Scientific Programming 11 (2003) 81–93
IOS Press

How good is OpenMP

Timothy G. Mattson
Intel Corporation, 2800 Center Drive, Mail Stop DP2-226, DuPont, WA 98327, USA
Tel.: +1 253 371 7094

Abstract. The OpenMP standard defines an Application Programming Interface (API) for shared memory computers. Since its introduction in 1997, it has grown to become one of the most commonly used API's for parallel programming. But success in the market doesn't necessarily imply successful computer science. Is OpenMP a "good" programming environment? What does it even mean to call a programming environment good? And finally, once we understand how good or bad OpenMP is; what can we do to make it even better? In this paper, we will address these questions.

1. Introduction

OpenMP [1] is an Application Programming Interface for writing multithreaded programs. It is not a replacement for low-level thread libraries. These libraries are well optimized for writing system level and middleware software. Programmers who need maximum control over the details of how multithreaded software executes should avoid OpenMP and continue to use thread libraries. The overwhelming majority of applications programmers, however, don't need to control the low level details of how threads execute. They need portability and maintainability coupled with convenience so they can hit tight delivery schedules. These people want applications that run fast, give the right answers, and can be affordably coded and maintained. These people are the target audience for OpenMP.

Since its introduction in 1997, OpenMP has grown to become one of the top four API's for expressing concurrency in programs (the others are MPI, POSIX threads, and the threads API used with Microsoft Operating systems). This is a major improvement over the early to mid 90's when there were dozens of programming environments to choose between. Now, one only needs to consider one of the four main environments and can move directly to the job of writing parallel software.

It is interesting that all four parallel programming API's are explicit. In other words, a programmer tells the computer precisely where to introduce parallelism and in most cases, how that parallelism is to be exploited. OpenMP sets itself apart from the other modern parallel programming API's, however, in that the constructs in OpenMP are for the most part semantically neutral compiler directives. This is a key feature of OpenMP. Since a non-OpenMP compiler can just ignore the OpenMP directives without changing a program's semantics, a programmer can adopt a discipline where the OpenMP program is sequentially equivalent; i.e. it is identical semantically to the original sequential code. This feature is one of the reasons OpenMP has been so successful.

While OpenMP has been successful commercially, however, there is much that could be improved. In this paper, we will look at a handful of improvements under consideration for OpenMP. But first, we will review the history of OpenMP and describe the mechanisms in place to drive new ideas into the language. We will then present an informal framework to be used in understanding the models behind OpenMP and how they help us understand the quality of OpenMP. In particular, we will provide an answer to the question "how good is OpenMP".

2. The historical roots of OpenMP

OpenMP was born from necessity. It all started in 1996. SGI had acquired Cray Research Inc. and needed to unify compiler directives for multithreaded programming across the two product lines. At about the same time, Kuck and Associates Inc. (KAI) completed work

ISSN 1058-9244/03/$8.00

on a suite of software tools for directive-driven parallel programming of shared memory machines. These tools were based on their earlier work with the Parallel Computing Forum (PCF) and later the ANSI X3H5 [2] committees. KAI was a small software company and to get the most from these new tools, they needed large markets for their products; ideally spanning the shared memory computing market.

Meanwhile, the scientists within the U.S. Department of Energy's ASCI program at Lawrence Livermore national laboratory were becoming increasingly frustrated. Every time they bought a new shared memory computer, they had to invest time moving to new directive sets. Codes became littered with "ifdefs" and maintenance costs soared. They needed a single API so they could write one code that would run on all shared memory machines.

The ASCI scientists urged SGI (a major vendor to the ASCI program at that time) to standardize compiler directives. SGI joined with KAI and together they started what eventually became OpenMP. Drawing heavily from ANSI X3H5, they created a strawman proposal for OpenMP. With the proposal in hand, they invited other vendors involved with shared memory computing. Intel, IBM, and DEC agreed to join and by late 1997, the full group had created the first version of OpenMP; OpenMP version 1.0 for Fortran.

OpenMP was practically guaranteed to succeed from the beginning. Vendors responsible for the majority of the shared memory computing market were involved with its creation. Within two years, OpenMP grew to become the standard API for programming shared memory computers. Today, it is difficult to find a shared memory computer for which an OpenMP compliant compiler is not available.

From the beginning, it was recognized that OpenMP should be a living language. As hardware changes and new programmers work with the language, OpenMP would need to evolve. To manage the long-term evolution of OpenMP, the OpenMP Architecture Review Board (ARB) was created. At the time this paper is being written, the members of the ARB include ASCI (DOE), Compaq, EPCC, Fujitsu, HP, IBM, Intel, KAI, NEC, SUN, and an OpenMP users group called Compunity.

The ARB agreed upon a collection of goals to guide work on new OpenMP specifications. It is important to keep these in mind anytime extensions to OpenMP are considered. The goals are:

- To produce API specifications that let programmers write portable, efficient, and well understood parallel programs for shared memory systems.
- To produce specifications that can be readily implemented in robust commercial products. i.e. we want to standardize common or well understood practices, not chart new research agendas.
- To whatever extent makes sense, deliver consistency between programming languages. The specification should map cleanly and predictably between C, Fortran, and C++.
- We want OpenMP to be just large enough to express important, control-parallel, shared memory programs - but no larger. OpenMP needs to stay "lean and mean".
- Legal programs under an older version of an OpenMP specification should continue to be legal under newer specifications.
- To whatever extent possible, we will produce specifications that are sequentially consistent. If sequential consistency is violated, there should be documented reasons for doing so.

The specifications produced so far are outlined in Table 1. All specifications are available from the OpenMP web site, www.openmp.org.

The most recent ARB member, Compunity, is an OpenMP users group and deserves special comment. The original membership in the OpenMP ARB was deliberately stacked in favor of vendors. We wanted ARB members to have a stake in OpenMP as a business. We hoped that by doing so, our specifications would be more likely to focus on products that could be immediately implemented rather than research agendas.

History has shown that we made the right decision. For each OpenMP specification, a conforming implementation was available within months of releasing the specification. Letting vendors dominate the ARB, however, created the impression that the ARB is closed and not open to input from the research community. In part, as a response to this criticism, we helped create the Compunity OpenMP users group. Compunity is not just a passive participant in the ARB. They are a full-fledged member of the organization with a full vote in ARB matters. Through Compunity, the academic community and other organizations unwilling to join the ARB can have a hand in shaping the future of OpenMP.

3. What is a "good" programming environment

The computer science literature is full of claims about how good various programming environments are. In most cases, claims of "goodness" are driven by engineering arguments:

Table 1
History of OpenMP specifications

Specification	Year	Description
OpenMP 1.0 Fortran	1997	The first OpenMP specification based closely on the work from X3H5 [2].
OpenMP 1.0 C/C++	1998	A mapping of the original OpenMP 1.0 Fortran specification onto C/C++
OpenMP 1.1 Fortran	1999	An update to the Fortran specification including bug fixes and clarifications. Responds to insights about OpenMP gained during the writing of the C/C++ specification.
OpenMP 2.0 Fortran	2000	A major upgrade of OpenMP to better meet the needs of Fortran95 and to correct some oversights from the earlier specs. The rationale behind OpenMP 2.0 is described in [17]
OpenMP 2.0 C/C++	2002	A major upgrade of the C/C++ specifcation with the goal of making it consistent with the OpenMP Fortran 2.0 specification and ANSI C 1999.

We designed it, we built it, and it worked. Therefore, it's good.

These types of arguments are of little value for advancing computer science. We need a more concrete set of criteria to objectively analyze the quality of a parallel programming environment.

Parallel computing is primarily concerned with performance, so most arguments of "goodness" emphasize performance to the exclusion of programmability. But performance should not be the primary criteria. With the advances in computer hardware over the last few decades, attaining the desired performance, while important, is no longer the supreme issue. Rather, the key issue is the ease with which a programmer can use an API to create high quality software.

This is especially the case for OpenMP. Essentially, for a well-written OpenMP program, the performance is largely determined by underlying threading runtime system. There are differences in particular algorithms used to implement an OpenMP runtime library, but these differences are issues of engineering quality. The best way to address performance issues in OpenMP is by using market forces to pressure the providers of OpenMP technology. This is best done by using benchmark suites such as the EPCC microbenchmark suite [3] or the more recent SPEComp2001 benchmark [4].

Therefore, we avoid performance based arguments in this paper. Instead, we focus on the more difficult issue of the quality of the programming API itself. What makes one programming API "bad" while another one is "good"? The issue comes down to the programming itself. In other words:

The quality of a programming environment is given by its ability to help programmers write high quality, correct code with the least effort.

Programmability is largely avoided in the computer science literature since it is hard to quantify. While quantitative metrics are desirable, we can understand a great deal about the "goodness" of a programming environment qualitatively. Hence, we will move forward with this argument using a qualitative framework to help us answer the question "what makes a programming environment good".

We start with the simple fact that all known programmers are human. The complex process of creating software is fundamentally tied to what goes on inside a programmer's head. So our framework must derive from our understanding of the human computer interface as it applies to software development. In other words, we are interested in the psychology of programming and what it tells us about the quality of different programming environments.

The body of research in the psychology of programming is small. One can come up to speed in this field by reading two short books [5,6] and by studying the proceedings from a series of workshops in which empirical techniques in clinical psychology are applied to understanding programming [7]. Space constraints don't permit us to review the literature here. Instead, we will just describe the results we will use and leave their development to the referenced literature.

Research in cognitive psychology has given us a clear picture of human reasoning. People understand the world by comparing their observations against a set of internally held models. These models are usually informal and change as understanding evolves. One of the benefits of model-based cognition is its support for reasoning from incomplete information. Once enough observations have been acquired to fix the right model and define how a problem maps onto that model, humans are able to reach conclusions even if the available information is incomplete.

Constructing models is an innate human characteristic, and for the most part we are not aware that it is taking place. When trying to bring order to a new field of study, however, or when trying to deepen understanding of an old one, it can be useful to make the models explicit.

In programming, we construct models of a problem and then map them onto abstract models of a computation. These models allow us to address the key issues at each stage of the programming process while

abstracting away non-essential details. Programmers push their models even further and apparently conduct simulations with their models as they design software [8] and choose alternative solutions.

Programming is a complex process. A single model applied over a simple domain cannot address the full range of issues that arise during software development. Hence, researchers investigating this field typically work in terms of a hierarchy of models:

- The top layer of the hierarchy is a specification model, which is a high-level, abstract view of a problem for use by the algorithm. This model implies an overall structure for the algorithm.
- The next layer is a programming model, which is used to map an algorithm onto the constructs of a programming language. In other words, a programming model abstracts how a programming environment presents the parallel computation to the programmer.
- The next layer is a computational model, which provides an abstract view of how a computation takes place on a computer system. It must be general enough to support a wide range of algorithms, but simple enough to support informal mental simulations during the design process.
- The bottom layer is a cost model that incorporates the detailed execution characteristics of real machines. This is the level in the hierarchy where, for example, the costs of access data across levels in a memory hierarchy are addressed.

This hierarchy of models is related to the work of [9] with additional influence from [10] and [11]. It is important to note, however, that there is no single "correct" way to organize the models we use in software engineering.

Theses layers of abstraction and how they relate to the fundamental domains addressed during the programming process are shown in Fig. 1. A programmer begins in the "problem domain". The objects residing in this domain are directly related to the objects in the actual problem being solved. For example, for a molecular modeling problem, the objects in the problem domain would be atomic coordinates and molecular forces. We map from the problem domain into the algorithm domain using the specification model. This high level, problem-dependent and usually informal model is used as we translate the mathematics of the problem into high-level data structures, an overall algorithm structure, and collections of software components.

As we translate those components into code, we need an abstraction of the parallel computation as supported by the target API. This is the job of the programming model. As with the all of our models, they are used during the reasoning process as we bridge from one domain to another - in the case, the algorithm domain into source code.

As the programmer writes code, numerous tradeoffs must be made between different constructs. At this point, some concept of how the program will execute must be addressed. Delving into low level details too early compromises portability, so a good programmer first considers an abstract view of the machine that maps across the full range of intended systems. This abstract view of the machine is handled with the computational model.

As some point in the software development process, the programmer needs to specialize their program to actual systems. In some cases, this is carried out as a separate optimization step at a later date or even by a different team of programmers. Expert programmers, however, consider these low level decisions throughout the software design process. This is done with a low level abstraction called the cost model. Specific performance characteristics of classes of machines or even specific systems are included in these models.

Understanding the models used when designing software and how they fit together provides much of what we need to understand the programming process. To finish the picture, we need to understand the strategy used by expert programmers as they work with these models.

As programmers move from the specification of the original problem, to source code, to optimization; they need to make progress at each level of the hierarchy. We are taught that good programmers work sequentially from top to bottom, starting at the specification level and working through the abstractions to the cost model. Empirical studies of actual programmers, however, show that a completely different approach is used [12]. It is well established in the literature that programmers bounce around between levels working at which ever level is most productive at a given time. This is called the Opportunistic refinement strategy of software development.

With our models and a core strategy in place, we can now state what distinguishes a good programming environment from poor one. The quality of an application programming interface is given by how well it supports effective programming for the target programming community. In our case, we will evaluate an API in terms of:

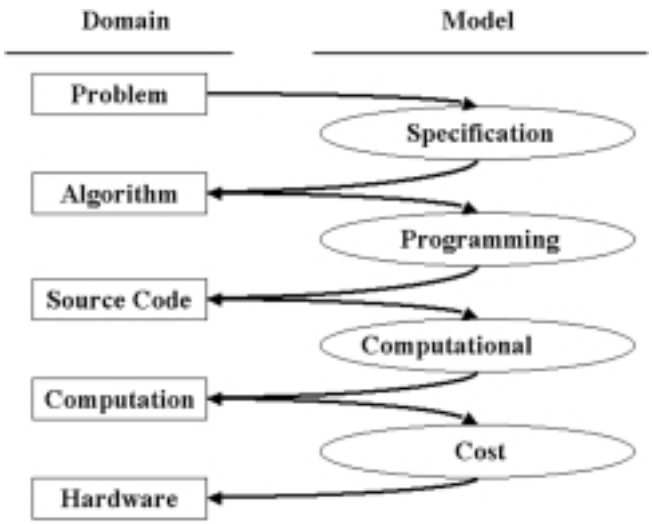

Fig. 1. The problem domains and how the models used in programming relate to them.

1. The models associated with the API must be accessible-to and helpful-for the target programming audience.
2. Transitions between models must be convenient, and clear; i.e. abstract the machine, don't hide it.
3. The programmer must be in control of where they are working within the in the hierarchy of models at a given time [13].
4. An API should make commonly used programming idioms convenient, but without cluttering the language.

4. Levels of abstraction in OpenMP

Now that we've established our framework for understanding API's, we'll apply it to OpenMP. We will start at the top (the specification model) and work our way to the lowest level layer (the cost model).

Specification model: OpenMP does not formally define a single specification model. OpenMP is a general API and hence supports a range of specifications models. However, we did have a couple broad specification models in mind as we created OpenMP. OpenMP was first and foremost targeted to array-driven scientific and engineering applications. Hence, we envisioned a task parallel specification model for which the tasks would correspond to loop iterations. While not our original target, it is clear that OpenMP is also effective for SPMD or Single Program Multiple Data or SPMD models [14].

Programming model: OpenMP is based on a fork/join programming model. This is displayed in Fig. 2. An OpenMP program begins life as a regular sequential program. This single thread of control is called the master thread. At some point in the program's execution, the master thread encounters tasks that can execute concurrently. At this point, the executing program forks a number of threads. These threads constitute a team and they execute in parallel across a set of statements called a parallel region. At the end of the parallel region, the team of threads terminates (joins) and the original, master thread continues.

Later on, additional opportunities to exploit concurrency might arise. At these points, the OpenMP programmer can cause additional teams to fork and execution across parallel regions proceeds much as we described before.

The fork/join programming model is the only model explicitly defined in the OpenMP specification. Other models are involved with OpenMP, but these models are implied.

Computational model: OpenMP implies a very simple computational model. OpenMP assumes that the machines OpenMP programs run on are symmetric multiprocessors. In other words, the OpenMP programming model assumes the computer consists of a single, shared address space that is available to each thread with equal-time access from each processor. OpenMP programmers usually assume sequentially consistent memory, but the specifications don't require this or any other specific memory consistency model.

Note that it is possible to map OpenMP programs onto non-uniform address spaces [15] or even clusters

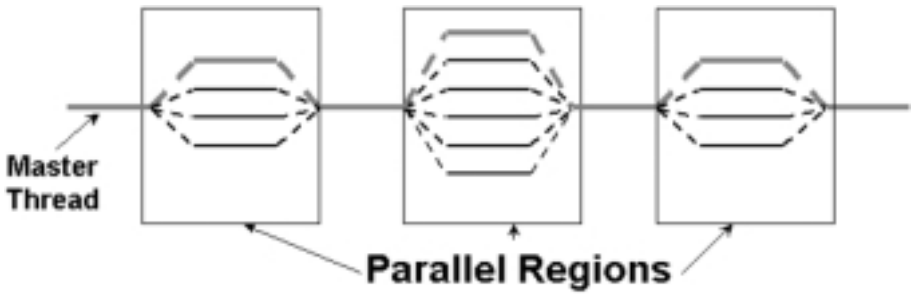

Fig. 2. OpenMP Programming model showing master thread running in sequential regions and parallel regions with multiple concurrent threads.

[16]. But this is done with extensions to the specification or severe performance restrictions the programmer must manage.

Cost Model: OpenMP does not define an abstract machine or how that machine maps onto real systems. Omitting a cost model was not due to laziness on the part of the language designers. In order to map OpenMP programs onto the largest range of systems without providing undue advantage to any one architecture, the language definition intentionally omitted a cost model.

5. How good is OpenMP?

Now that we understand the models used in OpenMP, we can ask the question, "how good is OpenMP?" We will do this by considering the criteria presented earlier.

– The models associated with the API must be accessible-to and helpful-for the target programming audience.

The target audience for OpenMP is general-purpose application programmers. In particular, we wanted to reach programmers who would not use lower level API's such as Pthreads or MPI. For the models we define within OpenMP – the programming and computational models – we were successful. They are indeed simple, accessible and helpful to our target programmers. The fork-join programming model, based on success teaching OpenMP over the last five years, is easy to learn and easy to apply when understanding a parallel algorithm. The shared memory model of computation implied by OpenMP is also an effective model that is familiar to most programmers.

Weaknesses for OpenMP emerge, however, when you consider the models we don't define. For example, we didn't define a cost model. Programmers must construct these models on their own – even if they only do so informally. Second, OpenMP's specification models are not general enough. For the loop-based codes common in scientific programming, OpenMP is highly effective. But for more general algorithms based, for example, on pointer-following or more complex structures, the OpenMP specification model is too narrow. Finally, the memory consistency model is ill defined in OpenMP. This leads to a great deal of confusion concerning the flush construct and when it must be used.

We consider the next two criteria together.

– Transitions between models must be convenient and clear; i.e. abstract the machine, don't hide it.
– The programmer must be in control of where they are working within the hierarchy of models at a given time.

Mapping from the fork-join model to the OpenMP model of computation is immediate and takes little effort on the part of the programmer. This is one of the major reason's OpenMP is so easy to learn. A programmer familiar with modern multithreaded operating systems has no problem understanding how to map the fork-join model onto the standard computational model for shared memory systems.

When a programmer's problem fits the OpenMP specification model, mapping onto the OpenMP programming model is straightforward as well. Once again, the biggest problem is with the cost model. Since OpenMP doesn't define a cost model, transitions onto this model is challenging and requires substantial effort on the part of the programmer.

– An API should make commonly used programming idioms convenient, but without cluttering the language.

In scientific applications, loop level parallelism is a very common programming idiom. OpenMP was specifically designed to handle these algorithm struc-

tures. With a simple, semantically neutral directive/pragma, an OpenMP programmer can create a team of threads and schedule loop iterations onto those threads. The solution in OpenMP is elegant and syntactically minimal.

The OpenMP specification also works well for SPMD programs [14]. The programming idioms required by SPMD programs are sparse; essentially all you need is the concept of a thread ID combined and a way to discover how many threads are present. The key to enabling SPMD programs is orphaning; the embedding of OpenMP constructs in compilation units that do not contain a surrounding parallel construct. Orphaning lets the programmer put a Parallel construct at the top of a program, an end parallel construct at the end of the program, and all other OpenMP constructs littered across multiple compilation units. In many cases, only minimal restructuring of the sequential code is required.

As you move away from loop structured codes or SPMD programs, however, OpenMP becomes increasingly difficult to apply. For example, OpenMP has a tough time with simple pointer following algorithms. While OpenMP supports loop-structured programs that use a simple *for* loop, *while* loops or *for* loops with dynamic loop indices are not easily handled in OpenMP.

Putting all these comments together, we see that the quality of OpenMP is mixed. For the algorithm structures we anticipated, OpenMP is a very effective programming environment, i.e. "it is good". For optimization, where you need to map onto a cost model, OpenMP is weak.

6. Making OpenMP even better

The OpenMP community is steadily working to improve OpenMP. We have two major venues for identifying improvements to the language. First, we hold workshops on OpenMP each year at locations around the world (go to www.compunity.org to learn more about future OpenMP workshops). We include an open discussion on the future of OpenMP at these workshops. Another source of input to OpenMP's evolution is the ARB futures committee. This committee of ARB members meets regularly to work out the details for changes to the OpenMP language.

In this section, we will discuss some of the changes that have been considered for OpenMP. The sources for these ideas are not always listed since much of this comes from random conversations over the last few years.

We will divide these changes into the following categories:

- Changes to make OpenMP more convenient for our target programmers.
- Increasing the range of the specification models supported by OpenMP.
- Making the cost model more visible in portable OpenMP programs.

We will consider each of these in the following subsections.

6.1. Making OpenMP more convenient

Ease of use is a high priority in OpenMP. Software engineers need to write quality software under tight schedules. If we want them to write parallel programs, then we need API's that are convenient to use and fit in well with good practices in software engineering. To address this goal, we went to great pains to make the constructs in OpenMP easy to understand. Most interactions between OpenMP constructs are straightforward and easy to explain.

Another goal with OpenMP is to minimize the number of source-code transformations when parallelizing a program. The bulk of OpenMP consists of semantically neutral compiler directives. In most cases, OpenMP programmers can write programs that are identical for sequential and concurrent readings.

6.2. Parallelize loop nests:

When dividing the iterations of a loop among many threads, it is important to have a large enough number of loop-iterations to (1) provide enough concurrency to support the number of concurrent threads and (2) to compensate for the parallel computing overhead. Many algorithms include nested loops that span a grid or some other regular data structure. In some cases, a single loop has too few iterations, but if the nested loops were merged into a single larger loop, there would be plenty of parallelism.

This is a common trick in parallel computing called loop coalescing. A programmer can do this by hand as shown in Fig. 3. In the first block of code in figure 3 we have two loops over N. Assume N is modest in size relative to the number of threads. If the body of the loop is independent in both loop indices, a programmer can rewrite the two loops into a single, much larger loop.

```
#pragma omp parallel for private (i,j)
 for(i = 0, i< N, i++) {
     for (j = 0; i<N; j++) {
     ! loop body that uses i and j
    }
 }
```

```
 #pragma omp parallel for private(ij, i, j)
 for(ij = 0; ij< N*N; ij++){
   i = ij / N;
   j = ij% N;
     ! loop body that uses i and j
 }
```

```
#pragma omp parallel for private(i,j) collapse(2)
for(i = 0, i< N, i++) {
   for (j = 0; i<N; j++) {
     ! loop body that uses i and j
    }
}
```

Fig. 3. Proposed constructs to coalesce loops; top panel, original code; second panel, transformation by hand; third panel, a proposed construct to coalesce the loops and share the loop-iterations among a team of threads.

This is shown in the second block of code in Fig. 3. Every time a programmer makes a chance to a program, there is a chance of errors being introduced. Hence, it would be nice if this program could be parallelized without any re-write at all.

This is the case for the third version of the program. We've added an extra clause,

collapse(2)

This clause tells the compiler to introduce the loop coalescing transformation over the immediately following two loops. This approach is very similar to that used in the compilers from SGI. In this case, the loops to be coalesced are indicated by a list of the indices that will be merged:

nest(i,j)

Further work is needed to decide how far we can push this construct. We believe transformations are well defined so compilers can handle triangular loop nests. Imperfectly nested loops, however, in which intervening statements occur between the loop statements, are more difficult to handle. In order to avoid complex rules for when imperfect nesting can be supported, we might initially drop support for this case. After we have more experience with nesting loops, we may figure out how to deal with imperfectly nested loops. In that case, we'd add support for imperfectly nested loops at a later date. This is typical of our approach to OpenMP specifications. It's much easier to extend a construct than to fix a poorly defined one. We'd rather go slow and get it right, then move too fast and clutter the language with awkward constructs.

6.3. Automatic scooping:

One of the most error-prone steps in creating an OpenMP program is deciding which data should be private and which data should be shared. For large blocks of code split between multiple compilation units, the analysis required to correctly determine the scope of data can be very difficult. If an error is made, the resulting program could contain race conditions. When there is a race condition, the program will produce different results depending on the scheduling of the threads. The program may work some of the time if the threads happen to be scheduled in the "correct" order. At other times with different thread schedules, however, the program may give erroneous results. This unpredictability makes race conditions particularly dangerous.

It has been proposed that the OpenMP specification be amended to include constructs that let programmers ask a compiler to automatically decide how data should be shared [18]. The idea is to add a new clause to OpenMP:

default(automatic)

If the compiler can safely decide how to share data, the compiler will do so and activate the OpenMP construct in question. If the compiler cannot safely determine the appropriate scope of the variables, the construct would be serialized (i.e. executed with one thread).

For example, consider the SPEComp'2001 benchmark AMMP [4]. A pragma taken from this program is shown in the first panel in Fig. 4. The chances of

```
#pragma omp parallel for private (n27ng0, nng0, ing0, i27ng0, \
 natoms, ii, a1, a1q, a1serial, inclose, ix, iy, iz, inode, \
 nodelistt, r0, r, xt, yt, zt, xt2, yt2, zt2, xt3, yt3, zt3, \
 xt4, yt4, zt4, c1, c2, c3, c4, c5, k, a1VP , a1dpx , a1dpy, \
 a1dpz , a1px, a1py, a1pz, a1qxx , a1qxy , a1qxz ,a1qyy , \
 a1qyz , a1qzz, a1a, a1b, iii, i, a2, j, k1, k2 ,ka2, kb2, \
 v0, v1, v2, v3, kk, atomwho, ia27ng0, iang0, o) schedule(guided)
```

```
#pragma omp parallel for scope(automatic) schedule (guided)
```

Fig. 4. The proposed Automatic scope clause. The top panel is a parallel for construct from the AMMP program; a molecular dynamics code from the Spec OMP benchmark suite [4].

producing such a complex private-list without introducing errors is slim. In the second panel, we show what the code could look like given a compiler that can automatically decide data scope.

In order for this to be acceptable as an OpenMP construct, we must provide well-defined semantics so compilers that cannot determine variable-scope can still conform to the specification. We believe it will be sufficient to allow a compiler to serialize the constructs in this case. When this happens, however, we expect high quality compilers to list the variables that didn't permit automatic data scope assignment.

Another common pattern is for all the scalar variables in a loop to be private while all arrays are shared. We could handle this case with a new clause:

default(mixed)

Unlike the automatic scope case, this construct is explicit and it would be up to the programmer to make sure this is a safe choice.

6.4. Expanding the range of specification models

OpenMP has been very successful for problems that map onto the loop-level parallelism specification model. This case is common for the array-dominated algorithms found in many scientific and engineering codes.

As you move away from scientific computing, however, a larger range of algorithms are encountered. Codes may loop over linked lists or have recursive structures. These codes do not map well onto OpenMP. If OpenMP is to have impact outside science and engineering, the specification must grow to include a larger range of algorithm structures.

We are concerned about keeping OpenMP simple and are hesitant to let it grow too rapidly. Hence, we only have one construct under consideration that expands the range of specification models.

6.5. Work queues:

The work queue is a flexible way to define work to be shared between a team of threads. In a work queue, the work to be carried out is packaged up into a task. These tasks are placed on a queue. The threads in a team pull tasks off the queue as the threads become free, carry out the indicated work, and then go back to the queue for additional tasks. This continues until the queue is empty and no more tasks are being placed on the queue.

How these tasks are defined is extremely flexible. Unlike iterative worksharing constructs, the creation of the work on the queue does not constrain the format of a loop's structure. Unlike a sections clause, the creation of the work can be dynamic: changing dramatically as the runtime conditions shift. If we are very careful, it should even be possible to use a work queue model with recursive programs.

There is broad agreement within the OpenMP community that some sort of work queue construct is required. As you move beyond this simple high-level model, however, the agreement ends. How should the work queue function with orphaning? Should it be possible to nest work queues (i.e. place work queues inside work queues)? How much new syntax do we need to support this concept?

We will consider the work queue proposal from KAI [19] in this paper. This construct was implemented within the C version of KAP/Pro from KAI and is currently available in the C compiler from Intel (http://developer.intel.com/). This existence of reference implementations is important to the OpenMP ARB. With very few exceptions, we will only standardize constructs for which reference implementations exist.

The work queue construct has two components: *taskq* and *task*. This is directly analogous to the famil-

iar OpenMp construct *sections* and *section*. When a team of threads encounters a *taskq* construct, the program shifts to "single thread" semantics. This means that the statements execute as if one thread from the team is executing. When this single thread encounters a *task* construct, the structured block within that construct defines a unit of work that is placed on the queue. The single thread continues execution; encountering *task* constructs and filling the queue until the end of the *taskq* construct is encountered.

Meanwhile, the other threads in the team wait until work appears in the queue. As work appears, the threads on the team pull work off the queue and continue until the queue is empty and closed.

An example will clarify the operation of these constructs. In the top panel of Fig. 5, we show a simple pointer following loop. We assume a linked list has been created the elements of which are of type *nodeptr. The members of nodeptr include a link to the next element of the list and two links to values that define a computation. Assume these calculations are completely independent.

In the second panel of Fig. 5, we show how this loop can be parallelized with a work queue. The first pragma forks the team of threads and creates the task queue. In most OpenMP constructs, all threads participate in the execution of program statements. The *taskq* construct, however, is different. In this case, the program executes with *single thread* semantics. One thread executes the code within the *taskq* outside of the enqueued *tasks* themselves. Hence, one thread executes the for-loop to traverse the linked list. When this thread encounters the *task* construct, the included structured block is placed as a unit of work on the task queue. This single thread continues execution until the end of the task queue construct is reached.

As work appears on the task queue, the other threads in the team pull a task off the queue, they do the assigned work, and then return to the queue for additional tasks. The data environment can be complicated. Basically, the taskq defines the environment for the threads. Any data modified within a unit of work must be shared among the team of threads or private to each task. Working with data private to a thread could result in race conditions since there is no control over which threads process which tasks.

When threads finish their work and the queue is empty, they wait on a barrier at the end of the taskq. As with any workshare construct, nowait clauses can be used to turn off this barrier and allow threads to proceed immediately beyond the end of the task queue.

This approach for the work queue construct is straightforward and deceptively simple. If it were really this simple, however, work queues would have made it into an earlier OpenMP specification. When you look more deeply into work queues, there are a number of tough issues that must be resolved.

- The tasks are placed in the queue with no concept of order. This may limit algorithms where a precedence relation exists between the tasks.
- For recursive programs, should there be one global task queue with orphaned task constructs within the recursive call tree? Or should tasks be required to occur lexically within a task queue with a hierarchy of task queues spanning the recursive call tree?
- Should we represent the work queue semantics by extending the syntax of the SECTIONS/SECTION construct? Or should we create syntactic elements to represent work queues?

These questions and more are under active discussion within the OpenMP ARB futures committee. It is likely that consensus on the ARB will be reached and this construct will appear in the next OpenMP specification.

6.6. Exposing cost models

The OpenMP standard does not define a formal cost model. For SMP computers, this might be an acceptable option. For computers that include memory or processor hierarchies, the lack of a well-defined cost model complicates program optimization.

Currently, programmers either tolerate diminished performance or utilize system dependent (and hence, non-portable) constructs to tune the program for a specific machine. This is a problem for OpenMP. A key attraction of OpenMP is its portability. If non-portable constructs are needed to make programs run efficiently, portability is compromised and the effectiveness of OpenMP is diminished.

It is not clear what should be done to address this problem. There are many options – including leaving the language alone and requiring the operating system in partnership with the OpenMP runtime library to make OpenMP programs run fast. It is most likely, however, that some changes will be needed in the OpenMP specifications. These changes involve exposing the memory hierarchy and how threads map onto that hierarchy. In other words, we need to (1) imply a cost model that includes a memory and/or proces-

```
nodeptr list;
for (nodeptr p=list; p!=NULL; p=p->next){
  work(p->right);
  work(p->left);
}
```

```
nodeptr list;

#pragma omp parallel taskq
for (nodeptr p=list; p!=NULL; p=p->next){
  #pragma omp task
  work(p->right);
  #pragma omp task
  work(p->left);
}
```

Fig. 5. The top panel shows a pointer following loop. Existing OpenMP specifications do not provide reasonable ways to parallelize such a loop even if the processing of each member of the list can be carried out independently. The work queue proposal shown in the second panel is a proposed mechanism to make processing of such loops easy to express within OpenMP.

sor hierarchy and (2) add constructs to the language to support this model.

We will consider a minimalist solution to this problem. An informal cost model can be implied based on an abstract view of a non-uniform memory architecture (NUMA) computer. In a NUMA computer, processors are organized around a memory hierarchy. The operating system provides a single address space accessible to all processors. Memory locality is managed by the operating system in terms of pages that can be mapped around the machine.

Different machines use different mapping strategies for the memory pages. Typically, they use a first touch strategy in which the first processor to access a page gains local access to that page. This strategy may be amended to move pages on the next access, replicate pages when multiple processors work with them, or remap them when the processor most frequently accessing a page changes.

We can therefore address the needs for most machines if the programmer first "touches" data inside the parallel region. By doing so, the first touch strategy spreads the pages out among the processors. Well-optimized programs can then be written if the processing can be kept close to the pages it uses.

This strategy has been explored in the Nanos threads project [20] and found to work quite well. To work best, however, two extra constructs are needed in OpenMP.

6.7. Thread affinity

To get the most performance, it is essential to maximize reuse of data close to a processor. This includes data in processor caches as well as the pages in a NUMA computer. For example, in scientific and engineering applications, programmers go to great lengths to block structure their algorithms to support better cache reuse.

On a shared memory system, however, the operating system will migrate threads to optimize the load balancing for the system. Once a thread is migrated to a different processor, all of a programmer's hard work to optimize data layout is wasted.

To address this problem, we need a portable way to ask the system to turn on or off thread migration. Consistent with existing OpenMP constructs to support dynamic mode or nesting of parallel regions, we need an environment variable and two runtime library routines:

OMP_THREAD_AFFINITY
int omp_get_thread_affinitiy()
int omp_set_thread_affinitiy(int)

If set to "logical true" the OS will attempt to lock threads to processors; i.e. thread migration will be turned off. Also in line with the rest of the OpenMP specification, if the OS is unable to provide this functionality, it can silently fail to do so.

6.8. Reusable loop schedules

When using a "first touch" page migration strategy to distribute the data at the beginning of a program, it is important that the same thread access the same blocks of data as the program moves from one loop to another. To support this, you need the same schedule to be used for each work-shared loop.

The Nanos project at UPC in Barcelona [20] has proposed a number of options to support this tech-

Table 2
Summary of OpenMP enhancements discussed in this paper

Modification category	Proposed extension to OpenMP
Make OpenMP more convenient	Parallelize nested loops Automatic data scooping – default(auto) and default(mixed)
Expand range of Specification models used with OpenMP	Work queues
Expose costs models in portable programs	Thread Affinity Reusable Loop Schedules

nique. One particularly interesting technique is to name a schedule and then in a later loop, reuse the schedule. For example, the schedule clause could have an additional argument to supply an optional name:

SCHEDULE(type, chunk) NAME(name)

Once defined, the programmer could specify that other loops schedule loop iterations onto threads in the same way by specifying:

SCHEDULE(name)

As long as the same threads continue to access the same pages, the combination of turning off thread migration, touching data in parallel to distribute pages, and then reusing schedules may provide enough information about the underlying NUMA machine to meet out needs.

7. Conclusion

How good is OpenMP? In this paper, we've discussed a framework to help us answer this question. We did this by looking at the core models behind OpenMP and how programmers use them as they develop software. Our conclusion? For the target audience and target applications and systems that reasonably approximate a symmetric multiprocessor architecture, the OpenMP models are well matched to programmer needs. Therefore, OpenMP is "good". As you move away from the target applications, however, or the system has significant non-uniformities, OpenMP has some serious problems.

Fortunately, the OpenMP language designers didn't "go away" after they created the specifications. The group is continuously working to enhance OpenMP and grow the range of applications that can use OpenMP. This group is called the OpenMP Architecture Review Board (ARB). Interested parties can suggest enhancements for OpenMP through the ARB web site (www.openmp.org) and in person at OpenMP workshops held around the world each year (see www.compunity.org for information about OpenMP workshops).

The ARB works on future enhancements to the language through its futures committee. This committee has a large number of new constructs under consideration. We have not provided an exhaustive review of the work of this committee in this paper. Rather, we have discussed a handful of constructs and showed how they mapped onto our framework for understanding OpenMP. The constructs discussed in this paper are summarized in Table 2.

In the future, we'd like to expand the approach used in this paper and consider a more detailed framework for accessing the quality of OpenMP. In particular, we are interested in working with Green's Cognitive dimensions Framework [21]. This framework was designed to with visual programming environments in mind. It would be interesting to amend the framework for parallel programming and apply it to OpenMP, MPI, and Pthreads. The results from this analysis would help us understand additional enhancements to future versions of OpenMP.

Acknowledgements

The OpenMP enhancements described in this paper emerged during discussions at the OpenMP workshops held each year around the world and during meetings of the OpenMP ARB futures committee. I couldn't possibly list everyone who contributed to these discussions. Certain people, however, have had a major impact on my thinking about the future of OpenMP and deserve special mention: Sanjiv Shah and Paul Petersen of Intel Corp., Larry Meadows of SUN Microsystems, Jesus Labarta and Eduard Ayguade of UPC in Barcelona, and Matthijs van Waveren of Fujitsu Ltd.

References

[1] All OpenMP specifications are available at www.openmp.org.
[2] American National Standards Institute, Parallel Extensions for Fortran, Technical report X3H5/93-SDI revision M. Accredited Standards Committee X3, April 1993.
[3] M. Bull, www.epcc.ed.ac.uk/research/openmpbench/.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.G. Jones and B. Parady, SPEComp: A new Benchmark suite for Measureing Parallel computer Performance, In Proc. of WOMPAT 2001, Workshop on OpenMP applications and Tools, *Lecture Notes in Computer Science* **2104** (July, 2001), 1–10.

[5] *Psychology of Programming*, edited by J.-M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore, Academic press, 1990.

[6] F. Defienne, *Software Design – Cognitive Aspects*, Springer, 2002.

[7] Empirical Studies of Programmers, Proceedings usually published by ACM Press. 1986, 1987, 1991, 1993, 1996, 1997.

[8] [Guindon90] R. Guindon, Knowledge exploited by experts during software system design, *Int. J. Man-machine Studies* **33** (1990), 279–304.

[9] B.M. Maggs, L.R. Matheson, R.E. Tarjan, Models of Parallel Computation: A Survey and Synthesis, *Proceedings of the 29th Hawaii International Conference on system sciences* **2** (1995), 61–70.

[10] D.K.G. Campbell and S.J. Turner. CLUMPS: A model of efficient general-purpose parallel computation, *Proceedings of the IEEE TENCON Conference* **2** (1994), 723–727.

[11] W.F. McColl, Bulk Synchronous Parallel Computing, *Second Workshop on Abstract Models for Parallel Computation*, Oxford University Press, 1993

[12] W. Visser, More or less following a plan during design: Opportunistic deviations in specification, *International Journal of Man-machine studies*, 1990.

[13] M. Petre, Expert Programmers and Programming Languages, in: *Psychology of Programming* J.-M. Hoc. T.R.G. Green, R. Samurcay and D.J. Gilmore, eds, Academic press, 1990.

[14] G. Krawezik, G. Alleon, and F Cappello, "APMD OpenMP versus MPI on a IBM SMP for 3 Kernels of the NAS Bench-Proceedings 4th International Symposium, ISHPC marks, in 2002, *Springer Lecture Notes in Computer Science* **2327** (May, 2002), Kansai City Japan.

[15] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta and E. Ayguade, Is data distribution necessary in OpenMP? *Proceedings Supercomputing*, IEEE Press, 2000.

[16] K. Kusano, S.Satoh and M. Sato, Performance Evaluation of the Omni OpenMP Compiler, *Proceedings 3rd International Symposium on High Performance Computing, Lecture Notes in Computer Science, Number 1940*, Springer, 2000 pp. 403–414, Tokyo Japan.

[17] T.G. Mattson, An Introduction to OpenMP 2.0, *Proceedings 3rd International Symposium on High Performance Computing, Lecture Notes in Computer Science, Number 1940*, Springer, 2000, pp. 384–390, Tokyo Japan.

[18] Dieter an May, email proposal to the OpenMP ARB futures committee, September 2001.

[19] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop, Flexible Control Structures for Parallelism in OpenMP, *Proceedings of the First European Workshop on OpenMP*, 1999.

[20] E. Ayguade and J. Labarta , www.cepba.upc.es/nanos

[21] T.R.G. Green and M. Petre, Usability Analysis of Visual Programming Environments: a cognitive dimensions framework, *J. Visual Languages and Computing* **7** (1996), 131–174.