Overview

OpenMP (Open Multi-Processing) is a popular open-source concurrency platform that supports multithreaded programming through Fortran and C/C++ language pragmas (compiler directives). Support for OpenMP is available in several popular compilers. By inserting pragmas into the code, the programmer identifies the sections of code that are intended to run in parallel.

OpenMP was designed to support parallel operation on arrays -- popular in many numerical applications. The loop-oriented nature of that design made OpenMP (versions 2.5 and earlier) unable to handle the kind of unbalanced parallelism that characterizes many modern problems. A new specification (OpenMP 3.0) attempts to address this deficiency by adding various task structures to the compiler extensions.

Consider the following C++ OpenMP code snippet which sums the corresponding elements of two arrays:

```cpp
#pragma omp parallel for
for (i=0; i<n; ++i) {
    c[i] = a[i] + b[i];
}
```

The pragma indicates to the compiler that the iterations of the loop that follows can run in openmp loops parallel. The loop specification must obey a certain set of patterns in order to be parallelized, and OpenMP does not attempt to determine whether there are dependencies between loop iterations. If there are, the code has a race. An advantage in principle to the pragma strategy is that the code can run as ordinary serial code if the pragmas are ignored. Unfortunately, some of the OpenMP directives for managing memory consistency and local copies of variables can cause the behavior of a serially executing OpenMP program to deviate from its behavior when compiled and executed as a C/C++ program. Thus, if you use these directives, you must redebug your OpenMP program for serial correctness, as well as debug it for parallel correctness. Moreover, the advantage of having a single source for both C/C++ and OpenMP is mitigated, since the two implementations may operate using different program logic, unless the code avoids these pragmas.

OpenMP schedules the loop iterations using a strategy called work sharing. In this model, parallel work is broken into a collection of chunks which are automatically farmed out to the various processors when the work is generated. Since work-sharing induces communication and synchronization overhead whenever a parallel loop is encountered, the loop must contain many iterations in order to be worth parallelizing.

Although OpenMP was designed primarily to support a single level of loop parallelization, alternating between serial sections and parallel sections, as illustrated on the right, some implementations of OpenMP support nested parallelism. The work-sharing scheduling strategy is often not up to the task, however, because it is sometimes difficult to determine at the start of a nested loop how to allocate thread resources. In particular, when nested parallelism is turned on, it is common for OpenMP applications to "blow out" memory at runtime because of the inordinate space demands. (The latest generation OpenMP compilers have started to address this problem.)

Summary: Cilk++ vs. OpenMP

If your code looks like a sequence of parallelizable Fortran-style loops, OpenMP will likely give good
speedups. If your control structures are more involved, in particular, involving nested parallelism, you may find that OpenMP isn't quite up to the job:

- Cilk++ has nested parallelism that works and provides guaranteed speed-up. OpenMP has nested parallelism, but can be a "memory hog" and unreliable.
- Cilk++ guarantees space bounds. On P processors, Cilk++ uses no more than P times the space of a serial execution. In OpenMP, not so.
- Cilk++ has a serial semantics. With OpenMP, you do not have this benefit - only a subset of OpenMP supports serial semantics.
- Cilk++ has a solution for global variables (a construct we invented called "hyperobjects"). OpenMP does not.

(A "deeper dive" of OpenMP's pitfalls and how to avoid them is available in this paper from the team at the University of Kassel.)

**Ease of Use**

Based on a reading of the [OpenMP 3.0 specification](http://www.openmp.org/resources/specification/), Cilk++ clearly maintains the advantage of simplicity.

Compare the Cilk++ language extensions...

- cilk_spawn
- cilk_sync
- cilk_for
- #pragma cilk_grainsize
- reducer <=
- set_worker_count()

...with a subset (!) of variables, directives and pragmas that you need to learn to use OpenMP 3.0:

- **Internal Control Variables (ICVs)**
  - dyn-var
  - nest-var
  - nthreads-var
  - thread-limit-var
  - max-active-levels-var
  - run-sched-var
  - def-sched-var
  - stacksize-var
  - wait-policy-var

- **Constructs and clauses**
  - parallel construct
  - num_threads clause
  - schedule clause
  - Loop construct

- **Interface routines**
  - omp_set_num_threads
omp_get_max_threads
omp_set_dynamic
omp_get_dynamic
omp_set_nested
omp_get_nested
omp_set_schedule
omp_get_schedule
omp_get_thread_limit
omp_set_max_active_levels
omp_get_max_active_levels

- **Environment variables**
  - OMP_SCHEDULE
  - OMP_NUM_THREADS
  - OMP_DYNAMIC
  - OMP_NESTED
  - OMP_STACKSIZE
  - OMP_WAIT_POLICY
  - OMP_MAX_ACTIVE_LEVELS
  - OMP_THREAD_LIMIT

And consider how to use just one of the constructs:

From the [OpenMP specification](http://sd-10807.dedibox.fr/show_items-feed=9b25aba75fb3d5442...):

The syntax of the parallel construct is as follows:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
```

where `clause` is one of the following:

- structured-block
- if(scalar-expression)
- num_threads(integer-expression)
- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- copyin(list)
- reduction(operator: list)

### Performance

OpenMP was designed to support parallel operation on arrays. The loop-oriented nature of that design made OpenMP (versions 2.5 and earlier) unable to handle the kind of unbalanced parallelism that characterizes many modern problems. A new specification (OpenMP 3.0) attempts to address this deficiency by adding various task structures to the compiler extensions.

Cilk++ has clear advantages over the limited parallel constructs available in OpenMP 2.5. In fact, a paper describing OpenMP 3.0 by the OpenMP tasking subcommittee highlights the problems quite well...
Many applications, ranging from document-based indexing to adaptive mesh refinement, have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. Irregular parallelism in these applications is often expressed in the form of dynamically generated units of work that can be executed asynchronously. The OpenMP Specification Version 2.5, however, does not provide a natural way to express this type of irregular parallelism, since OpenMP was originally "somewhat tailored for large array-based applications."

This paper compares a prototype implementation of OpenMP 3.0 with the 15-year-old MIT Cilk technology, and shows that the two are very close in performance for the problems chosen by the authors.

We have not had the opportunity to benchmark Cilk++ against a production version of OpenMP 3.0. Our experiences with OpenMP 2.5 confirm the conclusion in the paper, that OpenMP 2.5 is inadequate for dealing with unbalanced parallelism.

**Reliability**

- Cilk++ is bundled with Cilkscreen, a powerful tool that includes guaranteed race detection. Because Cilkscreen understands the parallel Cilk++ constructs in your program, Cilkscreen offers a very strong guarantee: "Cilkscreen will report any data races that are exposed by a test run of your program. In other words, if there is an possible task schedule on any number of processors that could ever result in a different result, Cilkscreen will report it." We know of no tool that can deliver these guarantees to OpenMP developers.

- Cilk++ has a solution for global variables (a construct we invented called "hyperobjects"). OpenMP attempts to address this problem with the reduction clause. While the general concept is similar, the implementation and features do vary. Not all reducers are created equal, and depending on what is being asked of them, it may be worth looking under the hood a bit. Cilk++ reducers are considerably more powerful: They feature lower overhead; deterministic ordering; are usable in general control structures rather than just loops; and are extensible - Cilk++ provides a hyperobject library containing many commonly used reducers - and you may also write your own.

**What are YOUR impressions?**

- Have you built parallel applications with OpenMP?
- What has your experience been, in terms of ease of use, performance, reliability?
- What have we missed? Are there other important considerations?
Demystifying Persistent OpenMP Myths - Part I

By Ruud on Feb 28, 2009

Unfortunately, the September 5, 2008 blog titled "The OpenMP Concurrency Platform" written by Charles Leiserson from Cilk Arts repeats some of the persistent myths regarding OpenMP.

Certain comments made also may give rise to a somewhat distorted view on OpenMP for those readers that are less into the aspects of parallel programming. For example, the statement that OpenMP is most suitable for loops only. This has never been the case and certainly the introduction of the flexible and powerful tasking concept in OpenMP 3.0 (released May 2008) is a big step forward.

In this article I would like to respond to this blog and share my view on the claims made. The format chosen is to give a quote, followed by my comment(s).

"OpenMP does not attempt to determine whether there are dependencies between loop iterations."

This is correct, but there are two additional and important comments to be made.

OpenMP is a high level, but explicit programming model. The developer specifies the parallelism. The compiler and run time system translate this into a corresponding parallel execution model. The task of the programmer is to correctly identify the parallelism. As far as I can tell, this is the case for all parallel programming models. In that sense, OpenMP is not any different than other models. It is therefore not clear what the intention of this comment is (note that the exception is automatic parallelization. In this case it is the responsibility of the compiler to identify those portions of the program that can be executed in parallel, as well as generate the correct underlying infrastructure).

Another aspect not mentioned is that one of the strengths of OpenMP is that the directive based model allows
compilers to check for possible semantic errors made by the developer. For example, several compilers perform a static dependence analysis to warn against possible data races. Such errors are much harder to detect if function calls are used to specify the parallelism (e.g. POSIX threads).

"Unfortunately, some of the OpenMP directives for managing memory consistency and local copies of variables affect the semantics of the serial code, compromising this desirable property unless the code avoids these pragmas."

I don't think OpenMP directives affect the semantics of the serial code, so how can this be the case? An OpenMP program can always be compiled in such a way that the directives are ignored, effectively compiling the serial code.

I suspect the author refers to the "#pragma omp flush" and "#pragma omp private" directives. These affect the semantics of the parallel version, not the serial code, but either or both could be required to ensure correct parallel execution. The need for this depends on the specific situation.

We can only further guess what is meant here, but it is worth doing so.

Part of the design of a shared memory computer system is to define the memory consistency model. Several choices are possible and have indeed been implemented in commercially available parallel systems, as well as in more research oriented architectures.

As suggested by the name, memory consistency defines what memory state the various threads of execution observe. They need not have the same view at a specific point in time.

The problem with this is that at certain points in a shared memory parallel program, the developer may want to enforce a consistent view to ensure that modifications made to shared data are visible to all threads, not only the thread(s) that modified this data.

This has however nothing to do with OpenMP. It is something that comes with shared memory programming and needs to be dealt with.

Ensuring a consistent view on memory is exactly what the "#pragma omp flush" directive does. This is guaranteed by the OpenMP implementation. Therefore, the developer has a powerful yet portable mechanism to achieve this. In other words, it is a strength, not a weakness. Also, for ease of development, many OpenMP constructs already have this construct implied. This dramatically reduces the need to explicitly use the flush directive, but if it is needed still, this construct is a nice feature to have.

Given what it achieves, this directive does not impact correct execution of the serial or single threaded version of an OpenMP program. Therefore this can also not explain the claim made in this blog.

The second item mentioned ("local copies of variables") is also not applicable to the serial version of the program, nor single threaded execution of the parallel version. The "#pragma omp private" directive allows the programmer to specify what variables are local to the thread. There are also default rules for this by the way. As a result of this directive, each thread has its unique instance of the variable(s) specified. This is not only a very natural feature to wish for, it also has no impact on the serial code.

Perhaps the author refers to the "firstprivate" and "lastprivate" clauses, but these can be used to preserve the sequential semantics in the parallel program, not the other way round. Their use is rare, but if needed, very convenient to have.
"Since work-sharing induces communication and synchronization overhead whenever a parallel loop is encountered, the loop must contain many iterations in order to be worth parallelizing."

Again some important details are left out. OpenMP provides several strategies to assign loop iterations to threads. If not specified explicitly, the compiler provides for one.

There is a good reason to provide multiple choices to the developer. The most efficient strategy is the static distribution of iterations over the threads. In contrast with the claim made above, the overhead for this is close to zero. It is also the reason why many compilers use this as the default in absence of an explicit specification by the user.

This choice may however not be optimal if the workload is less balanced. For this, OpenMP provides several alternatives like the "dynamic" and "guided" workload distribution schemes. It is true that more synchronization is needed then, but this is understandable. The run time system needs to make choices how to distribute the work. This is not needed with the static scheduling.

Of course the developer can always implement a specific scheme manually, but these 2 choices come a long way to accommodate many real world situations. Moreover, an implementor will try very hard to provide the most efficient implementation of these constructs, relieving the developer from this task.

"Although OpenMP was designed primarily to support a single level of loop parallelization"

I'm not sure what this comment is based on, because nested parallelism has been supported since the very first 1.0 release of OpenMP that came out in 1997. The one thing is that it has taken some time for compilers and run time systems to support this, but it is a widely available feature these days.

"The work-sharing scheduling strategy is often not up to the task, however, because it is sometimes difficult to determine at the start of a nested loop how to allocate thread resources."

OpenMP has a fairly high level of abstraction. I fail to see what is meant with "allocate thread resources". Actually, there is no such concept available to the user, other than various types of data-sharing attributes like "private" or "shared". It is also not clear what is really meant here. Nested parallelism works, each thread becomes the master thread of a new pool of threads, and resources are available whenever needed.

The next line gives us somewhat more of a clue as to what is really meant here.

"In particular, when nested parallelism is turned on, it is common for OpenMP applications to "blow out" memory at runtime because of the inordinate space demands."

In my experience this has not been an issue, but of course one can not exclude that an (initial) implementation of nested parallelism for a specific platform suffered from certain deficiencies. Even if so, that is a Quality Of Implementation (QoI) issue and has nothing to do with the programming model. Shared data is obviously not copied, so there are no additional memory resources, and by design (and desire) each (additional) thread gets a copy of its private data.

The fact this is really a QoI issue seems to be confirmed by the next statement.
"The latest generation OpenMP compilers have started to address this problem, however."

In other words, if there was a problem at all, it is being addressed.

"In summary, if your code looks like a sequence of parallelizable Fortran-style loops, OpenMP will likely give good speedups."

This is one of those persistent myths. OpenMP has always been more flexible than for "just" parallelizing loops. As for example shown in the book "Using OpenMP" (by Chapman, Jost and van der Pas), the sections concept can be used to overlap input, processing and output in a pipelined manner.

"If your control structures are more involved, in particular, involving nested parallelism, you may find that OpenMP isn't quite up to the job."

This is not only a surprisingly bold and general claim, some more specific information would be helpful. As already mentioned above, it is not at all clear why nested parallelism should not be suitable and performant. It actually is and is successfully used for certain kinds of algorithms.

Regrettably the author of this blog also does not seem to be aware of the huge leap forward made with OpenMP 3.0. The specifications have been released in May 2008 and are supported by several major compilers already.

The main new functionality added is the concept of tasking. A task can be any block of code. The developer has the responsibility to ensure that different tasks can be executed concurrently. The run time system generates and executes the tasks, either at implicit synchronization points in the program, or under explicit control of the programmer.

This adds a tremendous flexibility to OpenMP. It also uplifts the level of abstraction. Although never true in the past either, a claim that OpenMP is only suitable for a loop level style of parallelization is certainly way too restrictive.

In addition to tasking, numerous other features have been added, including enhanced support for nested parallelism and C++.

Last, but certainly not least, I can strongly recommend anybody interested in OpenMP to visit the main website.

Just how good are the implementations of tasks these days? For example suppose that I use the task models to implement doubly recursive fibonacci in parallel, how good is it? In MIT Cilk it's
cilk int fib (int n) {
}
if (n<2) return n;
else {
    int a = spawn fib(n-1);
    int b = spawn fib(n-2);
    sync;
    return a+b;
}
}

In OpenMP it's very similar.

Last time I measured an implementation (icc) the overhead of a parallelizable function call a task was an order of magnitude higher than for Cilk. You may argue that this is a quality-of-implementation issue, but I would counter that the semantics of OpenMP make it difficult to implement high-performance task parallelism.

Disclaimer: I am one of the developers of MIT Cilk, but not of Cilk++.

Posted by Bradley C. Kuszmaul on March 09, 2009 at 12:42 AM PDT #

Intel has freely admitted that their taskq implementation was not optimized. Unless Mr. Kuszmaul measured the task implementation using icc Version 11, I am afraid that his comparison doesn't mean much.

Posted by Eric Duncan on March 09, 2009 at 11:41 PM PDT #

Hi Bradley,

Thank you for your feedback. I'm afraid I fail to understand why the semantics of OpenMP inhibit an efficient implementation of tasking. Can you please elaborate?

Thanks. Ruud.

Posted by Ruud van der Pas on March 10, 2009 at 07:20 AM PDT #

Dear Mr. van der Pas:

I consider many of your criticisms of your article to be unfair and inaccurate. For a full response, please see http://www.cilk.com/multicore-blog/bid/8752/Debunking-an-OpenMP-Demystifier

Sincerely,

Charles E. Leiserson

Posted by Charles E. Leiserson on March 13, 2009 at 02:03 PM PDT #

Dear Mr. Leiserson, I have read your response. I don't think it is very productive if I were to post an elaborate answer to this, but I would like to make a few comments.

You have a valid point regarding preserving the semantics of the serial code versus the OpenMP version executed on a single thread. I should have been more careful in my wording. One can think of situations as shown in the example, but these are not only rare, this difference in behaviour can also be avoided by using local variables inside the code block.
The example given to measure the scheduling overhead is flawed I think. I maintain my point that static scheduling is very efficient. The reduction clause in your example implies a critical section and this of course significantly affects the performance.

I'm afraid I also do not find your example on nested parallelism very convincing, as there is no computational work performed.

Your comment on OpenMP 3.0 tasking support seems to be largely based on gcc. As far as I know, tasking is supported in several other widely used compilers.

Kind regards, Ruud van der Pas

Posted by Ruud van der Pas on March 19, 2009 at 07:48 AM PDT #
Debunking an OpenMP “Demystifier”

Date: Friday, 13 Mar 2009 22:41

During the summer of 2008, I posted several overviews of concurrency platforms on the Cilk Arts multicore-programming blog. Among them was a short summary of OpenMP. On February 28, 2009, Ruud van der Pas posted an article on his blog entitled, “Demystifying Persistent OpenMP Myths — Part I,” which was critical of my article. In this article, I respond to Mr. van der Pas’s criticisms, many of which I consider unfair and inaccurate.

In the following, my original comments are in blue, Mr. van der Pas’s criticisms are in red, and my responses are in black. Since the quotations from the two articles are taken out of context below, the reader may find it convenient to read the original articles before proceeding.

1. **OpenMP does not attempt to determine whether there are dependencies between loop iterations.**

   _This is correct, but there are two additional and important comments to be made._

   Funny to start an article on “demystifying” by saying that the first “myth” is actually true. Indeed, the paragraph characterizes something OpenMP does well, beginning as I do, “One of OpenMP’s strengths is parallelizing loops ….”

   Mr. van der Pas’s first subsidiary comment impugns my “intention” in stating this fact, which he seems to take (as with everything I say) as a criticism. Indeed, my statement can be made as well about the parallel loop construct in my own company’s Cilk++ concurrency platform. It is not a criticism - it is a statement of fact.

   Mr. van der Pas’s second subsidiary comment discusses an advantage of OpenMP’s directive approach. I agree with that advantage. In my 500-word summary, I did not attempt to mention every advantage, nor every disadvantage, of the OpenMP approach.

2. **Unfortunately, some of the OpenMP directives for managing memory consistency and local copies of variables affect the semantics of the serial code, compromising this desirable property unless the code avoids these pragmas.**

   _I don't think OpenMP directives affect the semantics of the serial code, so how can this be the case? An OpenMP program can always be compiled in such a way that the directives are ignored,_
effectively compiling the serial code.

I’m sorry for the misunderstanding. My language was not as clear as it could have been. I should have said more accurately that certain OpenMP directives affect the semantics of the serially executing OpenMP program (as opposed to the serial code), causing it to deviate from its behavior when compiled and executed as a serial C/C++ program. Thus, if you use these directives, you must redebug your OpenMP program for serial correctness, as well as debug it for parallel correctness. Moreover, the advantage of having a single source for both C/C++ and OpenMP is mitigated, since the two implementations may operate using different program logic, even when run serially.

As an example of how the behavior of a serially executing OpenMP program can differ from that of the C/C++ program on which it is based, page 233 of the OpenMP 3.0 specification contains the following code as Example A.30.1c:

```c
#include <stdio.h>
#include <assert.h>
int main()
{
    int i, j;
    int *ptr_i, *ptr_j;
    i = 1;
    j = 2;
    ptr_i = &i;
    ptr_j = &j;
    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
        assert (*ptr_i == 1 && *ptr_j == 2);
    }
    assert(i == 1 && j == 2);
    return 0;
}
```

This code operates differently depending on whether it is compiled as C or as OpenMP. Try it yourself.

3. Since work-sharing induces communication and synchronization overhead whenever a parallel loop is encountered, the loop must contain many iterations in order to be worth parallelizing.

*The most efficient strategy is the static distribution of iterations over the threads. In contrast with the claim made above, the overhead for this is close to zero.*

Close to zero? Nonsense. Here is a little program for measuring loop overhead. The inner loop is run n times, where n ranges from 1 million down to 1 by factors of 10. The program is constructed to run the body of the inner loop 1 billion times no matter what, so that the runtimes for different n’s correspond directly to loop overhead.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys time.h>
long long sumit(int n)
```
```c
{ 
    int i;
    long long sum = 0;

    #pragma omp parallel for private(i) reduction(+:sum) schedule(static)
    for (i=0; i<n; i++) {
        sum += i;
    }

    return sum;
}

int main(int argc, char *argv[]) {
    int n;
    for (n = 1000000; n >= 1; n /= 10) {
        int i, ub = 1000000000 / n;
        long long sum;
        struct timeval start, stop;

        gettimeofday(&start, 0);
        for (i = 0; i < ub; ++i)
            sum = sumit(n);
        gettimeofday(&stop, 0);

        printf("n=%10d sum=%15lld  time=%3.2fs
", n, sum,
               (stop.tv_sec - start.tv_sec) +
               1.0e-6 * (stop.tv_usec - start.tv_usec));
    }
}

Running the program (linux-2.6.28 amd64, 2.8GHz quadcore AMD Phenom 920) with various compilers produced the following results.

Without OpenMP

The first set of experiments use the native compilers to baseline the loop overhead for a serial execution. As we can see, the loop overhead is largely amortized by 10 iterations for Intel icc and gcc, and by around 100 iterations or so for Sun cc.

Intel icc 11.0 20090131 using optimization level -O3:

```
<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>499999500000</td>
<td>0.72s</td>
</tr>
<tr>
<td>100000</td>
<td>49999500000</td>
<td>0.72s</td>
</tr>
<tr>
<td>10000</td>
<td>4999500000</td>
<td>0.72s</td>
</tr>
<tr>
<td>1000</td>
<td>49950000</td>
<td>0.72s</td>
</tr>
<tr>
<td>100</td>
<td>4950000</td>
<td>0.77s</td>
</tr>
<tr>
<td>10</td>
<td>45000</td>
<td>0.79s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1.43s</td>
</tr>
</tbody>
</table>
```

Sun cc: Sun Ceres C 5.10 Linux_i386 2008/10/22, optimization -xO3:

```
<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>499999500000</td>
<td>0.39s</td>
</tr>
<tr>
<td>100000</td>
<td>49999500000</td>
<td>0.39s</td>
</tr>
<tr>
<td>10000</td>
<td>4999500000</td>
<td>0.39s</td>
</tr>
<tr>
<td>1000</td>
<td>49950000</td>
<td>0.40s</td>
</tr>
<tr>
<td>100</td>
<td>4950000</td>
<td>0.49s</td>
</tr>
<tr>
<td>10</td>
<td>45000</td>
<td>0.93s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4.28s</td>
</tr>
</tbody>
</table>
```

gcc 4.3.3 -O3:
With OpenMP (OMP_NUM_THREADS=1)

With OpenMP running on one core, the output below shows that the loop overhead is amortized for all three compilers when the number of iterations is somewhat larger than 1000.

Intel icc:

n= 1000000 sum= 499995000000 time=0.74s
n= 100000 sum= 4999500000 time=0.72s
n= 10000 sum= 49950000 time=0.74s
n= 1000 sum= 499500 time=0.93s
n= 100 sum= 4950 time=2.81s
n= 1 sum= 0 time=21.17s

Sun cc:

n= 1000000 sum= 499995000000 time=0.36s
n= 100000 sum= 4999500000 time=0.36s
n= 10000 sum= 49950000 time=0.39s
n= 1000 sum= 499500 time=0.71s
n= 100 sum= 4950 time=3.84s
n= 1 sum= 0 time=34.53s

gcc:

n= 1000000 sum= 499995000000 time=0.36s
n= 100000 sum= 4999500000 time=0.36s
n= 10000 sum= 49950000 time=0.39s
n= 1000 sum= 499500 time=0.71s
n= 100 sum= 4950 time=3.84s
n= 1 sum= 0 time=34.53s

With OpenMP (OMP_NUM_THREADS=4)

Finally, let’s run it on 4 cores and look at the impact when OpenMP actually has to schedule something. As the data below shows, the loop overhead is not amortized until somewhere in excess of 10,000 iterations for Intel icc, and until somewhere in excess of 100,000 iterations for Sun cc and gcc.

Intel icc:

n= 1000000 sum= 499999500000 time=0.20s
n= 100000 sum= 4999950000 time=0.20s
n= 10000 sum= 49995000 time=0.37s
n= 1000 sum= 499500 time=2.15s
n= 100 sum= 4950 time=21.17s
I stand by my statement that many iterations are needed to amortize the scheduling overhead. It is not a
myth. In all fairness, however, this example is difficult for all concurrency platforms, and all need some
number of iterations to amortize the overhead. But, the notion that the overhead is “close to zero” is a
myth.

4. Although OpenMP was designed primarily to support a single level of loop parallelization

I'm not sure what this comment is based on, because nested parallelism has been supported since
the very first 1.0 release of OpenMP that came out in 1997. The one thing is that it has taken
some time for compilers and run time systems to support this, but it is a widely available feature
these days.

The history of OpenMP specifications makes it clear that a single level of parallelism (i.e., no nesting)
was the focus of the design, although the specification accommodates nesting. Page 9 of the original
OpenMP 1.0 specification says, “Nested parallel regions are serialized by default.” The latest OpenMP
3.0 specification’s glossary on page 10 defines “supporting nested parallelism” to mean “Supporting
more than one level of parallelism,” which is distinguished from “supporting OpenMP,” meaning
“Supporting at least one level of parallelism.” That is, even the latest OpenMP implementations need
only support one level of parallelism to be compliant. As you acknowledge, support for nested
parallelism has been slow in coming, at the time I wrote my blog, and even today, turning on nested
parallelism in various implementations of OpenMP blow out memory on fairly straightforward recursive
codes that use nested thread parallelism. I give a concrete example in item 8 below.

5. In particular, when nested parallelism is turned on, it is common for OpenMP applications to
“blow out” memory at runtime because of the inordinate space demands.

In my experience this has not been an issue, but of course one can not exclude that an (initial)
implementation of nested parallelism for a specific platform suffered from certain deficiencies.
Even if so, that is a Quality Of Implementation (QoI) issue and has nothing to do with the
programming model.
Mr. van der Pas’s last statement underscores a philosophical difference between us. I was reticent to speculate in my original blog post, where I tried to characterize OpenMP accurately and fairly based on my and others’ experiences, but frankly, I’ve seen no evidence that OpenMP’s work-sharing concept admits nested parallelism with a high QoI. In my opinion, the design of a programming model for parallel computing, where the focus is on performance, should not be separated from its ability to be implemented efficiently.

6. The latest generation OpenMP compilers have started to address this problem, however.

In other words, if there was a problem at all, it is being addressed.

The most recent OpenMP specification (3.0) now provides for task parallelism, which addresses the problem of parallel nesting in a way that OpenMP’s earlier specifications for thread teams and work-sharing directives do not. Many OpenMP compilers, including gcc, do not yet support task parallelism, however. (See item 8 below.) I don’t think it’s fair to call my statement a myth.

7. In summary, if your code looks like a sequence of parallelizable Fortran-style loops, OpenMP will likely give good speedups.

This is one of those persistent myths. OpenMP has always been more flexible than for "just" parallelizing loops.

I stand by my statement. Mr. van der Pas quotes the word “just” as if I said it, but that word does not appear in my entire post. The myth seems to be in his mind, not in my words.

8. If your control structures are more involved, in particular, involving nested parallelism, you may find that OpenMP isn't quite up to the job."

This is not only a surprisingly bold and general claim, some more specific information would be helpful.

A fair request. Here are some specifics. Consider the following code, which generates an execution tree with about 1 million parallel leaves operating in parallel:

```c
void foo(int depth)
{
    if (depth > 0) {
        int i;

#pragma omp parallel for
        for (i = 0; i < 2; ++i)
            foo(depth - 1);
    }
}

main()
{
    omp_set_nested(1);
    foo(20);
}
```
All the experiments were run on the same quadcore machine as before with OMP_NUM_THREADS=4. Here are the results:

- gcc-4.3.3 (latest official release) — Crashes because it exceeds the thread limit.
- gcc-4.4.0 20090224 snapshot (unreleased) — Same as gcc-4.3.3.
- Intel icc — Does not crash, but the system became unusable, executing about 4 million context switches per second. Killed the program after two minutes.
- Sun cc — Runs to completion within 150ms. The debugger shows that the program creates about 16 threads total.

Although the Sun compiler outperforms the others, one can tweak the loop fanout to make it generate almost 200 threads, causing significant context switching. I invite readers to run this code themselves.

I stand by my statement that for nested parallelism, “you may find that OpenMP isn’t up to the job.” I don’t think that’s a myth, or even hyperbole. As I mention in my original summary and in item 6 above, however, the latest generation of compilers is addressing this issue.

In conclusion, it seems to me that the OpenMP “myths” of which Mr. van der Pas accuses me are unfair and largely of his own making. He repeatedly takes issue with things I said that are true as stated, but which he seems to defensively interpret as if I am implying something other than that which I actually said. (Let me apologize again for the one place where I could have been clearer.) Moreover, he seems prone to hyperbole — e.g., “close to zero” — and makes statements which, based on my studies, seem to be technically incorrect. I hope I have made it clear that my characterization of OpenMP was objectively grounded in concrete coding experience, and not based on “persistent” gossip, opinions, or myths.
One of the biggest challenges in building a parallel program is dealing with data races. Cilk++ offers several tools and techniques to find and eliminate races from your program. Reducers provide a powerful mechanism for eliminating races, but as Spiderman said, "with great power comes great responsibility."

**The Problem**

I recently looked at a customer problem that at first looked like an obvious application for reducers. Briefly, the task was to sort a collection of objects into separate bins based on certain properties of the objects. The simplest parallel implementation, converting the for loop to a `cilk_for` loop, introduces data races on the bins if two or more objects can be put into the same bin.

At first look, this seemed like a good opportunity to use reducers. The contents of each bin is a list of elements, so why not just have each bin hold a list reducer? Reducers are extremely efficient when used sparingly, but unfortunately can become expensive if you create thousands or even millions of reducers, as would be needed in this case.

The other safe approach to handling races is to use locks to protect access to the shared memory locations. In this case, that would either mean using a single lock to protect the entire collection (safe, but bad because the high contention on this lock would eliminate all advantages of parallelism) or to create and acquire a lock for each bin (also safe, but bad because it requires thousands or millions of expensive locks.)

Luckily, there is another way.

**The Approach**

Mutex locks provided by the operating system are expensive to create and acquire, and are generally more heavy weight than is required for this problem. However, there is a much lower-cost alternative. The underlying hardware offers *atomic instructions*, which are effectively locks around single hardware instructions. These are the same instructions that are used to implement mutexes, but we can use them directly to save a significant amount of overhead for our problem at hand. It is important to note that there is no free lunch - atomic instructions require a *memory barrier* that can cost 40 or more instruction cycles. This is far more expensive than the few cycles used by an unlocked read/write, but still much less than the thousands required to create and acquire mutexes. You can think of these as small locks that lock just the read/write hardware operation to ensure that no other processor can race on the memory between the read and the write.

My approach is to use the atomic instruction to safely update the bins in a way that will not create data races. As with any use of locks, this approach prevents data races, but does not guarantee a deterministic result. In this case, each bin will hold the correct list of elements, but the order of the elements in the list is indeterminate.

**The Solution**

An atomic swap operation for pointers is available under different names in both Gnu and Windows C/C++ compilers:
An implementation of a thread-safe sets can be implemented as a template with the class c of the set element as the template parameter:

```cpp
#include
#define AtomicSwap(A,X) __sync_lock_test_and_set (A, X)
#endif
#ifdef WIN32
#define AtomicSwap(A,X) InterlockedExchangePointer (A, X)
#endif

template  class SafeSetElement  {
public:
    C me;
    SafeSetElement * next;
    SafeSetElement (const C & x0) : me (x0) {};
};
template  class SafeSet {
public:
    SafeSetElement * elt0;  // Pointer to the first element
    SafeSet() {
        elt0 = 0;
    }
    void add (const C & x) {
        SafeSetElement * nel = new SafeSetElement (x);
        SafeSetElement * tmp = AtomicSwap (&elt0, nel);
        nel->next = tmp;
    }
};
```

Figure 1 shows the set before the update:

![Figure 1](image1)

The next diagram in Figure 2 shows the data after the new element (nel) has been created and "swapped in", while the address of the old set is stored in a temporary (tmp). Note that the data is in an inconsistent state at this stage, but, as Figure 4 shows later, it is still completely thread-safe:

![Figure 2](image2)

In Figure 3 the contents of tmp variable is copied to the next field of the nel object thus completing the update: 
The diagram in Figure 4 shows what happens if two updates happen in parallel. It assumes that right after the first atomic swap (Figure 2) another thread comes in and does its own atomic swap before the first thread had a chance to finish the update (as in Figure 3). The diagram in Figure 4 assumes that the second thread creates objects tmp2 and nel2. Note that objects tmp and nel are visible only to the first thread, while tmp2 and nel2 -- only to the second.

While the situation in Figure 4 looks messy, it becomes perfectly kosher once each thread completes its update, i.e. when the first thread moves tmp to nel.next and the second thread -- tmp2 to nel2.next:

If the order of the updates was swapped, i.e. thread 2 did the AtomicSwap before thread 1 the situation would be almost the same but the relative order of nel and nel2 in the set would be reversed. Which, of course, is perfectly OK for unordered sets.

From Sets to Hash Tables

Once we have implemented a "safe" unordered set we can extend the implementation to hash tables by using safe sets to chain the values in the hash table. In Figure 6 a hash table is shown as an array of pointers to individual chains:

Notes

1. Note that this implementation only guarantees that two add operations would never cause a determinancy race, but add operation could race with other operations on the set, like iteration: If another thread tries to iterate over the set before the next field of the new element nel is updated (Figure 2) then the result would be unpredictable since we haven't even initialized the next field of the new element.

2. When the above sample code is run under the Cilkscreen tool the latter will complain about the data race on the elt0 member field of the SafeSet object. This complaint can be suppressed by using the __cilkscreen_disable_checking and the __cilkscreen_enable_checking pseudo-calls. The details can be found in the Cilk++ Programmer's Guide, but the add function from the initial code fragment would look something like this:
Results and Conclusions

The example above illustrates that, while reducers might be very powerful and easy to use in a large set of algorithms, there are situations where reducers are not the best answer.

The example described above has been benchmarked on a quad-core Intel CPU. To make the benchmark simulate a real problem, we've added some cycle-burning code to represent the work that the application would do to select the objects of interest and calculate the appropriate bin. The actual benchmarking program can be found on this page.

The results are listed in the table below, which contains the times for different number of processors (specified by the CILK_NPROC environment variable) as well as for the serial C++ execution of the program. A couple of points are worth highlighting:

1. There is a slight overhead in Cilk++-compiled code when it is run on a single core (about 0.2% in the table below). This overhead can be caused by memory barriers built into the atomic swap instruction, as well as by the Cilk++ runtime system.
2. Telling Cilk++ that there are more cores than the actual number is often counter-productive (too many cooks spoiling the performance broth?). We specify the number of cores for benchmarking purposes, but in practice it is usually best to let Cilk++ use the actual number of available processors (4 in this case).

<table>
<thead>
<tr>
<th>Core Count</th>
<th>Execution Time</th>
<th>Speed-up relative to CILK_NPROC=1</th>
<th>Speed-up relative to serial execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>85.694</td>
<td>1.002</td>
<td>1.000</td>
</tr>
<tr>
<td>CILK_NPROC=1</td>
<td>85.828</td>
<td>1.000</td>
<td>0.998</td>
</tr>
<tr>
<td>CILK_NPROC=2</td>
<td>43.182</td>
<td>1.988</td>
<td>1.985</td>
</tr>
<tr>
<td>CILK_NPROC=3</td>
<td>28.805</td>
<td>2.980</td>
<td>2.975</td>
</tr>
<tr>
<td>CILK_NPROC=4</td>
<td>21.958</td>
<td>3.909</td>
<td>3.903</td>
</tr>
</tbody>
</table>

Author: "Alex Zatsman"

The Economy versus Multicore
Date: Friday, 06 Feb 2009 05:03

The economy is on all our minds these days. No doubt times are tough, and every day we hear about more companies reacting to the economic downturn with layoffs, drastic cost-cutting, and shifting