# OpenMP: A Proposed Industry Standard API for Shared Memory Programming

October 1997

## 1 Introduction

It has long been recognized that the two necessary elements for achieving parallel scalability in application performance are scalable hardware and scalable software. Both these elements have existed for some time in their lowest common denominator form. Specifically, scalable hardware in the context of physically distributed memories connected through a scalable interconnection network such as a multistage interconnect, k-ary n-cube, or fat tree, have been commercially available since the 1980's. When such systems are built without any provision for cache coherence they are essentially "zeroth order" scalable architectures. In other words, such systems provide only a scalable interconnection network, and the burden of scalability then falls on the software. As a result scalable software for such systems will exist, at some level, only in a message passing model. Message passing is the native model for these architectures, and higher level models can only be built on top of message passing.

Unfortunately, this situation has existed long enough that there is now an implicit assumption in the high performance computing world that the only way to achieve scalability in parallel software is with a message passing programming model. This is not necessarily true. There is now an emerging class of multiprocessor architecture with scalable hardware support for cache coherence. These are generally referred to as Scalable Shared Memory Multiprocessor, or SSMP, 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 the shared memory model every processor has direct access to the memory of every other processor in the system. This means a processor can directly load or store any shared address. The programmer also can declare certain pieces of memory to be private to the processor. This provides a very simple yet

powerful model to the programmer for expressing and managing parallelism in an application.

Despite its promise of simplicity and scalability, many developers have resisted adopting a shared memory programming model in parallelizing their applications for one simple reason: portability. Every vendor of a shared memory system has created its own proprietary extensions to Fortran or C for developers to use in developing parallel software. However, the absence of portability has forced many developers to instead adopt a portable message passing model like the Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). This paper presents a portable alternative to message passing: OpenMP.

## 2 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 MPP's came in vogue. Nonetheless it is supported to varying degrees by many hardware vendors. However, X3H5 has some important limitations that make it unsuitable for other than loop level parallelism. Consequently, applications adopting this model are often limited in their parallel scalability.

The message passing programming model has been very effectively standardized by MPI. MPI 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 in order to work with the partitioned data structures. There is no incremental path to parallelizing an application. Furthermore, modern multiprocessor architectures are increasingly providing 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 the low end. However it is not targeted at the Technical/HPC space. There is very 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 with very little support for data parallelism. Also, portability to unsupported platforms requires a stub library or equivalent workaround.

There have been many new languages defined for parallel computing, but largely they have not been adopted in the mainstream. HPF is the most popular multiprocessing derivative of Fortran but it is mostly geared to distributed memory systems.

Independent software developers of scientific applications as well as government laboratories have a large volume of Fortran77 code that needs to get parallelized in a portable fashion. The rapid and widespread acceptance of shared

| | **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 |

Table 1: Comparison of various standard parallel programming models.

memory multiprocessor architectures, from the desktop to the "glass houses", has created a pressing demand for a portable way to program these systems. This is driven largely by the desire to begin parallelizing existing code without having to completely rewrite it. This is not possible with most existing parallel language standards, as is evident in Table 1. It's clear from this table that only OpenMP and X3H5 allow incremental parallelization of existing code, and of those two only OpenMP is scalable. OpenMP is targeted to developers who need to quickly parallelize existing scientific code, but it remains flexible enough to support a much broader application set than just scientific computing. OpenMP provides an incremental path for parallel conversion of any existing software, as well as scalability and performance for a complete rewrite or an entirely new development.

# 3 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. The base language is left unspecified and vendors may implement OpenMP in any Fortran compiler. Naturally, Fortan 90 and Fortran 95 require the OpenMP implementation to include additional semantics over Fortran 77 in order to support pointers and allocatables.

OpenMP leverages many of the X3H5 concepts while extending them to

support coarse grain parallelism. Table 2 compares OpenMP with X3H5 and the SGI DOACROSS model [3].

The language extensions fall into one of three categories: control structures, data environment, and synchronization. The standard also includes a callable runtime library with accompanying environment variables. Table 2 summarizes the language extensions in terms of these categories.

### 3.1 A Simple Example

Figure 1 presents a simple code example for computing PI using OpenMP. This example is meant only to illustrate how a simple loop may be parallelized in a shared memory programming model. The code would look quite 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, a team of one or more processes is formed, and the data environment for each team member is created. The data environment here 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, non-shared, copies. The REDUCTION attribute takes an operator, such that at the end of the parallel region the private copies are reduced 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 you don't even need to know how many processors you will run it on.

Within the parallel region there may be additional control and synchronization constructs, but there are none in this simple example. The parallel region here terminates with the END DO which has an implied barrier. On exit of 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 in 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 may fork and join a number of 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, in a shared memory model it is 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 in an incremental fashion and thereby immediately realize the performance advantages of a multiprocessor system.

| | **X3H5** | **OpenMP** | **SGI** |
|---|---|---|---|
| **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 | _OPENMP<br>!$ | C$ |
| Sentinel | C$PAR<br>C$PAR& | !$OMP<br>!$OMP& | C$<br>C$& |
| **Control Structures** | | | |
| Parallel region | PARALLEL | PARALLEL | DOACROSS |
| Iterative | PDO | DO | DOACROSS |
| Non-iterative | PSECTION | SECTION | User coded |
| Grouping | PGROUP | User coded | User coded |
| Single process | PSINGLE | SINGLE,<br>MASTER | User coded |
| Early completion | PDONE | User coded | User coded |
| Work distribution | ORDERED PDO | ORDERED | None |
| **Data Environment** | | | |
| Autoscope | None | DEFAULT(PRIVATE)<br>DEFAULT(SHARED) | shared default |
| Global objects | INSTANCE PARALLEL<br>(p+1 instances) | THREADPRIVATE<br>(p instances) | Linker: -Xlocal<br>(p instances) |
| Reduction attribute | None | REDUCTION | REDUCTION |
| Private initialization | None | FIRSTPRIVATE<br>COPYIN | None<br>COPYIN |
| Private persistence | None | LASTPRIVATE | LASTLOCAL |
| **Synchronization** | | | |
| Barrier | BARRIER | BARRIER | mp_barrier() |
| Synchronize | SYNCHRONIZE | FLUSH | synchronize() |
| Critical section | CRITICAL SECTION | CRITICAL | mp_setlock,<br>mp_unsetlock() |
| Atomic update | None | ATOMIC | None |
| Locks | None | Full functionality | mp_setlock(),<br>mp_unsetlock() |

Table 2: Comparison of X3H5, OpenMP and SGI DOACROSS functionality.

```
      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 PI in parallel using OpenMP.

For comparison with message passing, Figure 2 presents the same code example using MPI. Clearly there is additional complexity just in setting up the problem for the simple reason that here we begin outright with a team of parallel processes. Consequently we need to isolate a root process to read and write stdout. Since there is no globally shared data, the we have to explicitly broadcast the input parameters (in this case, the number of intervals for the integration) to all the processors. Furthermore, we have to explicitly manage the loop bounds. This requires identifying each processor (`myid`) and knowing how many processors will be used to execute 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`. Note that the storage for `pi` is replicated across all processors, even though only the root process needs it. As a general rule, message passing programs are more wasteful of storage than shared memory programs [4]. Finally we can print out the result, again making sure to isolate just one process for this in order to avoid printing out `numprocs` messages.

Finally, it is interesting to see how this example would look using pthreads. Figure 3 presents the pthreads version for computing PI. Naturally it's written in C, but we can still compare functionality with the Fortran examples given in Figures 1 and 2.

Clearly the pthreads version is much more complex than either the OpenMP or the MPI versions of the code. The reasons are twofold. Firstly, pthreads is aimed at providing task parallelism, whereas the example is one of data parallelism, i.e. parallelizing a loop. It is evident from the example why pthreads has not been widely used for scientific applications. Secondly, 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 be executing the loop and be able to determine their id's so we can manage the loop bounds. We get the thread number as a command line argument, and then use it to allocate an array of thread id's. At this time we also initialize a lock, `reduction_mutex`, which we'll need for reducing our partial sums into a global sum for PI. The basic approach we use then 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, since there is no implied barriers on threads, we need to explicitly join all our threads before we can print out the result of the integration.

Notice in this example that all the data scoping is implicit. In other words, global variables are shared and automatic variables are thread private. There is no simple mechanism in pthreads for making global variables private. Implicit scoping also is more awkward in Fortran because the language is not as strongly scoped as C. We won't discuss pthreads further in this paper since the limitations

```
      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 PI 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()-tid[0];
/* 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 != 3) {
       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 = %.16f\n", pi);
}
```

Figure 3: Computing PI in parallel using pthreads.

of that model for expressing data parallelism should now be abundantly clear.

### 3.2 Scalability

Although simple and effective, loop level parallelism is usually limited in its scalability because it typically 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 (e.g. loop level versus coarse grained) and the programming model. The type of parallelism exposed in a program is dependent on the algorithm and data structures employed and not on the programming 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 very powerful concept of *orphan* directives that greatly 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 a single parallel region with most of the execution taking place within that single region. In implementing a coarse grained parallel algorithm, it becomes desirable, and often necessary, to have the ability to specify control or synchronization from anywhere inside the parallel region, not from just 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 is highly restrictive to the programmer and makes any non-trivial coarse grained parallel application virtually impossible to write.

## 4 A More Complicated Example

To highlight additional features in the standard we present a slightly more complicated example which computes the energy spectrum for a field. This is essentially a histogramming problem but with a slight twist in that the sequence also is being generated in parallel. The histogramming loop and the sequence generation both could be easily parallelized 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. The variables `field` and `ispectrum` are declared SHARED, and everything else is made PRIVATE with a DEFAULT clause. Note that this does not affect common blocks, so `setup` remains a shared data structure.

```
      parameter(N = 512, NZ = 16)
      common /setup/  npoints, nzone
      dimension field(N), ispectrum(NZ)
      data npoints, nzone / N, NZ /
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(field, ispectrum)
        call initialize_field(field, ispectrum)
        call compute_field(field, ispectrum)
        call compute_spectrum(field, ispectrum)
!$OMP END PARALLEL
      call display(ispectrum)
      stop
      end

      subroutine initialize_field(field, ispectrum)
      common /setup/ npoints, nzone
      dimension field(npoints), ispectrum(nzone)
!$OMP DO
      do i=1, nzone
         ispectrum(i) = 0.0
      enddo
!$OMP END DO NOWAIT
!$OMP DO
      do i=1, npoints
         field(i) = 0.0
      enddo
!$OMP END DO NOWAIT
!$OMP SINGLE
      field(npoints/4) = 1.0
!$OMP END SINGLE
      return
      end

      subroutine compute_spectrum(field, ispectrum)
      common /setup/ npoints, nzone
      dimension field(npoints), ispectrum(nzone)
!$OMP DO
      do i= 1, npoints
        index = field(i)*nzone + 1
!$OMP ATOMIC
        ispectrum(index) = ispectrum(index) + i
      enddo
!$OMP END DO NOWAIT
      return
      end
```

Figure 4: A more complicated example.

Within the parallel region we call `initialize_field()` to initialize the `field` and `ispectrum` arrays. Here we have an example of orphaning of the DO directive. With the X3H5 directives we would have to move these loops up to the main program so 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 there. 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. To compute the spectrum we histogram the field values using the ATOMIC directive to eliminate a race condition 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.

# 5 OpenMP Design Objective

OpenMP was designed to be a flexible standard, easily implemented across different platforms. As mentioned above, the standard is comprised of four distinct parts: control structures, data environment, synchronization, and run time library. In the following sections we briefly describe the design objectives for each of these.

## 5.1 Control Structures

OpenMP strives for a minimalist set of control structures. Experience has indicated that only a few control structures are truly necessary for writing most parallel applications. For example, in the Silicon Graphics 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.

The examples above 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 the code. This is nontrivial for a user to program.

## 5.2 Data Environment

Each process has associated with it a unique data environment that provides a context for execution. The initial process (i.e. at program start up) has associated with it an initial data environment that exists for the duration of the program. New data environments are constructed only for new processes created during program execution. The objects comprising a data environment may have one of three basic attributes: SHARED, PRIVATE, or REDUCTION.

The concept of REDUCTION as an attribute is new in OpenMP. Reduction operations are important to many applications and the REDUCTION attribute allows these to be efficiently implemented by the compiler. This is especially important on cache based systems where the compiler can take care to eliminate any false sharing. On large scale SSMP architectures the compiler also may choose to implement tree-based reductions for even better performance.

OpenMP has a very rich data environment. In addition to the REDUCTION attribute, OpenMP allows PRIVATE initialization with FIRSTPRIVATE and COPYIN, and some limited 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 (in Fortran) directive. In the interest of performance, OpenMP implements a "p-copy" model for privatizing global objects. In other words, 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 from a compound global object. In either X3H5 or the DOACROSS model this would require splitting the compound object. This can be a tedious and error prone process, greatly adding to the programming effort required in parallelizing existing software. OpenMP solves this problem by allowing individual elements of a compound global object to appear in a PRIVATE list.

## 5.3 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 all other control constructs (i.e. DO, SINGLE, MASTER and ORDERED). In the case of DO and SINGLE, the implicit synchronization can be removed with the NOWAIT clause.

Explicit synchronization is specified by the user to manage order or data dependencies. Synchronization is a form of interprocess communication and as such can greatly impact program performance. In general, best performance is achieved by minimizing the synchronization requirements (explicit and implicit) of a program. 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 feature that 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. The implicit barriers in the energy spectrum example could be eliminated and replaced by point-to-point synchronization for the ultimate in performance.

All the OpenMP synchronization directives may be orphaned. As discussed earlier, this is critically important for implementing coarse grained parallel algorithms.

### 5.4 Runtime Library and Environment Variables

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

The runtime functions also allow a developer to specify when to enable nested parallelism. Enabling nested parallelism allows the system act accordingly when it encounters a nested PARALLEL construct. On the other hand, by disabling nested parallelism 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 CPP and with a built in 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 RTL functions where it makes sense. This again is important to application developers who, in addition to creating portable applications also need a portable runtime environment. A standard set of environment variables greatly simplifies the start up scripts for portable applications.

## 6 What does this mean to me?

If you are a software developer, scientist or engineer, in need of portability and parallel performance, OpenMP is the answer. OpenMP will allow you to write portable, scalable programs without abandoning the simple and intuitive shared

memory programming model. If you've already abandoned the shared memory model in the pursuit of portability, OpenMP will allow you to return.

# 7 What happens next?

OpenMP is supported by a number of hardware and software vendors. We expect support to grow.

OpenMP has been designed to be extensible. In the future we expect to add and evolve the standard, e.g. adding nested parallelism and/or shaped arrays. A governing body will be formed to manage OpenMP extensions in the future.

# References

[1] D. E. Lenoski, W. -D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers Inc., San Francisco, 1995.

[2] B. Leasure, ed., *Parallel Processing Model for High Level Programming languages*, Draft Proposed American National Standard for Information Processing Systems, April 5, 1994.

[3] Silicon Graphics, Inc., *MIPSpro Fortran77 Programmer's Guide*, 1996.

[4] J. Brown, T. Elken, J. Taft, *SGI POWER CHALLENGE and POWER CHALLENGEarray in the High Throughput Environment*, Silicon Graphics Inc., 1995.