Extending the OpenMP standard for thread mapping and grouping

Guansong Zhang

$Date : 2006 - 04 - 2503 : 12 : 57$

Abstract. In this paper, we are exploring the idea of improving the OpenMP 2.5 standard to facilitate parallel programming on emerging architectures. This includes mapping threads to particular processors, improving the load balance among processors by work distribution, and supporting nested parallelism inherited from applications. We will demonstrate the performance gains of thread mapping with our experimental implementation, and propose new concepts in the standard to try addressing the issue in a broader sense.

Keywords: OpenMP, thread mapping, thread grouping, nested parallelism, SPMD programming

1 Introduction

As several research papers pointed out, OpenMP [1] is due to an update [2], [3]. The set of features in the existing specification of the API provides essential functionality that were mostly selected from previous shared memory parallel APIs. Recently, the shared memory architectures have interesting developments, from multi-core processors to hyperthread and SMT, from accelerate boards to CELL broadband Engine[4]. The emerging architectures force us to explore new features for the OpenMP standard.

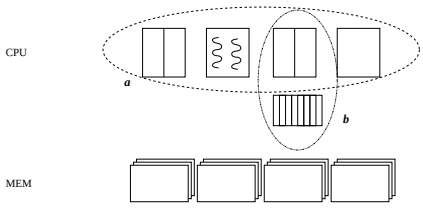

Fig. 1. Emerging architectures

Figure 1 is an example of an extreme case of the latest development of shared memory architectures. The picture includes dual core CPUs, hyperthreads or SMT proces-

sors. Although the system is not modeled on any real machine [1], it can illustrate the potential problems a programmer may face in the real world.

Programming on such complicated systems is going to require difficult compromise. The following items are elements that an ideal solution framework should address, quoted directly from OpenMP language committee discussion:

- *Modularity*: Modern software engineering makes heavy use of modular software. OpenMP's existing model, however, does not support compartmentalization of parallelism. For example, MPI defines a communicator. This can be passed to a library and the library is then restricted to the constraints defined by that communicator. Furthermore, the library developer can rename the communicator so the interactions between components of the library are not exposed outside the library.
- *Multi-level machines*: Shared memory machines will become increasingly hierarchical. The combination of SMT and NUMA all come together to create a nightmare for the existing "flat earth" model of OpenMP. The scalar `OMP_NUM_THREADS` has to expand to encompass a multi-level abstraction of some kind.
- *Mapping OpenMP threads onto processors*: Different machines will have different hierarchies of processors. A program must be able to query the system about the processor hierarchy and then adapt to it by controlling how the OpenMP threads map onto processors.
- *Worksharing between subteams*: To broaden the range of applications appropriate for OpenMP, we need to extend OpenMP so programmers can specify worksharing restricted to subteams.

This is a very ambitious goal. We are not sure if such a solution exist to fit all the requirements. Even if it does exist, the impact could be so large as to make all the previous OpenMP programs obsolete; or make the implementation of the standard so difficult that the drawbacks outweight the advantage it brings to the standard.

In this paper we will describe our attempt to solve this problem. It may not be the answer we are looking for, a lot of things need to be further improved.

To avoid a dramatic change in the OpenMP standard, we propose an incremental approach, to consider *thread mapping* as the first step and address the bigger issue in the second step. We will discuss these topics in the following sections.

2 Thread mapping

The objective of *thread mapping* or *thread binding* is to define a way to associate threads to physical processors. Once mapped, the thread will stay on the processor during the execution without being moved from one processor to another by the operating system. This is also referred to as *thread affinity*.

A typical situation that needs thread mapping is shown in Figure 2, we have two dual core processors, each core is capable of running two threads simultaneously. So the system can provide up to eight *physical threads* to an OpenMP application.

[1] Circle *a* and *b* may represent two different computers, or they are combined together to create an even more complicated structure. This is purely imaginary. We used the figure just to illustrate the complexity of the problem.

In this system, suppose each processor core has its own L2 cache, and each CPU core has its own L1 cache.

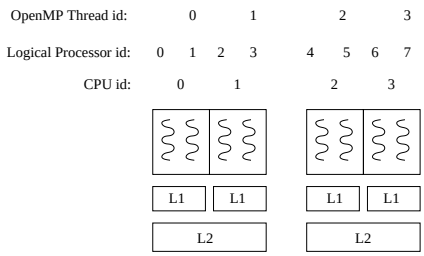

Fig. 2. Thread mapping

If an OpenMP application running on such system decides to use only four OpenMP threads, then there are at least two mapping options one may choose from,

- let each CPU core have one OpenMP thread, so every processor core is fully utilized. Or
- let each CPU core have two OpenMP threads, leaving two CPUs idle. In this way threads can share more data in cache among the adjacent ones.

It is hard to predict which mapping option can achieve better performance at an abstract level. Depending on the hardware implementation and the program itself, the answer may be different. So it will be desirable to let users define the mapping to get better performance.

2.1 Logical processors

Before we define the actual mapping, it is important to think from an application programmer's point of view, what kind of abstract view the hardware system should look like.

A parallel program itself may be complicated enough that a regular user may not want to know anything more than the number of processors offered in a system. This is partially reflected in the current OpenMP standard. In [1], an API function `omp_get_num_procs` returns the number of the processors available to the program at the time the routine is called. Most of the cases, it will be used to define the initial value for the internal control variable *nthreads-var*, which controls how many threads are going to be created when a parallel region is encountered later on.

The environment variable `OMP_NUM_THREADS` and the API function `omp_set_num_threads` provide ways to change this control variable. Yet in the current standard, there is no clear relationship between entities represented by this control variable and the number returned from the function `omp_get_num_procs`, especially when the two values are different.

The thread binding proposal we present here is to establish the affinity relations between the *OpenMP threads* controlled by the *nthreads-var* variable and the number of processors queried by `omp_get_num_procs`. In our proposal, we consider that the number returned in the function `omp_get_num_procs` is only for a group of *logical processors*. In Figure 2, this number can either be 4 or 8 depending whether the operating system made the extra threads available. It is in contrast to the *physical processors*, which may be only 4 traditionally.

As an initial step, we will organize the logical processor group as a linear array, giving each of them an id number, starting from 0, 1, and up to the total number of the processors minus one.

Then the thread binding problem becomes the problem of having *m* OpenMP threads, and *n* logical processor, how to assign those threads to the processors, especially when $m \neq n$.

2.2 The mapping definition and its effect

As explained above, we define the logical processors as a linear array.

We use two environment variables to specify the first processor number that the master thread binds to; and we specify the next processor number the second thread will bind on with a "stride" on the processor array.

For example, the following pair of environment variables

```
OMP_PROC_START=1; OMP_PROC_STRIDE=2
```

will give us the mapping drawn in Figure 2, where thread 0 is on logical processor 1, thread 1 is on logical processor 3, and so on. We will use a *round-robin* fashion to assign the other threads.

We used SPEC OpenMP benchmark to demonstrate the performance impact of thread mapping. In Figure 3, a 64 processor core POWER5 machine is used to measure the OpenMP suite. The SMT option of POWER5 was on. So each processor is capable of running two threads. We set `OMP_NUM_THREADS` to be 64, and compared the following situations

- No processor mapping specification.
- Stride as one by `OMP_PROC_START=0; OMP_PROC_STRIDE=1`, and
- Stride as two by `OMP_PROC_START=0; OMP_PROC_STRIDE=2`

The rest of the execution environment, including compilation flags, are all the same. From the figure, one can see that stride two setting has the best overall performance numbers; Stride one setting is the worst: No stride setting is in between. And some of the difference can be as big as 40%, such as in equak, apsi and wupwise.

2.3 More about the mapping

In the mapping scheme above, all the processors and threads are kept in a simple linear relation. More complicated mapping may be achieved though other structures. For

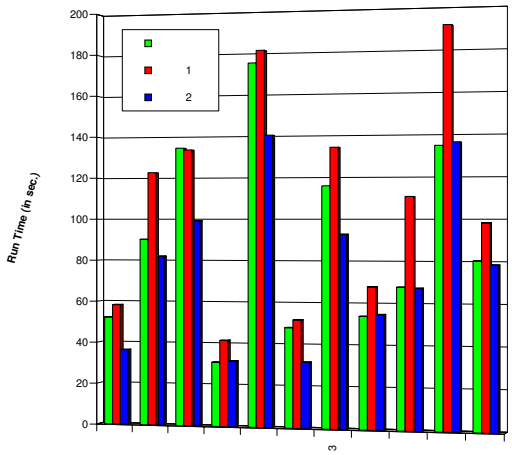

Fig. 3. SPEC OMP performance

example, we can define a arbitrary mapping as an integer array. We are not sure that such level of complexity is useful. We will see that more complicated mapping actually can be achieved in the next section when we extend the logical processor array to a *processor group*.

We also need to point out that the mapping we defined here is only the one from the OpenMP threads to the logical processors. We did not define how logical processors mapped to physical threads. It is left as an implementation defined issue.

3 Processor group

In the previous section, all the logical processors are considered as a linear array. This is a simplification which will hide all the architecture structure of a real machine. In this section, we will extend the view to represent more complex architectures. To support this, we need to introduce the following concepts in the OpenMP APIs.

- *Hierarchical* level: This represents a programmer's view of the hierarchy level that the hardware support. For example, a 4-way Pentium processor with hyper thread enabled can be viewed as either one group of eight processors on the same level, an 8 element linear array; Or two levels of processor groups, with four processor groups at the top level and two processors in each group at the second level.
- *Processor group*: A handle to an opaque data structure which represents a group of processors. It will be a new internal type in the language extension. We will use a temporary name `omp_procs` to represent this type. (A previous example in the language is a variable of `omp_lock` type.) As in HPF[5], a variable of this type can be used with Fortran array syntax.

– `omp run on` *construct*: This will specify on which processor group OpenMP threads will continue to execute and form a new group. When a program is running, the sequential part will be executed on a master thread in the *master group*. When a parallel region is encountered, the group members will execute in parallel and share the workshares bound to the region[2]. The mapping method introduced in the previous section will still work here for distributing the threads among the processors[3].

To express the three concepts in the OpenMP API, the following extensions are suggested[4]

```
const omp_procs * omp_get_procs(void);
```

This function will return a runtime variable, which has the data type as a pointer to `omp_procs`. This is the address of our special handle pointing to the underlining machine where the program will run. By default, all the program was executed as if it was run on this processor group. The `const` modifier indicating that a user can not change this variable.

The processor group provides a way to encapsulate detailed system informations for an application. We will use some annotated pseudo code as examples to further illustrate these ideas.

3.1 Group operation

Suppose we have a machine setup as Figure 1 *a*, and we assume for certain reasons the single core chips are faster.

These functions

```
const omp_procs & processor = * omp_get_procs();
const omp_procs & processor = * omp_get_running_procs();
```

at the beginning of the program will both give us a variable named `processor` to represent the logical view of a processor group. We can define a member function to get the number of members of the group

```
const int omp_procs::get_num_members(void) const;
```

We use the `[]` operator to overload the function to get member element

```
const omp_procs::get_member(int index) const;
```

Then we may have,

[2] More discussion is needed for the situation that another group is specified on the second level of parallel region. Possible issues may include thread migrating and thread stealing.

[3] We may decide later whether to allow different mapping for different processor groups through multiple *internal control variables*

[4] The function names are all tentative, and can be all different in different language bindings. And they may not be limited to these.

```
const omp_procs g0=processor[0];
const omp_procs g1=processor[1];
const omp_procs g2=processor[2];
const omp_procs g3=processor[3];
```

Suppose the processors were organized as two levels of hierarchy. So,

```
// g0 and g2 have two processors each
Assert(g0.get_num_members()==2 && g2.get_num_members()==2);
// g1 and g3 only have one processor each
Assert(g1.get_num_members()==1 && g3.get_num_members()==1);

// omp_procs is a hierarchy,
// it won't be 6 if queried at this level
Assert(get_omp_procs()->get_num_members()==4);
```

Again, we get members as,

```
const omp_procs g4=g1[0]; const omp_procs g5=g1[1];
const omp_procs g6=g3[0]; const omp_procs g7=g3[1];
```

We like to let the two faster processors each having two logical processors,

```
// Create new threads on g1 and g3
omp_procs * gptr0=new(g1) omp_procs[2];
omp_procs * gptr1=new(g3) omp_procs[2];

// Let operator [] work with a processor group address,
// same as ptr->get_member(int index)
const omp_procs g8=gptr0[0];
const omp_procs g9=gptr0[1];

const omp_procs g10=gptr1[0];
const omp_procs g11=gptr1[1];
```

We organize them as a new flat array of processors.

```
// This uses c++ array constructor
const omp_procs g12[]={g8,g9,g4,g5,g10,g11,g6,g7};
```

Now we can write code like this,

```
// On all the processors we grouped previously
#pragma omp parallel on g12[:]
{
  // all the old omp code should be here
  ...
}
```

A triplet `[::]` with optional *start*:*end*:*stride* is used here to get a "section" of the array as a group, the same as used in a Fortran array.

We can also write,

```
#pragma omp parallel on g12
{
  // On all the odd numbered processor
  #pragma omp run on g12[1::2]
  {
    ...
  }
}
```

For C++ completeness, we should do the following at the end of the program,

```
delete[] gptr0; delete[] gptr1;
```

In Fortran and C, we do not have objects, we can only access an address of an object. Besides, we do not have the operator overloading, the corresponding concepts have to be implemented through functions[5]. Suppose we have a user defined function can get the address of `g12[:]` defined in the previous code. And we want to start a new parallel region on all the odd numbered processors in Fortran,

```
      USE OMP_LIB  ! or INCLUDE "omp_lib.h"
      PARAMETER (N=10)
      INTEGER (OMP_PROCS_KIND) :: GROUP, NEWGROUP
      INTEGER (OMP_PROCS_KIND), DIMENSION(N) :: GROUPARRAY

!     User defined function, which calls a C routine.
      INTEGER (OMP_PROCS_KIND) GET_THE_C_GROUP_HANDLE
      EXTERNAL GET_THE_C_GROUP_HANDLE

      CALL OMP_INIT_PROCS(GROUP)
!     Call the user defined function,
!     to get the group handle in the previous C code.
      GROUP = GET_THE_C_GROUP_HANDLE()

!     In F77,  there is no allocatable array.
      IF (OMP_PROCS_GET_NUM_MEMBERS(GROUP) .GT. N) STOP

!     Use DO loop if we don't like an extra function name
      CALL OMP_INIT_PROCS_ARRAY(GROUPARRAY)
!     We know the group is g12, it is a flat 8 element array
!     Get the immediate members of the top level.
      CALL OMP_PROCS_GET_MEMBERS(GROUP, GROUPARRAY)

      CALL OMP_INIT_PROCS(NEWGROUP)
!     Lets use F90 array syntax,
      CALL OMP_PROCS_SET_MEMBERS(NEWGROUP,GROUPARRAY(1::2))

!OMP$ PARALLEL ON NEWGROUP
```

[5] To make it clear, `OMP_PROCS_` is used as the prefix for all these type of functions. In addition, like `omp_lock`, we may need `omp_init_procs` and `omp_destroy_procs`.

```
      ...

!OMP$ END PARALLEL

      CALL OMP_DESTROY_PROCS(GROUP)
      CALL OMP_DESTROY_PROCS_ARRAY(GROUPARRAY)
      CALL OMP_DESTROY_PROCS(NEWGROUP)
```

We need to use explicit functions to group an array of processors as a simple handle and convert it back. (This is done through the `[]` operator in the previous C++ coding.) In fact, in C/C++ and Fortran, if we define the `on` clause working on both scalar and array type of `omp_procs` variables, the previous snips can be further simplified, as some of the conversions are not needed.

3.2 Programming examples

We use a simple task to show a "real" program. Suppose we will calculate the sum of 100 numbers held in array `a[]` as a reduction. (This is not for performance, just to illustrate ideas.) And we assume that the machine is configured as two levels of hierarchy.

We can use the processor group denoted by the previous array `g12` to compute this with 8 threads,

```
#pragma omp parallel on g12
{
  #pragma omp for reduction(+:s)
  for (int i = 0; i < 100; i++) s+=a[i];
}
```

Alternatively, we can do

```
// get a flat processor array
#pragma omp parallel on (* processor.all())
{
  #pragma omp for reduction(+:s)
  for (int i = 0; i < 100; i++) s+=a[i];
}
```

Here `omp_procs * omp_procs::all(void) const` is a member function of the processor group, which returns an address of a processor group consist of all the lowest level processors available.

The differences of the two snips is that the first one will use the same number of threads as the available processors in group `g12`, i.e., 8 threads; While the second segment only uses 6 threads, since we did not create new processors explicitly, the function will return all the 6 physical processors available.

If we reconfigure the system, and let the two faster CPUs each offering two logical processors, as we specified that how to map local processors to physical processors is

still an implementation defined issue, then the second writing of the program should have the same effects as the first one before the reconfiguration[6].

Even though the processor group given by the system is hierarchical, all the previous examples are organizing processors as a flat array, now we will check ways to use nested levels explicitly. If a user want to take advantage of the architecture levels directly, (or organize a processor array as a hierarchy suitable to his nested parallel application) he may write the following code in Fortran,

```
      INTEGER (OMP_PROCS_KIND) :: GROUP

      ! Get the processor hierarchy
      CALL OMP_INIT_PROCS(GROUP)
      GROUP = OMP_GET_PROCS()

!OMP$ PARALLEL ON (GROUP) REDUCTION(+:S)
      ID = OMP_GET_THREAD_NUM()
      MYSTART = 1 + 25*ID  ! 25 should be calculated
      MYEND = MYSTART + 25 - 1

      !This is a real hardware, mapping rules are needed
      IF ((ID .EQ. 1) .OR. (ID .EQ 3)) THEN
        ! We are on the two faster single core CPUs
!OMP$   PARALLEL DO NUM_THREADS(2) REDUCTION(+:S)
        DO I = MYSTART, MYEND
          S = S + A(I)
        ENDDO
!OMP$   END PARALLEL DO
      ELSE
        ! We are on the two processors with 2 CPU cores
!OMP$   PARALLEL DO REDUCTION(+:S)
        DO I = MYSTART, MYEND
          S = S + A(I)
        ENDDO
!OMP$   END PARALLEL DO
      ENDIF
!OMP$ END PARALLEL

      CALL OMP_DESTROY_PROCS(GROUP)
```

If more member functions as query function are defined for the processor group, a user may get complete information from the handle. For example, communication routines can be customized directly by a user.

```
  void my_function(omp_procs * g) {
    // more query functions on *g
    ...
```

[6] So with the concept of the processor group, a well configured system hierarchy can hide all the machine details from regular users. The rather confusing `new` operator in the previous example should be regarded only as an advanced feature.

```
    #pragma omp parallel on *g
    ...
  }
```

4 Summary

In this paper, we first gave out an simple mapping mechanism for binding threads to particular processors. We showed that even with this simple scheme we can improve performance numbers for real benchmark suites.

We further extended the concept of logical processors to a processor group, which addressed the OpenMP programming issues raised earlier. The concepts we presented here are still in draft. A lot of issues, including syntax definition, still need to be refined.

We can summarize the main features of the framework as following,

- *Structured SPMD programming*: SPMD programming is the most common techniques used in parallel programming, especially for data parallel programming. As more processors are available in a real parallel system, and more applications begin to looking for speed up exploiting parallel processing, the traditional way of partition computation with data distribution alone can not fulfill users' need. Specifically work distribution or task distribution is needed among processors. Instead of just using the processor ID number as a conditional guard to distribute work, our programming model is based on a well designed structure — processor group. We call this programming style *structured SPMD* programming.
- *Backward compatibility*: When we introduce the concept of the processing group, we try to consider the backward compatibility. All the previous OpenMP codes will still be legal with out any change. They are running on the *default* processor group provided by the system.
- *Information encapsulation*: We hide most of the detailed information of a physical machine by the logical processor group. We believe this will help user to write portable code without too much machine specific information. In fact, we think most of the time the flat array machine configuration plus thread mapping are good enough for regular users to write efficient code.
- *System configurations*: Although we did not specify the details of system configuration here, it is possible to develop a resource manager that allows a physical system to be configured differently for different applications or partially available to particular applications. It is also possible to configure a system to have multiple groups with different attributes, so an application may target to a heterogeneous architecture.
- *Integrated solution*: The proposal we presented here actually addressed all the issues listed by the OpenMP language committee discussion group as in section 1. Currently we are not sure whether subgrouping threads will be merged in the future OpenMP programming, but we believe that providing a parallel context for subroutines and library functions with the concept of processor group is useful for the OpenMP programming model.
- *Incremental implementation*: There is no real runtime implementation to support the proposal in this paper yet. Some of the earlier work on distributed memory

system can be traced through [6]. We think the framework can be implemented with incremental steps. In the first step, the main structure `omp_procs` and its supporting functions will be added as C++ library functions. Most of the concepts in the proposal will be expressed as function calls. In the second step, we can improve the language syntax, to bind the programming model to Fortran and C/C++. In the third step, more compiler analysis will be used to improve the efficiency of the code. We hope that most of the operation overhead inside the runtime system can be optimized away or moved out of the hot spot with code motion. As those functions should not have any side effects on user variables.

Parallel programming was never an easy task, and new machine architectures posed even more challenges in it. Unfortunately the extra concepts in the language we present here do not make any of this simpler. Yet they provide a set of tools for users to have more controls over the system.

As we implemented the full features of the OpenMP 2.5 APIs, We are well aware of the possible overhead these extra concepts may bring to the standard. It is not the intention of this paper to discuss whether these kinds of complications are necessary. Rather we hope the paper can serve as a discussion base to see if this is the direction of the future OpenMP. It will be up to the users to decide which way OpenMP development should go.

5 Trademarks and copyright

References

1. OpenMP Architecture Review Board. Openmp application program interface version 2.5, 2005. http://www.openmp.org.
2. M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro. OpenMP Extensions forThread Groups and Their Run-time Support. In *LCPC 2000*, 13th International Workshop on Languages and Compil-ers for Parallel Computing, 2000.
3. Barbara M. Chapman, Lei Huang, Haoqiang Jin, and Gagriele Jost andBronis R. de Supinski. Support for flexibility and user control of worksharing in openmp, 2005. http://www.nas.nasa.gov/News/Techreports/2005/PDF/nas-05-015.pdf.
4. Cell broadband engine resource center, 2005. http://www-128.ibm.com/developerworks/power/cell.
5. Charles H. Koelbel, David B. Loveman, and Robert S. Schreiber. *The High Performance Fortran Handbook*. MIT Press, 1993.
6. Guansong Zhang, Bryan Carpenter, Geoffery Fox, Xinying Li, and Yuhong Wen. Structured SPMD programming, 1998. http://grids.ucs.indiana.edu/ptliupages/projects/HPJava/reports/structuredSPMD/javad.pdf.

[7] The opinions expressed in this paper are those of the authors and not necessarily of IBM.