Document: N3530
Date: 2013-03-15
References: *ISO/IEC JTC 1/SC 22/WG14/N1682*
Authors: Darryl Gove (darryl.gove@oracle.com)
Nawal Copty (nawal.copty@oracle.com)
Michael Wong (michaelw@ca.ibm.com)

Subject:

Leveraging OpenMP infrastructure for language level parallelisation

1 Introduction

This proposal suggests how language level parallelisation can be achieved using the existing OpenMP infrastructure. OpenMP is a mature, well established and widely used specification for writing portable multi-threaded applications on a shared memory system. All leading compilers (Oracle, Intel, IBM, gcc, Microsoft, PGI, etc.) already support OpenMP.

It has been around for over 15 years, the first formal version was published in 1997, the 4.0 version of the specifications is due to be released in 2013. The 4.0 specification will also include support for cc-NUMA systems and heterogeneous hardware architectures, like DSP and GPU accelerators.

OpenMP uses compiler directives to express the parallelism. However, the implementation of an OpenMP program relies on a library interface that can programmatically query the run-time system, alter settings and control how the parallelisation is undertaken.

This paper contains a proposal to leverage existing OpenMP infrastructure as part of a future language standard for parallelisation. In this proposal we have assumed some familiarity with OpenMP. A reader who is not familiar with OpenMP would be advised to read the brief overview contained in the appendix before reading the proposal.

2 Leveraging OpenMP as part of a language standard

The big advantage that OpenMP has over other approaches is that it is already supported by all major compilers. So there would be a low barrier to using its infrastructure as part of language specified parallelism. This would ensure timely availability in a wide range of compilers. It would also reduce the amount of new code needed to support the feature, which, together with the large reuse of existing debugged code, would ensure that initial implementations were of high quality.

Moreover, many developers are already familiar with the features of OpenMP. Language specified parallelism that used similar features would accelerates adoption, and greatly reduce the learning curve for users.

Although OpenMP uses **`#pragma`** based controls to describe the parallelism, this is not necessary to leverage the existing OpenMP infrastructure. It is trivial to formulate an alternative way of passing that information to the compiler and runtime.

The core of this proposal is that the language standards could adopt the already available run time OpenMP API and use keywords, or some other mechanism, instead of using **`#pragma`**s for identifying parallel regions.

For example, to define a parallel loop, the keyword "**`parallelfor`**" could be used instead of "**`#pragma omp parallel for`**". The example in Figure 1 illustrates this:

```
parallelfor (i=0; i<1000; i++)
{
 ...
}
```

Figure 1: Example of a parallel loop

As with OpenMP programs, the control of the characteristics of the parallel region could be set through the OpenMP API (optionally renamed to avoid confusion between specifications). In Figure 2, the developer uses the call to function "**`omp_set_schedule()`**" to control the distribution of the loop iterations over the threads. If the developer left out this function call, the loop would still execute in parallel, but the run time details would be decided by the system.

```
omp_set_schedule(omp_sched_dynamic,100);

parallelfor (i=0; i<1000; i++)
{
 ...
}
```

Figure 2: Use of the OpenMP API

The advantages of this approach are:

- Leverages the existing OpenMP infrastructure – this ensures rapid adoption by compilers.
- Does not require compiler vendors to support another parallelisation mechanism.
- All concepts and constructs are familiar to many developers already, greatly reducing the learning curve and accelerate user adoption of the new standard.
- Uses keywords rather than **`#pragma`** directives.
- Is compatible with OpenMP, without being constrained by OpenMP's decisions.

3 Mapping from OpenMP to language level parallelism

This section proposes how the language could be extended to leverage OpenMP features, without using the directive based approach favoured by OpenMP.

3.1 Parallelisation

Of the three approaches to parallel work that OpenMP supports, we are proposing the adoption of two: **`parallel for`** and **`parallel task`**. The third approach, **`parallel sections`**, can be implemented using **`parallel task`**, so does not warrant inclusion.

The first approach we have already met in Figure 1, which is the **`parallelfor`**. As proposed it could be translated directly to "**`#pragma omp parallel for \ for`**" by a preprocessor. It is not imagined that any implementation would do that direct translation. The index variable would naturally be scoped as **`private`**, as it currently is for OpenMP, in the example shown in Figure 3, the variable **`i`** would be scoped as private so each thread had a separate copy.

```
parallelfor (i=0; i<1000; i++)
{
 ...
}
```

Figure 3: Example of parallelfor

Although we are using **`parallelfor`** as a single keyword, there is no reason why it could not be a **`parallel`** keyword that could be placed before or after the existing **`for`** keyword. The objective here is to map the language level concept onto the existing OpenMP infrastructure, not for us to define a rigid specification for the language.

In the **`parallelfor`** keyword we are omitting support for OpenMP clauses that refine the behaviour in the parallel region. We are also combining the OpenMP concepts of a parallel region and a parallelisation directive. Finally we are not addressing here how variables should be scoped within the **`parallelfor`** loop.

It is important to realise that **`parallel for`** is probably the most widely used OpenMP parallelisation directive. Incorporating an equivalent into the language standard will enable the development of a significant number of parallel applications.

The second parallelisation approach is the **`parallel task`**. The keyword **`paralleltask`** would be used to indicate a block of code that should be placed on the task queue and executed by either the current thread or one of the threads available in the thread pool.

The user can insert a call to the **`taskwait()`** function to explicitly wait until all child tasks have completed. An example is shown in Figure 4.

```
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
  {
        paralleltask { i=fib(n-1); }
        paralleltask { j=fib(n-2); }
        taskwait();
        return i+j;
  }
}
```

Figure 4: Example of paralleltask

3.2 Controlling parallelisation behaviour

OpenMP has a set of clauses that control how parallelisation is performed within a parallel region. As an example, there is a clause that determines how the work is distributed across the threads, and another clause that determines how many threads are used to perform the work.

Fortunately OpenMP also defines an API that provides the same facilities for reading and modifying the runtime behaviour of parallel constructs. Rather than enumerate these routines, it is probably more useful to describe the set of facilities that the API provides:

- Obtaining and setting the number of available threads. This allows a program to change the number of threads used in the next parallel construct depending on runtime requirements. It would also allow a developer to change the algorithm depending on the number of threads available.
- Obtaining the id of the currently executing thread.
- Obtaining the number of processors available. This allows an application to spawn an appropriate number of threads for the given hardware.
- Determining whether a function is executing in a parallel or serial context.
- The scheduling for the next parallel region can be read or modified. Appropriate scheduling algorithms allow the program to ensure that the workload is evenly distributed across all available threads.

3.3 Variable scoping

OpenMP defines some default variable scoping rules, it also defines a set of clauses that can adjust the scoping of variables within parallel regions. The two most common scopes are **`shared`** and **`private`**. Fortunately these are very easy to accommodate within the current language defined variable scoping rules.

Variables that are defined outside of a parallel construct would naturally be shared within that construct, and variables which are defined inside a parallel construct would naturally be private to that construct. An example of this variable scoping can be seen in Figure 5. In the example, the variables **`length`**, **`in`**, and **`out`** are shared between the threads. The variable **`tmp`** is private to each thread.

```
void compute(int length, double * in, double * out)
{
  parallelfor (int i=0; i<length; i++)
  {
   double tmp = in[i];
   if (tmp>507.0) { tmp = tmp/2.0; } else { tmp = tmp * 2.0; }
   out[i]=tmp;
  }
}
```

Figure 5: Example of simple variable scoping

It is proposed that, like OpenMP, the loop induction variable is automatically private to the each thread.

3.4 Complex variable scoping

OpenMP also defines more complex OpenMP variables scopings of **`firstprivate`**, **`lastprivate`**, and **`reduction`**. Of these, reduction is the most important, this enables multiple threads to work together in reducing a volume of data down to a single value. The most recent version of the OpenMP specification enables a developer to provide user-defined reductions.

Ideally variable scoping would be included using some kind of language support. However, it is easy enough to support these variable scoping variants using local arrays. To do this each thread uses a single index into the array to hold its current value for the target variable. If we assume that the **`activethreads()`** call returns the number of threads that are available to execute the parallel region and the **`mythreadid()`** call returns the ID of the current thread, then we can get each thread to update its index into an array, and leave the serial code to manage initialising the array, and the finalisation. The example in Figure 6 shows a reduction operation coded this way.

```
double compute(int length, double * values)
{
  double total=0.0;
  double totalarray[activethreads()];

  // serial initialisation
  for (int i=0; i<activethreads(); i++) { totalarray[i] = 0.0; }

  parallelfor (int i=0; i<length; i++)      // parallel computation
  {
    totalarray[mythreadid()] += values[i];
  }

  for (int i=0; i<activethreads(); i++)     // serial finalisation
  {
    total += totalarray[i];
  }
  return total;
}
```

Figure 6: Reduction using thread-indexed arrays

One issue with the code as shown in Figure 6 is that there is false sharing of the array elements, which will have a performance impact. This indicates that although we can write functionally correct code, it might be more effective with language support. This initial proposal does not include a preferred approach to specifying variable scoping in a parallel region, although this would be addressed as part of a subsequent proposal.

4 Concluding remarks

Proposed here there are two approaches to parallelism, **`parallel for`** and **`parallel task`**. There is some syntax around how these are described, and there are some things we can do at the language level that make the use of these features easier.

The key take away from this proposal is that, through their OpenMP runtime libraries, all compilers already have the infrastructure to support language level parallelisation. What is proposed here is the outline of how we can leverage the existing code base to get parallelisation into the language.

The reuse of existing infrastructure reduces costs for compiler vendors, leading to more rapid availability across platforms, and a more robust implementation.

Using the same infrastructure also means that we can mix code written using the language provided parallelisation with code written using OpenMP. This will make it easier for people to adopt the language parallelisation constructs, and it will enable people to leverage the leading edge constructs provided by OpenMP if they need to.

5 What next?

This is a relatively bare-bones proposal. There are quite a few places where the details are undefined. The next step would be to produce a more detailed proposal and corresponding specification. Input on refinements to the specification would be welcomed.

6 Further reading

The current version of the OpenMP specification can be found at the OpenMP website http://www.openmp.org/

The current (3.1) specification can be found at

http://www.openmp.org/mp-documents/OpenMP3.1.pdf

The current draft of the 4.0 specification can be found at

http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf

Appendix

1 An overview of OpenMP features

1.1 The benefits of OpenMP

OpenMP is a combination of compiler directives and a supporting runtime library. The directives describe how the code should be made parallel, and the compiler works with the runtime library to make this happen.

Since the compiler and runtime library manage the threads, parallelisation can be applied to just the region of code where there is expected to be a benefit. This is known as incremental parallelisation since a developer can work through their application piecemeal. Similarly parallelisation can be applied in the main executable or libraries.

The OpenMP specification has a set of default behaviours, but enables a developer to override these. For example, a developer could specify that a region of code only be executed in parallel if there is sufficient work. This gives a developer a considerable amount of control over where and how an application is made parallel.

1.2 The parallel region

The parallel region is the key construct in OpenMP. Code within a parallel region is executed by all threads. Code outside of the parallel region is executed by a single thread.

Within the parallel region, the developer has a wealth of constructs to distribute the work over the threads, control synchronization, guard updates of shared data, etc. Often the parallel region is combined into a single directive that defines the region and defines the type of parallelisation used in the region; the examples shown in this paper use this approach.

In the remainder of this section we touch upon a subset of the features provided by OpenMP.

1.3 Types of parallelisation

OpenMP supports three mechanisms to specify the parallelism in an application

- Parallel sections
- Parallel for
- Parallel tasks

1.4 Parallel sections.

This is where the developer describes multiple sections of independent code. The run time system assigns a thread to each section and wraps around if there are fewer threads than sections.

For example, if an application needed a client thread and a server thread, then **`parallel sections`** could be used to start the two threads in parallel. An example of **`parallel sections`** is shown in Figure 7.

```
#pragma omp parallel sections
{
 #pragma omp section
 {
  // Client thread
  ...
 }
 #pragma omp section
 {
  // Server thread
  ...
 }
}
```

Figure 7: Example of two parallel sections

Although **`parallel sections`** are useful, they can be replicated using parallel tasks, which will introduce shortly. Therefore it is not proposed that **`parallel sections`** form the basis for any language level concepts.

1.5 Parallel for loops.

A **`for`** loop can be described as being parallel, and optional clauses on the directive can specify how the loop should be scheduled, the number of threads to be used etc. Defaults are used in the absence of any clauses. The **`parallel for`** is the most widely used OpenMP parallelisation directive. An example of a **`parallel for`** loop is shown in Figure 8.

```
#pragma omp parallel for
for (...)
{
 ...
}
```

Figure 8: Example of a parallel for loop

A constraint on the **`parallel for`** is that the iteration count for the loop must be known before the loop is entered. This constraint makes the **`parallel for`** construct less suitable for a more dynamic execution flow, e.g. a **`while`** loop. For this, the “tasking” concept has been supported since the OpenMP 3.0 specification.

1.6 Parallel tasks.

A **`task`** is a dynamically generated block of code that gets scheduled for completion some time in the future, either by the thread that created the **`task`**, or by another thread. For example, a web server could generate a **`task`** to handle each new request as it arrives.

An example of creating a parallel **`task`** is shown in Figure 9.

```
#pragma omp task
{
 ...
}
```

Figure 9: Example of parallel task

Parallel tasks are a great example of the general philosophy of OpenMP. All the developer needs to do to express concurrency is to embed the independent parts of the code in a task. From there on the compiler and run time system take care of generating and executing the tasks.

Tasks are completed in the parallel region where they are introduced. If needed, an explicit task execution point can be inserted, which forces all pending children tasks to be completed.

1.7 Other OpenMP directives

OpenMP also provides other directives for things like synchronisation. A selection of these are:

- The **`critical`** directive which indicates an optionally named critical section that only a single thread can enter at any one time.
- An **`atomic`** directive that indicates that the following operation should be completed atomically.
- A **`barrier`** directive which causes threads to wait until all threads have reached that point.
- A **`taskwait`** directive that causes a thread to wait until the completion of all its child tasks.
- A **`single`** directive delineates a block of code in a parallel region that will only be executed by one of the threads, by default the other threads will wait for this thread to complete the code. Similarly a **`master`** directive indicates a block of code that will only be executed by the master thread. [The **`master`** thread is the first thread that was created – it is responsible for executing the serial code in the application.]

1.8 Usage of variables

One of the very flexible features of OpenMP is the recognition that variables can be used in different ways in different regions of code. For example, a variable could be used to hold a running total in one region of code, and in the next region of code that same variable could be used to scale a vector to proportions of the total (see Figure 11).

In keeping with the idea that OpenMP is a syntactic layer added to an application to make it parallel, the specification allows the developer to define how a variable should be treated within in each parallel region. The example in Figure 10 uses the variable "**`total`**" in a reduction.

```
#pragma omp parallel for reduction(+:total)
for (i=0; i<1000; i++)
{
 total += a[i];
}
```

Figure 10: Example of scoping a variable as a reduction

The simple “**`reduction`**” clause describes a fairly complex process that is handled by the compiler. With this clause, each thread performs the reduction operation on a private copy of the variable. Upon completion, the run time system merges these partial results and delivers the final result (“**`total`**” in this example).

The **`reduction`** clause is actually more general than what this example might suggest and support for user defined reductions is included in the 4.0 specification.

In a subsequent region a variable can be differently scoped, for example in Figure 11, the variable “**`total`**“ is scoped in two different ways in two different parallel regions.

```
#pragma omp parallel for reduction(+:total)
for (i=0; i<1000; i++)
{
 total += a[i];
}
#pragma omp parallel for shared(total)
for (i=0; i<1000; i++)
{
 a[i] /= total;
}
```

Figure 11: Example of changing the scope of a variable between parallel regions

OpenMP has rules for the default scoping of variables in parallel regions. The **`shared`** scoping of “**`total`**” in Figure 11 is unnecessary, as it would be scoped as a **`shared`** variable by default.

OpenMP defines a number of variable scoping directives:

- A variable can be **`shared`** which means that all threads see, and can modify, the same variable. Used carelessly, this has the potential for introducing dataraces. Variables declared outside of a parallel region are (typically) shared by default.
- A variable can be **`private`**, so that each thread gets its own copy of the variable. Variables declared within a parallel region are, naturally, thread private.
- A variable can be **`firstprivate`**, which means that it is private, but it is initialised with the value that the original variable had before entering the parallel region.
- A variable can be **`lastprivate`**, which means that each thread gets a copy of the

variable, and the last value, in program order, calculated within the parallel region gets propagated back to the original variable after the parallel region has been executed.

- Finally, a variable can be scoped as a **`reduction`**. This gives each thread a private copy of the variable, and the value propagated to the original variable is the accumulation of the values from all these private copies.

There are a couple of other ways that variables can be manipulated using OpenMP directives:

A variable can be declared as **`threadprivate`**, this is equivalent to declaring a persistent thread local variable. The **`copyin`** directive can be used to copy the value of the master thread's **`threadprivate`** variable to all the other threads' values of the same variable.

A **`copyprivate`** clause can be combined with the **`single`** directive to broadcast the value of a variable, computed in the region declared by the **`single`** directive, to the other threads.

1.9 OpenMP environment variables

A set of diverse Operating System environment variables are available to define settings prior to program execution. Examples are **`OMP_NUM_THREADS`** to specify the number of threads and **`OMP_STACKSIZE`** to control the stack space allocated to each thread.

1.10 OpenMP runtime functions

OpenMP requires compiler support, together with an OpenMP specific runtime library. The runtime library provides extensive support to query the environment, change settings and alter run time execution behaviour.

For example the application can query the number of threads that are used to execute the next parallel region. This function is shown in Figure 12.

```
threads = omp_get_max_threads();
```

Figure 12: Example of querying number of threads

There are other programmatic interfaces that for example set the number of threads, allow the application to execute nested parallel regions, etc.

An example of a setting to change the run time behaviour is shown in Figure 13 where the developer specifies how the iterations of the next parallel loop are to be scheduled.

```
omp_set_schedule(omp_sched_dynamic,100);
```

Figure 13: Example of programmatically setting scheduling