OpenMP Extensions for Heterogeneous Architectures

Leo White

Computer Laboratory
University of Cambridge
Leo.White@cl.cam.ac.uk

Abstract. Modern architectures are becoming more heterogeneous. OpenMP currently has no mechanism for assigning work to specific parts of these heterogeneous architectures. We propose a combination of thread mapping and subteams as a means to give programmers control over how work is allocated on these architectures. Experiments with a prototype implementation on the Cell Broadband Engine show the benefit of allowing OpenMP teams to be created across the different elements of a heterogeneous architecture.

1 Introduction

Modern architectures are becoming more heterogeneous. Power dissipation issues have led chip designers to look for new ways to use the transistors at their disposal. This means multi-core chips with a greater variety of cores and increasingly complex memory systems. These modern architectures can contain a GPU or a number of slave processors, in addition to their CPUs. Developing programs for architectures containing multiple kinds of processors is a new challenge. The various memory systems connected to these processors often have non-uniform memory access costs or even partitioned address spaces (where each memory unit is only accessible to a subset of the processors).

OpenMP is a shared-memory parallel programming model that was first standardised in 1996. It was designed for scientific applications on large shared memory clusters. Parallelism is expressed using directives that preserve the sequential program. The original directives were built around loop-based parallelism. In version 3.0 dynamic task-based parallelism was added to the standard as part of an attempt to address the increasing complexity of parallel programs, especially in more mainstream applications. However, if OpenMP is to continue to help developers achieve high performance on modern architectures it must provide new mechanisms to handle their increasing complexity.

There are two main problems with implementing and using OpenMP on a modern heterogeneous architecture:

1. The model assumes that there is a single coherent memory space. Compiler techniques for mapping programs written for a single memory space onto architectures that have partitioned memory systems have had some success [3,

```
1  void a9(int n, int m, float *a, float *b, float *y, float *z)
2  {
3      int i;
4      #pragma omp parallel
5      {
6          #pragma omp for nowait
7          for (i=1; i<n; i++)
8              b[i] = (a[i] + a[i-1]) / 2.0;
9          #pragma omp for nowait
10         for (i=0; i<m; i++)
11             y[i] = sqrt(z[i]);
12     }
13 }
```

Fig. 1: First Example: Two parallel loops

4, 7]. However it seems that these techniques will not solve the problem in the general case, and some sort of extension to OpenMP will be required.

2. There is no mechanism for allocating work to specific processors on an architecture. This problem is orthogonal to the first one: whatever method is used to address partitioned memory systems, if OpenMP continues to use fork/join, loop-based and task-based parallelism then it will require mechanisms for mapping these models onto heterogeneous architectures.

It is the second problem that this paper attempts to solve. We present extensions that increase the expressivity of OpenMP to give programmers control over how work is allocated on a heterogeneous architecture.

All work in OpenMP is done by *teams* of *threads*. These teams are created using the `parallel` *construct*. Work can be divided amongst the threads in a team using *workshares*. One example of a workshare is the `for` construct, which allows a loop's iterations to be divided amongst the threads in a team. More dynamic forms of parallelism can be exploited using *tasks*. The `task` construct indicates that some work does not need to be executed immediately by the thread that encounters it, but can be deferred until later or executed by a different thread in the team.

We start with some illustrative examples of how OpenMP is used in practice, adapted from those found in the OpenMP Version 3.0 Specification [10]. Figure 1 shows how two loops can be divided between the threads in a team. It consists of a `parallel` construct containing two `for` constructs (each annotated with a `nowait` *clause*). Figure 2 shows a how a single thread in a team can traverse a list creating one task for each node. These tasks will then be executed by the other threads in the team.

The work in these examples could be allocated onto a heterogeneous architecture in a number of ways. The simplest allocation would allow the threads in the team to be executed by any of the processors in the architecture. However, it might be more efficient to restrict the threads to a selection of the processors – perhaps those sharing a single memory unit. This would require some form of

```
1  void process_list_items(node *head)
2  {
3      #pragma omp parallel
4      {
5          #pragma omp single
6          {
7              node *p = head;
8              while (p) {
9                  #pragma omp task
10                     process(p);
11                 p = p->next;
12             }
13         }
14     }
15 }
```

Fig. 2: Second Example: Using tasks to traverse a list

thread mapping extension, to allow the programmer to describe which threads in a team were restricted to which processors.

It is also possible that, in the first example, the processors best suited to execute the first loop differ from those best suited to execute the second. Perhaps there are two groups of processors, and the most efficient solution is to run the first loop on one group while the second loop runs on the other. In either case, the best allocation involves using different threads to execute the two workshares. This would either require two separate teams running in parallel (i.e. *nested parallelism*), or workshares restricted to subsets of the threads in the team (i.e. *subteams*).

The best allocation for the second example might involve allocating the processing tasks to accelerators, while the main loop is executed on the central processor. This would require the `single` workshare to be restricted to the subset of threads executing on the main processor and the `task` constructs to be restricted to the subset of threads executing on the accelerators. Note that nested parallelism could not be used to achieve this allocation.

OpenMP is currently incapable of expressing these possible allocations. In this paper, we propose a combination of *thread mapping* and *named subteams* to control how work is allocated amongst the different parts of a heterogeneous architecture. We also perform some experiments with a prototype implementation on the Cell Broadband Engine to show the benefit of giving the programmer control over the allocation of work onto a heterogeneous architecture.

The rest of the paper is structured as follows: Sect. 2 presents relevant related work, Sect. 3 describes our proposed extensions, Sect. 4 discusses how the extensions can be implemented, Sect. 5 presents an experiment to show some benefits of the extensions, Sect. 6 outlines our conclusions and Sect. 7 discusses future work.

2 Related Work

The need to extend OpenMP to handle the increasing complexity of modern processors has prompted a number of proposals for extensions. We discuss them here in relation to the examples discussed in the previous section.

Device-annotated tasks [1, 6] have been proposed to allow OpenMP tasks to be offloaded to accelerators. The OpenMP syntax is extended to allow a `device` clause to be attached to `task` constructs. This clause takes, as its argument, an identifier that represents a device capable of executing the task. Figure 3 shows how this extension can be used to assign the tasks in our second example to an accelerator. This extension is an effective way of offloading tasks to accelerators,

```
        ⋮
8       while (p) {
9           #pragma omp task target device(accelerator)
10              process(p);
11          p = p->next;
12      }
        ⋮
```

Fig. 3: Using device-annotated tasks to offload tasks to an accelerator

however it does not allow any of a team's threads to run on the accelerators. It also breaks the notion that all work in OpenMP is done by teams of threads, and forces the use of task-based parallelism where fork/join or loop-based parallelism might be more appropriate. These extensions also provide no mechanism to allow the programmer to specify how many instances of a given device should be used or if/when they should be initialised.

Zhang [13] proposes extensions to support thread mapping. The OpenMP execution model is extended to include the notion of a *logical processor*, which represents something on which a thread can run. An architecture is thought of as a hierarchy of these logical processors. The OpenMP syntax is extended by allowing an `on` clause to be used with `parallel` constructs. This clause takes an array of logical processors as its argument, then the team's threads are allocated from each processor in the list in turn. Figure 4 shows how this kind of extension can be used to execute our first example on a selection of the processors in an architecture (in this case the processors `0`, `2` and `4`). However these extensions do not allow different pieces of work in a parallel region to be allocated to different selections of processors. A new team would have to be created each time a different set of processors is required, which is potentially expensive. They also do not allow tasks to be created on one processor for execution on another. Furthermore, these extensions break the OpenMP rule that the thread that encounters a `parallel` construct becomes part of the new team.

```
1  void a9(int n, int m, float *a, float *b, float *y, float *z)
2  {
3      omp_group_t g[3];
4      omp_group_t procs = omp_get_procs();
5
6      assert(omp_get_num_members(procs) > 5);
7
8      g[0] = omp_get_member(procs, 0);
9      g[1] = omp_get_member(procs, 2);
10     g[2] = omp_get_member(procs, 4);
11
12     int i;
13     #pragma omp parallel on(g)
14     {
           ⋮
22     }
23 }
```

Fig. 4: Using thread mapping to control the allocation of the example from Fig. 1

Multiple levels of parallelism are already supported in OpenMP with nested `parallel` constructs, however the creation of new thread teams is often prohibitively expensive, and tasks cannot be exchanged between the threads in separate teams. Accordingly Huang et al. [8] propose allowing workshares to be executed by a subteam, as a cheaper alternative to nested parallelism. It works using an `onthreads` clause for workshares, e.g.

`#pragma for onthreads(`$first$`:`$last$`:`$stride$`)`

where $first$ to $last$ is the range of thread indices and $stride$ is the stride used to select which threads are members of the subteam that will execute the workshare.

Other directive-based programming models, similar to OpenMP, have been created for use with accelerators, especially GPUs. The PGI Accelerator Model [11] consists of directives for executing loops on an accelerator. HMPP [5] uses directives to allow remote procedure calls on an accelerator, these calls can be asynchronous, giving them some of the functionality of OpenMP tasks. Both these models only support a single model of parallelism and can only allocate work to either the main processor or the accelerators.

3 Extensions for Heterogeneous Architectures

This section describes our extensions to the OpenMP execution model and syntax for implementing OpenMP on heterogeneous architectures. These are centred around two complementary extensions: thread mapping and named subteams.

3.1 Thread Mapping and Processors

The current OpenMP execution model consists of teams of threads executing work. We propose extending this model with *thread mapping*. Thread mapping consists of placing restrictions, for each thread, on which parts of an architecture can participate in that thread's execution.

We define an *architecture* as a collection of *processing elements* (e.g. a hardware thread, a CPU, an accelerator, etc...), which are capable of executing an OpenMP thread. Each thread is mapped to a subset of these processing elements, called its *processing set*. A thread may migrate between processing elements within its processing set, but will never be executed by an element outside its set. Which subsets of the processing elements in an architecture are allowed as processing sets is implementation-defined.

Processing sets are represented by values of the new type `omp_procs_t`, which are created using implementation-defined expressions (typically macros or functions). We allow multiple `omp_procs_t` values to represent the same processing set, to allow them to represent additional restrictions or guidance about how a group of threads should be executed (e.g. `SCATTER(A)` might represent the same processing set as `A`, but groups of threads created using `SCATTER(A)` would be kept as far apart as possible). Some examples of possible `omp_procs_t` expressions are discussed in Sect. 4.

To provide some basic `omp_procs_t` expressions that are portable between implementations, an implementation is expected to allow processing sets that can be represented using the `omp_get_proc` routine (Sect. 3.5). These processing sets should approximately partition an architecture into its constituent hardware processors.

3.2 Subteams and Subteam Names

In the current OpenMP execution model tasks and workshares are executed by all of the threads in the team. We propose changing this model to allow tasks and workshares to be restricted to a subset of the threads in a team. To make creating these subsets easier we introduce the notion of *subteams*. Each team is divided into disjoint subteams, which are created when the team is created and remain fixed throughout the team's lifetime. The subset of threads associated with a task or workshare is specified by combining one or more subteams.

Subteams are referenced through the use of *subteam names*. These are identifiers with external linkage in their own namespace (similar to the names used by the `critical` construct). They exist for the duration of the program and can be used by different teams to represent different subteams. Each team maintains its own mapping between subteam names and subteams. Every subteam in a team must be mapped to a different subteam name.

Ideally, subteam names could be used as expressions within the base program, however these uses would be illegal within the base language (as an undeclared identifier). As an alternative, we propose adding a new type `omp_subteams_t`, whose values represent sets of subteam names. Values of this type can be manipulated by the runtime library routines and used with the `on` clause.

3.3 Syntax

The `subteams` *Clause.* Thread mapping, subteam creation and the mapping of subteams to subteam names is all done using a single clause for the `parallel` construct:

`subteams(`*name*$_1$`(`*procs*$_1$`)[`*size*$_1$`], `*name*$_2$`(`*procs*$_2$`)[`*size*$_2$`], ...)`

Each argument of the clause creates a new subteam containing $size_i$ threads, which is mapped to the subteam name $name_i$. All the threads in the subteam are mapped to the processing set represented by the `omp_procs_t` expression $procs_i$. If no name is given then the subteam is mapped to a unique unspecified name. If no processing set is given, or the keyword `auto` is used instead, the implementation chooses an appropriate processing set. The first subteam listed is the *master subteam* and contains the master thread of the team. The processing set used with the master subteam must be a superset of the processing set that the encountering thread was mapped to.

The `on` *Clause.* Subteams can be used to specify the subset of threads associated with a workshare or `task` construct by annotating that construct with an `on` clause:

`on(`*subteams*$_1$`, `*subteams*$_2$`, ...)`

Each argument $subteams_i$ is either an explicit subteam name or an expression with type `omp_subteams_t`. The threads that are members of the subteams mapped to these subteam names (according to the current team's mapping) are used to execute the task or workshare.

When a workshare or task construct without an `on` clause is encountered, it is associated with the subset of threads defined by the *default subteams set*. This is a set of subteam names, which is stored as an `omp_subteams_t` value in a *per-task internal control variable* (these are the variables that control the behaviour of an OpenMP implementation). By default, the default subteams set is the set of subteam names that represent the subteams executing the current piece of work.

In addition to the `on` clause, an `on` construct is also added, of the form:

`#pragma omp on(`*subteams*$_1$`, `*subteams*$_2$`, ...)`
structured block

Each argument $subteams_i$ is either an explicit subteam name or an expression with type `omp_subteams_t`. Threads that are members of the subteams mapped to these subteam names execute the structured block, all other threads ignore the construct. The `on` clause can also be attached to a `parallel` construct as syntactic sugar for a `parallel` construct containing a single `on` construct.

```
1  void a9(int n, int m, float *a, float *b, float *y, float *z)
2  {
3      int i;
4      #pragma omp parallel subteams(st1(PROC_1)[4],st2(PROC_2)[4])
5      {
6          #pragma omp for nowait on(st1)
7          for (i=1; i<n; i++)
8              b[i] = (a[i] + a[i-1]) / 2.0;
9          #pragma omp for nowait on(st2)
10         for (i=0; i<m; i++)
11             y[i] = sqrt(z[i]);
12     }
13 }
```

Fig. 5: Using subteams to divide work between processors in the example from Fig. 1

3.4 Examples

Figure 5 shows how these extensions can be used to allocate the loops of our first example onto different processors. Two subteams are created and each executes one of the loops. Here `PROC_1` and `PROC_2` are macros representing processing sets (each representing a different processor), and `st1` and `st2` are subteam names. Figure 6 shows how these extensions can be used to assign the tasks of our second example onto accelerators. The `subteams` clause is used to create two subteams. The first subteam contains only the master thread and allows the implementation to choose a suitable processing set. The second subteam contains five threads mapped to the processing set represented by the macro `ACC`. The `single` construct is associated with the first subteam, which forces the block to be executed by the master thread. The `task` construct is then associated with the second subteam so that all the tasks created by it will be executed by the threads on the accelerators. Here `main` and `accs` are subteam names.

3.5 Runtime Library Routines

The proposal adds some runtime library routines. We add some routines to provide some portable `omp_procs_t` expressions:

- `omp_procs_t omp_get_proc(int` *index*`);`
- `int omp_get_num_procs(void)`
- `int omp_get_proc_num(void);`

The first routine returns `omp_procs_t` values that should approximately represent the hardware processors in an architecture, each of which has an associated index. The second routine returns the number of hardware processors in the architecture. The third routine returns the index of the hardware processor that contains the processing element currently executing this thread[1].

[1] The result of this routine should only be used in threads that are mapped within a single processor, otherwise thread migration may cause a race condition.

```
1  void process_list_items(node *head)
2  {
3      #pragma omp parallel subteams(main[1], accs(ACC)[5])
4      {
5          #pragma omp single on(main)
6          {
7              node *p = head;
8              while (p) {
9                  #pragma omp task on(accs)
10                     process(p);
11                 p = p->next;
12             }
13         }
14     }
15 }
```

Fig. 6: Using subteams to offload tasks to an accelerator in the example from Fig. 2

We add a further three routines that deal with subteams and subteam names:

- `omp_subteams_t omp_get_subteam_name(int` *index*`);`
- `int omp_get_num_subteams(void);`
- `int omp_get_subteam_num(void);`

The first routine returns an `omp_subteams_t` value representing the singleton set whose member is the subteam name mapped to the subteam with the given subteam index. The second routine returns the number of subteams in the current team. The third routine returns the index of the subteam that the current thread is part of.

The final routine is used to find the set of subteams mapped within a given processing set:

- `omp_subteams_t omp_procs_subteam_names(omp_proc_t` *proc*`);`

Firstly, this routine locates all the subteams in the current team that are mapped to the given processing set, or a subset of that processing set. The routine then returns an `omp_subteams_t` value representing the subteam names that are mapped to each of these subteams.

Between them these routines provide the means for basic querying of the current allocation of subteams and processors. This allows for some simple dynamic tuning of programs to their environment. The `omp_procs_subteam_names` routine, in particular, should allow programmers to assign work to those threads mapped to a particular type of processor.

3.6 Integration with Future Error Model

Our proposed extensions become even more useful with the addition of an error model to OpenMP. OpenMP currently has no method for dealing with, and

allowing programs to adapt to, situations where the runtime is unable to meet the demands of the program. There are already plans to include such a model in the next version of the OpenMP standard [12].

Integrating our proposals with such a model should allow the programmer to decide what happens if a subteam cannot be mapped to the requested processing set, or if the subteam name associated with a workshare is not mapped to a subteam within the current team. This will allow programmers to attempt to use a certain allocation of work to processors, and if that allocation cannot be used revert to a less efficient allocation. This adaptability improves the portability of these extensions, which is a difficult problem when developing for heterogeneous architectures.

For example, an OpenMP implementation could be created for an architecture that allows GPGPU, but only if a certain type of GPU is available. Programs for this implementation could then attempt to use GPGPU for appropriate tasks, falling back on the main processor if a GPU is not available.

4 Implementation Aspects

To support these extensions an implementation must at least provide the ability to create subteams and associate them with workshares and tasks. This mostly requires annotating workshare and task data with a value representing the subteams associated with it, and then checking that a thread is a member of one of these subteams before allowing it to execute the work.

An implementation must decide how it is going to divide its target architectures into processors. The simplest method is to consider the whole architecture to be a single processor. It must also decide which other subsets of the processing elements in an architecture will be accepted as processing sets, and what expressions will be used to represent them.

Some possible examples of processing sets and expressions to represent them are:

- Group processors based on their functions (e.g. `MAIN` for the main processors and `ACC` for the accelerators).
- Arrange the processors in a tree and allow any subtrees of this hierarchy as processing sets. These processing sets could be represented by a variadic function or macro, where a subtree is represented by their child indices on the path from the root of the tree (e.g. `TREE(`n`,`m`)` represents the subtree that is the mth child of the nth child of the root of the hierarchy).
- Other patterns that specify groups of processors (e.g. `STRIDE(`$n1$`,`$n2$`,`s`)`).
- Allow any set of processing elements as a processing set, and provide a full range of functions to manipulate these sets. (e.g. `omp_union_procs(`p`,`q`)`, `omp_intersect_procs(`p`,`q`)`, etc...).
- Expressions that do not change the processing set of a given `omp_procs_t` value, but provide guidance about how groups of threads should be executed on these processing elements (e.g. `SCATTER(`p`)`, `COMPACT(`p`)`, etc...).

5 Experiments

To show the benefits of increasing the expressivity of OpenMP to allow the programmer to control how work is allocated to processors, a prototype implementation was created for the Cell Broadband Engine [2].

The Cell Broadband Engine processor includes one PowerPC Processor Element(PPE) and seven Synergistic Processor Elements(SPEs). The PPE has two hardware threads and accesses main memory through a cache. The SPEs cannot access main memory directly, instead they use 256kB local stores. The SPEs can perform DMA transfers between their local stores and main memory. The prototype implementation supports two processing sets: one mapping threads to the PPE and another mapping threads to the SPEs. The initial thread is mapped to the PPE.

As discussed in the introduction, the extensions proposed in this paper do not address the problem of a partitioned memory space. So the prototype implementation uses only simple mechanisms to move memory to/from the SPEs' local memories. Shared variables are accessed through simple software-managed caches in the local stores, while private variables are kept on the local stores within the call stack. Alternatives to simple software-managed caches have been shown to be more effective [3, 4, 7] and would be preferred in a more refined implementation.

The test programs are taken from the OpenMP C implementation of the NAS Parallel Benchmarks [9]. Each program was modified by adding a `subteams` clause to each of its `parallel` constructs. These clauses contain one subteam mapped to the PPE and one mapped to the SPEs. An `on` clause is also added to the `parallel` constructs to allow the parallel region to be executed by just the threads on the SPEs.

The test programs are:

EP Pairs of Gaussian random deviates are generated. The main part of the algorithm is a `for` workshare that performs computation on private data. There is very little communication between threads.

IS A large integer sort is performed. The main part of the algorithm is a `for` workshare that includes regular access to a shared array.

CG A conjugate-gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. It contains a series of `for` workshares, some including irregular access to shared arrays.

The speed-ups from adding threads to either the PPE or the SPEs are shown in Fig. 7. The greatest speed-up is highlighted in the table for each program.

The best speed-up for EP is obtained using a thread on the PPE and seven SPE threads. This is equivalent to one thread on each processor. This program is inherently very parallel, so the nearly linear relation between speed-up and number of threads is as expected.

The best speed-up for IS is obtained using two threads on the PPE and no SPE threads. This is probably due to the access to shared arrays in the main

EP

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	1.23	2.45	3.68	4.90	6.12	7.35	7.31
1	1	2.02	3.01	4.04	4.98	6.03	7.01	**7.98**
2	1.68	2.51	3.34	4.17	5.01	5.85	6.68	7.43
3	1.62	2.17	2.71	3.24	3.76	4.32	4.80	5.31

IS

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	0.07	0.14	0.20	0.27	0.33	0.39	0.36
1	1	0.14	0.20	0.27	0.33	0.39	0.44	0.42
2	**1.42**	0.20	0.27	0.33	0.39	0.45	0.50	0.45
3	1.18	0.27	0.33	0.39	0.45	0.50	0.55	0.49

CG

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	0.10	0.20	0.30	0.40	0.50	0.6	0.22
1	1	0.21	0.31	0.41	0.51	0.62	0.72	0.19
2	**1.64**	0.31	0.41	0.51	0.62	0.72	0.82	0.21
3	0.5	0.29	0.33	0.37	0.40	0.41	0.43	0.15

Fig. 7: Speed-ups obtained by using different numbers of PPE and SPE threads

loop. The simple software cache used by our implementation is an inefficient method for handling these regular loop accesses, and it is possible that a more refined implementation would actually get better performance from an allocation including SPEs.

The best speed-up for CG is also obtained using two threads on the PPE and none on the SPEs. In this case the access to shared arrays is irregular and very hard to optimise, so this result is unsurprising.

While memory sharing effects arguably dominate these figures, we argue that the size of the disparities shows the importance of allowing programmers to control how work is allocated onto a heterogeneous architecture. The performance of all three test programs is improved by using multiple threads, however the best thread mapping is not the same for all three. The tests also show that performance can be improved by allowing workshares to operate across the different elements of a heterogeneous architecture like the Cell.

Only the EP benchmark was able to improve performance by using the SPEs, but the other two benchmarks could both have their performance on the SPEs improved by using a more refined mechanism than a simple software-managed cache. However, some programs are simply not amenable to being split across

a partitioned memory space, so primitives to express thread mapping are still required.

6 Conclusions

We have proposed extensions to OpenMP for handling the heterogeneous elements of modern processors. These involve extending the execution model to allow threads to be mapped to processing sets. The execution model changed to allow workshares and tasks to be executed by a subset of the threads in a team. These extensions can be used to offload tasks to accelerators or to allow simple thread mapping. Unlike previous extensions proposed for accelerators, these extensions also allow workshares to be executed across the different elements of a heterogeneous architecture. To control which subsets of threads are used to execute workshares and tasks we introduced the concept of subteams and subteam names.

Our experiments show that allowing workshares to be executed across heterogeneous elements on an architecture like the Cell, can improve performance. They also show that this improvement in performance depends on how threads are allocated across such an architecture. This provides a strong argument for giving programmers the means to choose this allocation themselves.

The important conclusion of this work is that extensions that can map threads to processors and execute work on a selection of the threads in a team are sufficient for controlling the allocation of work amongst the various elements of a heterogeneous architecture. This increase in expressivity must be included in OpenMP to allow efficient programming on emerging modern architectures.

7 Future Work

In this paper, teams are divided into disjoint subteams that can only be created at the creation of the team and can only be referenced through subteam names. It would also be possible to implement subteams that could overlap and be created at any time. The `omp_subteams_t` type could then represent subteams directly, instead of representing sets of subteam names. This would be a more powerful model, allowing much greater ability for dynamic tuning through library routines. It might also be a simpler model for programmers to understand. However, it is more complicated to implement and makes programs harder to analyse statically.

Another possibility would be to make the programs much easier to statically analyse, by removing the `omp_subteams_t` type and forcing all `omp_procs_t` expressions to be compile-time constants. The increased ability to statically analyse programs could allow compilers to produce more efficient code. Programs could no longer be dynamically tuned using runtime library routines, however they could still be tuned using the error model (Sect. 3.6).

As discussed in Sect. 1, the proposals in this paper only address one of the two orthogonal problems for OpenMP on modern architectures. The problem of

maintaining the illusion of a single shared memory on architectures that have a partitioned memory space is yet to be solved in the general case. What code annotations are required to allow the compiler to maintain this illusion is still an open research problem. Until this problem is be solved, at least for common cases, OpenMP will be of limited use on architectures with partitioned memories.

Acknowledgements I thank Alan Mycroft and Derek McAuley for helpful discussions, and Netronome for funding this work through a PhD studentship.

References

1. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A proposal to extend the openmp tasking model for heterogeneous architectures. LNCS 5568, 154 (2009)
2. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation; a performance view. IBM Journal of Research and Development 51(5), 559 (2007)
3. Chen, T., Sura, Z., O'Brien, K., O'Brien, J.K.: Optimizing the use of static buffers for dma on a cell chip. LNCS 4382, 314 (2007)
4. Chen, T., Zhang, T., Sura, Z., Tallada, M.G.: Prefetching irregular references for software cache on cell. In: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization. p. 155 (2008)
5. Dolbeau, R., Bihan, S., Bodin, F.: Hmpp: A hybrid multi-core parallel programming environment. In: First Workshop on General Purpose Processing on Graphics Processing Units (2007)
6. Ferrer, R., Beltran, V., Gonzàlez, M., Martorell, X., Ayguadé, E.: Analysis of task offloading for accelerators. LNCS 5952, 322 (2010)
7. Gonzàlez, M., O'Brien, K., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A.E., Chen, T., Sura, Z., Zhang, T., O'Brien, K.: Hybrid access-specific software cache techniques for the cell be architecture. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08. p. 292 (2008)
8. Huang, L., Chapman, B., Liao, C.: An implementation and evaluation of thread subteam for openmp extensions. Workshop on Programming Models for Ubiquitous Parallelism (PMUP 06), Seattle, WA (2006)
9. Jin, H., Frumkin, M., Yan, J.: The openmp implementation of nas parallel benchmarks and its performance. Tech. rep. (1999), http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf
10. OpenMP Architecture Review Board: Openmp application program interface. Tech. rep. (2008), http://www.openmp.org/mp-documents/spec30.pdf
11. Wolfe, M.: Implementing the pgi accelerator model. In: GPGPU'10: Proceedings of the 3rd Workshop on GPGPUs. pp. 43–50. ACM, New York, USA (2010)
12. Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B., Churbanov, A.: Towards an error model for openmp. LNCS 6132, 70–82 (2010)
13. Zhang, G.: Extending the openmp standard for thread mapping and grouping. LNCS 4315, 435 (2008)