OpenMP

Using OpenMP : Portable Shared Memory Parallel Programming

Barbara Chapman, Gabriele Jost, and Ruud van der Pas

Cambridge, MA, USA: MIT Press, 2007.

Overview of OpenMP

- OpenMP API developed to enable shared memory parallel programming
- Uses the fork-join programming model
- Provides means for the user 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 them to perform certain operations exclusively (i.e., without interference by other threads)

Sharing Work among Threads

- If the programmer does not specify how the work in a parallel region is to be shared among the executing threads, they each redundantly execute all of the code
  - This approach does not speed up the program!
- All OpenMP strategies for sharing the work in loops assign one or more disjoint sets of iterations to each thread
  - The programmer may specify the method used to partition the iteration set
  - A loop is suitable for sharing among threads only if its iterations are independent
  - Data dependences prevent a loop from being parallelized

OpenMP Memory Model

- By default, data is shared among the threads and is visible to all of them
- Sometimes we need variables that have thread-specific values
- When each thread has its own copy of a variable, we say that the variable is private
  - When a team of threads executes a parallel loop, each thread needs its own values of the iteration variable
    - This case is so important that the compiler enforces it
  - In other cases, the programmer must determine which variables are shared and which are private
- The benefit of private variables is that they reduce the frequency of updates to shared memory, thus may help avoid hot spots
- The downside is that they increase the memory footprint of the program
- Threads need a place to store their private data at run time
  - For this, each thread uses its own thread stack

Caches in SMP

- Each processor of an SMP has a small amount of private memory, its cache
- OpenMP has rules about when shared data is visible to (or accessible by) all threads
  - Rules state that the values of shared objects must be made available to all threads at synchronization points
  - As a result, 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

Thread Synchronization

- Synchronizing, or coordinating the actions of, threads is sometimes necessary to:
  - ensure proper ordering of their accesses to shared data, and
  - to prevent data corruption
- OpenMP has a set of well understood synchronization features

Performance Considerations

- How much reduction of execution time can be expected?
- Denote by T_1 the execution time of an application on 1 processor
- In an ideal situation, the execution time on P processors should be T_1/P
- If T_P denotes the execution time on P processors, then the ratio $S = T_1/T_P$ is called the parallel speedup
  - It is a measure of success of the parallelization
- Virtually all programs have some regions suitable for parallelization and others that are not
  - Execution time is dominated by the time taken to compute the sequential portion which puts an upper limit on the expected speedup

Amdahl's Law

- This effect, known as Amdahl's law, is formulated as

$$S = 1/(f_{par}/P + (1-f_{par}))$$

where f_{par} is the parallel fraction of the code and P is the number of processors

- In the ideal case, when all the code runs in parallel, f_{par} =1 and the expected speedup is P
- Obstacles to speedup is the overhead introduced by
  - forking and joining threads,
  - thread synchronization, and
  - memory accesses

Writing a First OpenMP Program

- For C/C++, pragmas are provided by the OpenMP API to control parallelism
- In OpenMP these are called directives
- They always start with #pragma omp, then a specific keyword that identifies the directive, followed by zero or more clauses
- Clauses are used to further specify the behaviour or to further control parallel execution

Matrix times Vector

$$a_i = \Sigma_{j=1}^{n} B_{i,j} * c_{j,}\ i = 1, \ldots, m$$

- The row variant of this problem has a high level of parallelism
- It computes a value a_i for each element of vector a by multiplying the corresponding elements of row i and matrix B with vector c
- Since no two dot products compute the same element of the result vector and since the order in which the values of the elements a_i for i = 1,…,m are calculated does not affect correctness, these computations can be carried out independently
  - i.e., can parallelize over the index value I
  - See program source in mxv-omp.cc

Default, Private and Shared

- Data is either shared by threads in a team, in which case all member threads will be able to access the same shared variable, or it is private
- If private, each thread has its own copy of the data object, and hence the variable may have different values for different threads
  - Each thread accesses the same variable m but each will have its own distinct i
  - There are default data-sharing attributes but we recommend to use the default(none) clause
    - Informs the compiler that we take it upon ourselves to specify the data-sharing attributes
    - This forces us to explicitly specify, for each variable, whether it is shared by multiple threads
    - More work for the programmer but
      - Must carefully think about the usage of variables (helps avoid mistakes)
      - For good performance, use private variables as much as possible

Work-Sharing

- The loop following #pragma omp parallel (with loop variable i) should be distributed among threads
  - Different threads execute different iterations of this loop and each loop iteration is performed exactly once
  - If there are more iterations than threads, multiple iterations are assigned to the threads in the team
  - Distributing loop iterations among threads is one of the main work-sharing constructs OpenMP offers
  - By not specifying details of how the work should be distributed, we have left it up to the implementation to decide exactly which thread should carry out which loop iterations
  - OpenMP provides a schedule clause to allow the user to prescribe the way in which the loop iterations should be distributed

OpenMP Language Features

- OpenMP provides directives, library functions, and environment variables to create and control the execution of parallel programs
- First, we present constructs that suffice to write many different programs:
  - Parallel construct
  - Work-sharing constructs
    - Loop construct
    - Sections construct
    - Single construct
  - Data-sharing, no wait, and schedule clauses
- Then introduce features to orchestrate actions of different threads
  - Barrier construct
  - Critical construct
  - Atomic construct
  - Locks

Terminology

- OpenMP directive: in C/C++, a #pragma that specifies OpenMP program behaviour
- Executable directive: an OpenMP directive that is not declarative, i.e., it may be placed in an executable context
  - All directives except the threadprivate directive are executable directives
- Construct: an OpenMP executable directive and the associated statement, loop, or structured block, if any, not including the code in any called routines
  - i.e., the lexical extent of an executable directive

Structured Block

- In C/C++, a structured block is defined as an executable statement, possibly a compound statement, with a single entry at the top and a single exit at the bottom
- In C/C++ the following additional rules apply to a structured block:
  - The point of entry cannot be a call to setjmp()
  - Longjmp() and throw() (C++ only) must not violate the entry/exit criteria
  - Calls to exit() are allowed in a structured block
  - An expression statement, iteration statement, selection statement, or try block is considered to be a structured block if the corresponding compound statement obtained by enclosing it in { and } would be a structured block

Region of Code

- Another important concept is that of a region of code
- An OpenMP region consists of all code encountered during a specific instance of the execution of a given OpenMP construct or library routine
- A region includes any code in called routines, as well as any implicit code introduced by the OpenMP implementation
  - i.e., a region encompasses all the code that is in the dynamic extent of a construct

Parallel Construct

- The parallel construct plays a crucial role in OpenMP: a program without a parallel construct will be executed sequentially

```
#pragma omp parallel [clause[[,] clause]…]
    Structured block
```

- This construct is used to specify the computations that should be executed in parallel
- Although this construct ensures that the computations are performed in parallel, it does not distribute the work of the region among the threads in a team
- In fact, if the programmer does not use the appropriate syntax to specify this action, the work will be replicated
- Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution
  - One way to achieve this is to exploit the thread numbers
  - See program source in par.cc, par2for.cc

Clauses of the Parallel Construct

if(scalar-expression)

num_threads(integer-expression)

private(list)

firstprivate(list)

shared(list)

default(none|shared)

copyin(list)

reduction(operation:list)

Restrictions on the Parallel Construct

- There are several restrictions on the parallel construct and its clauses:
  - It is illegal to branch into or out of a parallel region
  - A program must not depend on any ordering of the 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
  - In C++, a throw inside a parallel region must cause execution to resume within the same parallel region, and it must be caught by the same thread that threw the exception

Active vs Inactive Parallel Region

- A parallel region is active if it is executed by a team of threads consisting of more than one thread
- If it is executed by one thread only, it has been serialized and is considered to be inactive
  - e.g., a parallel region can be conditionally executed, in order to be sure that it contains enough work for it to be worthwhile

Sharing Work among Threads

- OpenMP's work-sharing constructs are the next most important feature of OpenMP because they distribute work among the threads in a team
- C/C++ has 3 work-sharing constructs (clauses omitted)

Functionality	C/C++ Syntax
Distribute iterations over the threads	#pragma omp for
Distribute independent work units	#pragma omp sections
Only one thread executes the code block	#pragma omp single

Loop Construct

- The loop construct causes the iterations of the loop immediately following it to be executed in parallel
- At run time, the loop iterations are distributed across the threads
- The C/C++ syntax:

```
#pragma omp for [clause [[,] clause]…]
    for-loop
```

Loop Construct (cont'd)

- In C/C++ the use of the loop construct is limited to those kinds of loops where the number of iterations can be counted
  - i.e., the loop must have an integer counter variable whose value is incremented (or decremented) by a fixed amount at each iteration until some upper (or lower) bound is reached
  - This restriction excludes loops that process the items in a list

Loop Construct (cont'd)

- If the programmer does not say how to map the iterations to threads, the compiler must decide the strategy
- Another clause, schedule, is the means by which the programmer is able to influence the mapping
- See the program source parfor.cc

Clauses of a Loop Construct

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator:list)

ordered

schedule (kind[, chunk_size])

nowait

The Sections Construct

- The sections construct is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which is executed by one of the threads
- It consists of 2 directives:
  - #pragma omp sections to indicate the start of the construct, and
  - #pragma omp section to mark each distinct section
- Each section must be a structured block of code that is independent of the other sections
- See program source sections.cc

Syntax of Sections Construct

```
#pragma omp sections [clause[[,] clause]…]
{
    [#pragma omp section]
        structured block
    [#pragma omp section]
        structured block
    …
}
```

Sections (cont'd)

- 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 is 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
- Sections are most commonly used to execute function calls in parallel
  - This may lead to a load-balancing problem

Clauses of a Sections Construct

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator:list)

nowait

The Single Construct

- The single construct is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only
- It does not state which thread should execute the code block
  - Should be used when we don't care which thread executes this part of the application
  - The other threads wait at a barrier until the thread executing the single code block has completed
- See program source single.cc

Syntax for a Single Construct

#pragma omp single [clause[[,] clause]…]

structured block

- Clauses supported by the single construct:
  - private(list)
  - firstprivate(list)
  - copyprivate(list)
  - nowait

Combining Parallel Work-Sharing Constructs

- Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises exactly one work-sharing construct
- The semantics are identical to explicitly specifying the parallel construct immediately followed by the work-sharing construct
- A main advantage of these constructs is readability
  - There may also be a performance advantage

Combined Constructs

Full version	Combined construct

#pragma omp parallel
{
  #pragma omp for
  for-loop
}
	#pragma omp parallel for
	  for-loop

#pragma omp parallel
{
  #pragma omp sections
  {
  [#pragma omp section]
  structured block
  [#pragma omp section]
  structured block
  …
  }
}
	#pragma omp parallel sections
	{
	  [#pragma omp section]
	  structured block
	  [#pragma omp section]
	  structured block
	  …
	}

Clauses to Control Parallel and Work-Sharing Constructs

- The clauses are processed before entering the construct they are associated with
- Several clauses can be used with a given directive
  - The order in which they are given has no bearing on their evaluation
  - The evaluation order is arbitrary
    - The programmer should not make any assumptions about evaluation order

Shared Clause

- The shared clause is used to specify which data is shared among the threads executing the region it is associated with
- There is one unique instance of these variables, and each thread can freely read or modify the values
- The syntax for this clause is: shared(list)
- All items in the list are data objects that will be shared among the threads in the team
- An important implication is that multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another is updating
  - Special care must be taken to ensure that neither of these situations occurs and that accesses to shared data are ordered as required by the algorithm
  - OpenMP places the responsibility for doing so on the programmer and provides several constructs that may help
- See program source shared.cc

Private Clause

- There may be other data objects in a parallel region or work-sharing construct for which threads should be given their own copies
- The syntax of the private clause is: private(list)
  - Each variable in the list is replicated so that each thread in the team of threads has exclusive access to a local copy of this variable
  - Changes made to the data by one thread are not visible to other threads
  - By default, OpenMP gives the iteration variable of a parallel loop the private data-sharing attribute
    - In general, it is recommended not to rely on default rules for data-sharing attributes
    - Instead, specify them explicitly
- The value of private data are undefined upon entry to and exit from the specific construct
- The value of any variable with the same name as the private variable in the enclosing region is also undefined after the construct has terminated even if the corresponding value was defined prior to the region
- See program source private.cc

Lastprivate Clause

- The values of data specified in the private clause are undefined after the corresponding region terminates
- OpenMP offers a workaround if such a value is needed
- The lastprivate clause addresses this situation
  - It is supported on the work-sharing loop and sections constructs
- The syntax is: lastprivate(list)
  - It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution
  - What is “last”?
    - The object will have the value from the iteration of the loop that would be last in a sequential execution
    - If lastprivate is used on a sections construct, the object gets assigned the value that is has at the end of the lexically last sections construct
- See program source lastprivate.cc

Firstprivate Clause

- Private data is also undefined on entry to the construct
  - This could be a problem if we need to pre-initialize private variables with values that are available prior to the region in which they are used
- OpenMP provides the firstprivate construct to help in such cases
- Variables that are declared to be firstprivate are private variables, but they are pre-initialized with the value of the variable with the same name before the construct
  - The initialization is carried out by the initial thread prior to the execution of the construct
- See program source firstprivate.cc (run, e.g., with n=2)

Default Clause

- The default clause is used to give variables a default data-sharing attribute
- The syntax in C/C++ is: default(none|shared)
  - default(shared) assigns the shared attribute to all variables referenced in the construct
  - If default(none) is specified, the programmer is forced 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

Nowait Clause

- The nowait clause allows the programmer to fine-tune a program's performance
- There is an implicit barrier at the end of a work-sharing construct
  - The nowait clause overrides that feature of OpenMP
    - i.e., when threads reach the end of the construct, they immediately proceed to perform other work
  - Note that the barrier at the end of a parallel region cannot be suppressed
  - Some care is required when inserting this clause because its incorrect usage can introduce bugs

Schedule Clause

- The schedule clause is supported on the loop construct only
- It is used to control the manner in which loop iterations are distributed over the threads
  - This can have a major impact on the performance of a program
- The syntax is: schedule(kind[,chunk_size])
- The schedule clause specifies how the iterations of the loop are assigned to the threads in the team
  - The granularity of this workload distribution is a chunk, a contiguous, nonempty subset of the iteration space
  - The chunk_size parameter need not be a constant
    - Any loop invariant integer expression with a positive value is allowed
- See program source schedule.cc

Supported Schedule Kinds

Kind	Description
Static	Iterations are divided into chunks of size chunk_size. Chunks are assigned to the threads statically in a round-robin manner, in the order of thread number. When no chunk_size is specified, the iteration space is divided into chunks approximately equal in size. Each thread is assigned at most one chunk.
dynamic	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the chunk_size parameter), then requests another chunk until there are no more chunks to work on. When no chunk_size is specified, it defaults to 1.
guided	Similar to dynamic, but for a chunk_size of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1. For a chunk_size of k (k>1), the size of each chunk is determined the same way, with the restriction that the chunks do not contain fewer then k iterations.
runtime	If this schedule is selected, the decision regarding scheduling kind is made at run time. The schedule and (optional) chunk size are set through the OMP_SCHEDULE environment variable.

Schedule (cont'd)

- The most straightforward schedule is static
- It has the least overhead and is the default on many OpenMP compilers
- Both the dynamic and guided schedules are useful for handling poorly balanced and/or unpredictable workloads
  - The difference between them is that with the guided schedule, the size of the chunk (of iterations) decreases over time
- All workload distribution algorithms support an optional chunk_size parameter
  - The interpretation depends on the schedule chosen
  - It is not always easy to select an appropriate schedule and value for chunk_size up front
- Other than for the static schedule, the allocation is nondeterministic and depends on a number of factors including the system load

OpenMP Synchronization Constructs

- OpenMP has constructs that help organize accesses to shared data by multiple threads
  - Barrier
  - Ordered

Barrier Construct

- A barrier is a point in the execution of a program where threads wait for each other
  - No thread in the team of threads may proceed beyond a barrier until all threads in the team have reached that point
- Many OpenMP constructs imply a barrier
  - Thus, it is often unnecessary for the programmer to explicitly add a barrier
- The syntax is: #pragma omp barrier
- Two restrictions apply to 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
  - In C/C++, the barrier may only be placed in the program at a position where ignoring or deleting it would result in a program with correct syntax
- See program source barrier.cc

Ordered Construct

- Another synchronization construct, the ordered construct, allows one to execute a structured block within a parallel loop in sequential order
- This is sometimes used to enforce ordering on the printing of data computed by different threads
- The syntax is:

```
#pragma omp ordered
    structured block
```

- This ensures that the code within the associated structured block is executed in sequential order
- The code outside this block runs in parallel

Critical Construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously
  - i.e., the associated code is in a critical section
  - An optional name may be given to the CS
    - This name is global and must be unique. otherwise the behaviour of the application is undefined
  - The syntax of the critical construct:

    #pragma omp critical [(name)]

    structured block
  - When a thread encounters a critical construct, it waits until no other thread is executing a CS with the same name
    - There is no risk that multiple threads will execute the code contained in the same CS at the same time
- See program source critical.cc

Reduction Clause

- A critical construct was used to parallelize the summation operation

```
sum = 0;
for( i = 0; i < n; i++ )
        sum += a[i]
```

- OpenMP provides a reduction clause for specifying some forms of recurrence calculations (involving mathematically associative and commutative operators) so that they can be performed in parallel without code modification
- The programmer must identify the operations and the variables that hold the result values
  - The rest of the work can be left to the compiler
- In general, it is recommended to use a reduction clause rather than implementing a reduction operation manually
- The syntax in C/C+: reduction(operator:list)
- The order in which thread-specific values are combined is unspecified
  - Therefore, where floating-point data are concerned, there may be numerical differences between the results of a sequential and parallel run, or even of two parallel runs using the same number of threads
- See program source reduction.cc

Atomic Construct

- The atomic construct can be an efficient alternative to the critical region
- In contrast to other constructs, it is applied only to the (single) assignment statement that immediately follows it
  - This statement must have a certain form in order for the construct to be valid, and thus its range of applicability is strictly limited
- The syntax is:

```
#pragma omp atomic
        statement
```

- The atomic construct enables efficient updating of shared variables by multiple threads on hardware platforms that support atomic operations
  - It protects updates to the memory location corresponding to the left-hand side of the assignment
  - If the hardware supports instructions that read from a memory location, modify the value, and write back to the location all in one action, then atomic instructs the compiler to use such an operation
- See program source atomic.cc

Locks

- OpenMP provides a set of low-level, general-purpose locking runtime library routines, similar in function to the use of semaphores
  - These routines provide greater flexibility for synchronization than does the use of critical or atomic constructs
- The general syntax is:

  void omp_func_lock (omp_lock_t *lock)
- For a specific routine, func expresses its functionality
  - func may assume the values init, destroy, set, unset, test
  - The values for nested locks are init_nest, destroy_nest, set_nest, unset_nest, test_nest

Locks (cont'd)

- The routines operate on special-purpose lock variables, which should be accessed via the locking routines only
- There are two types of locks:
  - Simple locks, which may not be locked if already in a locked state
    - Declared with the special type omp_lock_t in C/C++
  - Nestable locks, which may be locked multiple times by the same thread
    - Declared with the special type omp_nest_lock_t in C/C++
  - In C/C++, lock routines need an argument that is a pointer to a lock variable of the appropriate type

Locks (cont'd)

- The general procedure to use locks is as follows:
  1. Define the (simple or nested) lock variable
  2. Initialize the lock via a call to omp_init_lock
  3. Set the lock using omp_set_lock or omp_test_lock
     - The latter checks whether the lock is actually available before attempting to set it and is useful to achieve asynchronous thread execution
  4. Unset a lock after the work is done via a call to omp_unset_lock
  5. Remove the lock association via a call to omp_destroy_lock

Interaction with Execution Environment

- OpenMP provides several means with which the programmer can interact with the execution environment, either to obtain information from it or to influence the execution of a program
- If a program relies on some property of the environment, e.g., a minimum number of threads will execute a parallel region, then the programmer must test for its satisfaction explicitly
- The OpenMP standard defines internal control variables
  - These are variables controlled by the OpenMP implementation that govern the behaviour of a program at run time in important ways
  - They cannot be accessed or modified directly at the application level
  - However, they can be queried and modified through OpenMP functions and environment variables

Internal Control Variables

- The following internal control variables are defined.
  - nthreads-var: stores the number of threads requested for the execution of future parallel regions
  - dyn-var: controls whether dynamic adjustment of the number of threads to be used for future parallel regions is enabled
  - nest-var: controls whether nested parallelism is enabled for future parallel regions
  - run-sched-var: stores scheduling information to be used for loop regions using the runtime schedule clause
  - def-sched-var: stores implementation-defined default scheduling information for loop regions

How to Access/Modify Internal Control Variables

- Library functions and environment variables can be used to access or modify the values of these variables
- The four environment variables defined by the standard may be set prior to program execution
- Library routines can also be used to give values to control variables
  - They override values set via environment variables
- In order to be able to use them, a C/C++ program must include the omp.h header file
- Once a team of threads is formed to execute a parallel region, the number of threads in it will not change

How to Access/Modify Internal Control Variables

- However, the number of threads used to execute future parallel regions can be specified in several ways:
  - At the command line, set the OMP_NUM_THREADS environment variable
    - The value specified is used to initialize the nthreads-var control variable
    - Its syntax is OMP_NUM_THREADS(integer), a positive integer
    - The command syntax is shell dependent
  - During program execution, the number of threads used to execute a parallel region may be set or modified via the omp_set_num_threads library routine
    - Its syntax is omp_set_num_threads(scalar-integer-expression), where the evaluation of the expression must result in a positive integer
  - Finally, it is possible to use the num_threads clause together with a parallel construct to specify how many threads should be in the team executing that specific parallel region
    - If this is given, it temporarily overrides both of the previous constructs