OpenMP: An Industry-Standard API for Shared-Memory Programming

LEONARDO DAGUM AND RAMESH MENON
SILICON GRAPHICS INC.

OpenMP, the portable alternative to message passing, offers a powerful new way to achieve scalability in software. This article compares OpenMP to existing parallel-programming models.

Application developers have long recognized that scalable hardware and software are necessary for parallel scalability in application performance. Both have existed for some time in their lowest common denominator form, and scalable hardware—as physically distributed memories connected through a scalable interconnection network (as a multistage interconnect, k-ary n-cube, or fat tree)—has been commercially available since the 1980s. When developers build such systems without any provision for cache coherence, the systems are essentially "zeroth order" scalable architectures. They provide only a scalable interconnection network, and the burden of scalability falls on the software. As a result, scalable software for such systems exists, at some level, only in a message-passing model. Message passing is the native model for these architectures, and developers can only build higher-level models on top of it.

Unfortunately, many in the high-performance computing world implicitly assume that the only way to achieve scalability in parallel software is with a message-passing programming model. This is not necessarily true. A class of multiprocessor architectures is now emerging that offers scalable hardware support for cache coherence. These are generally called *scalable shared memory multiprocessor* architectures.[1] For SSMP systems, the native programming model is shared memory, and message passing is built on top of the shared-memory model. On such systems, software scalability is straightforward to achieve with a shared-memory programming model.

In a shared-memory system, every processor has direct access to the memory of every other processor, meaning it can directly load or store any shared address. The programmer also can declare certain pieces of memory as private to the processor, which provides a simple yet powerful model for expressing and managing parallelism in an application.

Despite its simplicity and scalability, many parallel applications developers have resisted adopting a shared-memory programming model for one reason: portability. Shared-memory system vendors have created their own proprietary extensions to Fortran or C for parallel-software development. However, the absence of portability has forced many developers to adopt a portable message-passing model such as the Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). This article presents a portable alternative to message passing: OpenMP.

1070-9924/98/$10.00 © 1998 IEEE

OpenMP was designed to exploit certain characteristics of shared-memory architectures. The ability to directly access memory throughout the system (with minimum latency and no explicit address mapping), combined with fast shared-memory locks, makes shared-memory architectures best suited for supporting OpenMP.

Why a new standard?

The closest approximation to a standard shared-memory programming model is the now-dormant ANSI X3H5 standards effort.[2] X3H5 was never formally adopted as a standard largely because interest waned as distributed-memory message-passing systems (MPPs) came into vogue. However, even though hardware vendors support it to varying degrees, X3H5 has limitations that make it unsuitable for anything other than loop-level parallelism. Consequently, applications adopting this model are often limited in their parallel scalability.

MPI has effectively standardized the message-passing programming model. It is a portable, widely available, and accepted standard for writing message-passing programs. Unfortunately, message passing is generally a difficult way to program. It requires that the program's data structures be explicitly partitioned, so the entire application must be parallelized to work with the partitioned data structures. There is no incremental path to parallelize an application. Furthermore, modern multiprocessor architectures increasingly provide hardware support for cache coherence; therefore, message passing is becoming unnecessary and overly restrictive for these systems.

Pthreads is an accepted standard for shared memory in low-end systems. However, it is not targeted at the technical, HPC space. There is little Fortran support for pthreads, and it is not a scalable approach. Even for C applications, the pthreads model is awkward, because it is lower-level than necessary for most scientific applications and is targeted more at providing task parallelism, not data parallelism. Also, portability to unsupported platforms requires a stub library or equivalent workaround.

Researchers have defined many new languages for parallel computing, but these have not found mainstream acceptance. High-Performance Fortran (HPF) is the most popular multiprocessing derivative of Fortran, but it is mostly geared toward distributed-memory systems.

Independent software developers of scientific applications, as well as government laboratories, have a large volume of Fortran 77 code that needs to get parallelized in a portable fashion. The rapid and widespread acceptance of shared-memory multiprocessor architectures—from the desktop to "glass houses"—has created a pressing demand for a portable way to program these systems. Developers need to parallelize existing code without completely rewriting it, but this is not possible with most existing parallel-language standards. Only OpenMP and X3H5 allow incremental parallelization of existing code, of which only OpenMP is scalable (see Table 1). OpenMP is targeted at developers who need to quickly parallelize existing scientific code, but it remains flexible enough to support a much broader application set. OpenMP provides an incremental path for parallel conversion of any existing software. It also provides scalability and performance for a complete rewrite or entirely new development.

Table 1: Comparing standard parallel-programming models.

	X3H5	MPI	Pthreads	HPF	OpenMP
Scalable	no	yes	sometimes	yes	yes
Incremental parallelization	yes	no	no	no	yes
Portable	yes	yes	yes	yes	yes
Fortran binding	yes	yes	no	yes	yes
High level	yes	no	no	yes	yes
Supports data parallelism	yes	no	no	yes	yes
Performance oriented	no	yes	no	tries	yes

What is OpenMP?

At its most elemental level, OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express shared-memory parallelism. It leaves the base language unspecified, and vendors can implement OpenMP in any Fortran compiler. Naturally, to support pointers and allocatables, Fortan 90 and Fortran 95 require the OpenMP implementation to include additional semantics over Fortran 77.

Table 2: Comparing X3H5 directives, OpenMP, and MIPS Pro Doacross functionality.

	X3H5	OpenMP	MIPS Pro
Overview			
Orphan scope	None, lexical scope only	Yes, binding rules specified	Yes, through callable runtime
Query functions	None	Standard	Yes
Runtime functions	None	Standard	Yes
Environment variables	None	Standard	Yes
Nested parallelism	Allowed	Allowed	Serialized
Throughput mode	Not defined	Yes	Yes
Conditional compilation	None	!$OMP	C$
Sentinel	C$PAR	!$OMP	C$
	C$PAR&	!$OMP&	C$&
Control structure			
Parallel region	Parallel	Parallel	Doacross
Iterative	Pdo	Do	Doacross
Noniterative	Psection	Section	User coded
Single process	Psingle	Single	User coded
		Master	
Early completion	Pdone	User coded	User coded
Sequential Ordering	Ordered PDO	Ordered	None
Data environment			
Autoscope	None	Default(private)	shared default
		Default(shared)	
Global objects	Instance Parallel ($p + 1$ instances)	Threadprivate (p instances)	Linker, Xlocal (p instances)
Reduction attribute	None	Reduction	Reduction
Private initialization	None	Firstprivate	None
		Copyin	Copyin
Private persistence	None	Lastprivate	Lastlocal
Synchronization			
Barrier	Barrier	Barrier	mp_barrier
Synchronize	Synchronize	Flush	synchronize
Critical section	Critical Section	Critical	mp_setlock
			mp_unsetlock
Atomic update	None	Atomic	None
Locks	None	Full functionality	mp_setlock
			mp_unsetlock

OpenMP leverages many of the X3H5 concepts while extending them to support coarse-grain parallelism. Table 2 compares OpenMP with the directive bindings specified by X3H5 and the MIPS Pro Doacross model,[3] and it summarizes the language extensions into one of three categories: control structure, data environment, or synchronization. The standard also includes a callable runtime library with accompanying environment variables.

Several vendors have products—including compilers, development tools, and performance-analysis tools—that are OpenMP aware. Typically, these tools understand the semantics of OpenMP constructs and hence aid the process of writing programs. The OpenMP Architecture Review Board includes representatives from Digital, Hewlett-Packard, Intel, IBM, Kuck and Associates, and Silicon Graphics.

All of these companies are actively developing compilers and tools for OpenMP. Open MP products are available today from Silicon Graphics and other vendors. In addition, a number of independent software vendors plan to use OpenMP in future products. (For information on individual products, see *www.openmp.org*.)

A simple example

Figure 1 presents a simple example of computing π using OpenMP.[4] This example illustrates how to parallelize a simple loop in a shared-memory programming model. The code would look similar with either the Doacross or the X3H5 set of directives (except that X3H5 does not have a `reduction` attribute, so you would have to code it yourself).

Program execution begins as a single process. This initial process executes serially, and we can set up our problem in a standard sequential manner, reading and writing `stdout` as necessary. When we first encounter a `parallel` construct, in this case a `parallel do`, the runtime forms a team of one or more processes and creates the data environment for each team member. The data environment consists of one `private` variable, `x`, one `reduction` variable, `sum`, and one `shared` variable, `w`. All references to `x` and `sum` inside the parallel region address private, nonshared copies. The `reduction` at-

```
      program compute_pi
      integer n, i
      double precision w, x, sum, pi, f, a
c function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      print *, 'Enter number of intervals: '
      read *,n
c calculate the interval size
      w = 1.0d0/n
      sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+: sum)
      do i = 1, n
        x = w * (i - 0.5d0)
        sum = sum + f(x)
      enddo
      pi = w * sum
      print *, 'computed pi = ', pi
      stop
      end
```

Figure 1. Computing π in parallel using OpenMP.

tribute takes an operator, such that at the end of the parallel region it reduces the private copies to the master copy using the specified operator. All references to w in the parallel region address the single master copy. The loop index variable, `i`, is `private` by default. The compiler takes care of assigning the appropriate iterations to the individual team members, so in parallelizing this loop the user need not even on know how many processors it runs.

There might be additional control and synchronization constructs within the parallel region, but not in this example. The parallel region terminates with the `end do`, which has an implied barrier. On exit from the parallel region, the initial process resumes execution using its updated data environment. In this case, the only change to the master's data environment is the reduced value of `sum`.

This model of execution is referred to as the *fork/join model*. Throughout the course of a program, the initial process can fork and join many times. The fork/join execution model makes it easy to get loop-level parallelism out of a sequential program. Unlike in message passing, where the program must be completely decomposed for parallel execution, the shared-memory model makes it possible to parallelize just at the loop level without decomposing the data structures. Given a working sequential program, it becomes fairly straightforward to parallelize individual loops incrementally and thereby immediately realize the performance advantages of a multiprocessor system.

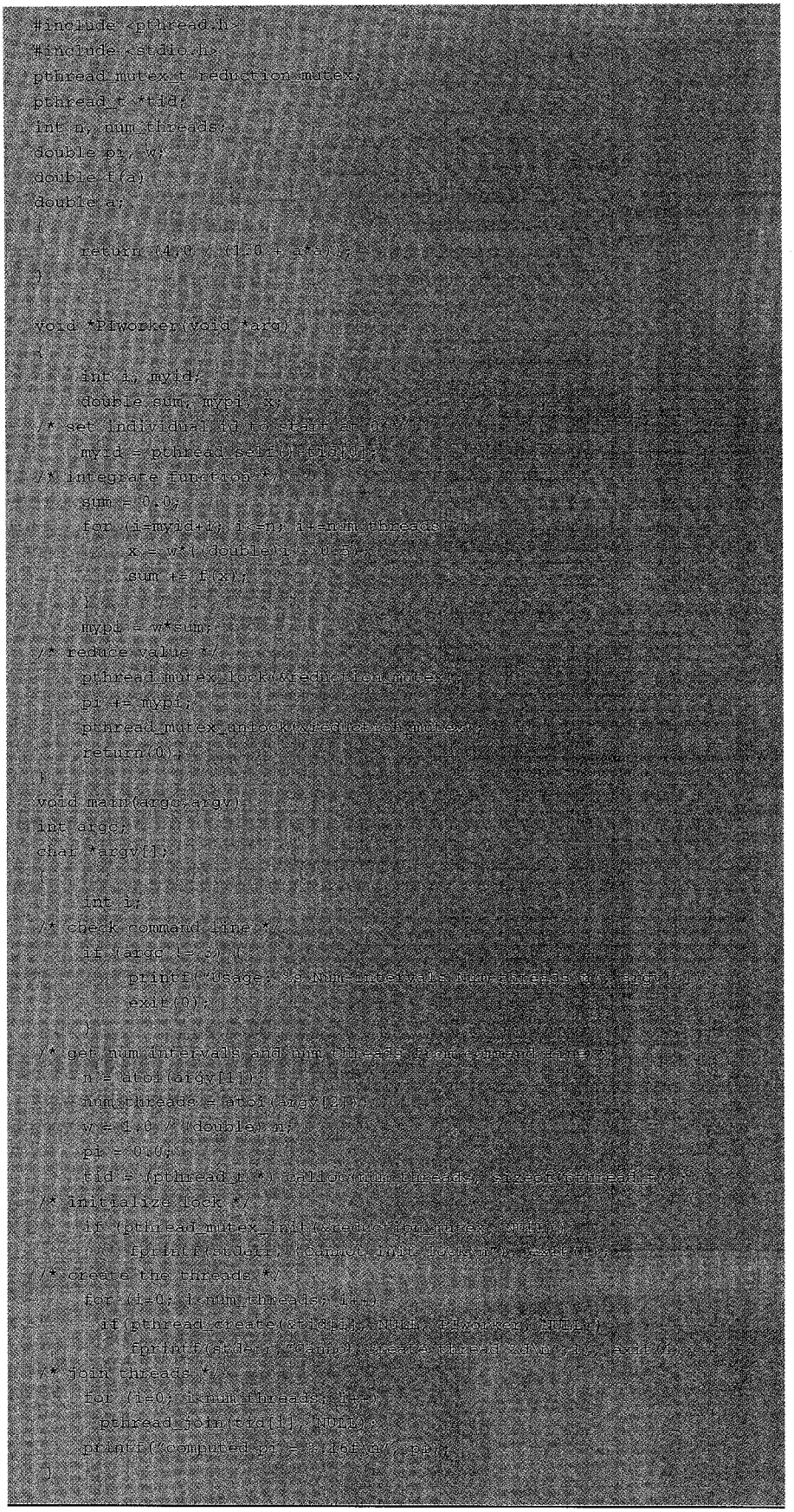
For comparison with message passing, Figure 2 presents the same example using MPI. Clearly, there is additional complexity just in setting up the problem, because we must begin with a team of parallel processes. Consequently, we need to isolate a root process to read and write `stdout`. Because there is no globally shared data, we must explicitly broadcast the input parameters (in this case, the number of intervals for the integration) to all the processors. Furthermore, we must explicitly manage the loop bounds. This requires identifying each processor (`myid`) and knowing how many processors will be used to ex-

```
      program compute_pi
      include 'mpif.h'
      double precision mypi, pi, w, sum, x, f, a
      integer n, myid, numprocs, i, rc
c function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

      if ( myid .eq. 0 ) then
         print *, 'Enter number of intervals:'
         read *, n
      endif
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c calculate the interval size
      w = 1.0d0/n
      sum = 0.0d0
      do i = myid+1, n, numprocs
         x = w * (i - 0.5d0)
         sum = sum + f(x)
      enddo
      mypi = w * sum
c collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $    MPI_COMM_WORLD,ierr)
c node 0 prints the answer.
      if (myid .eq. 0) then
         print *, 'computed pi = ', pi
      endif
      call MPI_FINALIZE(rc)
      stop
      end
```

Figure 2. Computing π in parallel using MPI.



```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t reduction_mutex;
pthread_t *tid;
int n, num_threads;
double pi, w;
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}

void *PIworker(void *arg)
{
    int i, myid;
    double sum, mypi, x;
/* set individual id to start at 0 */
    myid = pthread_self();
/* integrate function */
    sum = 0.0;
    for (i=myid+1; i<=n; i+=num_threads) {
        x = w*((double)i - 0.5);
        sum += f(x);
    }
    mypi = w*sum;
/* reduce value */
    pthread_mutex_lock(&reduction_mutex);
    pi += mypi;
    pthread_mutex_unlock(&reduction_mutex);
    return(0);
}
void main(argc,argv)
int argc;
char *argv[];
{
    int i;
/* check command line */
    if (argc != 2) {
        printf("Usage: %s Num-intervals Num-threads\n", argv[0]);
        exit(0);
    }
/* get num intervals and num threads from command line */
    n = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    w = 1.0 / (double) n;
    pi = 0.0;
    tid = (pthread_t *) calloc(num_threads, sizeof(pthread_t));
/* initialize lock */
    if (pthread_mutex_init(&reduction_mutex, NULL))
        fprintf(stderr, "Cannot init lock\n"), exit(1);
/* create the threads */
    for (i=0; i<num_threads; i++)
        if (pthread_create(&tid[i], NULL, PIworker, NULL))
            fprintf(stderr, "Cannot create thread %d\n", i), exit(1);
/* join threads */
    for (i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
    printf("computed pi = %16.14f\n", pi);
}
```

Figure 3. Computing π in parallel using pthreads.

ecute the loop (**numprocs**).

When we finally get to the loop, we can only sum into our private value for **mypi**. To reduce across processors we use the `MPI_Reduce` routine and sum into **pi**. The storage for **pi** is replicated across all processors, even though only the root process needs it. As a general rule, message-passing programs waste more storage than shared-memory programs.[5] Finally, we can print the result, again making sure to isolate just one process for this step to avoid printing **numprocs** messages.

It is also interesting to see how this example looks using pthreads (see Figure 3). Naturally, it's written in C, but we can still compare functionality with the Fortran examples given in Figures 1 and 2.

The pthreads version is more complex than either the OpenMP or the MPI versions:

- First, pthreads is aimed at providing task parallelism, whereas the example is one of data parallelism—parallelizing a loop. The example shows why pthreads has not been widely used for scientific applications.
- Second, pthreads is somewhat lower-level than we need, even in a task- or threads-based model. This becomes clearer as we go through the example.

As with the MPI version, we need to know how many threads will execute the loop and we must determine their IDs so we can manage the loop bounds. We get the thread number as a command-line argument and use it to allocate an array of thread IDs. At this time, we also initialize a lock, **reduction_mutex**, which we'll

need for reducing our partial sums into a global sum for π. Our basic approach is to start a worker thread, `PIworker`, for every processor we want to work on the loop. In `PIworker`, we first compute a zero-based thread ID and use this to map the loop iterations. The loop then computes the partial sums into `mypi`. We add these into the global result `pi`, making sure to protect against a race condition by locking. Finally, we need to explicitly join all our threads before we can print out the result of the integration.

All the data scoping is implicit; that is, global variables are shared and automatic variables are private. There is no simple mechanism in pthreads for making global variables private. Also, implicit scoping is more awkward in Fortran because the language is not as strongly scoped as C.

In terms of performance, all three models are comparable for this simple example. Table 3 presents the elapsed time in seconds for each program when run on a Silicon Graphics Origin2000 server, using 10^9 intervals for each integration. All three models are exhibiting excellent scalability on a per node basis (there are two CPUs per node in the Origin2000), as expected for this embarrassingly parallel algorithm.

Table 3: Time (in seconds) to compute π using 10^9 intervals with three standard parallel-programming models.

CPUs	OpenMP	MPI	Pthreads
1	107.7	121.4	115.4
2	53.9	60.7	62.5
4	27.0	30.3	32.4
6	17.9	20.4	22.0
8	13.5	15.2	16.7

Scalability

Although simple and effective, loop-level parallelism is usually limited in its scalability, because it leaves some constant fraction of sequential work in the program that by Amdahl's law can quickly overtake the gains from parallel execution. It is important, however, to distinguish between the type of parallelism (for example, loop-level versus coarse-grained) and the programming model. The type of parallelism exposed in a program depends on the algorithm and data structures employed and not on the programming model (to the extent that those algorithms and data structures can be reasonably expressed within a given model). Therefore, given a parallel algorithm and a scalable shared-memory architecture, a shared-memory implementation scales as well as a message-passing implementation.

OpenMP introduces the powerful concept of *orphan* directives that simplify the task of implementing coarse-grain parallel algorithms. Orphan directives are directives encountered outside the *lexical* extent of the parallel region. Coarse-grain parallel algorithms typically consist of only a few parallel regions, with most of the execution taking place within those regions.

In implementing a coarse-grained parallel algorithm, it becomes desirable, and often necessary, to be able to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained portion. OpenMP provides this functionality by specifying binding rules for all directives and allowing them to be encountered dynamically in the call chain originating from the parallel region. In contrast, X3H5 does not allow directives to be orphaned, so all the control and synchronization for the program must be lexically visible in the parallel construct. This limitation restricts the programmer and makes any nontrivial coarse-grained parallel application virtually impossible to write.

A coarse-grain example

To highlight additional features in the standard, Figure 4 presents a slightly more complicated example, computing the energy spectrum for a field. This is essentially a histogramming problem with a slight twist—it also generates the sequence in parallel. We could easily parallelize the histogramming loop and the sequence generation as in the previous example, but in the interest of performance we would like to histogram as we compute in order to preserve locality.

The program goes immediately into a parallel region with a `parallel` directive, declaring the variables `field` and `ispectrum` as `shared`, and making everything else `private` with a `default` clause. The `default` clause does not affect common blocks, so `setup` remains a shared data structure.

Within the parallel region, we call `initialize_field()` to initialize the `field` and `ispectrum` arrays. Here we have an example of

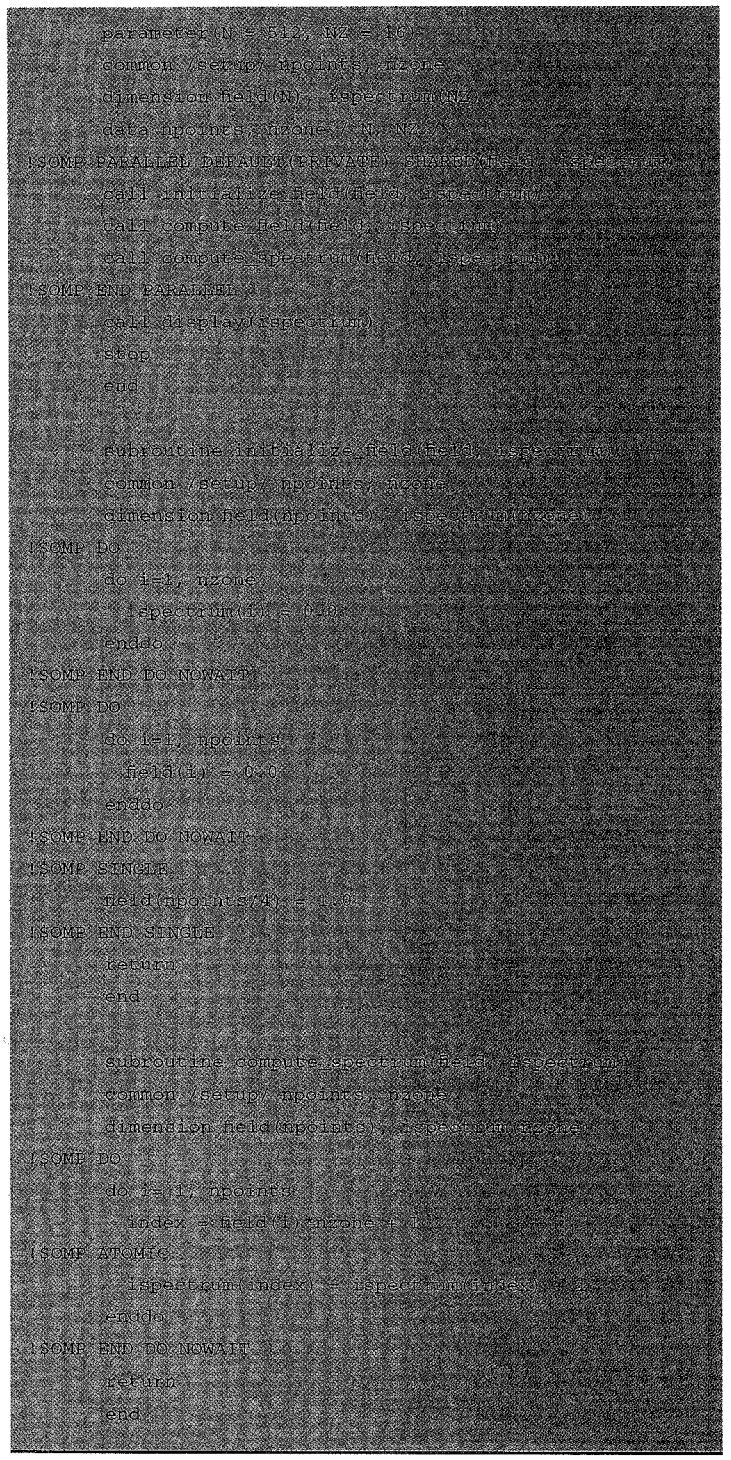


Figure 4. A coarse-grained example.

orphaning the do directive. With the X3H5 directives, we would have to move these loops into the main program so that they could be lexically visible within the parallel directive. Clearly, that restriction makes it difficult to write good modular parallel programs. We use the nowait clause on the end do directives to eliminate the implicit barrier. Finally, we use the single directive when we initialize a single internal **field** point. The end single directive also can take a nowait clause, but to guarantee correctness we need to synchronize here.

The **field** gets computed in **compute_field**. This could be any parallel Laplacian solver, but in the interest of brevity we don't include it here. With the field computed, we are ready to compute the spectrum, so we histogram the field values using the atomic directive to eliminate race conditions in the updates to **ispectrum**. The end do here has a **nowait** because the parallel region ends after **compute spectrum()** and there is an implied barrier when the threads join.

OpenMP design objective

OpenMP was designed to be a flexible standard, easily implemented across different platforms. As we discussed, the standard compriss four distinct parts:

- control structure,
- the data environment,
- synchronization, and
- the runtime library.

Control structure

OpenMP strives for a minimalist set of control structures. Experience has indicated that only a few control structures are necessary for writing most parallel applications. For example, in the Doacross model, the only control structure is the doacross directive, yet this is arguably the most widely used shared-memory programming model for scientific computing. Many of the control structures provided by X3H5 can be trivially programmed in OpenMP with no performance penalty. OpenMP includes control structures only in those instances where a compiler can provide both functionality and performance over what a user could reasonably program.

Our examples used only three control structures: parallel, do, and single. Clearly, the compiler adds functionality in parallel and do directives. For single, the compiler adds performance by allowing the first thread reaching the single directive to execute the code. This is nontrivial for a user to program.

Data environment

Associated with each process is a unique data environment providing a context for execution.

The initial process at program start-up has an initial data environment that exists for the duration of the program. It contructs new data environments only for new processes created during program execution. The objects constituting a data environment might have one of three basic attributes: `shared`, `private`, or `reduction`.

The concept of `reduction` as an attribute is generalized in OpenMP. It allows the compiler to efficiently implement `reduction` operations. This is especially important on cache-based systems where the compiler can eliminate any false sharing. On large-scale SSMP architectures, the compiler also might choose to implement tree-based reductions for even better performance.

OpenMP has a rich data environment. In addition to the `reduction` attribute, it allows `private` initialization with `firstprivate` and `copyin`, and private persistence with `lastprivate`. None of these features exist in X3H5, but experience has indicated a real need for them.

Global objects can be made `private` with the `threadprivate` directive. In the interest of performance, OpenMP implements a "*p*-copy" model for privatizing global objects: `threadprivate` will create *p* copies of the global object, one for each of the *p* members in the team executing the parallel region. Often, however, it is desirable either from memory constraints or for algorithmic reasons to privatize only certain elements because of a compound global object. OpenMP allows individual elements of a compound global object to appear in a `private` list.

Synchronization

There are two types of synchronization: implicit and explicit. Implicit synchronization points exist at the beginning and end of parallel constructs and at the end of control constructs (for example, `do` and `single`). In the case of

do sections, and single, the implicit synchronization can be removed with the nowait clause.

The user specifies explicit synchronization to manage order or data dependencies. Synchronization is a form of interprocess communication and, as such, can greatly affect program performance. In general, minimizing a program's synchronization requirements (explicit and implicit) achieves the best performance. For this reason, OpenMP provides a rich set of synchronization features so developers can best tune the synchronization in an application.

We saw an example using the Atomic directive. This directive allows the compiler to take advantage of available hardware for implementing atomic updates to a variable. OpenMP also provides a Flush directive for creating more complex synchronization constructs such as point-to-point synchronization. For ultimate performance, point-to-point synchronization can eliminate the implicit barriers in the energy-spectrum example. All the OpenMP synchronization directives can be orphaned. As discussed earlier, this is critically important for implementing coarse-grained parallel algorithms.

Runtime library and environment variables

In addition to the directive set described, OpenMP provides a callable runtime library and accompanying environment variables. The runtime library includes query and lock functions. The runtime functions allow an application to specify the mode in which it should run. An application developer might wish to maximize the system's throughput performance, rather than time to completion. In such cases, the developer can tell the system to dynamically set the number of processes used to execute parallel regions. This can have a dramatic effect on the system's throughput performance with only a minimal impact on the program's time to completion.

The runtime functions also allow a developer to specify when to enable nested parallelism, which allows the system to act accordingly when it encounters a nested parallel construct. On the other hand, by disabling it, a developer can write a parallel library that will perform in an easily predictable fashion whether encountered dynamically from within or outside a parallel region.

OpenMP also provides a conditional compilation facility both through the C language preprocessor (CPP) and with a Fortran comment sentinel. This allows calls to the runtime library to be protected as compiler directives, so OpenMP code can be compiled on non-OpenMP systems without linking in a stub library or using some other awkward workaround.

OpenMP provides standard environment variables to accompany the runtime library functions where it makes sense and to simplify the start-up scripts for portable applications. This helps application developers who, in addition to creating portable applications, need a portable runtime environment.

OpenMP is supported by a number of hardware and software vendors, and we expect support to grow. OpenMP has been designed to be extensible and evolve with user requirements. The OpenMP Architecture Review Board was created to provide long-term support and enhancements of the OpenMP specifications. The OARB charter includes interpreting OpenMP specifications, developing future OpenMP standards, addressing issues of validation of OpenMP implementations, and promoting OpenMP as a de facto standard.

Possible extensions for Fortran include greater support for nested parallelism and support for shaped arrays. Nested parallelism is the ability to create a new team of processes from within an existing team. It can be useful in problems exhibiting both task and data parallelism. For example, a natural application for nested parallelism would be parallelizing a task queue wherein the tasks involve large matrix multiplies.

Shaped arrays refers to the ability to explicitly assign the storage for arrays to specific memory nodes. This ability is useful for improving performance on Non-Uniform Memory architectures (NUMAs) by reducing the number of non-local memory references made by a processor.

The OARB is currently developing the specification of C and C++ bindings and is also developing validation suites for tesing OpenMP implementations. ♦

References

1. D.E. Lenoski and W.D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann, San Francisco, 1995.

2. B. Leasure, ed., *Parallel Processing Model for High-Level Programming Languages*, proposed draft, American National Standard for Information Processing Systems, Apr. 5, 1994.
3. MIPSpro Fortran77 Programmer's Guide, Silicon Graphics, Mountain View, Calif., 1996; http://techpubs.sgi.com/library/dynaweb_bin/0640/bi/nph-dynaweb.cgi/dynaweb/SGI_Developer/MproF77_PG/.
4. S. Ragsdale, ed., *Parallel Programming Primer*, Intel Scientific Computers, Santa Clara, Calif., March 1990.
5. J. Brown, T. Elken, and J. Taft, *Silicon Graphics Technical Servers in the High Throughput Environment*, Silicon Graphics Inc., 1995; *http://www.sgi.com/tech/challenge.html.*

Leonardo Dagum works for Silicon Graphics in the System Performance group, where he helped define the OpenMP Fortran API. His research interests include parallel algorithms and performance modelling for parallel systems. He is the author of over 30 refereed publications relating to these subjects. He received his MS and PhD in aeronautics and astronautics from Stanford. Contact him at M/S 580, 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389; dagum@sgi.com.

Ramesh Menon is Silicon Graphics' representative to the OpenMP Architicture Review Board and served as the board's first chairman. He managed the writing of the OpenMP Fortran API. His research interests include parallel-programming models, performance characterization, and computational mechanics. He received an MS in mechanical engineering from Duke University and a PhD in aerospace engineering from Texas A&M. He was awarded a National Science Foundation Fellowship and was a principal contributor to the NSF Grand Challenge Coupled Fields project at the Univeristy of Colorado, Boulder. Contact him at menon@sgi.com.