Unique Worker model for OpenMP

Raghesh Aloor
Dept of CSE, IIT Madras, Chennai, India
raghesh@cse.iitm.ac.in

V. Krishna Nandivada
Dept of CSE, IIT Madras, Chennai, India
nvk@iitm.ac.in

ABSTRACT

In OpenMP, because of the underlying efficient 'team of workers' model, each worker is given a chunk of *tasks* (iterations of a parallel-for-loop, or sections in a parallel-sections block), and a barrier construct is used to synchronize the workers (not the tasks). Naturally, the practitioners are discouraged from invoking barriers in these tasks; as otherwise, depending on the number of workers the behavior of the program can vary. Such a restrictive practice can adversely impact programmability and program readability. To overcome such a restrictive practice, in this paper, inspired from the more intuitive interaction of tasks and barriers found in newer task parallel languages like X11, HJ, Chapel and so on, we present an extension to OpenMP (called UW-OpenMP).

UW-OpenMP gives the programmer an impression that each parallel task has been assigned a unique worker, and importantly these parallel tasks can be synchronized using a barrier construct. Consequently, the semantics of the programs (using parallel-for-loops, sections and barriers) remains independent of the actual number of worker threads, at runtime. We argue that such a scheme allows the programmer to conveniently think and express his/her logic in parallel. We propose a source to source transformation scheme to translate UW-OpenMP C programs to equivalent OpenMP C programs that are guaranteed to not invoke barriers in any task. We have implemented our proposed translation scheme in the ROSE compiler framework. Our preliminary evaluation shows that the proposed extension leads to programs that are concise and arguably easier to understand. Importantly, the efficiency resulting from the 'team of workers' model is not compromised.

1 INTRODUCTION

The increasing popularity of multi-core systems has resulted in the advent of many task parallel languages like X10 [1], HJ [2], Chapel [3], Cilk [4], OpenMP [5] and so on. One of the mantras of parallel programming in many of these languages is that the programmer thinks in parallel and expresses the program logic in a parallel language. For instance, from a programmer's point of view, a parallel loop in HJ [2] or X10 [1] can be seen as one that creates multiple tasks, where each task will run in parallel. Figure 1a shows a sample stencil computation code (written in HJ), which creates four parallel tasks (by using the `forall` construct) and the goal of each task is to execute the `workload` function in parallel. Each instance of the `workload` function executes the first statement (`S1`) and then waits for the other tasks to reach the barrier (`next`). Once all the tasks reach the barrier, they continue with their execution of `S2`. The barrier (in the `workload` function) ensures that the execution of `S2` (by any task) starts only after all the instances of `S1` have been executed. The expected values in arrays `A` and `D` are [0 1 2 3 0] and [2 4 6 0], respectively.

```
A = new int[5];
B = new int[4];
B[0]=1;B[1]=2;
B[2]=3;B[3]=4;
D = new int[4];
void compute(){
 forall (i: 0..3) {
   workload(i);
 } }
void workload(int i){
 A[i]=B[i]-1; /*S1*/
 next; // barrier
 D[i]=A[i+1]*2;/*S2*/}
```

(a) HJ code

```
int A[5],D[4]; //zero init
int B[4]={1,2,3,4},i;
void compute(){
omp_set_num_threads(nthrds);
#pragma omp parallel
{
  #pragma omp for private(i)
  for(i=0;i<=3;i++){
   workload(i); } } }
void workload(int i){
  A[i]=B[i]-1;   /* S1 */
  #pragma omp barrier
  D[i]=A[i+1]*2; /* S2 */ }
```

(b) OpenMP code

Figure 1: Barrier inside parallel-for-loop.

It can be argued that the facility to place barriers inside parallel tasks (supported in many languages like X10 [1], HJ [2], Chapel [3], and so on) helps improve programmability. It helps the programmer to conveniently encode program logic such as "Create a number of tasks and each task performs a job that requires the tasks to (conditionally or unconditionally) synchronize at different points". In addition, from a programmer's point of view, it is natural to assume that the number of parallel worker threads (*workers*, for short) will match the number of parallel tasks. The HJ code shown in Figure 1a works according to this intuitive behavior, where the synchronization happens among the iterations of `forall` (parallel loop) rather than workers. However, we observe interesting challenges when a similar code is written in languages likes OpenMP C [5], PJ [6], and JOMP [7].

Figure 1b shows a code snippet (similar to that shown in Figure 1a) written in OpenMP C. OpenMP provides a simple yet powerfully efficient scheme of creating a team of workers and using them to execute various parallel tasks; in the context of OpenMP, we use 'task' to refer to either an iteration of a parallel-for-loop (specified using `#pragma omp for`) or a section (specified using `#pragma omp section`). The number of workers in a team can be set inside the code by invoking `omp_set_num_threads` (a library function). The OpenMP code in Figure 1b, compiled using GCC gives the expected output, if `nthrds=4` (= number of iterations). However, say, `nthrds=2` and the OpenMP scheduler

ICS'15, June 08 - 11, 2015, Newport Beach/Irvine, CA, USA.

http://dx.doi.org/10.1145/2751205.2751238.

(using static scheduling) assigns iterations 0 and 1 to worker 0, and iteration 2 and 3 to worker 1. Now worker 0 and 1 start executing iterations 0 and 2, respectively. The barrier works as a synchronization point between the workers. Once both the workers hit the barrier, they resume executing the code after the barrier. After a worker has finished executing an iteration, it continues executing the next assigned iteration. At the end of the execution of the parallel-for-loop, `D` contains [0 4 0 0], which does not match the expected value. The reason for such a mismatch is that the workers executed the statement at label `S2` (present after the barrier) of iterations 0 and 2, before they executed the statement at label `S1` (present before the barrier) of iterations 1 and 3. If `nthrds=3`, the program may not even terminate. Further, setting the number of workers to be greater than the number of tasks does not guarantee consistent results, either.

The above discussed inconsistency can be noted with all the implementations (and variations) of OpenMP we have tested (GCC [8], Cray CC [9], Clang/LLVM [10], ICC [11], PJ [6], and JOMP [7]). However, some compilers (like GCC 4.4) issue a warning to the programmer when a barrier is syntactically surrounded by a work-sharing construct (such as parallel-for-loop or parallel-sections). Similarly, GCC 4.7, Cray CC, ICC and Clang/LLVM throw an error in such a case. But if the barrier is present inside a function called within a work-sharing construct, no such warning/error is issued.

The correct way to encode the logic expressed in Figure 1a in OpenMP is to write conforming [5] OpenMP code that ensures that no barrier can execute in the body of the parallel-for-loop. A programmer can achieve this (in case of the Figure 1b) by manually inlining the function invoked in the loop-body and distributing the parallel-for-loop. In general, this leads to code bloat and arguably harder to read programs (cohesive pieces of program logic are spread across multiple parallel-for-loops). Importantly, it is not always straightforward to distribute the body of the parallel-for-loop, as the barrier may be present deep inside if-statements, or while-loops. For example, consider the illustrative iterative averaging (IA) code from [12] (modified by Shirako et al [13]) shown in Figure 2; it intends to perform iterative averaging on a one-dimensional array. Here, a simple distribution of the outer loop would not suffice, because barriers are present inside a while-loop.

To summarize, the use of barriers within parallel-for-loops and parallel-sections can improve the general programmability and readability of the OpenMP code. However, addressing the associated consistency related issues requires that such barriers are used to synchronize the iterations (or sections) of the parallel-for-loop (or parallel-sections), instead of the team of workers associated with the work-sharing construct. In this regard, we first define a runtime model called *UW* model. Note: we mainly focus on parallel-for-loops and later show that parallel-sections can be dealt similarly.

DEFINITION 1.1. *A parallel-for-loop is said to be executing in UW model if a unique worker executes each iteration therein. An OpenMP program is said to be executing in UW model if each constituent parallel-for-loop executes in UW model and each parallel-for-loop uses the same iterations to worker map.*

Any given OpenMP program may be run in UW model by setting the number of workers equal to the number of parallel tasks. To distinguish UW model from the current execution model of OpenMP (where a worker may execute one or more iterations of a parallel-for-loop), we term the latter as the *One-to-Many model* (in short OM model). To avoid the above discussed consistency related issues, we add a requirement (matching the popular advise [14]) that a program executing in OM model cannot invoke barriers inside work-sharing constructs.

```
iters = 0; delta = epsilon + 1;
#pragma omp parallel
{
 #pragma omp for
 for (j=0;j<N;j++){
   while(delta>epsilon){
     newA[j]=(oldA[j-1]+oldA[j+1])/2.0;
     diff[j]=abs(newA[j]-oldA[j]);
     #pragma omp barrier
     if(j==1){ delta=sum(diff); iters++;
               temp=newA;newA=oldA;oldA=temp;}
     #pragma omp barrier
   } if (j==0) printf("iters = %d\n",iters);
 }/*for*/ }/*parallel*/
```

Figure 2: Snippet of Iterative Averaging (IA) in OpenMP

In this paper, we propose to extend OpenMP with UW model; we call this extension UW-OpenMP. Similarly, we term the subset of OpenMP programs that can execute in OM model as OM-OpenMP programs. For the UW-OpenMP code shown in Figure 1b, an equivalent OM-OpenMP can be derived by performing function inlining and a simple loop-distribution [15] transformation across the barrier statement. However, in general, it may not be so straightforward to do the transformation, as the barrier may be present deep inside if-statements or while-loops (see for example, Figure 2). In such examples, a naive loop distribution may not lead to even syntactically correct code.

We present a source to source transformation scheme to translate UW-OpenMP C programs to equivalent OM-OpenMP C programs; the translation ensures that the semantics of the generated program is independent of the number of workers. And the generated code can still take advantage of the efficiency of the 'team of workers' model of OpenMP. Our proposed technique is inter-procedural and whole-program in nature. Another important feature induced by UW-OpenMP is that it allows OpenMP programmers to think in parallel, instead of first writing a serial program and then introducing parallelism as an afterthought.

Our Contributions:

- UW-OpenMP allows barriers to be placed inside parallel-for-loops, which helps the programmer encode the program logic in a convenient (more intuitive) way.
- A transformation scheme (based on a combination of traditional loop optimizations [15] and some novel mini-transformations) is presented that helps enforce the UW model behavior in the presence of barriers in conditional statements and loops.
- The correctness of the transformation is stated as a theorem and we present a proof thereof.
- We present a set of optimizations to generate efficient code.
- We have implemented the proposed techniques in the ROSE Compiler Framework [16].
- We present an evaluation over ten popular OpenMP kernels (taken from different benchmark suites), and show that (i) our proposed translation ensures that the generated code takes advantage of the underlying 'team of workers' model, (ii) the overheads are minimal, and (iii) the proposed optimizations are effective.

1.1 Related Work

Shirako et al [17] proposed a mechanism to adopt the concept of phasers in HJ to OpenMP. They extend the existing runtime to admit phaser related calls (e.g., registration, synchronization and drop) in OpenMP programs. In contrast, our extension does not need any modifications to the runtime, and realizes thread level synchronization by doing a source to source translation of UW-OpenMP programs to OM-OpenMP. Their proposed technique does not allow synchronization between iterations of different parallel-for-loops (or different instances of the same parallel-for-loop nested

1.	Parallel region	`#pragma omp parallel S`
2.	Atomic	`#pragma omp atomic S`
3.	Parallel-for-loop	`#pragma omp for for(i=0;i<N;i++) S`
4.	Parallel-sections	`#pragma omp sections S`
5.	Parallel section	`#pragma omp section S`
6.	Single	`#pragma omp single S`
7.	Barrier	`#pragma omp barrier`

Figure 3: Syntax of OpenMP Pragmas

inside a serial loop), or different parallel-sections, which is not the case with UW-OpenMP.

Presence of synchronization primitives inside parallel-for-loops poses interesting challenges while doing semantics preserving transformations. Nandivada et al [18] present techniques to do semantics preserving transformation of programs with task parallel loops with barriers. In this paper, we present a systematic approach to translate UW-OpenMP programs (that may contain barriers inside loops) to OM-OpenMP.

We extend many of standard loop optimizations (such as loop fusion, loop distribution) [15] in the context of OpenMP programs. Further, we propose a series of optimizations to improve the performance of the generated code. Many prior works try to optimize parallel programs [19, 20, 18, 21], by eliminating useless barriers, joins, task-creation and termination overheads. These techniques can be applied on our generated code in an orthogonal way.

2 BACKGROUND

Figure 3 lists the syntax of some of the most important pragmas supported by OpenMP C [5]. Each of the pragmas (except the one for barrier) is applied on the statement immediately after the pragma (referred to as the *body* of the pragma).

Parallel Region: A parallel region creates a team of workers, each of which executes the body `S` in parallel. For each parallel region, the number of workers is fixed (once per region) by either invoking the library routine `omp_set_numthreads`, or setting the environment variable `OMP_NUM_THREADS`.

Atomic Construct: It ensures mutual exclusion, important to concurrently update / read shared variables.

Work-sharing Constructs: OpenMP supports three types of work sharing constructs: (i) parallel-for-loop: the iterations of the enclosed for-loop are distributed among the workers of the current team. (ii) parallel-sections: This work-sharing construct distributes each section (declared using `#pragma omp section`) present in the body, among the workers of the current team. The exact distribution of the work (iterations / sections) to different workers of the team is dependent on the scheduling policy. The default policy is left to individual implementations (e.g., GCC uses a default blocked distribution). (iii) single: The body of the single construct is executed only by one of the workers in the team.

Barriers: A barrier within a parallel region acts as a synchronization point among all the workers of the current team. No worker in a team can cross a barrier, until all the workers of the team reach a barrier. There is an implicit barrier at the end of each parallel region. Each work-sharing construct (unless invoked with a special `nowait` clause) has an implicit barrier inserted at the end of it.

Memory Model: OpenMP programs support the relaxed memory consistency model [22, 23]. There is an implicit "flush" at the end of the barrier construct that ensures that all the workers crossing the barrier see a consistent view of the updates to the shared variables.

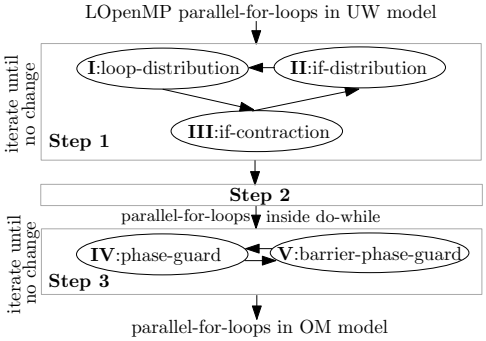


Figure 4: Block Diagram

3 FROM UW-OpenMP TO OM-OpenMP

In this section, for the ease of translation, we first define a subset of UW-OpenMP called LOpenMP and present our transformation scheme to translate LOpenMP programs to equivalent OM-OpenMP programs. Later in Section 3.3 we present a generalization.

DEFINITION 3.1. *An LOpenMP parallel region is an OpenMP parallel region which has its body comprising of one or more parallel-for-loops, each possibly nested within a sequential (while) loop. The body of a parallel-for-loop in an LOpenMP region may contain a serial loop. But the serial loop should not have further nested serial loops. An OpenMP program where each parallel region is an LOpenMP parallel region is called an LOpenMP program.*

DEFINITION 3.2. *In the execution trace of an LOpenMP parallel region in UW model, we group all the statement instances by the iteration number of the parallel-for-loop in which they execute: the sequence of statements executed by all the parallel-for-loops in their i^{th} iterations constitutes the i^{th} UW-group. We say that the i^{th} UW-group is generated by the i^{th}* agent *of the LOpenMP parallel region.*

DEFINITION 3.3. *We use b_k^i to denote the $k^{th}(\geq 1)$ barrier in the i^{th} UW-group. Let Ψ_k^i denote the set of statement instances executed before b_k^i. The k^{th} phase (Φ_k) is defined as follows. $\Phi_1 = \bigcup_i \Psi_1^i$; and $\Phi_k = \bigcup_i \Psi_k^i \setminus \Psi_{k-1}^i$, if $k > 1$. The statement instances of any phase Φ_k are said to be executed in the k^{th} phase of the parallel region.*

3.1 Translation Scheme

The execution of each UW-OpenMP task can be seen to be consisting of one or more phases; intuitively, a phase describes the barrier free code executed between the starting point and a barrier, between two barriers, between a barrier and the end point of the task, or between the starting and ending points. And when two or more UW-OpenMP tasks synchronize on a barrier, each one of them progresses to the next phase. Our proposed transformation scheme uses this intuition to generate equivalent OM-OpenMP code.

For ease of explanation, we introduce our transformation techniques assuming that the parallel-for-loops in the input LOpenMP program have no barriers nested inside while-loops; later in Section 3.2 we relax this requirement. We define `N` to be the maximum number of iterations present in any parallel-for-loop.

The overall block diagram of our transformation is shown in Figure 4. Our transformation consists of three steps. In *Step 1*, we repeatedly invoke loop-distribution, if-distribution, and if-contraction rules (Figure 5), till the body of each parallel-for-loop is a *simple-statement* (sequential code with no OpenMP pragmas), or a barrier, or a barrier surrounded by an `if` predicate (conditional-barrier). These rules are standard compiler transformations [24], applied in

Step 1	

I: Loop-distribution
#pragma omp for
for(i=0;i<N;i++){
  S1;S2;}
//S1 or S2 has barriers
	#pragma omp for
	for(i=0;i<N;i++){S1;}
	#pragma omp for
	for(i=0;i<N;i++){S2;}

II: If-distribution
#pragma omp for
for(i=0;i<N;i++) {
  if(e) { S1; S2; }}
//S1 or S2 has barriers
	#pragma omp for
	for(i=0;i<N;i++) {
	  v = e;
	  if(v) S1; if(v) S2;}

III: If-contraction
#pragma omp for
for(i=0;i<N;i++) {
  if(e1) if (e2) S1;}
//S1 has barriers
	#pragma omp for
	for(i=0;i<N;i++) {
	  if(e1 && e2) S1; }

Step 2	

#pragma omp parallel
  S
	H=calloc(M,
	  N*sizeof(int));
	P=calloc(N,
	  sizeof(int));
	cp=0;
	#pragma omp parallel
	do { S; break;
	} while (1);

Step 3	

IV: Phase-guard
#pragma omp for
for(i=0;i<N;i++) {
  S; }
	#pragma omp for
	for(i=0;i<N;i++) {
	 if(P[i]==cp &&
	  !H[⟦pfCnt⟧][i])
	 {H[⟦pfCnt⟧][i]=1; S}}

V: Barrier-phase-guard
#pragma omp for
for(i=0;i<N;i++) {
 if(e)
  #pragma omp barrier
}
	#pragma omp for
	for(i=0;i<N;i++){
	 if(P[i]==cp &&
	  !H[⟦pfCnt⟧][i])
	 { H[⟦pfCnt⟧][i] = 1;
	  if (e) P[i]++; }
	 if (P[i]!=(cp+1)){
	  #pragma omp atomic
	  flag=0;
	 }/*end if*/}//end for
	#pragma omp single {
	  tmpflag = flag;
	  if(flag) cp++;
	  flag = 1;}
	if(tmpflag) continue;

Figure 5: Transformation Rules

the presence of OpenMP parallel-for-loops. Note: Loop-distribution may perform scalar-expansion, if necessary.

Step 2 initializes two auxiliary variables (H and P used in Step 3) and embeds the complete body of the parallel region inside a do-while loop. This do-while loop helps avoid unstructured goto statements in the next step. The output from Step 2, consisting of a parallel region containing a series of parallel-for-loops (say, in total M) embedded inside a do-while loop, is given as input to *Step 3*, which in turn generates equivalent OM-OpenMP code.

Data structures: We use a shared variable cp (to indicate the *current phase* of the parallel region) , and a shared array P (to denote the *phase array*) with size equal to N. If the parallel region is in phase k, cp is set to k. The current phase of the i^{th} agent is stored in P[i] and is incremented to emulate the behavior of the i^{th} agent encountering a barrier. When all the agents increment the phase, we increment cp by 1 to indicate the change in the current phase of the parallel region. With the body of each parallel-for-loop (after Step 2), we maintain a unique identifier pfCnt $\in [0, M)$. We use ⟦pfCnt⟧ to indicate the value of pfCnt for the current parallel-for-loop. We use a two dimensional array H: M×N (to denote *statement execution history*). A statement with identifier ⟦pfCnt⟧ is executed by agent i, only if H[⟦pfCnt⟧][i]$= 0$ (that is, it has not already been executed). Thus, the maximum memory overhead = O(M×N).

Rules IV and V: The input to Step 3, is a parallel region containing a series of parallel-for-loops, each of whose body is either a simple-statement or a conditional-barrier; we treat unconditional-barriers also as conditional barriers, where the predicate always evaluates to true. Correspondingly, in Step 3, we present two transformation rules (IV and V) shown in Figure 5. In Rule IV, the i^{th} instance of the simple-statement S is executed only when P[i] (current phase of agent i) is equal to cp and H[⟦pfCnt⟧][i]=0. In Rule V, the barrier inside the conditional-barrier statement is replaced by the statement P[i]++, which is executed only when P[i] (current phase of agent i) is equal to cp and H[⟦pfCnt⟧][i]=0. We examine the complete phase array P to check if a phase change (increment) is in the offering (happens only if all agents increment P[i] by 1). The phase increment is reflected by the increment of cp by a single thread. After the phase increment, the control goes back to the beginning of the do-while loop. This is to execute those statements which were skipped because P[i] was greater than the value of cp at that instant.

Our proposed technique ensures that we replace the abstraction of synchronization among the workers of the team (executing a parallel region) with the abstraction of synchronization among the iterations of a parallel-for-loop. This helps the programmer to code in the UW-OpenMP, ignoring the complexities arising out of the need to know the mapping of iterations to the workers at the time of placing barriers.

Correctness of Transformation

We first extend the definition of barrier given in Definition 3.3, in the context of our transformed programs: We use b_k^i to denote the $k^{th}(\geq 1)$ instance of the statement P[i]++ in the i^{th} UW-group.

The definitions of Statement instance Ψ_k^i and phase set Φ_k, in the context of the transformed program, depend on this new definition of b_k^i. The following theorem establishes the correctness of our transformation scheme.

THEOREM 1. *Let $S_{i,n}^{uw}$ denote the n^{th} instance of statement S executed by agent i in UW model and $S_{i,n}^{bm}$ denotes the instance of statement S executed by agent i in OM model in the transformed code. If $R \rightarrow_d S$ denotes that statement S is dependent on R then*

1. *[Phase-dependence] $S_{i,n}^{uw} \in \Phi_k \Rightarrow S_{i,n}^{bm} \in \Phi_k$.*
2. *[Data-dependence] $R_{i,m}^{uw} \rightarrow_d S_{j,n}^{uw} \Rightarrow R_{i,m}^{bm} \rightarrow_d S_{j,n}^{bm}$.*

PROOF. *(Sketch)* [Phase-dependence]: The k^{th} increment of P[i] in the transformed code in OM model corresponds to b_k^i in UW model. Based on the type of $S_{i,n}^{uw}$ (a simple-statement or conditional-barrier), Rules IV or V ensure that $S_{i,n}^{bm}$ executes in the phase Φ_k. This behavior is ensured by the guard P[i]==cp, since cp is incremented only when all the agents increment P[i]. This is equivalent to all agents executing b_k^i in the code executed in UW model.

[Data-dependence]: The phase-dependence part ensures that for each $R_{i,m}^{uw}$ and $S_{j,n}^{uw}$ there are corresponding elements in the same phase in the OM model. Since we do not add/remove any new statements and for each statement in the UW model, there is a corresponding statement in the OM model. Thus, it implies that no new statement executes in between $R_{i,m}^{bm}$ and $S_{j,n}^{bm}$; thus the dependencies are preserved. □

3.2 While Transformation

In this section, we extend the transformation rules presented in Section 3.1, to admit parallel-for-loops with a while-loop as the body;

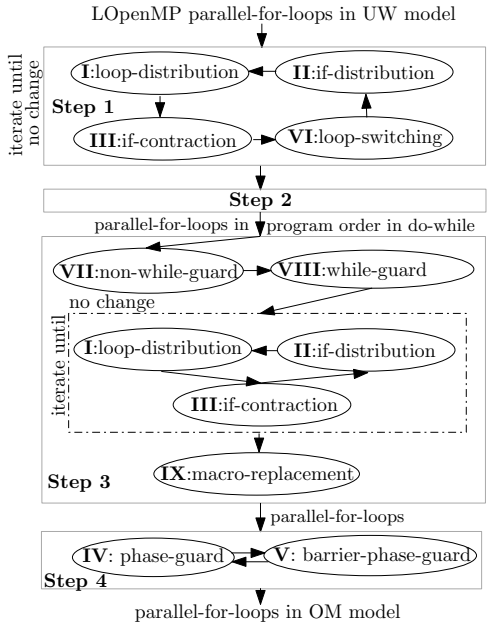


Figure 6: Block Diagram for While Transformation

the while-loop may in turn contain barriers within. Unlike a serial for-loop in normal form [24], while-loops introduce newer challenges because of the uncertainty of the number of iterations of the while-loop a priori during compilation. Note that a while-loop in LOpenMP has no nested serial loops inside.

Overall procedure: The overall block diagram for the translation of parallel-for-loops containing while-loops, carried out in four steps, is shown in Figure 6. In *Step 1*, we invoke the Rules I, II and III (Figure 5) along with the standard loop-switching rule (see Rule VI, Figure 7), till there is no change. *Step 2*: Similar to the Step 2 invoked in Section 3.1, we embed the body of the parallel region inside a do-while loop. We also initialize two new variables (`K` and `Rdy`) that are used for later bookkeeping (in some helper functions). *Step 3*: We first invoke the non-while-guard rule (Rule VII, in Figure 7), on all the parallel-for-loops whose body is not a while-loop containing barriers. We now process, in the program order, each of the parallel-for-loops that contains an immediately nested while-loop. Each such parallel-for-loop PF is processed through three sub-steps: (i) invoke the while-guard rule (Rule VIII, Figure 7) on PF, (ii) invoke Rules I, II, and III (again) on the generated code till there is no further change, and (iii) invoke the macro-replacement rule (Rule IX, Figure 7). *Step 4*: Invoke phase-guard and barrier-phase-guard (Rules IV and V, Figure 5).

We first explain the data structures and auxiliary functions used and then present an explanation of the transformation rules.

Data structures: Besides `pfCnt` (discussed in Section 3.1), we use an additional auxiliary counter `grCnt` (to indicate group-count). In Step 3, the value of `pfCnt` for a statement may change (due to the invocation of Rule I). We use $[\![\texttt{grCnt}]\!]$ to remember the `pfCnt` value (at the end of Step 2) for each statement. Note that the value of `grCnt` for any statement does not change in Step 3.

Consider a parallel-for-loop (PF), with an immediately nested while-loop (W); say W has statement `S` as the body and `e` as the predicate. Say, `S1` and `S2` are sequences of parallel-for-loops before and after PF, respectively. Each arbitrary iteration of PF, executes `S` until `e` evaluates to *false*, for that iteration. Thus, for any iteration `i` of PF, W contributes a sequence of statements (say, $s_{\texttt{i}}$) to the $\texttt{i}^{th}$ UW-group (say, denoted by $G_{\texttt{i}}$). In $G_{\texttt{i}}$, let $r_{\texttt{i}}$ (contributed by `S1`) and $t_{\texttt{i}}$ (contributed by `S2`) denote the sequence of statements executed before $s_{\texttt{i}}$ and after $s_{\texttt{i}}$, respectively.

Step 1

VI: Loop-switching

```
#pragma omp for
for(i=0;i<N;i++) {
 if(e1) {
  while(e2){ S;}}}
```

```
#pragma omp for
for(i=0;i<N;i++) {
  v = e1;
  while(e2){if(v){ S;}}
```

Step 2

```
#pragma omp parallel
    S
```

```
H=calloc(M,
  N*sizeof(int));
P=calloc(N,sizeof(int));
Rdy=calloc(N,
     sizeof(int));
cp=0; K=0;
#pragma omp parallel
do { S;
 if(Done(0))break;
} while (1);
```

Step 3

VII: Non-while-guard

```
do{ S1;
 #pragma omp for
 for(i=0;i<N;i++) S;
 S2;
 if(Done(N1)) break;
}while (1);
//S has no while
//loop with barrier
```

```
do { S1;
 #pragma omp for
 for(i=0;i<N;i++) {
  if(canEnter(i,[[grCnt]]))
   {S;iExit(i);}}
  S2;
  if(Done(N1+N))break;
}while (1);
```

VIII: While-guard:

```
do { S1;
 #pragma omp for
 for(i=0;i<N;i++)
  while (e) {S}
  //S has barriers
  S2
  if(Done(N1))
   break;
}while (1);
```

```
do { S1;
 #pragma omp for
 for(i=0;i<N;i++) {
  if(canEnter(i,[[grCnt]]))
  {v = e;
   if(v) {S;
    tmpreset(i,[[pfCnt]]);}
   if (!v){iExit(i);}}}
  S2
  if(Done(N1+N))break;
}while (1);
```

IX: Macro-replacement

```
#pragma omp for
for(i=0;i<N;i++)
  if(e){ S;
    tmpreset(i,C);}
```

```
#pragma omp for
for(i=0;i<N;i++)
 if(e) { S;
  reset(i,C,[[pfCnt]]);}
```

Figure 7: While Transformation Rules.

That is, $G_{\texttt{i}} = r_{\texttt{i}} \parallel s_{\texttt{i}} \parallel t_{\texttt{i}}$, where $\parallel$ is the concatenation operator. Our transformations ensure that the sequence of statements executed in the $\texttt{i}^{th}$ UW-group match $G_{\texttt{i}}$. Note that when the agent `i` has finished executing the code in $r_{\texttt{i}}$, statements in $r_{\texttt{i}}$ are said to be in *done* state, the statements in $s_{\texttt{i}}$ are considered ready to be executed (state = *ready*), and the statements in $t_{\texttt{i}}$ are considered to be not ready (state = *ignore*).

To realize the semantics discussed above, our transformation uses two auxiliary variables: i) a zero initialized integer array `Rdy` (of size `N`, the maximum number of iterations of the parallel-for-loops). For any iteration `i` of a parallel-for-loop PF, the value of `Rdy[i]` decides if the iteration `i` is 'ready' to be executed or not, by agent `i`. Say the group-count of PF is given by $[\![\texttt{grCnt}]\!]$, then if the value of `Rdy[i]` $= 2 * [\![\texttt{grCnt}]\!]$ or $2 * [\![\texttt{grCnt}]\!] + 1$, it indicates that the $\texttt{i}^{th}$ iteration of PF, is ready to be executed, for the first time or subsequent time, respectively. ii) we use a zero initialized shared counter `K` to keep a count of the number of parallel-for-loop iterations (spread across multiple parallel-for-loops in the parallel region) that have finished executing. Say, the parallel region contains exactly two parallel-for-loops having N_1 and N_2 iterations, then the maximum value of `K` is $N_1 + N_2$.

Auxiliary procedures: The transformed code uses four auxiliary procedures, shown in Figure 8: (i) `canEnter`, (ii) `iExit`, (iii)

```
int canEnter(int i,int g){
 if(Rdy[i]==2*g){
   Rdy[i]++; return 1;}
 if(Rdy[i]==2*g+1){
   return 1;}
 return 0; }
int Done(int maxCnt){
  #pragma omp barrier
  return K == maxCnt; }
```

```
int iExit(int i){
 Rdy[i]++;
 #pragma omp atomic
  K++; return 1;}
void reset(int i,
     int l,int h){
  for (j=l;j<=h;j++)
    H[j][i]=0;
}
```

Figure 8: The Auxiliary Procedures

1. **init:** (a) ⟦i,0⟧=*ready* (b) $\forall g \geq 1$, ⟦i,0⟧=*ignore*
2. ⟦i,g−1⟧=*exec* ∧ iExit(i) ⇒ ⟦i,g−1⟧=*done* ∧ ⟦i,g⟧= *ready*
3. ⟦i,g⟧=*ready* ∧ canEnter(i,g) ⇒ ⟦i,g⟧=*exec*
4. ⟦i,g⟧=*exec* ∧ canEnter(i,g) ⇒ ⟦i,g⟧=*exec*

Figure 9: State change rules for agent i.

reset, and (iv) Done. The i^{th} iteration of the parallel-for-loop with grCnt=g will execute the body of the parallel-for-loop, only if canEnter(i,g) returns 1. If the value of Rdy[i]=2*g, it increments Rdy[i] by 1 (indicates that the body has started executing). The function call iExit(i) increments Rdy[i] by 1 (indicates that the i^{th} agent has finished executing the i^{th} iteration of the current parallel-for-loop and is ready to start with the next parallel-for-loop). And the value of K is atomically incremented by 1, to indicate that the body of one more parallel-for-loop has been executed by agent i. Note: the value of Rdy[i] at the end of a iExit(i) call is guaranteed to be even. The function reset is used to reset a part of statement execution history H (Section 3.1). The function call reset(i,l,h) resets the statement execution history corresponding to the i^{th} agent, in the parallel-for-loops whose pfCnt values vary between l and h (both inclusive). The function Done checks if K matches its expected maximum value (passed as an argument). The barrier ensures that the routine waits for all the updates to K, by any parallel task, before evaluating the predicate. We now explain the Rules given in Figure 7.

Transformation rules: Rule #VII and VIII together provide an interesting mechanism to avoid the while-loop altogether. We start with Rule VIII for the ease of explanation.

In any iteration of the outer do-while loop (introduced in Step 2), the state ⟦i,g⟧ of an agent i, inside any parallel-for-loop with grCnt=g and body as S, can be one of the following: (a) *ready*: i is going to execute S for the first time, (b) *exec*: when the state changes from *ready* to *exec*, it indicates that the agent i will execute S one or more times. (c) *done*: i has finished executing S, or (d) *ignore*: i is in *exec* or *ready* state inside a parallel-for-loop with grCnt=k and g>k. The function canEnter(i,g) returns 1, only if ⟦i,g⟧ is *ready* or *exec*. Fig. 9 shows the state change rules.

In the transformation shown for Rule VIII, we can execute e and S only if canEnter(i,g) returns 1. After the first invocation of canEnter, the element Rdy[i] gets incremented by 1 and ⟦i,g⟧ changes to *exec*. In this *exec* state, agent i executes the statements in the parallel-for-loop with grCnt=g, and not execute any other statements in any other parallel-for-loop with grCnt≠ g (as their state is *done* or *ignore*). Once the predicate e becomes *false* for the agent i, the function iExit(i) is called to increment Rdy[i] by 1. As a result, ⟦i,g⟧ changes from *exec* to *done* and ⟦i,g+1⟧ changes from *ignore* to *ready*. Thus, the parallel-for-loops with grCnt=g+1 are now ready to execute. And this process continues till all the states are set to *done*.

As discussed in Section 3.1, the statement execution history map H ensures that there are no duplicate executions of any statement. However, in case of the body S of a while-loop nested inside a parallel-for-loop, S may have to be executed multiple times. Thus, at the end of one iteration of the while-loop, by a parallel iteration i, we have to reset the corresponding H value entries for i. Considering the possibility that the code generated in Rule VIII, may be distributed in later transformations (leading to a sequence of parallel-for-loops), we use a temporary function tmpreset to remember the starting index (pfCnt) of the sequence and replace it with the auxiliary function reset and supply the value of the last index in Rule IX.

Rule VIII also increments the argument to the Done function by N, to indicate that N parallel-iterations are executed as part of the current parallel-for-loop.

Rule VII: It handles the case when the body S does not contain a while-loop with a barrier inside. In the transformed code, agent i executes S when canEnter(i,⟦grCnt⟧) returns 1. Note that the agent i executes S only once. And once executed, in the future iterations of the outer do-while loop, agent i will never execute S. This is ensured by the call to iExit(i) after the execution of S.

Phase-guard rules: Similar to the Step 3 in Section 3.1, we invoke the rules IV and V (Figure 5). Note: this may lead to multiple invocations of the canEnter function (max 2 times) for the same statement. This does not alter the semantics (as canEnter has no side effects), and is optimized away by a later pass (Section 4).

In the transformed code, the body of the do-while loop may have different parallel-for-loops. The technique explained above ensures that the statements which were part of a while-loop in the original code, repeat as many number of times as executed by the original while-loop in UW model, and the statements which are not part of any while-loop (in the original code) are executed only once.

Example Translation: Figure 10a shows a simplified version of the code discussed in Figure 2. Figures 10b–10g show the impact of applying our transformation rules on the input UW-OpenMP code (Figure 10a), along with the sequence of the rules applied to obtain the code in the figure. For brevity, we show the code generated after some of the important transformations only.

We use a sample execution to show how the semantics of Figure 10g (executing in OM-OpenMP) matches that of Figure 10a (executing in UW-OpenMP). Say, the while-loop in the latter executes its body twice. Statement S3 is executed only after all the iterations finish the execution of the while-loop.

In Figure 10g, the arrays H and P, and the variable cp are initialized to 0. Similarly, the variable flag is initialized to 1. We term the predicate guarding the body of each of the six parallel-for-loops (L0-L5), within the do-while loop, as the *guarding-predicate*.

In Figure 10g, during the first iteration of the do-while loop, the guarding-predicate will evaluate to true for L0, L1 and L2. In L0, v[i] will be evaluated to true by all the iterations. In L1, S1 will be executed by all the iterations. In L2, all the iterations increment P[i] to 1 and as a result the predicate P[i]!=(cp+1) evaluates to false (as cp=0). Next, cp is incremented to 1, by a single thread. This indicates a phase increment; it is equivalent to finishing the execution of the barrier statement by all the agents during the first iteration of the while-loop in Figure 10a. Now, in Figure 10g, all the workers execute the continue statement and jump to the start of the do-while loop.

During the second iteration of the do-while loop, the last term in the guarding-predicate evaluates to false for L0, L1 and L2. And the control reaches L3. In L3, S2 and the function reset(i,0,3) is called by all the iterations, which resets the value of all H array elements corresponding to the parallel-for-loops generated because of while-loop. In L4, since the predicate !v[i] evaluates to false, iExit(i) is not executed which keeps canEnter(i,0) as true. In L5, S3 will not be executed because canEnter(i,1) evaluates to false. The execution of the do-while loop continues because Done(N+N) evaluates to false.

```
// Input code
#pragma omp parallel
{
 #pragma omp for
 for(i=0;i<N;i++)
 {
  while(delta>epsilon)
  {
   S1;
   #pragma omp barrier
   S2;
  }
  S3;
 }
}
```
(a)

```
//after Rule I
#pragma omp parallel
{
 #pragma omp for
 for(i=0;i<N;i++){
  while(delta>epsilon)
  {
   S1;
   #pragma omp barrier
   S2;
  }
 }
 #pragma omp for
 for(i=0;i<N;i++) S3;
}
```
(b)

```
//after Step 2
#pragma omp parallel
 do {
 #pragma omp for
 for(i=0;i<N;i++){
  while(delta>epsilon){
   S1;
   #pragma omp barrier
   S2;
  }
 }
 #pragma omp for
 for(i=0;i<N;i++) S3;
  if(Done(0)) break;
 } while(1);
```
(c)

```
//after Rule VII
#pragma omp parallel
do{
 #pragma omp for
 for(i=0;i<N;i++){
  while(delta>epsilon){
   S1;
   #pragma omp barrier
   S2; }}
  #pragma omp for
  for(i=0;i<N;i++)
   if(canEnter(i,1)) {
    S3;iExit(i);}
 if(Done(N)) break;
}while(1);
```
(d)

```
//after Rule VIII
#pragma omp parallel
do {
 #pragma omp for
 for(i=0;i<N;i++){
  if(canEnter(i,0)){
   v=delta>epsilon;
   if(v){
    S1;
    #pragma omp barrier
    S2;
    tmpreset(i,0);}
   if(!v)iExit(i);
  }}
 #pragma omp for
 for(i=0;i<N;i++)
  if(canEnter(i,1)){
   S3;iExit(i);}
 if(Done(N)) break;
}while(1);
```
(e)

```
//Rules II,I,II,I,III
#pragma omp parallel
do {
 #pragma omp for
 for(i=0;i<N;i++){
  v1[i]=canEnter(i,0);
  if(v1[i]) {
   v[i]=delta>epsilon;}}
 #pragma omp for
 for(i=0;i<N;i++){
  if(v1[i] && v[i]){S1;
   #pragma omp barrier
   S2; tmpreset(i,0);}}
 #pragma omp for
 for(i=0;i<N;i++){
  if(v1[i] && !v[i])
   iExit(i);}
 #pragma omp for
 for(i=0;i<N;i++)
  if(canEnter(i,1)){
   S3;iExit(i);}
 if(Done(N+N)) break;
 }while(1);
```
(f)

```
//Rules I,II,IX,IV,V
#pragma omp parallel
do{
 #pragma omp for // L0
 for(i=0;i<N;i++){
  if (canEnter(i,0) &&
   P[i]==cp&&!H[0][i]){
   H[0][i]=1;
   v1[i]=canEnter(i,0);
   if(v1[i])
    v[i]=delta>epsilon;
 }}
 #pragma omp for // L1
 for(i=0;i<N;i++)
  if (canEnter(i,0) &&
   P[i]==cp&&!H[1][i]){
   H[1][i]=1;
  if(v1[i] && v[i]) S1;}
 #pragma omp for // L2
 for(i=0;i<N;i++){
  if (canEnter(i,0) &&
   P[i]==cp&&!H[2][i]){
   H[2][i]=1;
   if(v1[i]&&v[i])P[i]++;}
   if(P[i]!=(cp+1)) {
    #pragma omp atomic
     flag=0;  }}
 //contd.  in the next col
```
(g)

```
//contd.  from the prev col
 #pragma omp single {
  tmpflag = flag;
  if(flag) cp++;
  flag = 1;}
 if(tmpflag) continue;
 #pragma omp for // L3
 for(i=0;i<N;i++)
  if (canEnter(i,0) &&
   P[i]==cp&&!H[3][i]){
    H[3][i]=1;
    S2;  reset(i,0,3);}
 #pragma omp for // L4
 for(i=0;i<N;i++)
  if (canEnter(i,0) &&
   P[i]==cp&&!H[4][i]){
    H[4][i]=1;
    if(v1[i] && !v[i])
       iExit(i);}
  #pragma omp for // L5
  for(i=0;i<N;i++)
  if (canEnter(i,1) &&
   P[i]==cp&&!H[5][i]){
    H[5][i]=1;
    if(canEnter(i,1)){
     S3;iExit(i);}}
 if(Done(N+N)) break;
 }while(1);
```

Figure 10: Effect of our transformation rules shown in Figures 5 and 7. (a) input UW-OpenMP code. (b)-(g) code after applying different rules.

The third and fourth iterations of the do-while loop execute similar to that of the first and second iterations respectively. In the fifth iteration of the do-while loop, the predicate `v[i]` will be set to false. Hence in L4, `iExit(i)` will be executed. As a result, in L5, `canEnter(i,1)` evaluates to true and consequently, `S3` and `iExit(i)` are executed. Now `Done(N+N)` evaluates to true and the control comes out of the do-while loop.

3.3 Generalizing the Translation

Given any general UW-OpenMP program, we first translate it to LOpenMP and then invoke the transformations discussed in Section 3 that deal with programs written in LOpenMP.

OpenMP sections: Each parallel-section inside a `#pragma omp sections` is executed by a worker available in the parallel region. We can translate it to an equivalent parallel-for-loop (number of iterations equal to the number of sections) such that its i^{th} iteration executes the i^{th} section.

Statements outside parallel-for-loop: Figure 11 shows a simple scheme to handle statements, in an OpenMP parallel region, not nested inside any work-sharing construct.

```
#pragma omp parallel
{
  ...
  S
  ... }
```

```
T=omp_get_numthreads();
#pragma omp parallel
{...
 #pragma omp for
 for(i=0;i<T;++i){S} ...}
```

Figure 11: Statement outside parallel for

```
while(e1){
 S1;
 while(e2)
   S2;
 S3;}
```

```
inInnner=0;
while((!inInner&&e1)||(inInner)){
 if(!inInner) S1;
 inInner=e2;
 if(inInner) S2; if(!inInner) S3;}
```

Figure 12: Nested while loops

Nested while loops: Figure 12 shows a simple transformation to convert a nested while-loop to a non-nested one. It uses an additional variable (`inInner`) to execute the correct code.

Barriers inside functions: If a non-recursive function containing a barrier is called from a work-sharing construct, it is first inlined (as a pre-processing step). In case of a recursive function, it is specialized to take additional arguments for the auxiliary data structures. We skip the details for lack of space.

```
#pragma omp for
for(i=0;i<N;i++) {
 if(P[i]==cp&&!H[C][i]){
   H[C][i] = 1; S1;}}
#pragma omp for
for(i=0;i<N;i++) {
 if(P[i]==cp&&!H[C+1][i]){
   H[C+1][i] = 1; S2;}}
```

```
#pragma omp for
for(i=0;i<N;i++) {
 if(P[i]==cp&&
      !H[C][i]){
   H[C][i] = 1;
   S1;
   S2;}}
```

Figure 13: Loop fusion

Scheduling policies: One of the advantages of our proposed translation scheme is that the semantics of the generated code is independent of the scheduling policy [5] used in the program (static, dynamic, guided, auto, or runtime). This is because the generated code (in OM-OpenMP) never invokes barriers inside any work-sharing construct. Note that to handle the possible issues arising due to multiple invocations of `omp_get_thread_num`, for each iteration, we cache the returned value of this function in a local data structure and use it in future invocations.

4 CLEANUP OPTIMIZATIONS

We now briefly describe a few optimizations to help generate efficient code; these optmizations mainly target reducing the overheads (in space and time) resulting from our translation scheme.

Direct loop distribution: In the context of the rules discussed in Section 3.1, if there exists a sequence of unconditional barriers inside a parallel-for-loop (may be interspersed with serial code in between), before any conditional barrier: we can apply loop-distribution around those unconditional barriers (incurs no overhead) and then apply the techniques of Section 3.1 on the rest of the parallel-for-loop starting with the conditional statement enclosing the barrier. We extend this logic further, to include conditional barriers guarded by single-valued expressions [25]. In such a case, we can apply the if-distribution rule, followed by the loop-distribution rule (Figure 5) and avoid any overhead, whatsoever.

Loop fusion: Figure 13 shows an extension to the standard loop fusion [15] technique to fuse some of the consecutive parallel-for-loops generated by our transformation. This helps in eliding the implicit barrier after the first parallel-for-loop, avoiding a predicate check. As shown in the text in **bold**, we are able to fuse the two loops even though the predicates are different. The underlying semantics of the `H` array guarantees the semantics preservation.

Removing avoidable atomic operations: We eliminate the atomic construct introduced for the write to the shared variable `flag` (in Rule V, Figure 5). Though this optimization may lead to non-conforming OpenMP code, we argue that it is acceptable, as the resulting race (between multiple writes to `flag`) is a benign one [26]: all of them write the same value 0. And since at most one bit may change (the old value of `flag` may only be either 0 or 1), it does not lead to partial writes [27].

Code specialization: Considering the possible overheads associated with our generated code (due to the While-transformation, discussed in Section 3.2), we specialize the transformed code (if the transformation involves the While-transformation) such that, when the number of workers matches the number of parallel iterations, the input parallel region is executed as it is, along with an option to set the chunk-size of the parallel-for-loop to 1. Though this optmization leads to non-conforming OpenMP code, the code will run according to the semantics of UW-OpenMP.

Barrier elimination: We make a pass over the generated code to check if there are two consecutive barrier statements (implicit or explicit), and eliminate one of them.

Redundant `canEnter` elimination: We make a simple pass over the code to eliminate calls to the `canEnter` function, nested within an if-statement whose predicate includes a call to `canEnter`.

```
#pragma omp for
for (i=0;i<n;++i)
  S1;
#pragma omp for
for (i=0;i<n;++i)
  S2;
```

(a)

```
#pragma omp for
for (i=0;i<n;++i){
  S1;
  #pragma omp barrier
  S2; }
```

(b)

```
while (cond) {
#pragma omp parallel
{
 #pragma omp for
 for (i=0;i<n;++i)
   S1;
 #pragma omp for
 for (i=0;i<n;++i)
  S2; } }
```

(c)

```
#pragma omp parallel
{
#pragma omp for
for (i=0;i<n;++i)
 while (cond) {
   S1;
   #pragma omp barrier
   S2; } }
```

(d)

Figure 14: (a) and (b) (and, similarly (c) and (d)) encode different program logics.

5 DISCUSSION

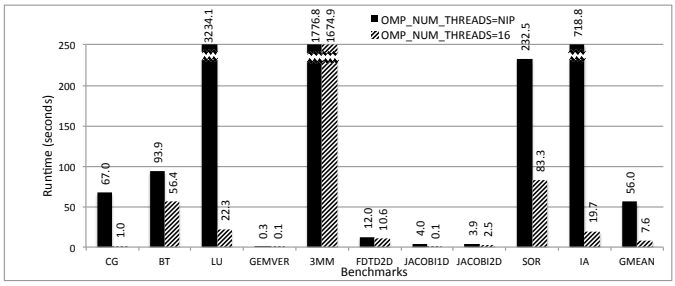
We now present a brief discussion on the expressiveness in our proposed extension UW-OpenMP. Depending on the specific problem under consideration, the intuitive mechanism to encode the solution may vary. Consider two typical solutions: i) Create `n` parallel tasks to execute `S1`; wait till all the tasks are completed; create `n` parallel tasks to execute `S2`. ii) Create `n` parallel tasks to compute `S1`, followed by `S2`, and ensure that each task waits for each other, after executing `S1`. Depending on the particular problem under consideration the programmer may find it convenient to encode the program logic using the first solution or the second. Unlike OM-OpenMP, where only the first solution can be coded (Figure 14a), UW-OpenMP allows the encoding of both the solutions (Figure 14a, and Figure 14b).

Similarly, consider two solutions typically applicable in many iterative computations (e.g., iterated averaging [12], Jacobi kernels from the polybench suite [28]): i) Repeat the following till convergence: in each iteration create `n` tasks to execute `S1`; wait till all the tasks are completed; and then create `n` tasks to execute `S2`; wait till all the tasks are completed. ii) Create `n` tasks, where each task executes the following code till convergence: execute `S1` followed by `S2`, and ensure that the tasks wait for each other after every execution of S_i (lock step synchrony). These two solutions are encoded in Figure 14c and Figure 14d, respectively. As has been argued in Section 1, unlike OM-OpenMP, where only the first solution can be coded (Figure 14c), UW-OpenMP allows the coding of both the solutions (Figure 14c, and Figure 14d).

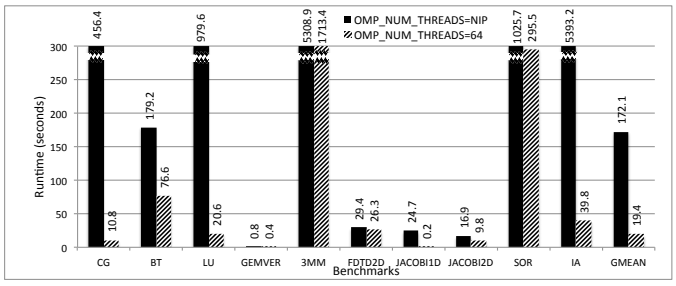
In general, we see that UW-OpenMP is more expressive than OM-OpenMP, and gives the programmer more choices to encode the program logic.

6 IMPLEMENTATION AND EVALUATION

We implemented our proposed techniques as an AST to AST transformation in the ROSE compiler framework [16], which supports OpenMP 3.0 specifications. It takes as input UW-OpenMP programs and outputs equivalent OM-OpenMP code. We use the GCC 4.4 compiler to compile any of these OpenMP programs. To understand the impact of our transformations on varying hardware configurations, we conducted experiments on two different machines: an Intel 16 core system (two E5-2670 processors, 64GB RAM, hyper threading disabled), and an AMD 64 core system (four AMD 6376 processors, 512GB RAM, no hyper threading). We found that the compilation time overheads were negligible.



(a) 16 core Intel system, #H/W cores = 16.



(b) 64 core AMD system, #H/W cores = 64.

Figure 16: Performance comparison of the input UW-OpenMP code (executed by setting `OMP_NUM_THREADS` to NIP) with the translated code (executed by setting `OMP_NUM_THREADS` to #cores).

	Bench	Source	#LOC			Input class and size
			UW	OMP	%less	
1	CG	NPB [29]	851	868	1.2	A=14k
2	BT	NPB [29]	3607	3616	0.2	B=102
3	LU	NPB [29]	3470	3481	0.3	B=102
4	GEMVER	PB [28]	177	181	2.2	L=8000
5	3MM	PB [28]	100	102	1.96	EL=8000
6	FDTD2D	PB [28]	152	154	1.2	EL=4000
7	JACOBI1D	PB [28]	116	117	0.8	ST=10000
8	JACOBI2D	PB [28]	123	124	0.8	EL=4000
9	SOR	[30]	75	77	2.6	3000
10	IA	[12]	78	80	2.5	500

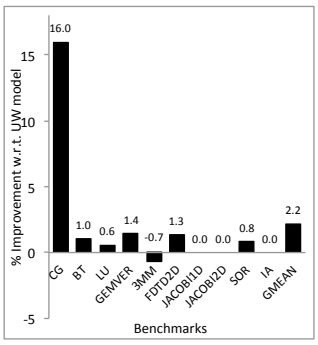
Figure 15: Characteristics of the benchmark kernels.

For our evaluation, we first created UW-OpenMP versions (UW) of ten popular benchmark kernels (see Figure 15). This involved, parallelizing five kernels (3MM, GEMVER, FDTD2D, JACOBI1D and JACOBI2D) from the PolyBench [28] suite (to UW-OpenMP), rewriting CG, BT, LU and SOR in UW-OpenMP, and designing a standalone kernel for IA (based on [12]). One common feature of all these kernels is that each kernel contains at least one parallel-for-loop with at least one nested barrier within; in case of IA, JACOBI1D, and JACOBI2D the barrier is nested inside a serial loop, which in turn is nested inside the parallel-for-loop. For all of the ten kernels, we also considered their equivalent OM-OpenMP variants (OMP). The main difference between these two versions is the way in which parallelism is expressed in the programs (see Section 5).

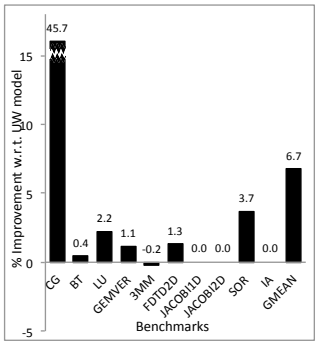
Since the OpenMP versions distribute the code to perform certain coherent computation across multiple parallel-for-loops, it led to codes that are lengthy (marginally higher – around 1-17 lines) and arguably more complex (less readable – related parts of the parallel-for-loop are not together). For NPB and PolyBench (PB) kernels, we chose the largest class of input for which we could run the kernels. In case of SOR and IA, we chose input sizes such that the code generated by our transformation technique (from the input UW-OpenMP kernels) took at least a few tens of seconds (on the Intel system, by setting the environment variable `OMP_NUM_THREADS` to 16). In all these ten kernels the input size also matches the number of iterations of the main parallel-for-loop (abbreviated as NIP).

The first part of our evaluation is to address a naive argument that instead of following the elaborate translation scheme, we could let the programmer write programs in UW-OpenMP and we just set the `OMP_NUM_THREADS` to NIP at runtime (guarantees a consistent behavior). Figure 16 presents a comparison of the input UW-OpenMP code (executed by setting `OMP_NUM_THREADS` = NIP) and the translated code (executed by setting `OMP_NUM_THREADS` = #cores).

Figure 16a shows that on the Intel system, the execution times of our translated code are clearly lower than that of the UW-OpenMP code; a geometric mean % difference of $100 \times (1 - \frac{7.6}{56.0}) = 86.4\%$. The exact difference for any particular benchmark depends on the value of NIP and the work done in each iteration. Figure 16b shows



(a) 16 core Intel system.



(b) 64 core AMD system.

Figure 17: Performance comparison of the generated and input code, both executed by setting `OMP_NUM_THREADS`=NIP.

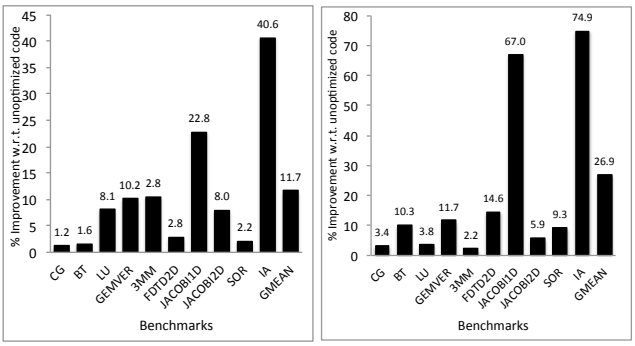
the corresponding comparison on the 64 core AMD system. On the AMD system the geometric mean % difference is 88.7%.

This behavior is consistent with the common knowledge that setting `OMP_NUM_THREADS` to a large number will lead to significant performance degradation. And we show that use of our proposed scheme is definitely better than using such a naive scheme.

To explain the impact of the overheads resulting from our generated code, Figure 17 shows the comparison of the performance of the generated code (α) and the input code (β), both executed by setting `OMP_NUM_THREADS` set to NIP. Figure 17a shows the performance improvement on the 16 core Intel system (% improvement = $100\times$(1 - exec time of α / exec time of β). Except for 3MM, it can be seen that most of our generated codes ran faster than the input codes, even though `OMP_NUM_THREADS` was set to NIP. This indicates that the effects of the overheads is eclipsed by the proposed optimizations. Interestingly, in case of 3MM the performance gains actually show a dip due to some of the overheads introduced by our transformations. Overall, we observe that the impact of the overheads in our proposed transformation is more than nullified by the optimizations discussed in Section 4. We note that in case of the 64 core AMD system (Figure 17b), the performance of our generated code is similar to that seen on the 16 core Intel system.

Effect of the Optimizations: We studied the effect of the proposed optimizations (Section 4) on the generated code. In this regard, we invoked our transformation phase, with and without the optimizations, on each of the benchmark kernels.

Figure 18a shows the effect of the optimizations on the Intel system: the gains varied between 1.2% (CG) to 40.6% (IA), and geometric mean gain = 11.7%. The significant gains in IA and JACOBI1D are due to the optimizations which reduced the number of redundant barriers. The gains due to optimizations on the AMD system (Figure 18b) are similar to the ones shown in Figure 18a: the gains varied between 2.2% (3MM) to 74.9% (IA), and geomet-



(a) Intel system, #cores=16. (b) AMD system, #cores=64.

Figure 18: Comparison of unoptimized and optimized codes (both executed by setting `OMP_NUM_THREADS` = #cores).

ric mean gain = 26.9%. Note that the gains on the AMD system are a bit more pronounced because of the increased number of cores.

Impact of code specialization: This optimization is applicable only when `OMP_NUM_THREADS`=number of iterations of the main parallel-for-loop, that has a while loop. For the evaluation shown in Figure 17, code specialization was applicable for JACOBI1D, JACOBI2D and IA. The percentage improvement varied between 51-91% on the AMD system and 43-65% on the Intel system.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose UW-OpenMP (an extension to OpenMP) that allows the placement of barriers anywhere inside the OpenMP work-sharing constructs (parallel-for-loop and parallel-sections), and helps the programmer to realize synchronization among the different parallel tasks (instead of the workers), which is arguably a natural way to express parallel algorithms. Our proposed method results in concise and simpler programs, without compromising on the efficiency resulting from the underlying team of workers model of OpenMP. We validate the same by presenting an evaluation on two different hardware systems.

Extending UW-OpenMP to support OpenMP 3.0 [5] tasks (defined using `#pragma omp task`) would be an interesting future work. Further, extending UW-OpenMP to guarantee deadlock freedom would be another interesting challenge.

Acknowledgments: The authors would like to acknowledge Indu K, Aman Nougrahiya, Manas Thakur, and Jyothi Krishna V S for their valuable comments on an initial draft of this manuscript. We also thank Babu Viswanath (@ IIT Madras) and Dave Strenski (@ Cray Inc) for making available the Cray based compiler tool chains for our experiments. This research is partially supported by the New Faculty Seed Grant, funded by IIT Madras CSE/11-12/567/NFSC/NANV, DAE research grant CSE/13-14/139/BRNS/NANV and DST Fasttrack grant CSE/13-14/140/DSTX/NANV.

8 References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *OOPSLA*, pages 519–538. ACM, 2005.

[2] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *PPPJ*, pages 51–61, 2011.

[3] Cray Inc. The Chapel Language Specification. Technical report, Feb 2013. http://chapel.cray.com.

[4] Intel Cilk Plus. https://www.cilkplus.org/.

[5] OpenMP Application Program Interface, Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[6] K. Alan. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *IPDPS*, pages 1–8, 2007.

[7] J. M. Bull and M. E. Kambites. JOMP-an OpenMP-like Interface for Java. In *Java Grande*, pages 44–53. ACM, 2000.

[8] GCC Compiler. https://gcc.gnu.org/.

[9] CRAY CC Compiler. https://www.nersc.gov/users/software/compilers/.

[10] OpenMP C/C++ Language Extensions in Clang/LLVM Compiler. http://clang-omp.github.io/.

[11] ICC Compiler. https://software.intel.com/en-us/c-compilers.

[12] S. Deitz. A Comparison of a 1D Stencil Code in Co-Array Fortran, Unified Parallel C, X10, and Chapel. In *IDRIS*, 2010. http://chapel.cray.com/presentations/Stencil1D.pdf.

[13] J. Shirako, J. Zhao, V. K. Nandivada, and V. Sarkar. Chunking Parallel Loops in the Presence of Synchronization. In *ICS*, pages 181–192. ACM, 2009.

[14] OpenMP discussion forum. http://openmp.org/forum/viewtopic.php?f=3&t=945/.

[15] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[16] D. J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Proc. Letters*, 10(2/3):215–226, 2000.

[17] J. Shirako, K. Sharma, and V. Sarkar. Unifying Barrier and Point-to-point Synchronization in OpenMP with Phasers. In *IWOMP*, pages 122–137. Springer-Verlag, 2011.

[18] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(1):3:1–3:48, April 2013.

[19] A. Nicolau, G. Li, A. V. Veidenbaum, and A. Kejariwal. Synchronization Optimizations for Efficient Execution on Multi-cores. In *ICS*, pages 169–180. ACM, 2009.

[20] C. Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. In *PPoPP*, pages 144–155. ACM, 1995.

[21] S. Satoh, K. Kusano, and M. Sato. Compiler Optimization Techniques for OpenMP Programs. *Sci. Program.*, pages 131–142, 2001.

[22] G. Bronevetsky and B. R. de Supinski. Complete Formal Specification of the OpenMP Memory Model. *Int. J. Parallel Program.*, 35(4):335–392, August 2007.

[23] J. P. Hoeflinger and B. R. De Supinski. The OpenMP Memory Model. In *IWOMP*, pages 167–177, 2008.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[25] A. Aiken and D. Gay. Barrier inference. In *POPL*, pages 342–354, 1998.

[26] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31. ACM, 2007.

[27] H. Boehm. How to Miscompile Programs with "Benign" Data Races. HotPar. USENIX Association, 2011.

[28] PolyBench/C Suite. http://www.cs.ucla.edu/ pouchet/software/polybench/.

[29] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[30] R. K. Karmani, N. Chen, B. Su, A. Shali, and R. Johnson. Barrier Synchronization Pattern. In *ParaPLOP*, 2009.