Structure and algorithm for implementing OpenMP workshares

Guansong Zhang, Raul Silvera, Roch Archambault *

IBM Toronto Lab
Toronto
ON, L6G 1C7, Canada

Abstract: Although OpenMP has become the leading standard in parallel programming languages, the implementation of its runtime environment is not well discussed in the literature. In this paper, we introduce some of the key data structures required to implement OpenMP workshares in our runtime library and also discuss considerations on how to improve its performance. This includes items such as how to set up a workshare control block queue, how to initialize the data within a control block, how to improve barrier performance and how to handle implicit barrier and nowait situations. Finally, we discuss the performance of this implementation focusing on the EPCC benchmark.

Keywords: OpenMP, parallel region, workshare, barrier, nowait

1 Introduction

Clearly OpenMP[1][2] has become the leading industry standard for parallel programming on shared memory and distributed shared memory multiprocessors. Although there are different hardware architectures to support the programming model, the implementation of an OpenMP programming language usually can be separated into two parts: compilation process and runtime support, shown in Figure 1. In this figure, the language-dependent frontends will translate user source code to an *intermediate representation (IR)*, to be processed by an optimizing compiler into a resulting *node program*, which interacts with a *runtime* environment that starts and controls the multi-threaded execution.

There are already papers introducing the compilation part of this structure [3], yet little discussion has been dedicated to the runtime environment itself. In fact, as the OpenMP standard continues evolving, there are still many open issues in this area, such as the mechanisms required to support nested parallelism, threadprivate etc.

In this paper, we will focus our attention on the runtime environment implementation. The runtime system we have developed is based on the pthreads library available on most major operating systems, including Linux, Mac OS and AIX©.

We try to avoid the controversial parts of the OpenMP runtime environment by concentrating mainly on structures and algorithms used to implement OpenMP *workshares*, the well-understood concept yet the most fundamental one in parallel programming. We will also discuss the performance considerations when we use standard benchmarks to test our implementation.

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

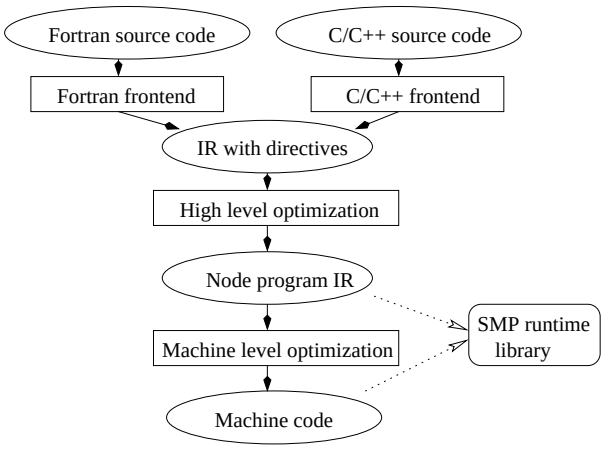


Fig. 1. compilation structure

2 Classifying OpenMP workshares

The parallelism in an OpenMP application program is expressed through parallel regions, worksharing constructs, and combined parallel constructs. A combined parallel construct can be considered as a parallel region with only one worksharing construct inside it, although its implementation may be more efficient from a performance point of view. In any case, a parallel region is the basic concept in OpenMP programming.

When a parallel region starts, multiple threads will execute the code inside the region concurrently[4].

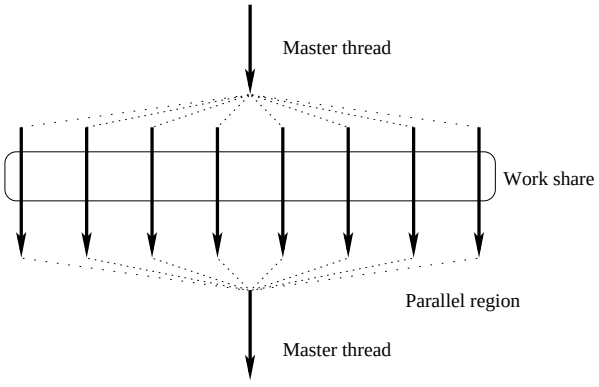


Fig. 2. Parallel region

In Figure 2, a master thread starts up a parallel region executed by 8 threads. Through parallel regions, multiple threads accomplish *worksharing* in an OpenMP program.

The simplest format of worksharing is replicated execution of the same code segment on different threads. It is more useful to divide work among multiple threads — either by having different threads operate on different portions of a shared data structure, or by having different threads perform entirely different tasks. Each cooperation of this kind is considered as a *workshare*[1] in this paper. In the figure, we use a round-cornered rectangular block to represent one workshare.

The most common forms of workshares in OpenMP specification are worksharing DO and SECTIONS, as shown in Figure 3 (a) and (b) respectively.

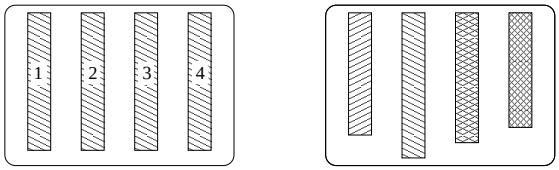


Fig. 3. Common workshare

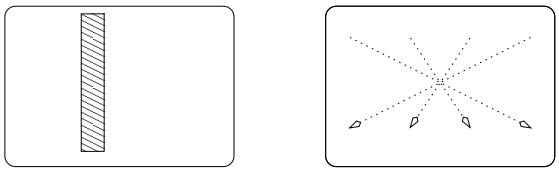
We use different numbers to mark the sequential order of the blocks in an OpenMP DO construct, but it does not mean the implementation will ensure that the corresponding threads will work on those blocks. In fact, the mapping of work to threads is determined by the schedule type of the DO construct as specified by the OpenMP standard. Similarly, the mapping between the sections in a SECTIONS construct and the threads is also decided by the schedule type. The fundamental difference between a DO construct and a SECTIONS construct is that in SECTIONS the code segments executed by threads could be entirely different.

Besides the common workshares introduced above, we can consider other OpenMP structures as workshares too. The typical examples are SINGLE constructs and explicit barriers, as in Figure 4 (a) and (b).

A SINGLE construct is semantically equivalent to a SECTIONS construct with only one SECTION inside, while the explicit barrier is like a SECTIONS construct with neither NOWAIT clauses nor any SECTION inside. For a SINGLE construct, the first thread that encounters the code will execute the block. This is different from a MASTER construct, where the decision can be made simply by checking the thread ID.

Worksharing DO, SECTIONS and SINGLE constructs have an important common feature: they have an implicit barrier at the end of the construct, which may be disabled by using the optional NOWAIT clause. An explicit BARRIER is semantically equiva-

[1] The concept of workshare here is different from the parallel construct, `WORKSHARE` in Fortran OpenMP specification 2.0, which can be treated semantically as the combination of `OMP DO` and `OMP SINGLE` constructs.



(a) (b)

Fig. 4. Synchronization workshare

lent to an empty SINGLE section without the NOWAIT clause. This observation leads us to classify explicit barriers in the same category of workshare constructs.

Different implementations may have a different view of this kind of classification. The real advantage of considering them all as workshares is for practical coding. From an implementation point of view, their common behaviors will lead to a common code base, thus improving the overall code quality.

From now on, we will use *workshare* to refer to any one of the workshares we have mentioned above.

3 Key data structures

In this section, we will examine the techniques for implementing workshares in a parallel region.

3.1 General requirement

The specific ways of implementing workshares in a parallel region may be different from one to another, but with the analysis we already have in previous sections, one can see that at least the following data segments should be in a *control block* for workshares,

- *Structure to hold workshare-specific information* This is needed to store information regarding an OpenMP DO construct or SECTIONS construct. Things like initial, final value of the loop induction variable, the schedule type, etc.
- *Structure to complete possible barrier synchronization* This is used to implement any barriers that may be needed during the lifetime of the workshare construct.
- *Structure to control access to the workshare control block* This is typically a *lock* to ensure only one thread modifies the information of the shared control block, for example, to mark the workshare started or show that a particular section code is already done.

Since different workshare constructs can be put into the same parallel region, the first item must be a "*per workshare*" value, i.e. for each workshare in the parallel region, there should be a corresponding one. Because of the existence of NOWAIT clauses, multiple workshares can be *active* at the same time — executed by different threads

simultaneously. So, multiple instance of this data structure must exist simultaneously; one for each active workshare. The same observation can be made for the third item in the list. Each thread will access its own active workshare control block, which may or may not be different from each other.

In general, there has to be a queue of workshare control blocks for each parallel region.

3.2 The control block queue length

Once we understand the basic structures needed for a parallel region, we have to consider its construction, destruction and initialization. At the same time, we need to minimize the overhead of creating and manipulating such structures, as noted in [5].

It cannot be statically predicted how many parallel regions a program may explore and how many of them will be active at the same time because of nested parallelism or explicit usage of threads in user codes. Therefore, the workshare control block queue will be allocated dynamically. In addition to discussions in section 3.1, a workshare control block queue has to be constructed whenever a parallel region is encountered, and it will be destroyed when the parallel region ends.

Before we go any further, we need to decide the length of this queue. For example, if we map each workshare to its own control block in a straightforward manner, the following parallel region

```
!$OMP PARALLEL
!$OMP DO
      DO i = istart, iend
        ...
      END DO
!$OMP END DO
!$OMP DO
      DO i = istart, iend
      ...
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO i = istart, iend
      ...
      END DO
!$OMP END DO
!$OMP DO
      DO i = istart, iend
      ...
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

will need the workshare control block queue as shown in Figure 5, where there are four block elements in total.

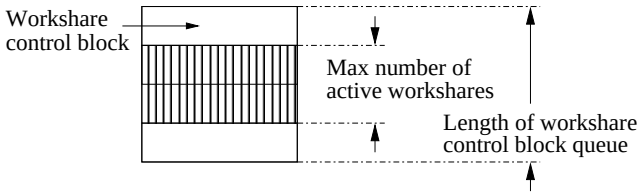


Fig. 5. Workshare control block queue

A simple way of handling the queue is to create a new control block whenever a workshare is encountered, more accurately, whenever a *workshare instance* is encountered. If a workshare is defined inside a sequential loop body, we have as many workshare instances as the number of iterations in that loop.

Since the workshare control block is a shared construct for multiple threads, to constantly increase its size will be detrimental to the runtime overhead for the creation of parallel regions. Moreover, doing so will create higher memory impact on the overall program execution.

A better solution is to allocate a chunk of workshare control blocks at once, use them as a block pool, and try to reuse the pool whenever possible, only increasing its size when absolutely necessary.

In the queue, we will set up extra fields, such as *next available workshare control block pointer*, and set it back to the beginning whenever an explicit or implicit barrier is met. Thus we only need two block structures instead of the four in Figure 5 for the code snippet above. When the first workshare instance finishes, a barrier synchronization occurs, and the next available workshare control pointer will be set back to the first block structure. Then, two block structures are needed at most because of the NOWAIT clause. After that, another barrier will set the pointer back to the beginning again.

Furthermore, resizing the pool may need extra book-keeping structures and work. If we artificially introduce a barrier on a nowait workshare when the queue becomes full, we do not need to increase the size at all[2].

In real applications, a special case of using of NOWAIT clauses is to form a coarse-grain software pipeline, such as the one in APPLU of SPECOMP2001[6], thus one cannot degrade any nowait workshare. But in such cases, the length of the pipeline is often set as the number of the hardware processors to get best performance. Choosing a proper fixed-size queue to avoid extra overhead is still a valid consideration.

Regardless of resizing the queue or not, the number of hardware processors or its multiples, to be conservative, is a good candidate for the initial chunk size of the pool. Then we can resize the queue if needed.

[2] We do need to mark the head and tail position of the active workshares in the queue, so the next available pointer can be set, from the position next to the head, back to its tail when a barrier is encountered.

3.3 Fine-grained optimization

From the analysis in section 3.1, a lock has to be defined within the control block structure. Whether it is hand coded as in [7] or used directly from Pthread library[8], the lock itself needs to be initialized, along with other structures for the workshare-specific information.

The master thread can initialize all the structures at once when they are allocated during the setup time. But for parallel regions where the maximum number of active workshares is small — only one, when the NOWAIT clause does not exist, this process will bring extra overhead. It is more desirable to defer the initialization of the block items until their use becomes necessary.

Algorithm 1, used to execute a workshare, achieved this by always keeping the queue have one extra block ready during initialization phase. In the scheme, the master thread initializes the first control block, thus making the queue ready to use for the first workshare. Then, when the first thread exclusively accesses the available control block through the lock, it will make the next control block available by initializing it.

Algorithm 1: Execute a workshare in parallel regions

```
Data      : Workshare control block queue
Data      : Next available block pointer

begin
    if this workshare is not locked then
        lock it;
        if this workshare is already started then
            release the lock;
        else
            mark this workshare has been started;
            init the next available control block if it is new;
        end
    end
    execute the corresponding workshare and release the lock if I have locked it;
    if barrier is needed then
        do barrier synchronization and reset the next available control block pointer;
    end
end
```

The next control block may never be used. But by keeping it ready, we do not need another lock process to initialize a shared item. The lock is shared with the exclusive access to the previous control block. The exclusive access here is needed anyway, since each thread needs to know if the workshare has been worked on.

3.4 Barrier implementation

There are many papers dedicated to the implementation of a barrier synchronization already. Here we will use Algorithm 2 described in [9].

Algorithm 2: Barrier with distributed counter and local sensor

```
Data : Distributed counter with each element as one
Data : Local sensor with each element as one
begin
    Decrease my own distributed counter element;
    if I am the master thread then
        repeat
            foreach element in distributed counter do check if it is zero
        until all distributed counter elements are zero;
        foreach element in distributed counter do
            set it back to one
        end
        foreach element in local sensor do
            set it to zero
        end
    else
        repeat
            check my local sensor element
        until it is zero;
    end
    Set my own local sensor element back to one;
end
```

Since both distributed counter and local sensor are padded cachelines, we would like to avoid allocating two cache line arrays for every barrier by letting all the barriers in a parallel region share the same pair of counter and sensor.

Before a barrier starts, all the elements of the counter array will be set to one, as will the local sensor counter array. We let one thread in the group, for instance the master thread, act as if it is the last thread. It will decrease its own element of the distributed counter array and then spin to check whether all of the counter elements are zero. The rest of the threads will decrease their own counter elements and then spin on checking their own local sensors.

When the designated thread finds the counter elements are all zero, it will set all the counter elements back to one and then zero all of the elements in the local sensor array. Finally, when all of the threads leave the barrier after their local sensor is zeroed, they reset their local sensor back to one.

4 Other techniques and performance gain

We used the EPCC microbenchmarks[5] to show the performance gains coming from optimization considerations we discussed in previous sections. The hardware systems we tested on are a 16-way 375MHz POWER3^{TM} and a 32-way 1.1GHz POWER4^{TM}.

The EPCC benchmarks have two parts, the synchronization suite and the schedule suite. Since we are concentrating on workshare implementation, we have focused

on the synchronization benchmark. It includes separate test cases for parallel region, workshares (which are OMP DO, SINGLE and explicit barrier), and reductions.

For a parallel region, as we show in Figure 6, the "on demand" initialization method introduced in section 3.3 reduces its overhead on both POWER3 and POWER4 systems.

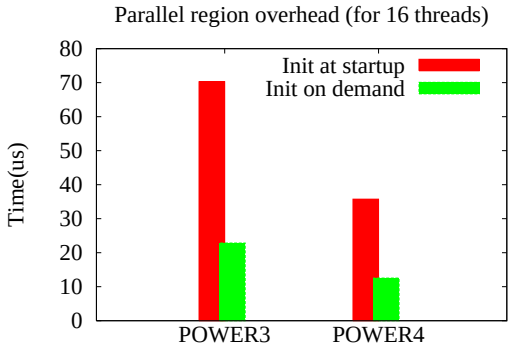


Fig. 6. Performance difference

For a barrier operation, we compare a primitive fetch-and-add barrier with the one using algorithm 2 and the results are shown in Figure 7. The test was done on the POWER4 system. As the number of threads increases, the advantage of using distributed counters becomes more apparent.

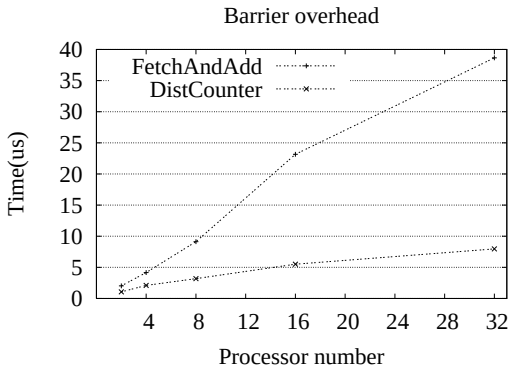


Fig. 7. POWER4 barrier overhead

In fact, the whole sub-suite improved significantly with the methods introduced in the previous sections. Again the data in Figure 8 was collected on the POWER4 system.

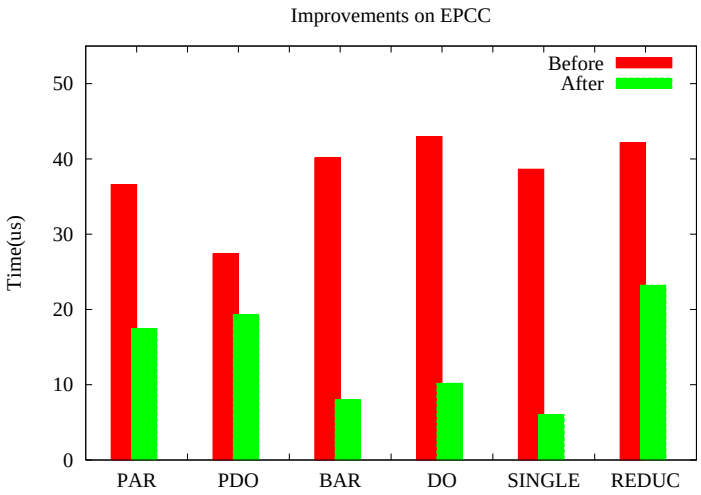


Fig. 8. EPCC improvement

Note that in this figure, the reduction case is special; aside from the improvement from a better parallel region and synchronization method, we have also implemented a partial-sum mechanism to implement the reduction, which allows us to minimize the required synchronization.

5 Summary and future work

In this paper, we listed important data structures needed in our runtime library to support workshares in OpenMP standard, and the corresponding algorithms to reduce the runtime overhead.

We explained that by carefully choosing the length of the workshare control block queue, and reusing the queue whenever a barrier synchronization occurred, we do not need to map control block structures to memory for each workshare instance. Also by preparing the next workshare control block in advance, we do not need a separate locking phase to initialize a control block, neither do we need to initialize the whole queue at startup, which saves the parallel region overhead for most of the real application cases. Although these are just "small techniques", they do have significant impact on performance benchmarks for runtime libraries.

We also introduced our view on OpenMP workshare, and algorithms to implement barrier synchronization, which was considered as a special case of the workshare. All

these are commercial available in our XL Fortran and VisualAge® C/C++ compilers. We are actually further improving the barrier performance by taking advantage of more accurate cacheline alignment of our internal data structures.

6 Trademarks and copyright

References

1. OpenMP Architecture Review Board. Openmp specification FORTRAN version 2.0, 2000. http://www.openmp.org.
2. OpenMP Architecture Review Board. Openmp specification C/C++ version 2.0, 2002. http://www.openmp.org.
3. Michael Voss, editor. *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2003, Toronto, Canada, June 26-27, 2003, Proceedings*, volume 2716 of *Lecture Notes in Computer Science*. Springer, 2003.
4. Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, 2001.
5. J. M. Bull. Measuring synchronization and scheduling overheads in OpenMP. In *First European Workshop on OpenMP*, October 1999.
6. Standard Performance Evaluation Corporation. SPECOMP2001 benchmarks. http://www.spec.org/omp2001.
7. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.
8. David R. Butenhof. *Programming with POSIX threads*. Addison-wesley, 1997.
9. Guansong Zhang et al. Busy-wait barrier synchronization with distributed counter and local sensor. In Michael Voss, editor, *WOMPAT*, volume 2716 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2003.