0] += 4.0
   }
} // end #pragma omp pa
   for (int i = 0; i < n
     pi += (sum[i][0] * step);
   return pi;
 }
```

The amount of padding depends on the cache line size and the data type. Hard coding the padding amount is not portable, because it may not work across different cache configurations, architectures, and data types.

Optimize the Pi Program: Use Thread-Local Sum

```
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
     sum += 4.0 / (1.0 + x * x);
   }
   pi += (sum * step);
} // end #pragma omp parallel
   return pi;
 }
```

Optimize the Pi Program: Use Thread-Local Sum

```
 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
     sum += 4.0 / (1.0 + x * x);
   }
   pi += (sum * step);   Data Race
} // end #pragma omp parallel
   return pi;
 }
```

This program is now wrong. Why?

Synchronization Constructs

critical and atomic Constructs

```
  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);
  }
}
```

```
  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;
  }
}
```

- If the code has multiple unnamed critical sections, they are all mutually exclusive
  - Can avoid this by naming critical sections
  - `#pragma omp critical (optional_name)`

- Uses hardware atomic instructions; much lower overhead than using critical sections
- Expression operation can be of type
  - `x binop= expr`, x++, ++x, x−, −x
  - x is a scalar type, `binop` can be +, *, -, /, &, ˆ, |, «, or »

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
     sum += 4.0 / (1.0 + x * x);
   }
#pragma omp critical
   pi += (sum * step);
} // end #pragma omp parallel
   return pi;
 }
```

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 `#pragma omp for` or `#pragma omp parallel for`

```
  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";
  }
}
```

Clauses master and single

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 skip it

no barrier is implied

single

```
#pragma omp parallel
{

  do_many_things();
#pragma omp single
{

  reset_boundaries();
}

  do_many_other_things();
}
```

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

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

Reductions in OpenMP

- `reduction` clause specifies an operator and a list of reduction variables (must be shared variables)
  - ▶ OpenMP compiler creates local variables for each thread, divides work to form partial reductions, and generates code to combine the partial reductions
  - ▶ Local copy for each reduction variable is initialized to operator's identity
  - ▶ Final result is placed in the shared variable

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

```
  double sum = 0.0;
#pragma omp parallel reduction(+ : sum)
  sum += func(omp_get_thread_num());
```

- Predefined set of associative operators (e.g., +, *, -, min, max) can be used with the `reduction` clause

Computing Pi with OpenMP: Another version

```
 double omp_pi_with_fs() {
   omp_set_num_threads(NUM_THRS);
   double sum[NUM_THRS] = {0.0}, 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;
   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;
 }
```

Synchronization with Locks

- More flexible than the `critical` construct
  - `critical` locks a code segment, while locks lock data
- More error-prone, can deadlock if a thread does not unset a lock after acquiring it
- Nested locks can be acquired if it is available or owned by the same thread
  - E.g., `omp_init_nest_lock()`

```
  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);
```

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

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 (from v3+)

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 xs fall out-of-scope after the parallel region
  - The base global variable with the same name is unaffected

```
  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 are 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)`

Why does OpenMP forbid use lastprivate in "#pragma omp parallel"?

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
  }
```

```
  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

```
int 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.cpp
Makefile

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
  - Mostly allocated on the heap in thread-local storage
- Enables persistent private variables, not limited in lifetime to one parallel region

```
// Applicable to file-scoped variables
int a, b;
# pragma omp threadprivate(a, b)
  // a and b are thread-private
```

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

Worksharing Construct

Coarse-grained parallelism

Worksharing Construct

Sequential version

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

Manual worksharing

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

Worksharing construct

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

Worksharing Construct

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

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

Variable `i` is made private to each thread (can be specified explicitly with `private(i)` clause)

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

Dependences and Worksharing

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;
  }
}
```

Yet Another 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;
  }
```

Finer Control on Work Distribution

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

Advantages with `schedule` Clause

- `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.

What's 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 `iterations/num_threads`
- `#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

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	1000	1	0

Understanding the `schedule` Clause

`schedule`	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

Nested Loops

- We can parallelize multiple loops in a perfectly nested rectangular loop nest with the `collapse` clause
- OpenMP will form a single loop of length N × M and then parallelize the loop, useful when there are more than N threads

```
#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

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

Does not compile with GCC 7.4 but compiles with GCC 13

collapse.cpp

Sections

- Allows 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
  ...
}
```

Difference between section and task openmp

Explicit Tasks

Dealing with Non-canonical Loops

- OpenMP can only parallelize loops in canonical form with loop counts known at run time
- Not all programs have canonical loops

Consider a program to traverse a linked list

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

How can we modify the program to parallelize with OpenMP?

Possible Idea

❶

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

❷

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

❸

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

Possible Idea

1

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

2

```
  p = head;
  for (int i=0; i<count; i++) {
```

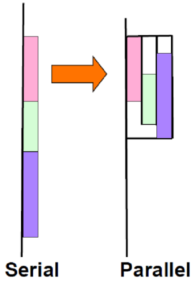
This works, but is inelegant (had to use a vector or array as an intermediate) and is inefficient (requires multiple passes over the data)

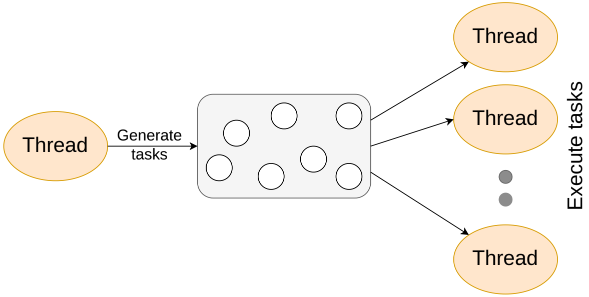
3

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

Tasks in OpenMP

- Explicit tasks were introduced in OpenMP 3.0
- Tasks are independent units of work and are composed of (i) code to execute, (ii) data to compute with, and (iii) 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





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
- `#pragma omp taskwait` waits for child tasks to complete

```
#pragma omp parallel
{

  #pragma omp master
  {
    #pragma omp task
    tapti();
    #pragma omp task
    tista();
    #pragma omp task
    damodar();
  }

}
```

Thread 0 packages data

Tasks executed by threads in some order

All tasks complete before the barrier ends

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!";
  }
}
```

Tasks are executed in any order

```
#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!";
  }
}
```

- array-sum.cpp
- fibonacci.cpp
- Makefile

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");
      // Task 2 is a child of Task 1
    }
    #pragma omp taskwait


    #pragma omp task
    printf("task 3\n");
  }
}
```

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]);
     }
   }
}
  }
```

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

- If the number of tasks reaches a limit, the task generator thread can stop 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

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
  (i) Perform work division in parallel loops
  (ii) Query thread ID and `num_threads`
  (iii) Partition work among threads

With SIMD, threads execute the same instruction. With SPMD, threads may be executing different instructions.

`simd` Construct

- `#pragma omp simd`
  - Introduced in version 4.0
  - Can be applied to a loop to indicate that the loop can be vectorized
  - Partitions loop into chunks that fit a SIMD vector register
  - Does not parallelize the loop body with threads

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

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

```
#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 parallel 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]
```

omp simd features

SIMD Function Vectorization

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

- Declares 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;
}
```

```
// 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

simd-function.cpp

Code Examples

- hello-world.cpp
- set-num-threads.cpp
- pthread.cpp
- compute-pi.cpp
- array-sum.cpp
- data-sharing.cpp
- collapse.cpp
- worksharing.cpp
- fibonacci.cpp
- simd-function.cpp
- Makefile

References

- P. Pacheco and M. Malensek. An Introduction to Parallel Programming. Chapter 5, 2nd edition, Morgan Kaufmann.
- OpenMP Application Programming Interface v6.0.
- OpenMP Application Programming Interface Examples v6.0.
- T. Mattson. A “Hands-on” Introduction to OpenMP.
- T. Mattson et al. The OpenMP Common Core: A hands on exploration.
- Blaise Barney. OpenMP.