Math 4370/6370 Lecture 4: Shared-Memory Parallel Programming with OpenMP

Daniel R. Reynolds

Spring 2015
reynolds@smu.edu
SMP Review

Multiprocessors:
- Multiple CPUs are attached to the bus.
- All processors share the same primary memory.
- The same memory address on different CPUs refers to the same memory location.
- Processors interact through shared variables.

Multi-core:
- Replicates substantial processor components on multiple chips
- Allows the processor to behave much like a shared-memory parallel machine.
- Standard on modern personal computers.

All of the parallel decomposition strategies that we previously discussed are possible on SMP computers, though some make better use of the shared address space.
Origins of OpenMP

After inventing SMPs, vendors needed to make them accessible to users.

Although compilers are typically responsible for adapting programs to hardware, for SMPs this is difficult since parallelism is not easily identifiable by a compiler.

To assist the compiler, a programmer could identify independent code regions to share among the processors; mostly focusing on distributing work in loops.

- Programs have dependencies where one processor requires another’s results.
- Thanks to shared memory, this is OK if things happen in the right order.
- However, since processors operate independently, this is not always the case.

In the 1980s vendors provided the ability to specify how work should be partitioned, and to enforce the ordering of accesses by different threads to shared data.

- The notation took the form of directives that were added to sequential programs.
- The compiler used this information to create the execution code for each process.
Although this strategy worked, it had the obvious deficiency that a program could not necessarily execute on SMPs made by different vendors.

In the late 1980s, vendors began to collaborate to improve this portability issue.

Eventually, OpenMP was defined by the OpenMP Architecture Review Board (ARB): a vendor group who joined forces to provide a common means of programming a broad range of SMP architectures. The first version, consisting of a set of Fortran directives, was introduced to the public in late 1997.

Since then C/C++ bindings have been added, and features have been added.

Today, almost all major computer manufacturers, major compiler companies, several government laboratories, and groups of researchers belong to the ARB.

A primary advantage of OpenMP is that the ARB continually ensures that OpenMP remains relevant as technology evolves. OpenMP is under continuous development; and features continue to be proposed for inclusion into the API.
What is OpenMP?

OpenMP is a shared-memory API, based on previous SMP programming efforts. Like its predecessors, OpenMP is not a new language, nor is it a library; it is a notation that can be added to a sequential program in C++, C or Fortran to describe how work should be shared among threads, and to order accesses to shared data as needed.

OpenMP Goals:

- Support the parallelization of applications from many disciplines.
- Be easy to learn and use, with added functionality for more advanced users.
- Permit an incremental approach to parallelizing a serial code, where portions of a program are parallelized independently, possibly in successive steps.
- Enable programmers to work with a single source code for both serial/parallel, to simplify program maintenance.
The OpenMP Approach

OpenMP follows a threaded approach for parallelism: a *thread* is a runtime entity able to independently execute an instruction stream.

With threads, the OS creates separate processes to execute a program:
- Allocates resources to each process (memory pages and cache registers).
- Threads collaborate by sharing resources (including address space).
- Individual threads need minimal resources of their own: a program counter and an area in memory to save private variables of its own.

OpenMP expects the programmer to specify parallelism at a high-level in the program, and to request a method for exploiting that parallelism.
- Provides notation for indicating regions that should be executed in parallel
- Provides optional specification on how this is to be accomplished.

It is OpenMP’s job to sort out the low-level details of creating independent threads and assigning work according to the strategy specified by the program.
The Fork-Join Programming Model

OpenMP’s approach to multithreaded programming supports the “fork-join” model:

- The program starts as a single execution thread, just like a sequential program. This thread is referred to as the *initial thread*.

- When a thread encounters a parallel construct, it creates a team of threads (the *fork*), becomes the team’s master, and collaborates with other team members to execute the enclosed code.

- At the end of the construct, only the master thread continues; all others terminate (the *join*).

- Each portion of the code enclosed by a parallel construct is called a *parallel region*. 
OpenMP Program Execution Possibilities

Uniform:

Dynamic:
Alternate SMP Approaches

OpenMP is not the only choice for utilizing an SMP system:

Automatic Parallelism (compiler dependent)

- Many compilers provide a flag for automatic program parallelization.
- Compiler searches program for independent instructions (loops with independent iterations), to generate parallelized code.
- This is difficult, since compiler may lack the necessary information to do a good job.
- Compilers assume a loop is sequential unless it can prove otherwise.
- The more complex the code, the more likely it is that this will occur.
- For programs with very simple structure, auto-parallelism may be an option, but in my experience it rarely works well.
Alternate SMP Approaches

OpenMP is not the only choice for utilizing an SMP system:

MPI (use distributed parallelism model on a shared memory system)

- MPI is designed to enable efficient parallel code, be broadly customizable and implementable on multiple platforms.
- The most widely used API for parallel programming in high-end technical computing, where large parallel systems are common.
- Most SMP vendors provide MPI versions that leverage shared address space, by copying between memory addresses instead of messaging.
- Requires significant reprogramming (no incremental parallelization).
- Modern large parallel MPPs consist of multiple SMPs, and MPI is increasingly mixed with OpenMP (so-called “hybrid parallelism”).
Alternate SMP Approaches

OpenMP is not the only choice for utilizing an SMP system:

Pthreads (C++/C only, though some Fortran interfaces exist)

- IEEE-designed threading approach for a Portable Operating System Interface (POSIX).
- Shared-memory programming model using a collection of routines for creating, managing and coordinating a collection of threads.
- Library aims to be highly expressive & portable; a comprehensive set of routines to create/terminate/synchronize threads, and to prevent different threads from modifying the same values at the same time.
- Significantly more complex than OpenMP; requires large code changes from a sequential program.
Alternate SMP Approaches

OpenMP is not the only choice for utilizing an SMP system:

Java threads (only available for Java)

- The Java programming language was built from scratch to allow for applications to have multiple concurrently-executing threads.
- However, Java was not designed to allow the compiler much room to optimize code, resulting in significantly slower execution than comparable C++/C/Fortran code.
- Hence, there are very few scientific or high-performance applications written in Java.
Alternate SMP Approaches

OpenMP is not the only choice for utilizing an SMP system:

Co-Processor computing (GPU, Intel Xeon Phi):

- Recent SMP systems with many processing units \(O(100)-O(1000))\), but small memory \(O(1)\) GB).

- GPUs were historically relegated to single-precision computations, though many current GPUs natively perform double-precision arithmetic.

- GPU programs are typically written in either CUDA or OpenCL. CUDA is a proprietary C++-like language by NVIDIA. OpenCL is a rapidly-developing open standard for GPU computing, similar to C.

- Some compilers for other languages attempt to auto-generate GPU code (PGI, IBM, Cray).

- The newest OpenMP standard (4.0) includes *portable* constructs for offloading calculations onto a co-processor, although most compilers do not yet support these features.
The OpenMP Programming Style

- OpenMP uses directives to tell the compiler which instructions to execute in parallel, and how to distribute them among the threads.

- These are pragmas/comments that are understood by OpenMP compilers only (ignored otherwise), allowing the code to compile in serial as well.

- The API is relatively simple, but has enough to cover most needs.

- C/C++ use pragmas to specify OpenMP code blocks:

  ```
  #pragma omp command
  { ... }
  ```

- Fortran uses specially-formatted comments to specify OpenMP code blocks:

  ```
  !$omp command
  ...
  !$omp end command
  ```
Creating an OpenMP Program

- A huge benefit of OpenMP is that one can incrementally apply it to an existing sequential code to create a parallel program.
  - Insert directives into one portion of the program at a time.
  - Once this has been compiled/tested, another portion can be parallelized.
- Basic OpenMP usage can create relatively efficient parallel programs.
- However, sometimes the basic directives are insufficient, leading to code that doesn't meet performance goals, with the fix not altogether obvious.
- Here, advanced OpenMP programming techniques can be applied.
- OpenMP therefore allows the developer to optionally specify increasing amounts of details on how to parallelize the code, through specifying additional options to the basic constructs.
Creating an OpenMP Program

- Step 1: identify any parallelism contained in a sequential program.
  - Find sequences of instructions that may be executed concurrently.
  - Sometimes this is easy, though it may require code reorganization, or even swapping an entire algorithm with an alternative one.
  - Sometimes this is more challenging, though strategies for exploiting certain types of parallelism are built-in to the OpenMP API.

- Step 2: use OpenMP directives and library routines to express the parallelism that has been identified.
  - We’ll go through these OpenMP constructs and functions throughout the rest of this lecture.

In these lectures, we'll only cover a subset of the OpenMP functionality within the 3.1 standard (the version 4.0 standard has been released, but is not yet widely supported by compilers).
The OpenMP Feature Set

OpenMP provides compiler directives, library functions, and environment variables to create/control the execution of shared-memory parallel programs.

Many directives are applied to a *structured block* of code, a sequence of statements with a single entry point at the top and a single exit at the bottom.

Many applications can be parallelized by using relatively few constructs and one or two functions.

OpenMP allows the programmer to:

- create teams of threads for parallel execution,
- specify how to share work among the members of a team,
- declare both shared and private variables,
- synchronize threads, and
- enable threads to perform certain operations exclusively.

OpenMP requires well-structured programs; where constructs are associated with statements, loops or structured blocks.
parallel – Create Thread Teams

Syntax:

```c
#pragma omp parallel [clause[[,] clause]...]
{ ... }
```

- This specifies computations that should be executed in parallel. Parts of the program not enclosed by a `parallel` construct will be executed in serial.

- When a thread encounters this construct, it forks a team of threads to execute the enclosed `parallel` region.

- Although ensuring that computations are performed in parallel, it does not distribute the work of the region among the threads in the team.

- Each thread in a team is assigned a unique number ranging from 0 (the master) up to one less than the number of threads within the team.

- There is an implied `barrier` at the end of the `parallel` region that forces all threads to wait until the enclosed work has been completed.

- Only the initial thread continues execution after the join at the end of the region.
#pragma omp parallel
{
    printf("The parallel region is executed by thread %i\n", omp_get_thread_num());
    if ( omp_get_thread_num() == 2 )
        printf(" thread %i does things differently\n", omp_get_thread_num());
}

produces the results (using 4 threads in the team):

The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
  thread 2 does things differently
The parallel region is executed by thread 1

Note that threads don’t necessarily execute in order!
**parallel – Create Thread Teams**

Clauses supported by the *parallel* construct (we’ll get to *clauses* soon):

- `if(scalar-logical-expression)`
- `num_threads(scalar-integer-expression)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `default(none|shared|private)`
- `copyin(list)`
- `reduction({operator|intrinsic_procedure_name}:list)`

Restrictions on the *parallel* construct and its clauses:

- A program cannot branch into or out of a *parallel* region.
- A program must not depend on the ordering of evaluations of the clauses of the *parallel* directive, or on any side effects of the evaluations of the clauses.
- At most one `if` clause can appear on the directive.
- At most one `num_threads` clause can appear on the directive.
Execution Control – Environment

The variable `OMP_NUM_THREADS` sets the number of threads to use for parallel regions.

- Alternatively, we may control this on a per-parallel-region basis, with the `parallel` clause `num_threads(nthreads)`.

Additional environment variables that help control execution in OpenMP programs:
- `OMP_SCHEDULE` – Sets the runtime schedule type and chunk size (more later)
- `OMP_DYNAMIC` – Enables dynamic control over # of threads in parallel regions
- `OMP_PROC_BIND` – Controls whether threads are bound to physical processors
- `OMP_NESTED` – Enables nested parallelism
- `OMP_STACKSIZE` – Sets the size of the stack for spawned threads
- `OMP_WAIT_POLICY` – Controls the behavior of waiting threads (active/passive)
- `OMP_MAX_ACTIVE_LEVELS` – controls the max # of nested active parallel regions
- `OMP_THREAD_LIMIT` – controls the max overall # of threads available for the program

Note: if any variable is not specified in the user's environment, then the behavior of the program is *implementation-defined* (i.e. it depends on the compiler).
Execution Control – Functions

We have more fine-grained control with OpenMP functions:

- `omp_get_wtime()` returns a double with the current wallclock time (precise timer)
- `omp_get_num_procs()` returns the total number of available processors
- `omp_get_thread_limit()` gets the max # of threads available to the program
- `omp_in_parallel()` returns `true` if called from inside a parallel region
- `omp_get_num_threads()` retrieves the number of threads in a current team
- `omp_set_num_threads()` overrides `OMP_NUM_THREADS`
- `omp_get_thread_num()` retrieves the integer ID of the calling thread (0-based)
- `omp_set_dynamic()` overrides `OMP_DYNAMIC`
- `omp_set_nested()` overrides `OMP_NESTED`
- `omp_get_max_active_levels()` returns the # of nested active parallel regions
- `omp_set_max_active_levels()` overrides `OMP_MAX_ACTIVE_LEVELS`
- `omp_get_level()` returns the current # of nested parallel regions
- `omp_get_ancestor_thread_num()` returns the ID of a thread's ancestor
- `omp_get_team_size()` returns the size of the ancestor's thread team
Conditional Compilation

A strong appeal of OpenMP is that directives allow code to work in serial and parallel. However, if a program uses OpenMP function calls (e.g. `omp_get_thread_num()`), this is no longer straightforward.

Therefore, OpenMP allows special characters to be inserted into a program to enable conditional compilation. An OpenMP-aware compiler will compile the optional code, while other compilers will treat the lines as comments.

In C++/C, the preprocessor will define the `_OPENMP` variable for OpenMP-aware compilers, so typical `#ifdef` statements may be used for conditional compilation:

```c
#ifdef _OPENMP
int nthreads = omp_get_num_threads();
#endif
```

In Fortran90, the characters `!$` declare a line for conditional compilation:

```fortran
!$ nthreads = omp_get_num_threads()
```
Sharing Work Among Threads

- If the distribution of work is left unspecified, each thread will redundantly execute all of the code in a parallel region (Note: this does not speed up the program).

- Work-sharing directives allow specification of how computations are to be distributed among threads.

- A work-sharing construct specifies both a region of code to distribute among the executing threads, and how to distribute that work.

- A work-sharing region must be in a parallel region to have any effect.

- OpenMP includes the work-sharing constructs: loop, sections, single, task and workshare (F90 only)

Two main rules regarding work-sharing constructs (excluding task):

1) Each work-sharing region must be encountered by all/no threads in a team.

2) The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.
Sharing Work – for Loop Construct

For loop construct syntax ("omp do" in Fortran):

```plaintext
#pragma omp for [clause][]clause]...
for-loop
```

- This causes the enclosed loop iterations to be executed in parallel.
- At run time, the loop iterations are distributed across the threads.

Supported clauses:

- `private(list)`, `firstprivate(list)`, `lastprivate(list)`
- `reduction({operator|intrinsic_procedure_name}:list)`
- `schedule(kind[,chunk_size])`
- `collapse(n)`
- `nowait`
- `ordered`
Sharing Work – for Loop Construct

Example:

```c
#pragma omp for shared(n) private(i)
for (i=0; i<n; i++)
  printf("Thread %i ran iteration %i\n", omp_get_thread_num(), i);
```

When run on 3 threads with n=8, we may have

- Thread 1 ran iteration 0
- Thread 0 ran iteration 2
- Thread 0 ran iteration 3
- Thread 2 ran iteration 1
- Thread 2 ran iteration 6
- Thread 1 ran iteration 4
- Thread 0 ran iteration 5

Notes:
- We cannot expect results to be printed in a deterministic order.
- Threads 1 & 2 executed 2 iterations, while thread 0 executed 3.
- Unless specified, the distribution will be implementation-dependent.
- A compiler can pick different strategies for different loops in the same application.
Sharing Work – sections Construct

sections construct syntax:

```c
#pragma omp sections [clause[[], clause]...]
{
  #pragma omp section
  { ... }
  #pragma omp section
  { ... }
  ...
}
```

Supported clauses:

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction({operator|intrinsic_procedure_name}:list)
- nowait
A simple way to enable MPMD programming, since we can specify different code regions, each of which is executed by only one of the threads.

Consists of two directives:

- `#pragma omp sections { ... }` indicates the bounds of the construct.
- `#pragma omp section { ... }` marks each distinct section.

A section is a structured block of code that is independent of other sections. At run time the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once.

If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks.

If there are fewer code blocks than threads, the remaining threads will be idle.

The assignment of code blocks to threads is implementation-dependent.
Sharing Work – sections Construct

Although it provides a general mechanism for MPMD programming, it is most commonly used to execute subroutine or function calls in parallel:

```c
#pragma omp sections
{
  #pragma omp section
  funcA();
  #pragma omp section
  funcB();
}
```

Example notes:

- Contains only two sections (limiting parallelism to 2 threads).
- If at least two threads are available, `funcA()` and `funcB()` are executed in parallel. If only one thread is available, both calls will be executed sequentially.
- We cannot predict the order in which the sections are run – even if run with one thread, `funcB()` may be called before `funcA()`.
- Can lead to load balancing issues sections’ work loads differ dramatically.
Sharing Work – single Construct

Construct syntax:

```c
#pragma omp single [clause[[[],] clause]...]
{ ... }
```

- Specifies that the encompassed block should be executed by one thread only.
- Does not state which thread executes the block – it could vary between runs or even different single constructs within one application.
- Should be used when it does not matter which thread executes the code block.
- Remaining threads wait at a barrier until the single code block has completed.

Supported single construct clauses:

- `private(list)`
- `firstprivate(list)`
- `copyprivate(list)`
- `nowait`
Sharing Work – single Construct

Example demonstrating the single construct to initialize a shared variable:

```c
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp single
    {
        a = 10;
        printf("Single construct executed by thread %i\n", omp_get_thread_num());
    }
    #pragma omp for
    for (i=0; i<n; i++)  b[i] = a;
}
printf("After the parallel region:\nb =");
for (i=0; i<n; i++)
    printf(" %g",b[i]);
```
Sharing Work – single Construct

Results:

Single construct executed by thread 3
After the parallel region:
\[ b = 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \]

- Only thread 3 initializes the variable \( a \), others wait until \( a \) is initialized.
- The shared variable \( a \) is then used to initialize vector \( b \) in the parallel for loop.

Notes:
- Could argue the necessity of \textit{single} here since all threads would write the same value to the same variable \( a \).
  - However, this assumes that the machine does the “right thing” when all threads attempt to write the same variable at once.
  - Plus, multiple stores to the same memory location are bad for performance.
- A barrier is needed before the parallel for loop, otherwise some threads would begin to assign elements of \( b \) before \( a \) has been assigned a value. The implicit barrier at the end of the \textit{single} obviates the need for an explicit barrier.
Sharing work – task Construct

task construct syntax:

```c
#pragma omp task [clause[ [, ] clause]...] 
{ ... }
```

- Specifies that the encompassed block should be executed by one thread only (like sections, but typically used inside a loop).
- Does not state which thread executes the task (like sections).
- Tasks may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.
- Programs may not branch into, or out of, a task region.
- Synchronization occurs with a taskwait construct.
- Adds task to queue at execution point, unlike section (all created at sections).

Supported clauses:

- `if(scalar-logical-expression)`
- `final(scalar-logical-expression)`
- `private(list), shared(list), firstprivate(list)`
- `untied`
- `mergeable`
Sharing work – task Construct

Simple example:

```c
#pragma omp parallel
{
    #pragma omp single
    {
        printf("A ");

        #pragma omp task
        printf("race ");

        #pragma omp task
        printf("car ");

        #pragma omp taskwait
        printf("is fun to watch\n");
    } // end of single region
} // end of parallel region
```
Sharing work – task Construct

Either one of two possibilities can occur:

- A race car is fun to watch
- A car race is fun to watch

- Note that since task is inside single, each task is only created once.
- The tasks are synchronized at the taskwait.
- Since it's still inside the single construct, “is fun to watch” is only printed once.

Notes:

- Tasks may be suspended by the runtime environment during execution; resuming tasks are “tied” to the executing thread unless untied (helpful for private data).
- As the computation progresses, the work performed per task may shrink:
  - E.g. recursive algorithms, where each recursion spawns tasks.
  - This is where the final clause may come handy.
- Alternatively, since tasks can themselves spawn new tasks, the data environment can grow too much – this is where mergeable is useful.
workshare Construct (F90 only)

workshare construct syntax:

```fortran
!$omp workshare
  structured block
!$omp end workshare [nowait]
```

Supported clauses:

- `nowait`

Notes:

- Enables the parallel execution of F90 whole array operations.
- Statements in the construct are divided into units of work, which are executed in parallel in a manner that respects the semantics of F90 array operations.
- The definition of “unit of work” depends on the Fortran work construct, e.g. if applied to an assignment statement, the assignment of each entry in the array is a separate work unit.
Combined Parallel Work-Sharing Constructs

If a work-sharing construct comprises all code in a parallel region, the parallel directive may be combined with the work-sharing directive, allowing for directives that are easier to read and that may be slightly more efficient.

| Loops:          | Combined construct: #pragma omp parallel for for-loop { ... }
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>#pragma omp for</td>
<td></td>
</tr>
<tr>
<td>for-loop {</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sections:</th>
<th>Combined construct: #pragma omp parallel sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>#pragma omp sections</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>#pragma omp section</td>
<td></td>
</tr>
<tr>
<td>structured block</td>
<td></td>
</tr>
<tr>
<td>#pragma omp section</td>
<td></td>
</tr>
<tr>
<td>structured block</td>
<td></td>
</tr>
<tr>
<td>#pragma omp section</td>
<td></td>
</tr>
<tr>
<td>structured block</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Example: Vector Operations $z = ax + y$

Interactive demo of the serial version and its parallelization with OpenMP.

Here:
- $z$, $x$ and $y$ are vectors (1D arrays)
- $a$ is a scalar

Key ideas:
- parallel construct
- for construct
- combined parallel for
Customizing OpenMP Constructs and Memory Control: Clauses

As seen in the earlier constructs, OpenMP allows optional clauses that provide a simple but powerful way to control behavior of their construct.

These include syntax to specify between shared and private variables, in addition to others that we have previewed in the earlier slides.
Nested Loops: the collapse clause

The collapse clause specifies that an omp for (or omp do) can be applied to all iterations within a set of nested loops. The argument defines the nesting level.

Syntax: collapse(n)

The argument must be a positive integer expression, indicating the number of nested loops to collapse within one larger iteration space, that is then divided among threads.

```cpp
#pragma omp parallel for collapse(3)
for (k=0; k<nz; k++)
  for (j=0; j<ny; j++)
    for (i=0; i<nx; i++)
      a[i][j][k] = foo(u[i][j][k]);
```

- The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.
- The iteration count for all associated loops is computed before entry to the outermost loop, so loop extents should not be modified within any associated loops.
Shared vs. Private Memory

OpenMP is based on the shared-memory model:

- By default, data is shared among all threads and is visible to all of them.
- Sometimes, we need variables with thread-specific values. When each thread has its own copy of a variable (with different values), the variable is private.
- Each thread needs its own value of a loop iteration variable. The compiler enforces this by default; otherwise the user must declare private variables.
- Data can be shared/private for a parallel region, or just a work-sharing region.
- Private variables can reduce the frequency of updates to shared memory. Thus, they may help avoid competition for access to certain memory locations.
- However, private variables increase the program’s memory footprint.
- Each thread has its own thread stack to store private data at run time.
- Most compilers give the thread stack a default size. Sometimes private data requirements can be quite large, so the default may not be large enough.
- Fortunately, it is possible to increase the stack size.
Shared vs. Private Memory

We previously discussed cache-coherency issues on SMP architectures.

- Fortunately, one need not know how a specific system deals with this problem, since OpenMP has its own rules about when shared data is visible to all threads.
- OpenMP requires that shared values must be available at synchronization points. Between synchronization points, threads may keep updated values in local cache.
- Hence, threads may temporarily have different values for some shared objects.
- If one thread needs a value that was created by another thread, then a synchronization point must be inserted into the code.

OpenMP has an additional operation called flush to synchronize memory.

- This ensures that the thread has the same shared data values as in main memory.
- New values of shared objects updated by the thread are written back to shared memory, and the thread gets any new shared values produced by other threads.
- In some programming languages, a flush is known as a memory fence, since reads and writes of shared data may not be moved relative to it.
The shared Clause

The shared clause specifies which data will be shared among threads in a region.

Syntax: \texttt{shared(list)}

All items in the comma-separated \textit{list} will be shared among the threads in the team.

\begin{verbatim}
#pragma omp parallel for shared(a)
for (i=0; i<n; i++)
a[i] += i;
\end{verbatim}

- Here, \texttt{a} is declared to be shared, so all threads can read and write elements of \texttt{a}.
- Within the \texttt{parallel for} loop, each thread will access the pre-existing values of those elements \texttt{a[i]} that it is responsible for, and will compute their new values.
- \textbf{WARNING:} multiple threads can try to simultaneously update the same memory location, or one thread might try to read from a location that another is updating (though not in this example).
The **private** Clause

The private clause specifies which data should be duplicated so that each thread has their own copy.

**Syntax:** `private(list)`

- Each variable in the comma-separated `list` is replicated so that each thread in the team has exclusive access to a local copy.
- Changes made to private data by a thread are not visible to other threads.
- It is always safest to specify data-sharing attributes explicitly.

**Example:**

```c
#pragma omp parallel for private(i,a)
for (i=0; i<n; i++) {
    a = i+1;
    printf("Thread %i has value of a = %g for i = %i\n", omp_get_thread_num(), a, i);
}
```
The private Clause

This example has output:

Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 1 for i = 0
Thread 1 has a value of a = 3 for i = 2
Thread 0 has a value of a = 6 for i = 5
Thread 1 has a value of a = 4 for i = 3
Thread 2 has a value of a = 5 for i = 4

Notes:

- Values of private data are **undefined** upon *entry to*, and *exit from*, the construct.
- Declaring i as “private” is redundant, since the index in a parallel for loop is *always* private.
- The value of any generally-accessible variable with the same name as a private variable becomes undefined after the construct has terminated, **even if the corresponding variable was defined prior to the region.**
The default Clause

The default clause is used to give variables a default data sharing attribute.

Syntax: default(none|shared|private)

- default(shared) assigns the shared attribute to all variables in the construct – this is the default in OpenMP.
- default(private) makes all variables private by default. It is applicable to the parallel construct only.
- default(none) forces the programmer to specify a data-sharing attribute for each variable in the construct.

This clause is most often used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed:

```c
#pragma omp parallel default(shared) private(a,b,c)
```

- Declares all variables to be shared, with the exception of a, b and c.

Although the OpenMP default is shared, I will require that attributes be specified for all variables in a construct via use of default, private and/or shared.
The reduction Clause

Syntax: reduction(operator:list)

- Allows performing some kinds of recurrence calculations (involving mathematically associative and commutative operators).
- Identify operations and result variables; the rest is up to the compiler.
- Result variable type must be valid for the given operator or intrinsic procedure, cannot be array-valued (unless in F90).
- Results will be “shared” by default; doesn't require an explicit shared declaration.
- Highly preferable to implementing reduction manually (as we’ll see later).

Example: dot-product

```c
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<n; i++)
    sum += a[i]*b[i];
```
The reduction Clause

Notes:

- Compiler generates code similar to manual code, but may be more efficient.

- Depending on the operator/intrinsic specified, the initial value of the shared reduction variable may be updated and not overwritten (as in this example).

- The order in which thread-specific values are combined is unspecified, so floating-point roundoff can give different results between runs.
Example: Dot-Product  \( a = x^T y \)

Interactive demo of the parallelization with OpenMP.

Here:
- \( x \) and \( y \) are vectors (1D arrays)
- \( a \) is a scalar

Key ideas:
- parallel construct
- for construct
- reduction clause
Example: Computing $\pi$

Interactive demo of the parallelization with OpenMP.

Key ideas:

- parallel construct
- for construct
- private clause
- reduction clause
More Advanced Memory Clauses

lastprivate(list)

- Variables in list are accessible after the construct [for, sections] has completed, containing the last value it would have had if executed in serial (may result in a slight performance penalty).

firstprivate(list)

- Variables in list are private, but are pre-initialized with the existing value before entry to the construct [parallel, for, sections, task, single].

copyin(list)

- Copies the values of list from the master thread into private variables of other threads; occurs after threads created but before execution of parallel region [parallel only].

copyprivate(list)

- Broadcasts values of private variables in list from one thread to all others in team [single only].

flush(list)

- Makes each thread's private values in list consistent with the current value in main memory, but does not synchronize threads.
Thread Synchronization

Synchronizing threads is sometimes necessary in order to ensure the proper ordering of accesses to shared data, and to prevent data corruption.

- Ensuring proper thread coordination is one of the toughest challenges of SMP programming.
- Synchronization points are those places in the code where synchronization has been specified, either explicitly or implicitly.
- They have an additional function for OpenMP code: at synchronization points the system ensures that threads have consistent values of shared data objects.
- OpenMP’s synchronization points include explicit and implicit barriers, the start and end of critical regions, points where locks are acquired or released, and anywhere the programmer has inserted a flush directive.

Thread synchronization constructs include: barrier, taskwait, ordered, critical, atomic, lock and master.
The barrier Construct

Syntax:

```c
#pragma omp barrier
```

- A barrier is a point where all threads in a team wait for each other: no thread in the team may proceed beyond a barrier until all threads have reached that point.
- While many constructs have an implied barrier, this explicitly defines one.
- `taskwait` is a barrier that is specific to task-based parallelism.

Two important restrictions on the barrier construct:

- Each barrier *must* be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

Barriers are commonly used to avoid *data race conditions*:

- Inserting a barrier between the writes to, and reads from, a shared variable guarantees that data accesses are appropriately ordered.
- For example: ensuring a write is completed before another thread reads the data.
The master Construct

Construct syntax:

```cpp
#pragma omp master
{
  ...
}
```

- Like `single`, `master` specifies that the code will be executed by one thread only; in this case it is thread 0 (the master thread) that executes the code.

- Unlike `single`, `master does not have` an implied barrier on either entry to or exit from the code block.

- If used to initialize data, care must be taken to ensure that initialization is performed before other threads in the team use the data. Typically, one relies on an implicit barrier later on in the code, or specifies an explicit barrier.

- Does not support additional clauses.
The ordered Construct/Clause

Syntax:

```c
#pragma omp ordered
{
  ...  
}
```

- Allows execution of operations in a parallel loop in sequential order.
- When the thread executing the first iteration of the loop encounters the construct, it enters the region without waiting.
- When a thread executing any subsequent iteration encounters the construct, it waits until the previous iterations in the sequence have left the region.
- An ordered clause must also be added to the for loop region containing the construct, to inform the compiler of an ordered construct contained therein.
- Useful to enforce an ordering on the data I/O.
- Also helpful in debugging for data race conditions.
- Will result in a performance hit.
The ordered Construct-Clause

Example:

```c
#pragma omp parallel for ordered
for (i=0; i<n; i++) {
    a[i] = func(i);
    #pragma omp ordered
    printf("a[%i] = %g\n",a[i]);
}
```

Notes:

- The ordered clause on the parallel for loop instructs the compiler that the loop contains an ordered construct.
- The ordered construct inside the loop forces the `printf()` to be executed in iteration order, i.e. first for `i=0`, then 1, etc.
- Since `func()` is *not* inside an ordered construct, it is executed in parallel among the threads.
The critical Construct

Construct syntax:

```c
#pragma omp critical [(name)]
{
    ...
}
```

- Ensures that multiple threads do not update shared data simultaneously.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name before entering.
- The associated code is referred to as a critical region, or a critical section.
- An optional name can be given to a critical construct. This name is GLOBAL to the entire program and should be unique!
The critical Construct

Example: dot-product (without using the reduction clause)

```c
sum = 0.0;
#pragma omp parallel private(sumlocal,i)
{
    sumlocal = 0.0;
    #pragma omp for
    for (i=0; i<n; i++)
        sumlocal += a[i]*b[i];
    #pragma omp critical (update_sum)
    sum += sumlocal;
}
```
Example: Computing \( \pi \) (again)

Interactive demo of the parallelization with OpenMP.

Key ideas:

- Manual reduction via \textit{private} variables.
- \textit{critical} clause for thread contributions to global result.
The schedule Clause

Syntax: schedule(kind [,chunk_size])

The schedule clause specifies how the iterations in a loop are assigned to the threads in the team; it is supported on the for construct only.

- Can have a major impact on the performance of a program.
- The granularity of this workload distribution is a chunk – a contiguous, nonempty subset of the iteration space.
- chunk_size need not be a constant; any positive integer expression is allowed.
- There are 5 ‘kinds’: static, dynamic, guided, auto and runtime.
- static is the simplest, having the least overhead (often the default).
- Both dynamic and guided help for imbalanced and unpredictable workloads.
- auto is implementation-dependent.
- runtime saves the determination of scheduling until the program is run, at which time it obtains the schedule from an environment variable.
The schedule Clause

Static:
- Iterations are divided into chunks of size $chunk_size$.
- Chunks are assigned to threads statically in a round-robin manner.
- The last chunk to be assigned may have a smaller number of iterations.
- If $chunk_size$ is unspecified, iteration space is divided into equally-sized chunks.
- Each thread is assigned at most one chunk.

Dynamic:
- Iterations are assigned to threads as the threads request them.
- A thread executes a chunk of iterations (of size $chunk_size$), then requests another chunk, etc., until no more chunks remain.
- The last chunk may have fewer iterations than $chunk_size$.
- When $chunk_size$ is unspecified, it defaults to 1.

Auto:
- You can define the $chunk_size$, but that's about it.
The schedule Clause

Guided:
- Iterations are assigned to threads as the threads request them.
- Unlike dynamic, the chunks decrease in size.
- If chunk_size equals 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1.
- If chunk_size equals k (with k>1), the size of each chunk is determined as above, except that the minimum chunk size is k iterations (except for the last one).
- When unassigned, chunk_size defaults to 1.

Runtime:
- If this is selected, the decision regarding scheduling is made at run time.
- Schedule and chunk size set through the OMP_SCHEDULE environment variable.
- This variable takes the form:
  \[ \text{kind} [, \text{chunk} \_ \text{size}] \]
  where kind is one of static, dynamic, auto or guided.
- The optional parameter chunk_size is a positive integer.
### Schedule Examples

#### Key:
- **I** = iteration
- **S** = Static
- **D** = Dynamic
- **G** = Guided
- (N) = chunk size

Non-deterministic allocations depend on many factors, including the system load.

Static is often most efficient (less overhead), if load-balancing is nearly uniform.

<table>
<thead>
<tr>
<th>I</th>
<th>S</th>
<th>S(2)</th>
<th>D</th>
<th>D(2)</th>
<th>G</th>
<th>G(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: Global Minimization

Interactive demo of the parallelization with OpenMP.

Key ideas:
- parallel construct
- for construct
- schedule clause for load-balancing
- private variables
Orphan Directives

We may insert directives into procedures that are invoked from inside a parallel region.

Known as orphan directives, since they are not in the routine where the parallel region is specified.

If the orphan is executed outside a parallel region, it is just ignored.

```c
int main() {
   // initialize vectors
   double a[100], b[100], c[100];
   for (int i=0; i<100; i++) {
      a[i] = 1.0;
      b[i] = 2.0*i;
      c[i] = -1.0*(100-i);
   }

   #pragma omp parallel addvecs(n,a,b,c);

   return 0;
}

void addvecs(int n, double *a,
              double *b,
              double *c)
{
   #pragma omp for
   for (int i=0; i<n; i++)
      c[i] = a[i] + b[i];

   return;
}
```
Example: Chemical Kinetics

Interactive demo of the parallelization with OpenMP.

Key ideas:

- parallel construct.
- for construct.
- schedule clause for load-balancing.
- orphan directives for subroutines within a parallel region.
The nowait Clause

The nowait clause allows us to fine-tune a program’s performance:

- May be added to the constructs: for, sections, single, workshare; cannot be combined with the copyprivate clause.
- Recall that work-sharing constructs (except task) have an implicit barrier at the end.
- The nowait clause overrides that feature: if it is added to a construct, the barrier at the end of the associated construct will be suppressed.
- When threads reach the construct end, they immediately continue to other work.
- Note: the barrier at the end of a parallel region cannot be suppressed.

To use this construct:

- Once a parallel program runs correctly, try to identify places where a barrier may not be needed. Insert the nowait clause, and test thoroughly.
- Care must be exercised, because its incorrect use can introduce nasty bugs.
Programming Styles/Considerations

Low-level programming:

- OpenMP encourages high-level structured programming. However, sometimes loop-level ||-ism is limited, or directive-based ||-ization can lead to high overheads.

- It's possible to write OpenMP programs that do not rely on work-sharing directives but rather assign work explicitly to different threads using their thread numbers.

- This can lead to highly efficient code, but synchronization must be inserted manually to ensure that accesses to shared data are correctly controlled. Otherwise, errors such as deadlock (when all threads wait for each other in perpetuity) may occur.

- Approaches requiring manual assignment of work to threads and that need explicit synchronization are called “low-level programming.”

- This can be very effective and is broadly applicable, but it requires much more programming effort and careful design to ensure program correctness.
Programming Styles/Considerations

Data race conditions:

- It can be difficult in SMP programming to ensure program correctness, since SMP programs can contain new types of bugs.
- Fortunately, structured, directive-based programming may help prevent problems; for low-level programming, however, more care is needed.
- One kind of error in particular, a data race condition, can be extremely difficult to detect, manifesting itself in SMP code through silent data corruption.
- Essentially, two or more threads access the same shared variable without any synchronization to order the accesses, and at least one of the accesses is a write; sometimes the other threads get the new value, other times the old.
- Unfortunately, data race conditions may not be easily reproducible: erroneous data from one run may not show up the next time.
- Since it's easy to create a parallel loop without noticing that multiple iterations reference the same array element, a programmer must take care.
- The more complex the code, the harder it is to guarantee it is error-free, especially as codes are “optimized” to squeeze out additional performance.
Data race conditions (continued):

- The order of operations for a code with a data race condition may depend on the system load and the relative timing of the threads involved.
- Since threads may execute their instructions at different speeds, the order in which they reach certain code can vary between runs.
- Sometimes, the problem occurs only for a specific number of threads, thereby escaping detection during testing and even for some production runs.
- Thus it is critical to take care during program development to avoid this problem.

Other potential errors in OpenMP programs:

- If the program is written for a specific number of threads, but run using a different number, results may be unexpected.
- OpenMP expects you to ensure that the execution environment is set up correctly, and provides runtime routines for such queries.
- Incorrect use of synchronization constructs can lead to hard-to-identify problems. One must carefully think through the logic of explicit synchronizations if used.
Programming Styles/Considerations

Performance Considerations:

- Recall Amdahl’s law: the speedup for a parallel program using $p$ processors, in which $f$ is the fraction of non-parallelizable work, is given by

$$S = 1 / (f + (1-f)/p)$$

- The forking and joining of threads, thread synchronization, and shared-memory accesses all cause overheads that increase $f$ beyond those in the serial algorithm alone.

- On the other hand, the ability to fit more program data into cache may offset some of this overheads (recall the superlinear speedup homework question).
OpenMP Performance Concerns

OpenMP application performance is attributable to a number of factors:

- **Thread memory accesses:** if each thread accesses a distinct portion of memory consistently throughout a program, it can effectively utilize the thread-local cache.

- **The overhead required in handling OpenMP constructs:** it's helpful to understand parallel overheads: create/start/stop threads, task distribution, idling at barriers, redundant computations, etc.

- **The fraction of parallelizable work.**

- **Load imbalance.**

- **Synchronization costs:** critical regions, atomic updates, or waiting to set a lock.

- **System over-subscription:** it's usually recommended to use fewer than the number of available processors (since at least one must run the OS), though on some applications it can be useful to over-decompose the parallelism by a factor of 3 or more.
Best Practices: Optimize Barrier Use

All barriers incur overhead. Fortunately, `nowait` allows us to prune unneeded barriers.

- First, ensure the OpenMP program works correctly as-is.
- Add in `nowait` clauses where possible, carefully inserting explicit barriers as needed.
- Test the code as you go, ensuring consistency of results along the way.

```c
#pragma omp parallel default(shared) private(i)
{
    #pragma omp for nowait
    // the implied barrier is not needed
    for (i=0; i<n; i++)  a[i] += b[i];

    #pragma omp for
    // implied barrier is needed for c in next loop
    for (i=0; i<n; i++)  c[i] += d[i];

    #pragma omp for reduction(+:sum) nowait
    // implied barrier not needed, but enforced anyway
    for (i=0; i<n; i++)  sum += a[i] + c[i];
}
```
Avoiding Over-Synchronization

Try to avoid the ordered construct:

- This is a serial operation, and it is expensive to implement.
- In many cases, work from an ordered construct can often be moved outside a parallel region, resulting in faster execution.

Avoid large critical regions – these ensure that no two threads execute code simultaneously.

- The larger this code region, the greater probability that threads will sit idle waiting to enter.
- In many cases, an atomic update (in a few slides) can be more efficient since it only enforces exclusive access to a single memory location.
Minimizing Threading Overhead

Due to the overhead of spawning and terminating threads using OpenMP, there are a number of simple techniques to make programs more efficient.

Maximize Parallel Regions – It may be better to encompass more of a code in a single parallel region than to use multiple different ones.

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

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

    #pragma omp parallel for
    for (i=0; i<n; i++)
        c[i] += a[i];

    #pragma omp parallel for
    for (i=0; i<n; i++)
        c[i] += a[i];
}
```
Minimizing Threading Overhead

Avoid Creating Parallel Regions inside Loops

Similarly, placing parallel constructs inside of loops may result in repeated overheads for spawning and terminating threads.

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        aval = 0.0;

        #pragma omp parallel \
        for reduction(+:aval)
        for (k=0; k<n; k++)
            aval += B[i][k]*C[k][j];

        A[i][j] = aval;
    }
}
```

```c
#pragma omp parallel
{
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            #pragma omp single
            aval = 0.0;

            #pragma omp for \
            reduction(+:aval)
            for (k=0; k<n; k++)
                aval += B[i][k]*C[k][j];

            #pragma omp single
            A[i][j] = aval;
        }
    }
}
```
Example: Advection

Interactive demo of the parallelization with OpenMP.

Key ideas:
- parallel construct
- for construct
- default clause
- shared clause
- private clause
- Dual parallelism approaches: outer loop vs. inner loop
Best Practices (cont)

Address Poor Load Balancing

- In some parallel algorithms, threads may have different amounts of work, so static scheduling may result in large load-balancing overheads.
- One way to overcome such problems is through using a dynamic or guided schedule.

Single vs. Master

- Both the single and master constructs have similar functionality.
- Single can be executed by any thread, but has an implied barrier.
- Master can only be performed by thread 0, but has no implied barrier, so it may be more efficient since single incurs increased overhead.
- But if thread 0 is not the first to reach a code block, single could be more efficient.
- If no barrier is needed, then single + nowait may be the best of both.
OpenMP Benchmarking Demo

Run through the omp_benchmarking example programs to get an idea about the overheads associated with various OpenMP constructs.
Many believe that OpenMP’s fork-join model causes unnecessary overheads for threaded computation, especially for computations with a low amount of work compared to each fork/join operation.

A helpful goal is to push toward coarse-grained parallelism. Recall granularity from lecture 1: the size of tasks into which a problem is decomposed:

Many small tasks → fine-grained; Few large tasks → coarse-grained.

If possible, try to enclose an entire computation in one large parallel region.

Can be challenging, since loop-level parallelism is typically fine-grained, whereas coarse-grained programs often require MPMD programming.

MPMD programming typically involves more effort, but can provide greater scalability:

The codes contain a small number of large parallel regions.

Work sharing is controlled by the user, often based on the thread identifier (TID).

Major data structures distributed among threads, with most data as private, utilizing cache more optimally.
SPMD Example – Stencil Operation

Serial code:

```c
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        u[i][j] += (v[i-1][j] + v[i+1][j] +
                    v[i][j-1] + v[i][j+1])/4.0;
```

Simple OpenMP parallelization:

```c
#pragma omp parallel for private(i,j)
{
    for (i=0; i<n; i++)
        for (j=0; j<m; j++)
            u[i][j] += (v[i-1][j] + v[i+1][j] +
                        v[i][j-1] + v[i][j+1])/4.0;
}
SPMD Example – Stencil Operation

MPMD:

```c
int NT, NTi, NTj;
#pragma omp parallel default(shared)
{
#pragma omp single
{
    NT = omp_get_num_threads();
    NTi = (int) (sqrt(1.0*NT));
    NTj = NTi;
}
// note: since declared inside the parallel
// region, these are private (C++ only)
int TID = omp_get_thread_num();
int il = n*TID/NTi;
int ir = n*(TID+1)/NTi;
int jl = m*TID/NTj;
int jr = m*(TID+1)/NTj;
for (int i=il; i<ir; i++)
    for (int j=jl; j<jr; j++)
        u[i][j] += (v[i-1][j] + v[i+1][j] +
                    v[i][j-1] + v[i][j+1])/4.0;
```
Advanced OpenMP

The following slides discuss some useful, though more advanced, OpenMP techniques.
The **atomic** Construct

**Syntax:**

```c
#pragma omp atomic statement
```

- Also enables multiple threads to update shared data (like critical), but applies only to the single assignment statement immediately following it.
- Enables efficient SMP updating on hardware that supports **atomic** operations.
- Instructs the computer to read from a memory location, modify the value, and write back to the location *all in one action* (if supported by the hardware).
- If a thread is atomically updating a value, no other thread *in the entire program* (not just the team) may update the same value at that time.
- Programmer must mark *all* potentially simultaneous updates to a memory location as **atomic**.

**statement** must have one of the following forms:

- `x binop= expr, x++, ++x, x--, --x`

  Here, `x` must have scalar type, `expr` cannot reference `x`, and `binop` must be one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<` or `>>`
The **atomic** Construct

Example:

```c
#pragma omp parallel private(i,i2)
{
    for (i=0; i<n; i++) {
        #pragma omp atomic
        i1 += 1;
        #pragma omp atomic
        i2 *= func();
    }
}
```

- Multiple threads update `i1`, with one thread updating it at a time.
- Multiple threads update `i2`, but `func()` will still be executed by multiple threads simultaneously.
- If `func()` can only be run by one thread at a time, must use `critical`. 
The if Clause

- Supported under the parallel and task constructs; determines whether to create the indicated parallel region or spawn the indicated task.
- Useful for testing whether there is enough work in the region to warrant parallelization (to make up for the extra overhead of spawning threads).

```c
#pragma omp parallel if (n>5) default(none) \ 
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("value of n = %i\n",n);
        printf("number of threads in region = %i\n", omp_get_num_threads());
    }
    printf("I am thread %i\n",TID);
}
```
Locking Functions

Syntax:

```c
void omp_func_lock(locktype *lock);
int omp_testfunc_lock(locktype *lock);
```

- Low-level general-purpose locking library routines (for synchronization).
- `func` expresses the routine's functionality: init, destroy, set, unset. For nested locks these are: init_nest, destroy_nest, set_nest, unset_nest. Similarly, `testfunc` is either test or test_nest.
- Allows greater flexibility than critical or atomic.
- Two types: simple locks, which may be locked if not already, and nestable locks, which may be locked multiple times.
- Simple lock variables have locktype `omp_lock_t`
- Nested lock variables have locktype `omp_nest_lock_t`
- Special care is required if a programmer wishes to synchronize thread actions via locks. If used improperly, many errors are possible, including deadlock.
Locking Functions

General procedure for using locks:

1) Define the lock variables (simple or nested).
2) Initialize the lock with the call `omp_init_lock()`.
3) Set the lock using either `omp_set_lock()` or `omp_test_lock()`. The second checks the lock’s availability before attempting to set it.
4) Unset a lock after the work is completed using `omp_unset_lock()`.
5) Remove the lock association using `omp_destroy_lock()`.

**Note:** OpenMP locks are very similar to using PThreads for SMP programming.
Locking Example

```c
omp_lock_t lok;
int TID;
omp_init_lock(&lok);
#pragma omp parallel shared(lok) \\
private(TID)
{
    TID = omp_get_thread_num();
    while ( !omp_test_lock(&lok) )
        work2();
    work(ID);
    omp_unset_lock(&lok)
}
omp_destroy_lock(&lok);
```

- Allows asynchronous thread execution via explicit locking.
- Create the lock outside the parallel region, and share it among threads.
- Threads will attempt to create the lock:
  - If successful they call `work()` and then release the lock,
  - Otherwise they continue calling `work2()` until they can set the lock.
- The lock is destroyed outside of the parallel region.