A BRIEF OVERVIEW OF PARALLEL PROGRAMMING MODELS

Pavan Balaji, Argonne National Laboratory

Table of Contents

Chapter 1

OpenMP

Barbara Chapman, University of Houston

1.1 Introduction

The OpenMP Application Programming Interface (API) is a parallel programming model for shared-memory computer systems intended to provide a straight-forward means of exploiting concurrency inherent in many algorithms. With the insertion of compiler directives, the programmer directs the compiler to parallelize portions of the code at a high level.

Originally designed to target loop-centric algorithms, version 3.0 of the API introduced the ability to define explicit tasks that may be executed concurrently. The OpenMP API is an agreement among hardware and software vendors that make up the OpenMP Architecture Review Board (ARB). The origins of the API are found a set of compiler directives for writing parallel Fortran compiled by the the Parallel Computing Forum (PCF) in the late 1980s. In 1997, the newly formed ARB introduced the first OpenMP specification for a set of directives for use with Fortran and OpenMP compilers soon followed. Bindings for C and C++ have since been defined and the feature set has grown. Version 3.0, the most recent version, was ratified in 2008. The ARB is a nonprofit corporation comprised of members from industry and academia that owns the OpenMP brand and manages the OpenMP specification

The OpenMP does not require the programmer to explicitly decompose data and control flow to produce parallel computation. This fragmented style of programming is characterized by low-level programming to control the details of the concurrency. Rather, OpenMP allows the programmer to take a high-level view of parallelism and leave the details of the concurrency to the compiler. With the insertion of OpenMP directives, the programmer can specify what portions of sequential code should be executed in parallel. To a non-OpenMP compiler, the directives look like comments and are ignored. So by observing a few rules, one application can be both sequential and parallel, a very good quality that is quite useful in development and testing. Another benefit is the ability to incrementally apply OpenMP constructs to create a parallel program from existing sequential codes. Rather than start from scratch, the programmer can insert parallelism into a portion of the code and leave the rest sequential, repeating this process until the desired speedup is realized. This also means that to use OpenMP one need only learn a small set of constructs, not an entirely new language.

For a sample of how to use OpenMP, consider this fragment of C code that multiplies two matrices, a and b, and storing the result in a third matrix, c.

```
#pragma omp parallel private(i,j,k) shared(a,b,c)
{
#pragma omp for schedule(auto)
for (i = 0; i < N; i++)
    for (k = 0; k < K; k++)
        for (j = 0; j < M; j++)
            c[i][j]= c[i][j]+a[i][k]*b[k][j];
```

```
} /* end omp parallel */
```

As with any sequential program, execution starts with a single thread of control. The #pragma omp identifies a line of code to be an OpenMP parallel directive, which specifies a structured block of code that should be treated as a parallel region. In this case the parallel region is enclosed with {}. When the initial thread of control reaches a parallel directive, it creates a team of threads, all of which will then execute the code in the parallel region. The initial thread becomes the teams master thread while the other are the slaves. The entire team will be available to share the work of the parallel region. This parallel construct includes a private clause and a shared clause. These are used to designate how variables are to be treated in the parallel region. In this case the loop iteration variables will be treated local to individual threads while the matrices will be shared. Improperly characterized data can lead to incorrect results and errors, so this must be carefully considered. These and other clauses will be discussed later. This need to designate private and shared data is essentially what sets shared-memory programming from other methods.

The next directive identified by #pragma omp identifies the for construct, which designates that the execution of the immediately following loop is to be executed in parallel. Iterations of the loop will among the threads of the team in the enclosing parallel region according to the specified loop schedule. In this example, the auto schedule will allow the compiler and runtime to decide on the actually scheduling of the loop iterations.

1.1.1 OpenMP - A brief history

A group of vendors, now popularly known as the OpenMP Architecture Review Board (ARB) joined forces during the latter half of the 1990s to provide common means for programming a broad range of SMP architectures defining OpenMP. OpenMP based on the earlier PCF work [8] has become a widely-used portable programming interface facilitating shared memory parallel programming unveiled in 1997, proposed by scientists from industry, academia and research laboratories at a major conference on High Performance Computing, Networking, and Storage. OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (OpenMP ARB) jointly defined by a group of major hardware and software vendors including Intel, IBM, AMD, Convey Computer, Texas Instruments, Cray, NVIDIA, Red Hat, ST Microelectronics, NEC, and more. OpenMP is not a new programming language but is a notation that can be added to a sequential program written in Fortran, C or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed.

The first version, consisting of a set of directives that could be used with Fortran, was introduced to the public in late 1997. OpenMP compilers began to appear shortly thereafter. Since that time, bindings for C and C++ have been introduced, and the set of features has been extended. Compilers are now available for virtually all SMP platforms. The number of vendors involved in maintaining and further developing its features has grown. Today, almost all the major computer manufacturers, major compiler companies, several government laboratories, and groups of researchers belong to the ARB.

1.1.2 Phases of Development

The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. In October, the following year, the ARB released the C/C++ standard. Fortran and C/C++ specifications version 2.0 was released in 2000 and 20002 respectively. In 2005, a combined specification for C/C++ and Fortran version 2.5 was released. Originially designed to target loop-centric algorithms, the specification released in 2008, version 3.0, introduced the ability to define explicit tasks that may be executed concurrently. In July 2011, 3.1 was released that provided support for min/max reductions in C/C++. Support for accelerators, atomics, error handling, thread affinity, task extensions, user-defined reduction, SIMD support was included in version 4.0 that was released in July 2013.

One of the biggest advantages of OpenMP is that the ARB continues to work to ensure that OpenMP remains relevant as computer technology evolves. OpenMP is under cautious, but active, development; and features continue to be proposed for inclusion into the application programming interface. Applications live vastly longer than computer architectures and hardware technologies; and, in general, application developers are careful to use programming languages that they believe will be supported for many years to come. The same is true for parallel programming interfaces.

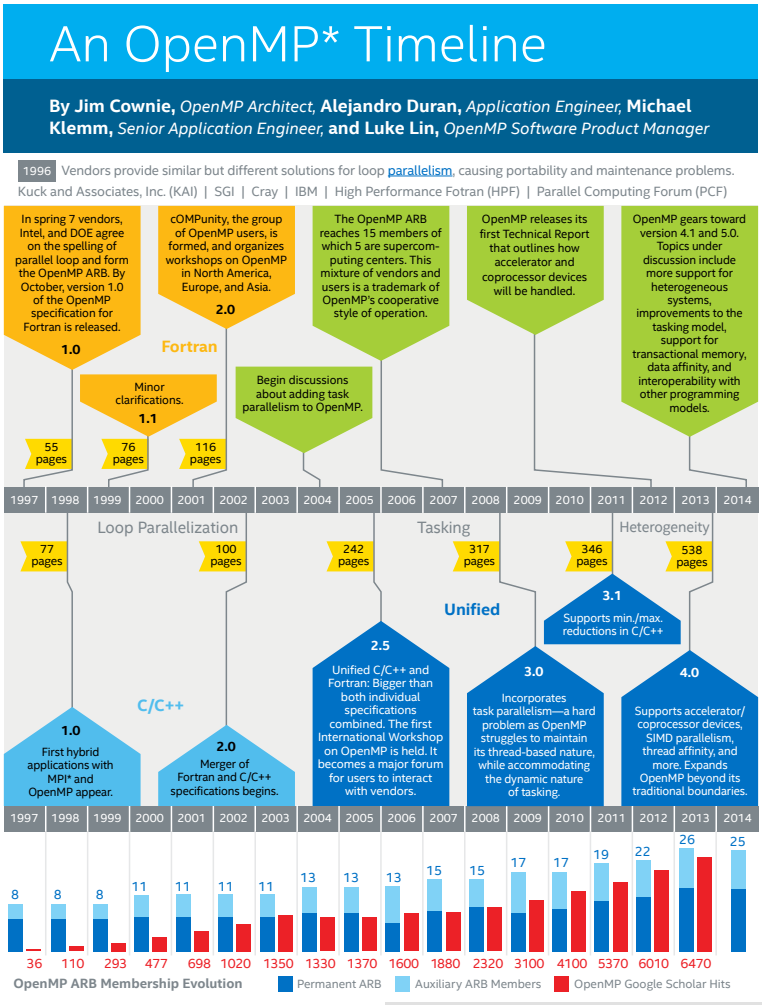


Figure 1.1: An OpenMP timeline

Figure 1.1 gives a nice overview of the timeline of OpenMP created by Jim Cownie, OpenMP Architect, Alejandro Duran, Application Engineer, Michael Klemm, Senior Application Engineer, and Luke Lin, OpenMP Software Product Manager.

Although the OpenMP specification can be downloaded from the web, the specification itself may not provide suitable starting point and hints to write real programs. The goal of this chapter is to discuss basic, mostly used and recently developed features of OpenMP and discuss some critical factors that will influence a program's performance that are not mentioned in the specification.

1.2 Overview

The OpenMP API relies on directives, library routines, and environment variables for expressing the parallelism desired in a given program. Some of these were mentioned previously. A directive and its accompanying structured block form a construct. Directives may have various clauses associated to provide further information to the OpenMP implementation. As seen above, directives for C and C++ begin with #pragma omp. Directives in Fortran can begin with !$omp, but other options are available.

The fundamental parallel directive defines a parallel region, marking the structured block of code therein should be executed by multiple of threads. Without a parallel construct, the code will be executed sequentially. Each parallel construct designates whether data should be treated as private or shared. Data not explicitly designated with a clause

is shared by default. Since a parallel region is form with a structured block, it must have only one entry point and one exit point. A program is non-conforming if it branches into or out of a parallel region.

Various constructs may be used with the parallel construct to designate worksharing, tasks, synchronization, and more. The worksharing constructs include the loop, sections, single, and workshare constructs. Each worksharing construct must bind to a active parallel region before its effects will be realized. Worksharing directives encountered by a single thread are ignored and the accompanying code will be executed sequentially, i.e. outside of a parallel region or inside a parallel region with a team of one thread. Clauses used with a parallel directive may be used to designate conditional use, the number of threads, data sharing properties, and reductions, some of which will be discussed below. Worksharing constructs end with an implicit barrier causing all threads to wait for the construct to complete. A worksharing construct that appears in a function or subroutine that is invoked from within a parallel region is called an orphan directive. Orphan directives may also be called from outside a parallel region allowing sequential or parallel execution of the function.

Worksharing constructs are used to define how the work in a block of code should be sdistributed across the executing threads. The loop construct (in C/C++ for, in Fortran do) is used to execute iterations of loops concurrently. The sections construct allows multiple blocks of code to be executed concurrently, each by a single thread. The single construct associates with the immediately following structured block for execution by a single, arbitrary thread. The workshare construct is supported only for directing the parallel execution of Fortran programs written using Fortran 90 array syntax.

Synchronization of threads in a team is achieved through another set of constructs. A barrier is a point of execution where threads must wait for all other threads is the current team before proceeding. While many constructs have implicit barriers, an explicit barrier is accomplished with the barrier directive. Barriers are used to avoid data race conditions. Access to shared data can be isolated to a critical region and protected using the critical directive or single assignment statements with the atomic directive. Explicit locks are also available for more flexible access of shared data.

Data in OpenMP programs is shared by the threads in the team by default. Any modification of shared data is guaranteed to be available to other threads after the next barrier or other synchronization point in the program. However, this is not always appropriate and some data will need to be local, or private, to each thread. A variable designated as private is replicated among the threads as a local copy. If the private variable needs to be initialized by the value of the corresponding variable immediately prior to the construct, the variable should be designated firstprivate. If the final value of a private variable is needed after the construct completes, the variable can be designated as lastprivate, whereby the final value of the variable inside the construct will be saved for the corresponding variable. Variables may also be designated as reduction variables with an associated operator. These variables will be used privately by each thread to perform it share of the work and then combined according to the operator as each thread completes its chunk of work.

OpenMP also specifies how a programmer may interact with the runtime environment. The API defines a set of internal control variables (ICVs) for controlling the execution at runtime. These include controlling the the number of threads in a thread team, enabling nested parallelism, and specifying the scheduling of iterations in loop constructs. Theses ICVs may be accessed by setting environment variables or using runtime library routines. In some cases, the ICV may be set with a clause in the proper directive.

1.2.1 Terminology

Some terms are used quite often in the OpenMP standard that we will also be using in this chapter. The following definitions are cited verbatim from Section 1.2.2 of the OpenMP 4.0 standard:

- OpenMP directive: In C/C++, a *#pragma* and in Fortran, a comment, that specifies program behavior
- Executable directive: An OpenMP directive that is not declarative; that is, it may be placed in an executable context.
- Construct: An OpenMP executable directive (and, for Fortran, the paired end directive, if any) and the associated statement, loop, or structured block, if any, not including the code in any called routines, that is, the lexical extent of an executable directive.

- Team: A set of one or more threads participating in the execution of a *parallel* region. For *an active parallel region*, the team comprises the master thread and at least one additional thread. For *an inactive parallel region*, the team comprises only the master thread.

OpenMP requires well-structured programs, and, as can be seen from the above, constructs are associated with statements, loops, or structured blocks. In C/C++ a structured block is defined to be an executable statement, possibly a compound statement, with a single entry at the top and a single exit at the bottom. In Fortran, it is defined as a block of executable statements with a single entry at the top and a single exit at the bottom. For both languages, the point of entry cannot be a labeled statement, and the point of exit cannot be a branch of any type. In C/C++ the following additional rules apply to a structured block:

- The point of entry cannot be a call to *setjump()*
- *longjump()* 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.

In Fortran the following applies:

- STOP statements are allowed in a structured block.

Another important concept in OpenMP is that of a region of code. This is defined as follows by the standard: 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. In other words, a region encompasses all the code that is in the dynamic extent of a construct.

Most OpenMP directives are clearly associated with a region of code, usually the dynamic extent of the structured block or loop nest immediately following it. A few (barrier and flush) do not apply to any code. Some features affect the behavior or use of threads. For these, the notion of a binding thread set is introduced. In particular, some of the runtime library routines have an effect on the thread that invokes them (or return information pertinent to that thread only), whereas others are relevant to a team of threads or to all threads that execute the program. We will discuss binding issues only in those few places where it is important or not immediately clear what the binding of a feature is.

1.2.2 Execution Model

OpenMP uses the fork-join model of parallel execution. A major task of the runtime library is the mechanism to create and manage threads in a team.

An OpenMP program begins with a single sequential thread of execution, which is termed as *master* thread. The *master* thread executes in a serial region until the first *paralle* construct is encountered. When this construct is encountered, theh *master* thread creates a team of *worker* threads. Each thread in the team executes the statements in the dynamic extent of a parallel region. This is not applicable for the work-sharing constructs. These constructs require they be enountered by all threads in the team in the same order. The statements within the associated structured block will be executed by one or more threads. At the end of the parallel region, all the *worker* threads will wait for each other until all the threads have finished execution, i.e. all threads in the team will synchronize at the implicit barrier, after which the *master* thread will continue execution of the code. Any number of parallel constructs can be specified in a single program. Therefore, one of the important roles of the OpenMP runtime library is to create and manage the underlying threads.

During the execution process, if a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. This step is guaranteed to be complete, by the other threads, at the next sequence point only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after the first modifying thread, and then (or concurrently) the other threads, encounter a *flush* directive that specifies the object (either implicitly or explicitly). There may be cases where the programmer needs to provide additional explicity *flush* directives.

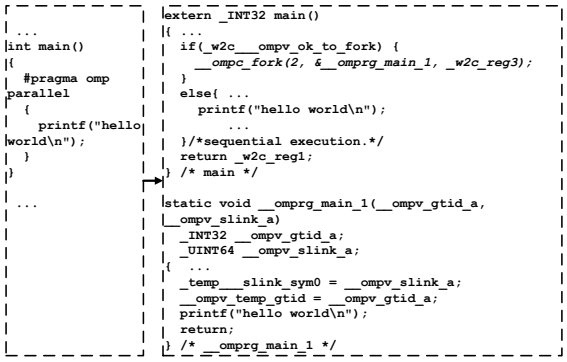


Figure 1.2: Example of OpenMP translation: transforming the `parallel` construct to corresponding runtime library function calls.

An OpenMP compiler will usually translate an OpenMP directive into corresponding runtime library function calls. One major approach is called outlining; an independent function will be created by the compiler that encapsulates the task within that directive region, this is adopted by most of the open source compilers such as OpenUH [6] and Omni [7]. For instance, as shown in Figure 1.2, the `parallel` will be translated into the runtime library function call, say *_ompc_fork()*, which will pass the parameters like how many threads are created, what are the assigned tasks and the pointer that maps the tasks to the thread. The function call consists of three parameters: number of threads requested, microtasks, a pointer that points the microtasks to the threads. The traditional OpenMP runtime implementations for HPC usually utilize POSIX threads [5] or Solaris threads to create and manage the OpenMP threads.

1.2.3 Memory Model

OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them. Sometimes, however, one needs variables that have thread-specific values. When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is private. For example, when a team of threads executes a parallel loop, each thread needs its own value 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. Data can be declared to be shared or private with respect to a parallel region or work-sharing construct.

The use of private variables can be beneficial in several ways. They can reduce the frequency of updates to shared memory. Thus, they may help avoid hot spots, or competition for access to certain memory locations, which can be expensive if large numbers of threads are involved. They can also reduce the likelihood of remote data accesses on cc-NUMA platforms and may remove the need for some synchronizations. The downside is that they will increase the programs memory footprint. Threads need a place to store their private data at run time. For this, each thread has its own special region in memory known as the thread stack. The application developer may safely ignore this detail, with one exception: most compilers give the thread stack a default size. But sometimes the amount of data that needs to be saved there can grow to be quite large, as the compiler will use it to store other information, too. Then the default size may not be large enough. Fortunately, the application developer is usually provided with a means to increase the size of the thread stack (for details, the manual should be consulted). Dynamically declared data and persistent data objects require their own storage area.

OpenMP also has a feature called flush, to synchronize memory. A flush operation makes sure that the thread calling it has the same values for shared data objects as does main memory. Hence, new values of any shared objects updated by that thread are written back to shared memory, and the thread gets any new values produced by other threads for the shared data it reads. 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.

1.3 OpenMP Features

1.3.1 Parallel Regions

As with any sequential program, execution starts with a single thread of control. The `#pragma omp` defines a line of code to be an OpenMP parallel directive, which specifies a structured block of code that should be treated as a parallel region. In this case the parallel region is enclosed within {}.

```
#pragma omp parallel [clause[[,]clause] ...]
structured-block
```

When the initial thread of control reaches a parallel directive, it creates a team of threads to execute the associated parallel region, which is the code dynamically contained within the parallel construct. The thread encountering the `parallel` construct becomes the *master* of the new team. Each thread in the team is assigned a unique thread number (also referred to as the "thread id"). Each thread is allowed to follow a different path of execution. But although this construct ensures that 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. A list of clauses may be used along with the parallel construct, they are: *private, shared, firstprivate, copyin, reduction, num_threads, if, default* clauses. Clauses such as *private, shared, firstprivate* are used to designate how variables are to be treated in the parallel region.

Since a parallel region is formed with a structured block, it must have only one entry point and one exit point. A program is non-conforming if it branches into or out of a parallel region.

Synchronization of threads in a team is achieved through another set of constructs. At the end of a parallel region, there is an implied barrier that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel region. A barrier is a point of execution where threads must wait for all other threads is the current team before proceeding. While many constructs have implicit barriers, an explicit barrier is accomplished with the `barrier` directive (discussed in more detail in Section 1.3.5.

1.3.2 Nested Parallelism

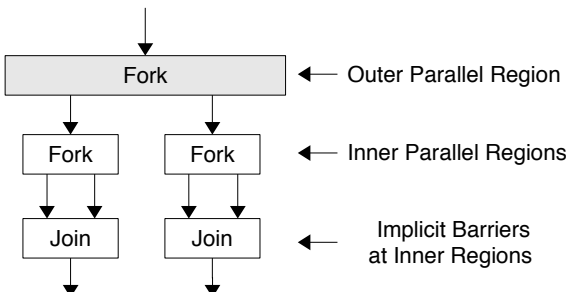
This is an advanced OpenMP construct considered to be special-purpose and their usage depends on the application used. If a thread in a team executing a parallel region encounters another `parallel` construct, it creates a new team and becomes the master of that new team. The new team will consist of multiple threads. Only the master thread of the inner team continues execution after the barrier. There is an implicit barrier synchronization at the end of an inner parallel region. This is generally referred to in OpenMP as nested parallelism. This is illustrated in Figure 1.3. In contrast to the other features of the API, an implementation is free to not provide nested parallelism. In this case, `parallel` constructs that are nested within other `parallel` constructs will be ignored and the corresponding parallel region serialized (executed by a single thread only): it is thus inactive. Increasingly, OpenMP implementations support this feature. We note that frequent starting and stopping of parallel regions may introduce a non-trivial performance penalty. Nested parallelism offers a way to exploit multiple levels of parallelism in order to increase the parallel scalability.

The default data-sharing attributes for nested regions are as follows:

- If a variable is shared in the outer region, it is shared by all threads in the teams of the inner regions.
- If a variable is private in the outer region and accessible by a thread that becomes the master thread of the team executing the inner parallel region, then this variable is by default shared by all threads executing the inner region.

1.3.3 Thread Affinity

Depending on the topology of the machine, thread affinity can have a dramatic effect on the execution speed of the program. Thread affinity can be used to get better locality, lesser false sharing and more bandwidth. There



have been a number of varying proposals from different vendors introducing their own extensions to support threa d affinity. These varying extensions were standardized along with the addition of more capabilities in the 4.0 release of the specification.

Thread affinity requires to determine the number of threads to be utilized and secondly how to bind these threads to the specific processor cores. It should not be the task of the user to think about explicit mapping of threads to cores, since this step is too low-level and tedious for most of the OpenMP programmers. Instead the user will specify an affinity policy via the *affinity* clause that can be used on a parallel region. In order to do so concepts of *place* and *place-list* are introduced. A *place* is a like a set of cores and a *place-list* is an ordered list of places. When creating a team for a parallel region, the *proc_bind* clause specifies a policy for assigning OpenMP threads to places within the current place partition, that is, the place listed in the *place-partition-var* ICV for the implicit task of the encountering thread. Once a thread is assigned to a *place*, the OpenMP implementation should not move to another *place*. There is more to it. There are three affinity policies that helps exploit *place-list*:

- The *master* thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread. The place partition is not changed by this policy, and each implicit task inherits the place-partition-var ICV of the parent implicit task.
- The *close* thread affinity policy instructs the execution environment to assign the threads to places closer tothe place of the parent thread. The master thread executes on the parents place and the remaining threads in the team execute on places from the place list consecutive from the parents position in the list, with wrap around with respect to the place partition of the parent threads implicit task. The place partition is not changed by this policy, and each implicit task inherits the place-partition-var ICV of the parent implicit task.
- The *spread* thread affinity policy is to create a sparse distribution of a team of threads *T*, among *P* places of the parent's place partition. In other words, OpenMP threads are spreaded evenly among the places. The place-list is partitioned so that subsequent threads (i.e. nested OpenMP) will only be allocated within the partition. This policy would provide the most dedicated hardware resources to an OpenMP program.

When creating a team for a parallel region, the *proc_bind* clause specifies a policy for assigning OpenMP threads to places within the current place partition, that is, the place listed in the *place-partition-var* ICV for the implicit task of the encountering thread. Once a thread is assigned to a place, the OpenMP implementation should not move to another place.

1.3.4 Work Sharing

Work can be divided among multiple threads, different threads can operate on different portions of a shared data structure or different threads can even work on totally different tasks. Distribution of work among the threads is called work-sharing, a general term in OpenMP. A work-sharing region must bind to an active `parallel` region, otherwise the work-sharing region is simply ignored.

1.3.4.1 Loop Parallelism

Since a significant amount of work occurs in loops, it is important to exploit any parallelism present. Any work that can be executed without depending on the outcome of other work, can be executed concurrently. This kind of concurrency often occurs in the iterations of the loops. As long as an each iteration does not depend in some way on the outcome of a previous iteration, iterations can be executed concurrently. Existing loop-carried dependences can often be removed with some reorganization to reveal concurrency.

The most commonly used worksharing construct is the loop construct, which is marked with:

```
#pragma omp for
```

where iterations of a loop can be distributed among multiple threads.

The number of iterations in the loop must be countable with an integer and use a fixed increment. This means only for loops in C/C++ and do loops in Fortran. Some rewriting of the loop may be necessary to expose parallelism. Clauses included can be used to prescribe the loop constructs data environment and loop scheduling as well as others. Loop scheduling refers to how loop iterations are assigned to threads in the team. These are easily specified using the clause:

```
schedule(kind[, chunk_size])
```

The *kind* of clause can be `static, dynamic, guided, auto`, or `runtime`. If no loop schedule is specified, then by default 'static' schedule will be used as shown in 1.1.

Listing 1.1: Static scheduling

```
#pragma omp for schedule (static)
      for (int n=0; n<10; ++n)
      printf("\%d", n);
```

Upon entering the loop, each thread independently decides which chunk of the loop to process. Each chunk will be assigned to a thread in a round-robin fashion. The size of a chunk is set by the chunk_size value. If the chunk size is not specified, all chunks will be approximately the same size and at most one chunk will be assigned to each thread in the team. For varying workloads, the 'dynamic' and 'guided' schedules may be more appropriate. 1.2 shows a sample code for dynamic scheduling.

Listing 1.2: Dynamic scheduling

```
#pragma omp for schedule (dynamic, 5)
      for (int n=0; n<10; ++n)
      printf("\%d", n);
```

With the 'dynamic' schedule, each thread will continually grab a chunk iterations until all chunks have been executed, using a chunk size of one if none is specified. This is a low value, almost guaranteeing overhead by threads going back for more work. In the code above, a chunk size of 5 is specified, each thread asks for an iteration number, executes 5 iterations of the loop, then asks for another, and so on. The last chunk may be smaller than 5, though. Similar to this is the 'guided' schedule except that the chunk size decreases as threads subsequent chunks. Then chunk size is set to be a proportion of the remaining iterations.

The 'auto' schedule allows the implementation to decide the schedule, which may be any possible distribution of iterations amongst the thread team. The runtime schedule defers the actual scheduling of threads until execution, at which time the schedule and chunk size are read from environment variables via a library routine.

There is no one scheduling technique that works best. The choice of scheduling depends heavily on the code. There are a few restrictions that is applied to the work-sharing directives:

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

1.3.4.2 Section

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 will be executed by one of the threads. Each *section* must be a structured block of code that is independent of the 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. Note that the assignment of code blocks to threads is implementation-dependent.

Although the sections construct is a general mechanism that can be used to get threads to perform different tasks independently, its most common use is probably to execute function or subroutine calls in parallel. Listing **??** gives an example of such a code. The immediate observation is that this limits the parallelism to two threads. If two or more threads are available, function calls *funcA* and *funcB* are executed in parallel. If only one thread is available, both calls to *funcA* and *funcB* are executed, but in sequential order. Note that one cannot make any assumption on the specific order in which section blocks are executed. Even if these calls are executed sequentially, for example, because the directive is not in an active parallel region, *funcB* may be called before *funcA*.

Listing 1.3: Example of parallel sections

```
#pragma omp parallel
{
#pragma omp sections
      {
#pragma omp sections
            (void} funcA();
#pragma omp sections
            (void} funcB();
      } /*-- End of sections block --*/
} /*-- End of parallel region --*/
```

1.3.4.3 Single

The *single* construct specifies that the encapsulated code can only be executed by a single thread. Therefore, only the thread that when encounters the single construct will execute the code within that region. It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another. It can also differ for different *single* constructs within one application.

1.3.4.4 Workshare

This construct is supported in Fortran only, where it serves to enable the parallel execution of code written using Fortran 90 array syntax. The statements in this construct are divided into units of work. These units are then executed in parallel in a manner that respects the semantics of Fortran array operations. The definition of unit of work depends on the construct. For example, if the workshare directive is applied to an array assignment statement, the assignment of each element is a unit of work. We refer the interested reader to the OpenMP standard for additional definitions of this term.

1.3.5 Synchronization

1.3.6 Task Parallelism

More flexibility is often needed to exploit parallelism in code, especially where the amount of parallelism is unknown, as with recursive algorithms or processing pointerbased structures. When the task directive is encountered by a thread, the associated block of code will be made ready for execution at sometime in the future. There is no guarantee when the task will be executed. When the task is given to a thread for execution, it will be executed sequentially. The implementation is allowed to suspend the execution and resume at a later time. By default the thread that begins a task must complete the task in its entirety. Any suspension in the execution of the task must be resumed on the thread on which it began. This is called a tied task. A task designated untied may resume suspended execution on any thread. There is no guarantee of the ordering of task execution or completion, on that tasks will complete with the use of task synchronization constructs. Tasks may be suspended or resumed when a thread reaches a task scheduling point. Task scheduling points occur at the point immediately following explicit task creation, the end of the task construct, any barriers, and in the taskwait region. The taskwait directive causes the task to wait on all child tasks it has generated.

The OpenMP standard further extended the capability of the programming model by allowing programmers to specify 'explicit' tasks in their applications. An explicit task is defined as a specific instance of executable code and its data environment, generated whenever a thread encounters a task construct. An explicit task may be executed by any

thread in the current thread team, in parallel with other tasks, and the execution can be immediate or deferred until later [4, 3]. Version 3.0 also introduced the *task* directive to widen the applicability of the OpenMP programming model beyond loop-level and functional parallelism and include support for expressing unstructured parallelism and for defining dynamically generated units of work. The C/C++ syntax of a task construct (as specified in [1]) is as follows:

```
#pragma omp task [clause[[,]clause] ...]
structured-block
```

The execution of a task gives rise to task region and may be done by any thread in the current parallel team. A task construct can be nested inside any other OpenMP construct including worksharing constructs, nested inside another task region (where the task region of the inner task is not a part of the task region of the outer task) and can also be present outside any explicit parallel construct. Completion of a task can be guaranteed using task synchronization constructs.

1.3.6.1 Why are OpenMP tasks useful?

Explicit OpenMP tasks have the potential to parallelize irregular problems commonly observed in codes containing unbounded loops, recursive parallelism and list/tree traversal. For example, consider the code in Listing 1.4 that illustrates the traversal of a linked list using worksharing constructs. Parallelizing this code without the use of explicit tasks involves calculating the number of nodes in the list followed by the transformation at runtime of the linked list into an array, incurring overhead of the additional while loop and array construction. Finally with the use of a worksharing construct we distribute the work in parallel across the for-loop iterations.

Listings 1.4 and 1.5 describe code without and with tasks respectively. In the version of 1.5 we see a single thread creating a new task (each node of the list) when it encounters the task construct. Each task is assigned to a thread that will execute it. All tasks are bound to complete at the barrier at the end of the single construct. Tasks in this example, are dynamically created and scheduled asynchronously allowing scope for additional concurrency as well as improving the overall program structure. It is to be remembered that a thread that creates a task may or may not execute it.

Listing 1.4: Traversal of a linked list using OpenMP worksharing constructs

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

p = head;
for(i=0; i<count; i++)
{
   parr[i] = p;
   p = p->next;
}
#pragma omp parallel
{
   #pragma omp for schedule(static,1)
   for(i=0; i<count; i++)
   processwork(parr[i]);
}
```

Listing 1.5: Traversal of a linked list using OpenMP tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
   p=head;
   while (p) {
  #pragma omp task firstprivate(p)
   processwork(p);
   p = p->next;
     }
  }
}
```

1.3.6.2 Data Scoping within tasks

Any task may contain the following optional clauses: *untied*, *shared* (list), *firstprivate* (list), *private* (list), *default* (list), *if* (expression), *final* (expression), *final* (expression).

Threads are allowed to suspend the current task region (at a task scheduling point) to execute a different task. A task can either be *tied* or *untied*. By default, an explicit task is a *tied* task, i.e. if the task were to be suspended, it must resume execution on the same thread. On the other hand, an *untied* task can resume execution on any thread in the team. A tied task can only be suspended at a scheduling point.

The task directive accepts the *private*, *firstprivate*, *shared*, and *default* clauses to influence the data environment. The default for tasks is usually *firstprivate*, because the task may not be executed until later (and variables may be out of scope). However if a variable is determined to be shared in all enclosing constructs up to and including the innermost parallel construct, it can be considered *shared*, as seen in Listing 1.6 where the scope of A is *shared*, B is *firstprivate* and C is *private*.

Listing 1.6: Scope of data in OpenMP tasks

```
#pragma omp parallel shared(A) private(B)
{
  .........
  #pragma omp task
  {
     int C;
     compute (A,B,C);
  }
}
```

On a task construct, when an *if* clause argument is false, the new task is executed immediately by the encountering thread and the current task region is suspended. The suspended task region does not resume until the new task finishes its execution. It could be a user directed optimization if the cost of the deferring the task is more compared to the cost of executing the task code. As seen in Listing 1.7 [1], the *if* clause does not affect descendant tasks.

Listing 1.7: Use of *if* and *final* clauses

```
{
  int i;
  #pragma omp task if(0) // This task is undeferred
  {
    #pragma omp task  // This task is a regular task
    for (i = 0; i < 3; i++) {
       #pragma omp task // This task is a regular task
        bar();
    }
  }
  #pragma omp task final(1) // This task is a regular task
  {
    #pragma omp task  // This task is included
    for (i = 0; i < 3; i++) {
       #pragma omp task // This task is also included
       bar();
    }
  }
}
```

If a task construct contains the *final* clause and the final clause expression evaluates to true, the generated task will be a final task. All task constructs encountered during execution of a final task will generate final and included tasks [1] as seen in Listing 1.7.

[1] An included task is undeferred and executed immediately by the encountering thread

1.3.6.3 Task Synchronization

All explicit tasks generated within a parallel region, in the code preceding an explicit or implicit barrier, are guaranteed to be complete on exit from that barrier region [4]. Tasks are guaranteed to be complete at a thread barrier:

- `#pragma omp barrier`: All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
- `#pragma omp taskwait`: Suspends execution of the current task until all children tasks of the current task, generated since the beginning of the current task, are complete. This only applies to child tasks of the current task and not descendant tasks.

1.3.6.4 Task Scheduling

When a thread encounters a task scheduling point, it is allowed to suspend the current task and perform a task switch, thus enabling the beginning or resuming execution of a different task (tied/untied) bound to the current team. It also permits a task to return to the original task and resume execution. Instances of common task scheduling points include [2]: a) the point immediately following the generation of an explicit task, b) after the point of completion of a task region, c) in a taskwait region, d) in an implicit and explicit barrier region, and e) at the end of a taskgroup region.

1.3.6.5 Extended task model

Task Dependencies:
The OpenMP spec 4.0 further extends the task model with the concept of task dependencies. The optional *depend* clause on a task construct, enforces additional constraints on the scheduling of tasks by establishing dependencies only among sibling tasks. It allows the specification of task-level granularity for synchronization of asynchronous tasks sharing the same parent. This clause consists of a dependence-type with one or more list variables (where the list items may also include array sections), where the dependence-type can be either *in*, *out* and *inout*. An *in* clause denotes that the generated task shall be dependent on all previously generated sibling tasks that reference at least one of the list items in an *out* or *inout* clause. An *out* and *inout* clause denotes that the generated task will be a dependent on all previously generated sibling tasks that reference at least one of the list items in an *in*, *out*, or *inout* dependence-type list.

As seen in Listing 1.8, a specific ordering of task execution is enforced wherein Task 1 has to be completed before Task 2 and Task 3 can be executed. Tasks 2 and 3 can be executed in parallel in any order.

Listing 1.8: Specification of task dependencies

```
void processng_in_parallel) {
  #pragma omp parallel
  #pragma omp single
  {
       int x = 1;

       for (int i = 0; i < N; i++) {
       #pragma omp task shared(x) depend(out: x) // Task 1 created
         preprocess_some_data();

       #pragma omp task shared(x) depend(in: x) // Task 2 created
         do_something_with_data();

       #pragma omp task shared(x) depend(in: x) // Task 3 created
         do_something_independent_with_data();
       }
   }
}
```

Taskgroup construct: This construct specifies a wait on the completion of child tasks of the current task and their descendant tasks. Since the end of the *taskgroup* region signifies a task scheduling point, the current task encountering this construct is suspended until all child tasks that it generated in the taskgroup region and all of their descendant tasks have completed execution.

1.3.7 Support for Accelerators

OpenMP 4.0 introduced significant changes into the OpenMP programming model, mostly pertaining to support for accelerators, generically referred to as "target" devices. Recall that an OpenMP program beings as a single thread of execution, executing as a task within an implicit, single-threaded parallel region which encompasses the whole program. The accelerator model within OpenMP makes a distinction between a *host* device, from which execution begins, and one or more *target* devices upon which work may be offloaded. Parallel regions and task regions may be defined to exploit task parallelism and data parallelism within the host device. New constructs are now available for defining and executing *target* regions. Target regions are similar to task regions in that they define a region of code with an accompanying data environment. However, while task regions are to be executed by a thread within the current thread team on the host device, target regions are intended to execute on some target device by a thread (or threads) which are distinct from the threads in the current thread team. The task on the host device which encountered the target construct will wait until the target region has completed execution on the target device. If the system can not execute the target region on a separate target device, than it may instead execute on the host device.

There are two issues which are performance critical for accelerator programming and are addressed in OpenMP 4.0 – explicit management of data movement between the host and its target devices, and work distribution and vectorization for handling data parallelism in the target region.

1.3.7.1 Data Movement

The accelerator model in OpenMP consists of a host device and one or more target devices. The model treats each of these devices as having their own memory, such that code that executes on any such device will have its own data environment which resides within the device's memory. Since target regions are launched onto target devices from the host device, some mechanism is needed to efficiently transfer data between the device memories and/or allocate data in the target device's memory. This can be done through the definition of the device *data environment*, either via the **target data** or the **target** constructs.

The device data environment defines data which will reside in the device memory for the extent of associated region, and also mappings between this data and corresponding data objects in the host memory. A **map** clause may be used to specify how data is initialized within the environment. It may be allocated in device memory with uninitialized values using the *alloc* map type. Or it may be allocated and initialized with values from the host memory, using the *to* map type. With the *tofrom* map type, the data will be allocated and initialized as with the *to* map type, and additionally, the values will be copied back to the host memory at the end of the region. Finally, the *from* map type may be used to allocate data that is uninitialized, but then to copy back its values to host memory at the end of the region.

1.3.7.2 Thread Teams and Work Distribution

The teams construct creates a league of thread teams where the master thread of each team executes the region. Each of these master threads is an initial thread, and executes sequentially, as if enclosed in an implicit task region that is defined by an implicit parallel region that surrounds the entire teams region.

1.3.7.3 SIMD Parallelism

1.3.7.4 Multi-level Parallelism

1.3.8 Error Handling

Codes with considerable execution times are sensitive to internal failures. Errors could be from the code that the compilers insert to transform a serial code with user directives. Use of OpenMP intrinsics can be another source of error. It is important to identify what is the kind of error; errors can be manyfold (such as memory exhaustion, invalid argument, . . .) . Once the error is identified, it is important to classify errors based on the predicted impact they will have upon application execution. Once the error has been identified and classified, some action needs to be executed,

the term used is *error handling*. OpenMP 4.0 includes error handling capabilities to improve the resiliency and stability of OpenMP applications in the presence of system-level, runtime-level, and user-defined errors. The API includes features to abort parallel OpenMP execution, based on conditional cancellation and user-defined cancellation points. The *cancel* construct activates cancellation of the innermost enclosing region of the type specified. At the *cancellation point* construct, a user-defined cancellation point, implicit and explicit tasks check if cancellation of the innermost enclosing region of the type specified has been requested, if so, the tasks abort the current region and jump to the of it. Cancellation can be requested via the *cancel* construct, which is able to cancel either the whole parallel region, the innermost sections worksharing construct, the innermost for worksharing construct, or the innermost taskgroup, with innermost always defined regarding to the thread team encountering the cancel construct. A thread or task requesting cancellation does not lead to immediate abort of all other threads or tasks in the respective region, instead will only abort execution once a cancellation points has been reached. This construct may not be used in place of the statement following an if, while, do, switch, or label. The user is responsible for releasing locks, and similar data structures that might cause a deadlock when a *cancel* construct is encountered and blocked threads cannot be cancelled. A point to note is that a programmer is required to set the environment variable explicitly, OMP_CANCELLATION=True in order to use the construct.

1.4 Best Programming Practices

While OpenMP makes parallel programs easy to write, some extra work is usually necessary to allow complex programs to scale to high numbers of processors.

- **Avoid unnecessary overhead**: Inherent in parallel programming is the presence of overhead, or work that would not otherwise be done in a sequential execution of the code. Each construct used will require some overhead to set up the parallel execution environment, so limiting their use to the presence of sufficient parallel work is advised. In addition to avoiding unnecessary overhead, the use of the memory hierarchy needs some careful attention.

- **Loop scheduling**: OpenMP does not specify how iterations are assigned to threads in the absence of a schedule clause. Typically, if we have 'N' loop iterations and 'M' threads, where N is much larger than M, it is upto the OpenMP runtime to figure out how to assign iterations to threads. Program characteristics have a major influence on loop scheduling. Work can be either distributed uniformly, statically (not necessarily uniformly) and dynamically. Choosing the right scheduling technique and the chunksize can help improve performance, reduce threading overhead and address load imbalances. The argument *chunksize* can mean different for different schedules. Static scheduling is best used when the workload is predictable and each iteration performs similar work. In the cases where variable amount of work (or some cores are faster than the others) goes to each iteration, either dynamic or guided is preferred. Dynamic scheduling is typically applicable for unpredictable cases or highly variable work per iteration; this can cause higher threading overhead due to synchronization costs per chunk. Guided scheduling can offer the most flexibility, especially when the computation gets progressively more time consuming with lesser overhead associated with scheduling. They have the advantage of typically requiring fewer chunks (lesser synchronizatio). Note: The initial chunk size is roughly the number of iterations divided by the number of threads.

- **Balanced workload**: Ensuring a balanced workload among the threads will also lead to better performance. The time spent in computation between synchronization points should be comparable. Use of the runtime schedule is helpful in experimenting to find the best loop schedule to balance the load in a given program.

- **Optimizing the sequential portion**: Good performance of an OpenMP program begins with the optimizing of its sequential counterpart. This often involves reorganizing array accesses in loops and removal of redundant code which may not be done by the compiler. Once a satisfactory version of the sequential code is obtained, OpenMP constructs may be inserted one at a time, ensuring with each insertion the resulting execution remains valid. This ability to add parallelism incrementally is one of OpenMPs strong points. One guiding principal in this process should be to keep all threads doing meaningful work as much as possible.

- **Use of `parallel`**: Using this construct is generally quite expensive as it must deal with the management of the thread teams. The number of `parallel` constructs should be few, enclosing as much code as possible. An *if* clause is available to create a parallel region only on the condition that sufficient parallel work can be achieved.

- **Synchronization**: Synchronization of threads is also expensive, so it is important to consider carefully when it is necessary and avoid it otherwise. Since workshare constructs have implied barriers, it is easy to have redundant barriers occur in code without being obvious. The nowait clause can be used to remove such implied barriers.

- **Use of critical**: A `critical` region is typically the most expensive synchronization construct. So careful programming choices need to be made before using this construct. Putting too much work into the `critical` region can potentially block other threads from using it leading to performance issues. So a careful code analysis is needed in such cases. More often going through the `critical` region than necessary can lead to higher maintenance costs. Usage of `flush`, that is significantly faster than `critical` can be a workaround in certain cases. Using a simple `atomic` can sometimes do the trick instead of `critical`, advantage being the compiler able to optimize `atomic` and providing better performance. Nevertheless any protection slows down the execution of the program, be it `atomic` or `critical`. So it is advisable to not use memory protection when not needed. Cases such as a variable being local (or private, threadprivate, first and lastprivate) to a thread or variable accessed in a code fragment guaranteed to be executed by a single thread only (in a master or single section) clearly do not need memory protection.

- **False sharing**: Cache coherent systems have a side effect known as false sharing, the interference among threads writing to different locations within the same cache line. A write by one thread will cause the system to notify the other caches of this modification causing them to update their copy of that entire cache line. Other threads accessing different locations in the cache line will have to wait for this update. While the threads in this case are not accessing shared data, the cache line is shared. This is a matter of performance, not correctness, but can prevent a program from scaling to a large number of threads.

  In the following example, 1.9, a `parallel do` construct exhibits false sharing. If N is the number of threads, each thread will execute one iteration of the loop given the chunk size of 1. When each thread updates one element of b it also invalidates the cache line containing it. All other threads accessing that cache line will be affected and will need to get a new copy before their respective array elements are even modified.

  Listing 1.9: False Sharing

```
! $OMP PARALLEL DO shared(b,N)
  schedule(static,1)
    DO i = 1,N,1
      b(i) = 4 * SIN(i*1.0)
    END DO
```

  While array padding can help prevent false sharing, a better solution is to avoid the conditions in which false sharing will occur. When multiple threads are modifying shared data in the same cache line(s) in rapid succession, false sharing is likely to considerably impact performance.

- **Other tips**: Since OpenMP is a shared-memory programming model some forethought as to how shared variables will be accessed is necessary. It is also prudent to minimize cache misses. Ensuring that array accesses in loops occur according to the base-language specification is one easy method. For instance, Fortran arrays are stored in memory in column-major order while C and C++ arrays are row-major. Nested loops should take this into account. Otherwise loops will likely access non-contiguous data on each subsequent iteration, causing a cache miss each time the array is accessed.

1.5 Implementation

Implementations of OpenMP usually consists of an OpenMP-aware compiler and its runtime library (RTL). The compiler is responsible for recognizing and properly translating the programs OpenMP constructs while the RTL is responsible for thread creation and management at execution time. Most mainstream compilers that implement C, C++, and Fortran are capable of translating OpenMP. Any directives and RTL calls are translated along with the base-language program when the proper compiler options are used. Omission of these options will cause OpenMP directives to be ignored and result in a sequential program.

Some constructs are merely replaced with a call to the RTL, like the barrier and taskwait. Implementations typically convert the structured block marked by a parallel directive into a separate procedure in a process called outlining. Any

references to shared variables are replaced with pointers to the variables memory locations and passed to the outlined procedure as arguments. Private variables are local to the outlined procedure. Values for firstprivate variables are also passed as arguments. A runtime library routine will fork the necessary threads and pass the outlined procedure to each thread with the needed arguments. Threads often sleep when not involved in an active team. The user can direct whether the threads will sleep or busy-wait when idle for performance considerations.

A parallel loop will likely have each thread execute a runtime routine to determine iteration chunks prior to executing the iterations, followed by a call to a barrier routine. A guided or dynamic schedule may have threads carry out more than one set of iterations by invoking a routine to get a remaining set of iterations. These schedules will have slightly more overhead because of this additional coordination. However, these schedules may execute the work fast enough to justify the added overhead.

A task can also be outlined to a procedure and variables passed as arguments similar to the parallel construct. Variables in tasks are firstprivate by default and their values must be saved at the time of task generation. Shared variables may be replaced with pointers to their memory locations as with the parallel construct. Each task is saved for execution at some time in the future, likely in a queue. The actual scheduling of the execution of the tasks is left to implementation dependent and vary greatly. OpenMP allow for work stealing, allowing a thread to take unfinished from other threads. By default, a task must be executed in its entirety by exactly one thread. If task is defined to be untied, it may finish execution on a different thread, making it a candidate for work stealing. As such, implementations may actually employ more than one schedule.

1.6 Performance

While OpenMP makes parallel programs easy to write, some extra work is usually necessary to allow complex programs to scale to high numbers of processors. Inherent in parallel programming is the presence of overhead, or work that would not otherwise be done in a sequential execution of the code. Each construct used will require some overhead to set up the parallel execution environment, so limiting their use to the presence of sufficient parallel work is advised. In addition to avoiding unnecessary overhead, the use of the memory hierarchy needs some careful attention.

Good performance of an OpenMP program begins with the optimizing of its sequential counterpart. This often involves reorganizing array accesses in loops and removal of redundant code which may not be done by the compiler. Once a satisfactory version of the sequential code is obtained, OpenMP constructs may be inserted one at a time, ensuring with each insertion the resulting execution remains valid. This ability to add parallelism incrementally is one of OpenMPs strong points. One guiding principal in this process should be to keep all threads doing meaningful work as much as possible.

Use of the parallel construct is generally quite expensive as it must deal with the management of the thread teams. The number of parallel constructs should be few, enclosing as much code as possible. An if clause is available to create a parallel region only on the condition that sufficient parallel work can be achieved. Synchronization of threads is also expensive, so it is important to consider carefully when it is necessary and avoid it otherwise. Since workshare constructs have implied barriers, it is easy to have redundant barriers occur in code without being obvious. The nowait clause can be used to remove such implied barriers.

Since OpenMP is a shared-memory programming model some forethought as to how shared variables will be accessed is necessary. It is also prudent to minimize cache misses. Ensuring that array accesses in loops occur according to the base-language specification is one easy method. For instance, Fortran arrays are stored in memory in column-major order while C and C++ arrays are row-major. Nested loops should take this into account. Otherwise loops will likely access non-contiguous data on each subsequent iteration, causing a cache miss each time the array is accessed.

Cache coherent systems have a side effect know as false sharing, the interference among threads writing to different locations within the same cache line. A write by one thread will cause the system to notify the other caches of this modification causing them to update their copy of that entire cache line. Other threads accessing different locations in the cache line will have to wait for this update. While the threads in this case are not accessing shared data, the cache line is shared. This is a matter of performance, not correctness, but can prevent a program from scaling to a large number of threads.

In the following example, a parallel do construct exhibits false sharing. If N is the number of threads, each thread will execute one iteration of the loop given the chunk size of 1. When each thread updates one element of b it also invalidates the cache line containing it. All other threads accessing that cache line will be affected and will need to get a new copy before their respective array elements are even modified.

```
!$OMP PARALLEL DO shared(b,N) schedule(static,1)
DO i=1,N,1
    b(i) = 4 * SIN(i*1.0)
END DO
```

While array padding can help prevent false sharing, a better solutions is to avoid the conditions in which false sharing will occur. When multiple threads are modifying shared data in the same cache line(s) in rapid succession, false sharing is likely to considerably impact performance.

Ensuring a balanced workload among the threads will also lead to better performance. The time spent in computation between synchronization points should be comparable. Use of the runtime schedule is helpful in experimenting to find the best loop schedule to balance the load in a given program.

1.7 Correctness Considerations

It is actually quite easy to inadvertently introduce bugs into OpenMP programs. Bugs in shared-memory programming are usually very subtle and difficult to find. One such bug, the data race condition, involves the silent corruption of data during execution that is hard to detect as it may not manifest itself but on only a small fraction of the time. A data race condition occurs when multiple threads concurrently access the same shared data with at least one of the threads attempting to modify the data. Since there is no guarantee of the order in which the threads access data, this may lead to an incorrect result. Data race conditions may result from the lack of proper synchronization, like using a nowait incorrectly or neglecting to enclose such data access in a critical region.

It is usually good practice to minimize the sharing of variables since this can be problematic. By privatizing variables that do not need to be shared, data race conditions and other issues can be avoided. And, since variables are shared by default, overlooked variables may be unintentionally shared. For example, in a Fortran loop the index variables are always treated as private, but they are only private in a parallel-for loop in C. So in C/C++, the index variable of an inner loop will be shared if not explicitly privatized. Overlooked shared variables are also a source of the data race condition.

Another problem is the misunderstanding of which constructs have an implied barrier and which do not. In many cases, the single and the master constructs may be used for the same purpose. However, the former has an implied barrier and the latter does not. When using the master construct it is important to know if an explicit barrier is needed afterwards. With the single, a nowait may be warranted to remove an unnecessary barrier.

Barriers may also lead to deadlock during execution if they are not used properly. An explicit barrier must be executed by all thread in the current thread team. If any thread does not reach the barrier, all the other threads will be waiting at the barrier for a thread that will never arrive. Another source of deadlock is the improper nesting of explicit locks. Deadlocked code will appear to be executing work but will never finish. These conditions are easily avoided with careful use of these constructs.

1.8 Future Directions

The ARB is actively considering new strategies and features to add to or enhance the current API, deliberating on changes frequently being proposed by researchers. The ARB seeks to only make changes in the API that are generally acceptable to all vendors on all platforms. The challenge they face is to keep the API small enough to be relatively easy to use yet robust enough to offer sufficient expressivity for existing parallelism. Proposals under consideration include error handling mechanisms, support for performance tools, enhancing the current task interface, providing the mapping of work and data to threads, and the ability to exploit heterogeneous architectures such as GPUs and accelerators.

1.9 Future Directions

Need input from Dr.Chapman here

References

[1] The openmp 4.0 api examples document. `http://openmp.org/mp-documents/OpenMP4.0.0.Examples.pdf`.

[2] OpenMP 4.0 Public Review Release Candidate Specifications. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in openmp. In *A Practical Programming Model for the Multi-Core Era*, pages 1–12. Springer, 2008.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[5] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[6] Chunhua Liao, Oscar Hernandez, Barbara M. Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: an Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[7] Mitsuhisa Sato, Mitsuhisa Sato Shigehisa, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *In EWOMP 99*, pages 32–39, 1999.

[8] CORPORATE The Parallel Computing Forum. Pcf parallel fortran extensions. *SIGPLAN Fortran Forum*, 10(3):1–57, September 1991.