Parallel Programming Principle and Practice

Lecture 6 — Shared Memory Programming OpenMP

Jin, Hai

School of Computer Science and Technology
Huazhong University of Science and Technology
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
Architecture for Shared Memory Model

Non-Uniform Memory Access

Uniform Memory Access
Thread Based Parallelism

- OpenMP programs accomplish parallelism exclusively through the use of threads

- A thread of execution is the smallest unit of processing that can be scheduled by an operating system
  - The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is

- Threads exist within the resources of a single process
  - Without the process, they cease to exist

- Typically, the number of threads match the number of machine processors/cores
  - However, the actual use of threads is up to the application
Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.
OpenMP Overview

OpenMP: An API for Writing Multithreaded Applications

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

- **1997**: OpenMP Fortran 1.0
- **1998**: OpenMP C/C++ 1.0
- **1999**: OpenMP Fortran 1.1
- **2000**: OpenMP Fortran 2.0
- **2002**: OpenMP C/C++ 2.0
- **2005**: OpenMP 2.5
- **2008**: OpenMP 3.0
- **2011**: OpenMP 3.1

- **1998**: A single specification for Fortran, C and C++
- **2002**: Tasking, other new features
- **2005**: A few more features and bug fixes
- **2008**: A few more features and bug fixes
- **2011**: A few more features and bug fixes
OpenMP Core Syntax

- Most of the constructs in OpenMP are compiler directives
  
  `#pragma omp construct [clause [clause]...]`

  - Example

  ```
  #pragma omp parallel num_threads(4)
  ```

- Function prototypes and types in the file

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

- Most OpenMP constructs apply to a **structured block**

  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - It’s OK to have an exit() within the structured block
OpenMP Overview: How do Threads Interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions
  - Race condition: when the program’s outcome changes as the threads are scheduled differently
- To control race conditions
  - Use synchronization to protect data conflicts
- Synchronization is expensive
  - Change how data is accessed to minimize the need for synchronization
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
OpenMP Programming Model

Fork-Join Parallelism

- All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.
Thread Creation: Parallel Regions

- Create threads in OpenMP with the `parallel` construct
- For example, to create a 4 thread in parallel region

```c
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
```

Each thread executes a copy of the code within the structured block.

- **Each thread calls** `pooh(ID, A)` for `ID = 0` to `3`
Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```c
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
omp_set_num_threads(4)
pooh(0, A)
pooh(1, A)
pooh(2, A)
pooh(3, A)
printf("all done\n");
```

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a \textit{barrier}).
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
Loop Worksharing Constructs

- Loop worksharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
{
  #pragma omp for
  for (I=0; I<N; I++){
    NEAT_STUFF(I);
  }
}
```

Loop construct name

- C/C++: `for`
- Fortran: `do`

The variable `I` is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause.
Loop Worksharing Constructs
A Motivating Example

Sequential code:

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

OpenMP parallel region:

```c
#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];}
}
```

OpenMP parallel region and a worksharing for construct:

```c
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```
Loop Worksharing Constructs: The *schedule* clause

- The *schedule* clause affects how loop iterations are mapped onto threads
  - *schedule*(static[,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread
  - *schedule*(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled
  - *schedule*(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds
  - *schedule*(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library)
  - *schedule*(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above)
## Loop Worksharing Constructs: The `schedule` clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
<tr>
<td>AUTO</td>
<td>When the runtime can “learn” from previous executions of the same loop</td>
</tr>
</tbody>
</table>

**Least work at runtime:** scheduling done at compile-time

**Most work at runtime:** complex scheduling logic used at run-time
Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```c
int i, j, A[MAX];
j = 5;
for (i=0; i< MAX; i++) {
j += 2;
A[i] = big(j);
}
```

```c
int i, A[MAX];
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Note: loop index `i` is private by default

Remove loop carried dependence
Nested Loops

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

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

- Will form a single loop of length \(N \times M\) and then parallelize that.
- Useful if \(N\) is \(O\)(no. of threads) so parallelizing the outer loop may not have good load balance.
Rules for Collapse Clause

- Only one collapse clause is allowed on a worksharing DO or PARALLEL DO directive.
- The specified number of loops must be present lexically. None of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested. There is no intervening code nor any OpenMP directive between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an EXIT statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a CYCLE statement, and there must be no branches to any of the loop termination statements except for the innermost associated loop.
Reduction

- How do we handle this case?

```c
double ave=0.0, A[MAX]; int i;
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed
- This is a very common situation … it is called a reduction
- Support for reduction operations is included in most parallel programming environments
Reduction

- OpenMP *reduction* clause
  
  reduction (op : list)

- Inside a parallel or a worksharing construct
  - A local copy of each list variable is made and initialized depending on the *op* (e.g. 0 for “+”)
  - Updates occur on the local copy
  - Local copies are reduced into a single value and combined with the original global value

- The variables in *list* must be shared in the enclosing parallel region

```c
double ave=0.0, A[MAX];  int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```
OpenMP: Reduction Operands/Initial Values

- Many different associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

C/C++ only

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fortran Only

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.IEOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IAND.</td>
<td>All bits on</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
<tr>
<td>MIN</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>MAX</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>


Example: Numerical Integration

Mathematically, we know that:

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

We can approximate the integral as a sum of 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 $i$. 

$$F(x) = \frac{4.0}{1+x^2}$$
Numerical Integration: Serial PI Program

```c
static long num_steps = 100000;
double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```
#include <omp.h>
static long num_steps = 100000;    double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
            for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
            }
    }
    pi = step * sum;
}
Single Worksharing Construct

- The *single* construct denotes a block of code that is executed by only one thread (not necessarily the master thread)
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause)

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
Synchronization

- High level synchronization
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush

Synchronization is used to impose order constraints and to protect access to shared data.
Synchronization: Critical

- Mutual exclusion: Only one thread at a time can enter a critical region

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

Threads wait their turn – only one at a time calls consume()
Synchronization: Atomic

- `atomic` provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```c
#pragma omp parallel
{
  double tmp, B;
  B = DOIT();
  tmp = big_ugly(B);
  #pragma omp atomic
  X += tmp;
}
```

Atomic only protects the read/update of X
Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```c
#pragma omp parallel shared (A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc1(id);

    #pragma omp barrier

    #pragma omp for
    for(i=0;i<N;i++){
        C[i] = big_calc3(i, A);
    }

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

    #pragma omp barrier
}
```

- Implicit **Barrier** at the end of a for worksharing construct.
- No implicit **Barrier** due to `nowait`.
- Implicit **Barrier** at the end of a parallel region.
Master Construct

- The *master* construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    {     exchange_boundaries();   }
#pragma omp  barrier
    do_many_other_things();
}
```
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
Data Environment: Default Storage Attributes

- Shared memory programming model
  - Most variables are shared by default

- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)

- But not everything is shared...
  - Stack variables in subprograms (Fortran) or functions (C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE
Data Sharing: Examples

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

```c
extern double A[10];
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
Data Sharing: Changing Storage Attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - √ SHARED
  - √ PRIVATE
  - √ FIRSTPRIVATE

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with
  - √ LASTPRIVATE

- The default attributes can be overridden with
  - √ DEFAULT (PRIVATE | SHARED | NONE)

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.
Data Sharing: Private Clause

- `private(var)` creates a new local copy of `var` for each thread
  - The value of the private copies is uninitialized
  - The value of the original variable is unchanged after the region

```c
void wrong() {
  int tmp = 0;
  #pragma omp parallel for private(tmp)
  for (int j = 0; j < 1000; ++j)
    tmp += j;
  printf("%d\n", tmp);
}
```

`tmp` was not initialized
`tmp` is 0 here
The original variable’s value is unspecified if it is referenced outside of the construct.

Implementations may reference the original variable or a copy …... a dangerous programming practice!

```c
int tmp;
void danger() {
    tmp = 0;
#pragma omp parallel private(tmp)
    work();
    printf("%d\n", tmp);
}
```

```c
extern int tmp;
void work() {
    tmp = 5;
}
```

`tmp` has unspecified value

unspecified which copy of `tmp`
Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

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

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

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```c
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 $i=(n-1)$)
Data Sharing: Default Clause

- Note that the default storage attribute is $\text{DEFAULT(SHARED)}$ (so no need to use it)
  - Exception: `#pragma omp task`
- To change default: $\text{DEFAULT(PRIVATE)}$
  - Each variable in the construct is made private as if specified in a private clause
  - Mostly saves typing
- $\text{DEFAULT(NONE)}$: no default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).
C/C++ only has default(shared) or default(none).
Outline

- OpenMP Overview
- Creating Threads
- Parallel Loops
- Synchronization
- Data Environment
- Tasks
What are Tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
- Tasks may be deferred, may be executed immediately
- The runtime system decides which of the above
- Tasks are composed of
  - code to execute
  - data environment
  - internal control variables (ICV)
Task Construct – Explicit Task View

- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – let's call this thread “L”
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region’s single construct

```c
#pragma omp parallel
{
  #pragma omp single
  {
    node * p = head;
    while (p) {
      // block 2
      #pragma omp task private(p)
      process(p);
      p = p->next; // block 3
    }
  }
}
```
Simple Task Example

```c
#pragma omp parallel num_threads(8)
// assume 8 threads
{
  #pragma omp single private(p)
  {
    ... 
    while (p) {
      #pragma omp task
      {
        processwork(p); 
      }
      p = p->next;
    }
  }
}
```

- A pool of 8 threads is created here.
- One thread gets to execute the while loop.
- The single “while loop” thread creates a task for each instance of `processwork()`.
Why are Tasks Useful?

Have potential to parallelize irregular patterns and recursive function calls.

```c
#pragma omp parallel
{
    #pragma omp single
    { // block 1
        node * p = head;
        while (p) { // block 2
            #pragma omp task
            process(p);
            p = p->next; // block 3
        }
    }
}  
```
When are Tasks Guaranteed to Complete

- Tasks are guaranteed to be complete at thread barriers
  
  `#pragma omp barrier`

- … or task barriers
  
  `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here
Consider recursive matrix multiplication, described in next 3 slides

➢ How would you parallelize this program using OpenMP tasks?

➢ What data considerations need to be addressed?
Recursive Matrix Multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions

\[
\begin{align*}
C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\
C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\
C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}
\end{align*}
\]
How to Multiply Submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply

\[
\begin{align*}
C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{11,1} &= A_{11,1} \cdot B_{11,1} + A_{11,2} \cdot B_{11,2}
\end{align*}
\]
Recurively Multiply Submatrices

\[
\begin{align*}
C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\
C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\
C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}
\end{align*}
\]

- Need range of indices to define each submatrix to be used

```c
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl, double **A, double **B, double **C)
{
    // Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]    C[mf..ml][nf..nl]
    // C11 += A11*B11
    matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);
    // C11 += A12*B21
    matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pf, A, B, C);
    // C11 += A21*B12
    matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nf, pf, pl, A, B, C);
    // C11 += A22*B22
    matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);
    ...
}
```

- Also need stopping criteria for recursion
Recursive Solution

Could be executed in parallel as 4 tasks

- Each task executes the two calls for the same output submatrix of C
13th International Workshop on OpenMP

Scaling OpenMP for Exascale Performance and Portability

Wang Center, Stony Brook University, Stony Brook, NY, USA

For many years, OpenMP has provided a very rich and flexible programming model for shared memory architectures. OpenMP 4.0 is a major advance that adds new forms of parallelism: device constructs for accelerators, SIMD constructs for vector units, and several significant extensions for work-sharing and task-based parallelism. OpenMP 4.5 further extends OpenMP for use with today’s complex heterogeneous architectures.

The International Workshop on OpenMP (IWOMP) is an annual workshop dedicated to the promotion and advancement of all aspects of parallel programming with OpenMP. It is the premier forum to present and discuss issues, trends, recent research ideas, and results related to OpenMP. We solicit submissions of unpublished technical papers detailing innovative, original research and development related to OpenMP.

IWOMP is co-located with its sister conference: OpenMPCon.

The OpenMP Developers’ Conference is the premier annual forum to present and discuss applications, tools, techniques, libraries and trends relating to parallel programming with OpenMP.
References

- The content expressed in this chapter comes from:
  - Clay Breshears, Intel Corp. clay.breshears@intel.com
  - J. Mark Bull, EPCC, The University of Edinburgh, markb@epcc.ed.ac.uk
  - Tim Mattson, Intel Corp. timothy.g.mattson@intel.com
  - Lawrence Livermore National Laboratory, OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/