Efficiency and Expressiveness in UW-OpenMP

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

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

ABSTRACT

OpenMP uses the efficient 'team of workers' model, where workers are given chunks of *tasks* (iterations of a parallel-for-loop, or sections in a parallel-sections block) to execute, and worker (not tasks) can be synchronized using barriers. Thus, OpenMP restricts the invocation of barriers in these tasks; as otherwise, the behavior of the program would be dependent on the number of runtime workers. To address such a restriction which can adversely impact programmability and readability, Aloor and Nandivada proposed UW-OpenMP by taking inspiration from the more intuitive interaction of tasks and barriers in newer task parallel languages like X10, HJ, Chapel and so on. UW-OpenMP gives the programmer an impression that each parallel task is executed by a unique worker, and importantly these parallel tasks can be synchronized using a barrier construct. Though UW-OpenMP is a useful extension of OpenMP (more expressive and efficient), it does not admit barriers within recursive functions invoked from parallel-for-loops, because of the inherent challenges in handing them. In this paper, we extend UW-OpenMP (we call it UWOmp++) to address this challenging limitation and in the process also realize more efficient programs.

We propose a source to source transformation scheme to translate UWOmp++ C programs to equivalent OpenMP C programs that are guaranteed not to invoke barriers in any task. Our translation uses a novel intermediate representation called UWOmpCPS, which represents a parallel program written in OpenMP in an extended CPS format (admits parallel-for-loops and barriers). The use of this intermediate representation leads us to handle recursive functions within parallel-for-loops efficiently. We have implemented our proposed translation scheme in the ROSE compiler framework. Our preliminary evaluation shows that the proposed language extension to allow recursion helps plug an important gap in expressiveness, without compromising on the efficiency resulting from the 'team of workers' model.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; *Compilers*; • **Computing methodologies** → *Parallel programming languages.*

KEYWORDS

OpenMP, Unique Worker model, Recursive task parallelism

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
CC '19, February 16–17, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6277-1/19/02...$15.00
https://doi.org/10.1145/3302516.3307360

ACM Reference Format:
Raghesh Aloor and V. Krishna Nandivada. 2019. Efficiency and Expressiveness in UW-OpenMP. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3302516.3307360

1 INTRODUCTION

The advent of multi-core systems has brought along an increased interest in task parallel languages like X10 [?], HJ [?], Chapel [?], OpenMP [?] and so on. These task parallel languages advocate that the programmers should think in parallel and use the language to express the parallel logic; not treat parallelism as an afterthought. For example, in languages like X10, HJ and Chapel, from the point of view of a programmer, a parallel loop creates many tasks that can execute in parallel; and the program logic can remain oblivious to the runtime-threads. Languages like X10, HJ and Chapel also let the tasks synchronize among themselves.

For example, Consider the code snippets (written in HJ) shown in Figure 1a; it performs distributed iterative averaging (IA) [? ?] over a one-dimensional array. In HJ, a `forall` loop is used to create parallel tasks and a `next` statement is a barrier used to synchronize the tasks. The programmer here can visualize each task as an independent computation (conveniently packaged inside the function `iterAvg`) and consider synchronizing these tasks using barriers at appropriate points. In Figure 1a, all the tasks synchronize twice in each iteration of the while-loop (see Lines 8 and 13).

Figure 1b shows a variation of IA that uses recursion. Here each parallel task invokes the recursive function `iterAvgR` (Line 2, Figure 1b). Similar to the iterative version, these tasks invoke the barrier (`next`) at two positions (Lines 8 and 12) and repeat the computation and synchronization till convergence (Line 5). Compared to the iterative version, this way of visualizing tasks as recursive functions may be intuitive and convenient for some programmers.

The codes shown in Figure 1a and Figure 1b support a natural understanding of the programmer that the number of worker threads (*workers* in short) will match the number of tasks. Further, the synchronization happens among the tasks (iterations of the `forall` loop) rather than workers. However, in contrast, the OpenMP specification [?], because of the underlying 'team of workers' model, defines barriers for synchronizing workers, which prevents the programmer from placing barriers within parallel-for-loops to synchronize its iterations; hereafter we refer to the iterations of the parallel-for-loops and sections in parallel-sections as tasks. Consequently, the equivalent OpenMP C codes for the code-snippets shown in Figures 1a and 1b, (e.g., Figures 1c and 1d) are OpenMP non-conforming. Aloor and Nandivada [?] show that such OpenMP programs (or equivalent programs in languages like PJ [?], and JOMP [?]) may produce varying outputs, or even hang, for varying number of workers; here the expected output is guaranteed,

```
1  ... // in some function
2  forall (int i: [1..N]) {iterAvg(i);}
3  ...
4  void iterAvg(int i){
5    while(delta>epsilon){
6     newA[i]=(oldA[i-1]+oldA[i+1])/2.0;
7     diff[i]=Math.abs(newA[i]-oldA[i]);
8     next;
9     if (i==1) {
10     delta=sum(diff); iters++;
11     temp=newA;newA=oldA;oldA=temp;
12    } /* if */
13    next;
14   } /* while */ }/*iterAvg*/
```

(a) Non-recursive version in HJ; `oldA` and `newA` are shared

```
1  ... // in some function
2  forall (int i: [1..N]) {iterAvgR(oldA, newA, i);}
3  ...
4  void iterAvgR(double[] oldA, double[] newA, int i){
5   if (delta<=epsilon) { return; }
6   new[i]=(old[i-1]+old[i+1])/2.0;
7   diff[i]=Math.abs(newA[i]-oldA[i]);
8   next;
9   if (i==1) {
10    delta = sum(diff); iters++;
11  } /* if */
12  next;
13  iterAvgR(newA,oldA,i);
14 } /* iterAvgR */
```

(b) Recursive version in HJ

```
1  ... // in some function.
2  #pragma omp parallel
3  { #pragma omp for
4     for(i=0;i<N;i++){iterAvg(i);}
5  }/*parallel*/
6  ...
7  void iterAvg(int i) {
8    while(delta>epsilon){
9     newA[i]=(oldA[i-1]+oldA[i+1])/2.0;
10    diff[i]=fabs(newA[i]-oldA[i]);
11    #pragma omp barrier
12    if (i==1) {
13     delta=sum(diff); iters++;
14     temp=newA;newA=oldA;oldA=temp;
15    } /* if */
16    #pragma omp barrier
17   } /* while */ } /* iterAvg */
```

(c) Non-recursive version in UW-OpenMP; `oldA` and `newA` are shared

```
1  ... // in some function.
2  #pragma omp parallel
3  { #pragma omp for
4     for(i=0;i<N;i++){iterAvgR(oldA,newA,i);}
5  }/*parallel*/
6  ...
7  void iterAvgR(double *oldA, double *newA, int i) {
8   if (delta<=epsilon) { return; }
9   new[i]=(old[i-1]+old[i+1])/2.0;
10  diff[i]=fabs(newA[i]-oldA[i]);
11  #pragma omp barrier
12  if (i==1) {
13    delta = sum(diff); iters++;
14  }
15  #pragma omp barrier
16  iterAvgR(newA,oldA,i);
17 } /* iterAvgR */
```

(d) Recursive version in UWOmp++

Figure 1: Variations of IA (iterative averaging) in HJ and OpenMP syntax; `delta` and `iters` are shared variables.

only when the number of workers is set to the number of tasks – potentially inefficient.

The correct way to encode the logic expressed in Figure 1c in OpenMP is to write conforming [?] OpenMP code where no barrier executes in the body of the parallel-for-loop. While this can be achieved by some complex code reorganization (inlining the function, interchanging the loops, and involved loop-distribution), this may also lead to code bloat and arguably harder to read programs (cohesive pieces of program logic spread across multiple parallel-for-loops).

Considering the impact of the restrictive practice of disallowing barriers within tasks, which can adversely impact programmability, Aloor and Nandivada [?] propose an extension called UW-OpenMP. This extension 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 (that is, UW-OpenMP admits barriers within work-sharing constructs like parallel-for-loops and parallel-sections). For example, UW-OpenMP admits codes such as the one shown in Figure 1c, and runs much faster (97% faster [?]) than the inefficient alternative of running it as an OpenMP code by setting the number of workers to `N` (number of tasks).

Though UW-OpenMP is a more expressive and efficient extension of OpenMP, it does not admit barriers within recursive functions invoked from parallel-for-loops, because of the inherent challenges in handling them (cannot be inlined). This can be a limitation for expressing recursive algorithms – a common approach to realize conciseness, readability, and intuitiveness [? ? ?]. Further, once we allow such recursive tasks, it is natural to admit synchronization therein. For example, consider the recursive version of IA (Figure 1d). One may argue that the recursive version is more succinct and readable than the iterative version[1]. However, UW-OpenMP does not admit such recursive codes.

Further, in some cases, it may be natural for a programmer to think each task as a recursive function where all these tasks may synchronize using barriers. Hence, disallowing the feature of synchronization within recursive functions invoked from parallel-for-loops limits the scope of the class of algorithms that can be expressed using UW-OpenMP. In this paper, we extend UW-OpenMP (we call it UWOmp++) to address this challenging limitation and in the process also realize more efficient programs.

The set of programs that can be written in UWOmp++ is a superset of the programs that can be written in UW-OpenMP. For example, the two codes shown in Figure 1c and Figure 1d can be expressed using UWOmp++. We propose a transformation scheme from UWOmp++ to OpenMP that avoids the complex bookkeeping code emitted by the UW-OpenMP translator and generates efficient OpenMP code. Our translation not only ensures that the semantics of the generated program is independent of the number of workers, but also the generated code can still take advantage of the efficient 'team of workers' model of OpenMP, in an effective manner.

[1]Note: we may use the OpenMP reduction clause (in place of the `sum` call), inside the parallel-for-loop; even then, we still need the barriers as shown.

Our translation is based on a novel extension to the popular concept of CPS (Continuation Passing Style [**?**]) to handle parallel constructs like parallel-for-loops and barriers. We name this extension as UWOmpCPS. To the best of our knowledge, we are not aware of any other variation of CPS that handles parallel-for-loops and barriers. We first translate the input UWOmp++ code to an intermediate representation (IR) in UWOmpCPS form and then translate the generated UWOmpCPS code to equivalent efficient conforming OpenMP code. For example, for Figure 1c, compared to the time taken by the code generated by the UW-OpenMP compiler, our translated code leads to an improvement of 93% on a 64 core AMD system. Interestingly, our translation of Figure 1d also leads to similar execution times (not much overhead, that is).

The expressibility and efficiency aspects of UWOmp++ helps to encode a wide classes of programs (stencil computations, wavefront parallelization, dynamic programming, and so on [**?**]), without compromising on performance. Note: the claim is not that using barriers within parallel-for-loops is the only/best way to encode such computations. Instead, the proposed extension (a feature common in languages like X10, HJ and so on) provides the programmer an additional way to encode parallelism, that is otherwise missing in UW-OpenMP (and OpenMP), while continuing to take advantage of the efficient 'team of workers' model of OpenMP.

Our Contributions:

• We propose UWOmp++ to admit barriers anywhere within parallel-for-loops and a systematic transformation scheme that uses a novel IR (UWOmpCPS), to generate efficient code.

• We present a series of optimizations to overcome the typical overheads associated with programs in the extended CPS format. Further, we present a scheme to generate efficient specialized code when static scheduling (one of the most popular schemes) is used in parallel-for-loops.

• We have implemented our proposed translation scheme and the optimizations in the ROSE Compiler Framework [**?**].

• We present an evaluation over ten benchmarks (from different benchmark suites) and show that (i) our generated code takes advantage of the underlying 'team of workers' model, (ii) compared to the scheme of Aloor and Nandivada [**?**], our translation leads to significant improvements, and (iii) the proposed optimizations are effective.

2 BACKGROUND

We now briefly describe some relevant OpenMP constructs. More details can be found in the OpenMP specification [**?**].

Parallel-Region: `#pragma omp parallel S`: creates a team of workers (set by `OMP_NUM_THREADS`); each worker executes `S`.

Parallel-for-loop: A for-loop can be annotated using `#pragma omp for`. This is a work-sharing construct and distributes the iterations of the for-loop among the workers. The scheduling policy (`static` – default, `dynamic` or `guided`) determines the exact distribution.

Barrier: `#pragma omp barrier`: is used to synchronize the workers in the team. No worker can cross a barrier unless all the other workers execute a barrier.

Hereafter, we abbreviate the above mentioned three pragmas as `#ompparallel`, `#ompfor` and `#ompbarrier`, respectively.

We now restate four relevant definitions, given by Aloor and Nandivada [**?**] that are also applicable for UWOmp++.

Definition 2.1. An OpenMP parallel-for-loop is said to be executing in *UW model* if a unique worker executes each iteration therein.

Definition 2.2. To distinguish UW model from the default OpenMP execution model (where a worker may execute one or more iterations of a parallel-for-loop), the latter is termed as the *One-to-Many model* (in short *OM model*). A program executing in OM model (termed as an OM-OpenMP program) cannot invoke barriers inside work-sharing constructs.

Definition 2.3. In the execution trace of a parallel-region in a UW-OpenMP program, all the statement instances are grouped 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 ith *UW-group*. The ith UW-group is said to have been generated by the ith *agent* of the UW-OpenMP parallel region.

Definition 2.4. The ith task of a UW-OpenMP parallel-region is defined as i^{th} task executing inside any of the constituent parallel-for-loops. In the execution of a parallel-region in UW-OpenMP, let b_k^i denote the $k^{th}(\geq 1)$ barrier executed by the ith task. Let Ψ_k^i denote the set of statement instances executed before b_k^i. The kth *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 kth phase of the parallel region.

3 TRANSLATING BARRIERS IN UWOMP++

We now discuss our proposed translation scheme to translate a program written in UWOmp++ to OM-OpenMP. Our scheme is based on an extension to CPS (Continuation Passing Style [**?**]), a well established intermediate representation (IR) with inherent support to wait and continue. We first translate the input UWOmp++ program to an IR called UWOmpCPS. A program in UWOmpCPS is similar to a program in the CPS form, except that the former may include parallel-for-loops and barriers. In the second step, the UWOmpCPS program is translated to OM-OpenMP.

During the execution of a UWOmp++ program, in each parallel-for-loop, each worker executes a unique task. In contrast, in an OM-OpenMP program, each worker may execute a "chunk" of tasks. Importantly, in UWOmp++ a barrier can be used to establish synchronization among tasks. To ensure semantics preservation, the generated OM-OpenMP code must satisfy the following two properties: (i) if a worker w_i (executing a task in a phase ϕ_k) hits a barrier, and in the scheduled chunk of w_i, it has at least one task to execute in the same phase ϕ_k, then w_i must continue executing one of those tasks; and (ii) if w_i has no more tasks to execute in phase ϕ_k, it needs to wait for other workers to complete the execution of their scheduled tasks in phase ϕ_k. To realize the above properties, the generated code should be able to suspend the tasks at different barrier statements – while waiting for other tasks to complete the current phase. Similarly, the generated code should be able to resume the suspended tasks – when all the tasks complete the current phase. We exploit the power of continuations (in programs in CPS form) to achieve such kind of suspend and resume operations.

We first extend the notion of CPS to OpenMP programs that have constructs like barriers, parallel-for-loops, parallel-regions, and use

```
Program     ::= (FuncDecl)* MainFunc
MainFunc    ::= int main() { ParRegion }
FuncDecl    ::= Type ID (Args){(Stmt)* RetStmt}
ParRegion   ::= #pragma omp parallel
                { (ParForLoop)* }
ParForLoop  ::= #pragma omp for nowait
                for(ID=0;ID<ID;ID++){ CallStmt }
                BarrierStmt
BarrierStmt ::= #pragma omp barrier
Stmt        ::= AssignStmt |CallStmt |
                RetStmt | BarrierStmt | Seq(Stmt)
```

Figure 2: Grammar for mUWOmp++.

the 'team of workers' model at runtime. We use this extension to derive our IR (UWOmpCPS), and translate the input UWOmp++ code to UWOmpCPS. We also present a scheme to translate code in UWOmpCPS form to OM-OpenMP. The overall transformation flow is shown below.

For the ease of explaining the transformations we use a representative subset of UWOmp++ called *miniUWOmp++* (in short *mUWOmp++*), as the input language; in Section 3.5, we discuss how we translate a general UWOmp++ program to mUWOmp++. We now describe the input mUWOmp++ language, the UWOmpCPS IR and the two-step (Step 1 and Step 2) transformation scheme.

3.1 Input Language mUWOmp++

Figure 2 lists the grammar of the mUWOmp++ language. It accepts a sequence of function declarations followed by the `main` function. The body of a function declaration can have an assignment-statement, a function call, a return-statement, a barrier-statement, or a statement generated by `Seq(X)`; for a non-terminal `X`, `Seq(X)` denotes the program formed from `X` closed under the sequential constructs. The rules for the non-terminals `ID` (denotes an identifier), `CallStmt`, `Args`, `AssignStmt` and `RetStmt` are standard, and we skip those for brevity. The body of the `main` function consists of a single parallel-region, which in turn has a sequence of parallel-for-loops in its body. A parallel-for-loop contains a normalized loop [?], whose body is a function-call. The `FuncDecl` production ensures that each input function returns a value (no "void" functions). Though we do not discuss the details, parallel-sections are handled similar to parallel-for-loops.

3.2 The UWOmpCPS IR

The UWOmpCPS IR grammar is given in Figure 3. A program in UWOmpCPS has a sequence of function declarations (`CPSFuncDecl`) followed by the `main` function. Besides the set of parameters (in `Args`), the formal parameters of a (non-`main`) function declaration include a parameter of type *Closure*. The fields of the *Closure* are: (i) a pointer to a function (say `pCPS`), (ii) type of the closure (explained in Section 3.3), (iii) a set of free variables used in `pCPS` (iv) another closure which stores the continuation function of `pCPS`. A function body consists of a sequence of simple statements followed by a tail call (given by `TailCallStmt`). A `SimpleStmt` can be either an `AssignStmt` or an `IfStmt`. The body of an `IfStmt` can be a sequence of assignment statements. In UWOmpCPS, an assignment statement can only have a simple-expression on the right-hand side. A `TailCallStmt`

```
Program        ::= (CPSFuncDecl)* MainFunc
CPSFuncDecl    ::= void ID(Args, Closure ID){
                     (SimpleStmt)* TailCallStmt}
MainFunc       ::= int main() { CPSParRegion }
CPSParRegion   ::= #pragma omp parallel
                   { CPSParForLoop }
CPSParForLoop  ::= #pragma omp for nowait
                   for(ID=0;ID<ID;ID++) {
                     (SimpleStmt)* CPSFunCall }
TailCallStmt   ::= CPSFuncCall | CPSIfStmt | CPSBarrierStmt
                 | CPSParForLoop
SimpleStmt     ::= AssignStmt | IfStmt
CPSFunCall     ::= ID(ActualParamList, ID);
CPSBarrierStmt ::= #pragma omp barrier (ID)
CPSIfStmt      ::= if(SimpleExpr){(AssignStmt)* TailCallStmt}
IfStmt         ::= if(SimpleExpr){ (SimpleStmt)*}
SimpleExpr     ::= ID <BinOp> ID | <UnaryOp> ID
```

Figure 3: Grammar for UWOmpCPS.

can be a function-call (`CPSFunCall`), an if-statement (`CPSIfStmt`) ending with a `TailCallStmt`, a parallel-for-loop (`CPSParForLoop`), or a barrier-statement in CPS form (`CPSBarrierStmt`). Note: the last argument of `CPSFunCall` is a continuation. A `CPSBarrierStmt` is a new type of barrier introduced in UWOmpCPS, which contains a barrier along with a closure which encapsulates a continuation; each agent synchronizes on the barrier and then invokes the continuation.

The main function consists of a single parallel-region in CPS form (`CPSParRegion`), which in turn contains a single `CPSParForLoop`. A `CPSParForLoop` consists of a series of simple statements, followed by a function call in CPS form.

The semantics of a UWOmpCPS program is similar to that of a UWOmp++ program, except in case of nested parallel-for-loops (not allowed in UWOmp++ or OpenMP): when an agent i encounters a nested parallel-for-loop, it only executes the ith task (iteration).

3.3 Step 1: mUWOmp++ to UWOmpCPS

We transform a mUWOmp++ program to UWOmpCPS using the rules in Figure 4. A rule of the form $[\![\texttt{X}]\!] \Rightarrow \texttt{Y}$ indicates that the input code `X` in mUWOmp++ is transformed to the output code `Y` in UWOmpCPS. A right-hand side code with one or more occurrences of the $[\![\ ]\!]$ operator means that these terms need to be further transformed.

Rules 1 to 10 are standard CPS transformation rules [?]. For instance, Rule 10 handles a function call `fun` when it appears as part of an assignment or a return statement. Say, the input code is of the form `X fun(`*args*`) Y`, where `X` and `Y` to refer to arbitrary code pieces. First, we emit a new function `pCPS` (using the macro *mkProc*) that takes two arguments V_1 (of type T_1, matching the return type of `fun`) and a closure K. Then we emit code to create a closure C using the macro *mkClsr*. The closure C is passed as an additional argument to `funCPS`. We explain the functionality of *mkClsr* (used by Rules 1, 10, 12, 14 and 15) by using the invocation in Rule 10 as the example. The macro takes the following four arguments: (i) the function pointer `pCPS` (ii) the type of the closure (referred by C`->type`) (iii) the free variables in `X` and `Y` (given by the function FV), and (iv) the closure K whose continuation function needs to be invoked after `pCPS` is executed. Note: the value of C`->type` is set to one of the following types, based on the first statement (say S1) to

#	Input		Output
1.	⟦**int** main() {S}⟧	⇒	int main() { K=$mkClsr$(id, T_e, $null$, $null$); ⟦K S⟧}
2.	⟦T fun($TypeArgLst$) {S}⟧	⇒	**void** funCPS($TypeArgLst$, $Closure$ K){⟦K S⟧}
3.	⟦K fun(a_1, ...a_n)⟧	⇒	funCPS(a_1, ...a_n, K)
4.	⟦K S1; S2⟧ // S1 *has no call, return or pragmas inside.*	⇒	S1; ⟦K S2⟧
5.	⟦K {S}⟧	⇒	{⟦K S⟧}
6.	⟦K S⟧ // *S is an AssignStmt*	⇒	S
7.	⟦K **if**(e){S1} **else**{S2} Y⟧ // e, S1 *and* S2 *do not contain any calls.*	⇒	**if** (e){⟦K S1⟧} **else** {⟦K S2⟧} ⟦K Y⟧
8.	⟦K **if**(e){S1} Y⟧ // e *and* S1 *do not contain any calls.*	⇒	**if** (e){⟦K S1⟧} ⟦K Y⟧
9.	⟦K **return** x⟧	⇒	K x
10.	⟦K α⟧ // α *is not a RetStmt or* // *AssignStmt.* // α=X fun($args$) Y // X *has no calls.*	⇒	// *Say* T_1 *is the return type of fun.* $mkProc$(void pCPS (T_1 V_1, $Closure$ K){⟦K X V_1 Y⟧}); C=$mkClsr$(pCPS, T_{seq}, FV(X, Y), K); funCPS($args$, C)
11.	⟦K **#ompparallel** {S}⟧	⇒	**#ompparallel** {⟦K S⟧}
12.	⟦K **#ompfor nowait** for(Header){ fun($args$);} **#ompbarrier** S⟧	⇒	$mkProc$(void pCPS($Closure$ K){⟦K**#ompbarrier** S⟧}); **#ompfor** for(Header) { C=$mkClsr$(pCPS, T_b, FV(S), K); funCPS($args$, C);}
13.	⟦K **#ompbarrier**⟧	⇒	**#ompbarrier** K
14.	⟦K **#ompbarrier** S⟧ // S *is a seq of* // *parallel-for-loops*.	⇒	$mkProc$(void pCPS($Closure$ K){⟦K S⟧}) C=$mkClsr$(pCPS, T_{pf}, FV(S), K); **#ompbarrier** C
15.	⟦K **#ompbarrier** S⟧ // S *is not a seq of* // *parallel-for-loops*.	⇒	$mkProc$(void pCPS($Closure$ K){⟦K S⟧}); C=$mkClsr$(pCPS, T_{seq}, FV(S), K); **#ompbarrier** C

Figure 4: mUWOmp++ to UWOmpCPS Translation.

be executed in pCPS (i) T_e (if S1 is empty), (ii) T_b (if S1 is a barrier), (iii) T_{pf} (if S1 is a parallel-for-loop) or (iv) T_{seq} (otherwise). We use this type-field in Step 2 of our transformation scheme. Note that Rule 10 enforces left to right processing of functions present inside any simple statement. Also note that in Rule 6, in the right hand side, no application of closure K is performed on S. In any function, finally K will be applied at all return points with the return value as argument by Rule 9.

Rules 11 to 15 are used to transform various OpenMP constructs which can appear in the input mUWOmp++ code. Rule 11: The body of the parallel region S is transformed to ⟦K S⟧. Rule 12: Here, the body of first parallel-for-loop is the function call fun(a_1, ..., a_n). The statement S contains the remaining sequence of parallel-for-loops. Similar to Rule 10, we create a new function pCPS (using *mkProc*) with body ⟦K ompbarrier S⟧, which is used to create the closure C (using *mkClsr*). The type of the closure C is set to T_b (first statement inside pCPS here is a barrier). Each task in the first parallel-for-loop calls the function funCPS, which takes the closure C as an additional argument. Rule 13: A barrier in the mUWOmp++ program is transformed to a UWOmpCPS barrier that takes a closure K as an argument. Rule 14 and Rule 15 deal with a barrier followed by a sequence S, where S is a sequence of parallel-for-loops and S is not a sequence of parallel-for-loops, respectively. Hence, Rule 14 sets the type of closure to T_{pf} and Rule 15 sets it to T_{seq}. The barrier is then transformed to accept C as its argument.

#	Input		Output
16	**#ompfor** for(Header) { C=$mkClsr$ (X); funCPS($args$, C); }	→	**#ompfor** for(Header) { C=$mkClsr$ (X); K=$mkClsr$(funCPS, T_b, FV(Header, $args$), C); addToRdyWL(K); }// for execRdyWL();
17	**#ompbarrier** K	→	uwBarrier(K, tid)

Figure 5: UWOmpCPS to OM-OpenMP Translation.

3.4 Step 2: UWOmpCPS to OM-OpenMP

Here we transform the code in UWOmpCPS to OM-OpenMP. In the generated OM-OpenMP code each worker has to execute zero or more closures. At runtime, we maintain a ready-worklist (rdyWL) and a pending-worklist (pendingWL) per worker to store the closures to be executed by that worker. The closures in the rdyWL can be executed in the current phase Φ_k. A worker executes closures only from the rdyWL. The pendingWL contains entries that can only be executed in the next phase (Φ_{k+1}). Note: the semantics of barriers in UWOmp++ ensures that the code executing in phase Φ_k can generate closures to be executed only in phases Φ_k or Φ_{k+1}.

We use four helper macros to manipulate worklists of the current worker: (i) addToRdyWL: adds an entry to rdyWL. (iii) movePndngToRdy: moves all the entries in pendingWL to rdyWL, (ii) addToPndngWL: adds an entry to pendingWL, and (iv) execRdyWL: executes the first closure from rdyWL.

We transform a program in UWOmpCPS to OM-OpenMP using the two rules shown in Figure 5. A rule of the format X → Y indicates that the input code X in UWOmpCPS is transformed to the output code Y in OM-OpenMP.

A parallel-for-loop in the UWOmpCPS program is transformed to OM-OpenMP, using Rule 16. In the transformed OM-OpenMP code, each task creates a new closure K with the following parameters: (i) funCPS as continuation function, (ii) T_b as the type of K, (iii) the free variables of Header and *args*, and (iv) closure C. The closure K is added to the rdyWL. The generated code ensures that for each task in the parallel-for-loop (scheduled to a particular worker W) a closure is added to the ready-worklist (rdyWL of W). After adding all the tasks of the parallel-for-loop, the function execRdyWL (invoked from a parallel-region, and not from the parallel-for-loop), is executed by each worker (not each task). Note that execRdyWL should not be invoked from within another parallel-for-loop (a work-sharing construct), as the rdyWL contains the closures to be executed by the respective workers and further work-sharing is not desired. Such a design would have led to OpenMP

```
1  // isAdded[tid] is initialized to 0
2  void uwBarrier(
3   Closure newK, int tid){
4   if (newK->type==T_pf){
5    if (!isAdded[tid]){
6     isAdded[tid]=1;
7     addToPndngWL(newk);}
8   }else{addToPndngWL(newk);}
9   if (rdyWL[tid] is empty){
10   #ompbarrier
11   isAdded[tid]=0;
12   movePndngToRdy();}
13  execRdyWL();}
```

Figure 6: `uwBarrier` **Code**

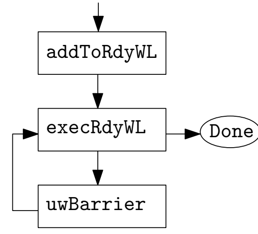


Figure 7: Execution flow of an agent

non-conforming [?] nested parallel-for-loops, if any of the closures stored in `rdyWL` contained a parallel-for-loop.

Rule 17 translates each barrier (`#ompbarrier` K) to a call to the function `uwBarrier` (see Figure 6). The function `uwBarrier` takes as argument a closure `newK` containing the continuation of the barrier and the current thread id `tid`. This continuation can be (1) a function which has a parallel-for-loop as the first statement in its body (`newK->type` = T_{pf}), or (2) a function which has a statement other than parallel-for-loop as the first statement in its body. The code we generate has a property that `uwBarrier` is executed in each task. However, in Case (1) the parallel-for-loop should not be executed by all tasks, but only once by each worker. So, if `newK->type` is T_{pf}, we store `newK` in the `pendingWL` only by the first task of each worker that calls `uwBarrier` (done by the code in Lines 5 - 7). If `newK->type` is not T_{pf} then `newK` is added to `pendingWL` in all the tasks, by their corresponding worker (Line 8).

If a worker has finished executing all the code of its assigned tasks scheduled to be executed in the current phase, then it waits for the other workers on a barrier (Line 10). Once all the workers reach the barrier each worker (i) resets corresponding entry in the `isAdded` array and (ii) moves all the entries from its `pendingWL` to its `rdyWL` using the function `movePndngToRdy` (Line 12). This indicates the beginning of the next phase.

The overall execution of the generated OMOmp++ code by an agent follows the flowchart shown in Figure 7. The agent starts by adding the first task to the `rdyWL`. After that it continuously executes tasks from the `rdyWL` and waits at the `uwBarrier` (if present) till all the tasks have been completed.

Note: while executing the OM-OpenMP code, to achieve the UWOmp++ semantics, the generated code must ensure that an agent performing computation in the current phase (say Φ_k), must (i) store the continuation corresponding to the computation of the next phase Φ_{k+1}, if the agent executes a barrier; and (ii) wait for all the agents to finish executing the scheduled computations of Φ_k, and then start executing the stored continuations of phase Φ_{k+1}. Our implementation of `uwBarrier` ensures that these two requirements are met.

Example transformation. For the better understanding of our transformation scheme, we now describe the steps in transforming a UWOmp++ code shown in Figure 8a to OMOmp++. We now discuss some of the salient points therein. We process the two functions individually. First, an initial closure KId (identity function) is applied to the parallel-region as shown in Figure8b. In Figure 8d, each task in the parallel-for-loop creates a closure C having continuation function `pCPS0`. The closure C is passed as an extra parameter to the function `fCPS`. In the body of `pCPS0` the barrier is transformed such that it accepts the closure K as argument. Figure 8e shows the final code for the function `main`. In Figure 8h, we apply rule 10 on the call to `f` in the `return` statement and then rule 17 is applied on `#ompbarrier` to get the code in Figure 8i. On Figure 8i, we apply rule 9 to get the final code in Figure 8j.

3.5 UWOmp++ to mUWOmp++ Simplification

In this section, we discuss how we transform any general UWOmp++ program to a mUWOmp++ program. We invoke the following steps until there is no further change.

Step 1. *A sequence of statements as the body a parallel-for-loop.* The full body is moved to a separate function and a call to that function is inserted in the parallel-for-loop.

Step 2. *One or more serial-loops inside the code invoked from a parallel-for-loop.* We transform each such serial-loop to a recursive function and replace the loop with a call to that function.

Step 3. *The body of the parallel-region is not a sequence of parallel-for-loops.* Similar to Step 1, we first move the full body to a separate function (say, `xFun`). Then, since the code has to be executed by all the workers, we replace the body of the parallel-region, with the following code:

```
#ompfor
for (int i=0;i<omp_num_threads();++i) {xFun(···);}
```

Note: The arguments to `xFun` are the list of free variables.

3.6 Correctness of Transformation

Theorem 3.1 *Let $S^{uw}_{i,n}$ denote the nth instance of statement S executed by agent i in UW model and $S^{om}_{i,n}$ 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] $S^{uw}_{i,n} \in \Phi_k \Rightarrow S^{om}_{i,n} \in \Phi_k$.*
2. *[Data-dependence] $R^{uw}_{i,m} \rightarrow_d S^{uw}_{j,n} \Rightarrow R^{om}_{i,m} \rightarrow_d S^{om}_{j,n}$.*

Proof. *(Sketch)* [Phase-dependence]: The barriers present in the input UWOmp++ code is also present in the generated OM-OpenMP code within the `uwBarrier` function (Line 10 in Figure 6). Our transformation scheme does not introduce or remove any barriers in the generated OM-OpenMP code. In the generated OM-OpenMP code a barrier is executed only if the `rdyWL` of the current worker W is empty (Line 9 in Figure 6). Otherwise W executes another entry (part of a strand corresponding to another agent) in its `rdyWL`. This shows that at any point of time the `rdyWL` of W contains entries (parts of strands) to be executed in the current phase, Φ_k. Thus there is a one-to-one correspondence between the parts of strand executed by agent i in Φ_k in the OM-OpenMP code, with parts of UW-group executed by agent i in the input UWOmp++ code. This proves that if $S^{uw}_{i,n} \in \Phi_k$ then $S^{om}_{i,n} \in \Phi_k$.

[Data-dependence]: From the above proof of phase-dependence it is guaranteed that for each $R^{uw}_{i,m}$ and $S^{uw}_{i,n}$, the corresponding statements are also executed in the OM-OpenMP code. Also the transformation rules listed in Figure 4 do not add any new statements, remove any existing statements, or change the order of the statements. Thus, no new statements are executed between $R^{uw}_{i,m}$

(a)
```
int f(args) {
 if (cond) return 1;
 S1;
 #ompbarrier
 return f(args)+1;
}

int main() {
 #ompparallel
 {
  #ompfor nowait
  for(Header){f(args);}
  #ompbarrier
 } }
```

(b)
```
int main() {
⟦KId
 #ompparallel
 {
  #ompfor nowait
  for(Header) {f(args);}
  #ompbarrier
 } ⟧ }
```

// (c) apply rule 11
```
int main() {
 #ompparallel
 { ⟦KId
   #ompfor nowait
   for(Header){f(args);}
   #ompbarrier
   ⟧ } }
```

// (d) apply rules 13,12
```
void pCPS0(Closure *K){
 #ompbarrier K }
int main() {
#ompparallel
{
 #ompfor nowait
 for(Header){
  C=mkClsr(pCPS0,T_b,FV,KId);
 fCPS(args,C);
 } } }
```

// (e) apply rules 16,17
```
void pCPS0(Closure *K){
  uwBarrier(K); }
int main() {
#ompparallel
{
 #ompfor nowait
 for(Header) {
  C=mkClsr(pCPS0,T_b,FV,KId);
  K=mkClsr(fCPS,T_b,FV,C);
  addToRdyWL(K); }
 execRdyWL(); } }
```

// (f)
```
⟦ int f(args) {
 if (cond) return 1;
 S1;
 #ompbarrier
 return f(args)+1;
}⟧
```

// (g) apply rule 2
```
void fCPS(args,Closure *K){
⟦K if (cond) return 1;
 S1;
 #ompbarrier
 return f(args)+1;
⟧}
```

// (h) apply rules 7,9,4,15
```
void pCPS1(int v2,Closure *K){
 ⟦K
 return f(args)+1;
 ⟧
}
void fCPS(args,Closure *K){
 if (cond) K 1;
 S1;
 C1=mkClsr(pCPS1,T_seq,FV,K);
 #ompbarrier C1;
}
```

// (i) apply rules 10, 17
```
void pCPS2(int v3,Closure *K){
 ⟦K return  v3+1;⟧
}
void pCPS1(int v2,Closure *K){
 C2=mkClsr(pCPS2,T_seq,FV,K);
 fCPS(args,C2);
}
void fCPS(args,Closure*K){
 if (cond) K 1;
 S1;
 C1=mkClsr(pCPS1,T_seq,FV,K);
 uwBarrier(C1);
}
```

// (j) apply rule 9
```
void pCPS2(int v3,Closure *K){
  K (v3+1);
}
void pCPS1(int v2,Closure *K){
 C2=mkClsr(pCPS2,T_seq,FV,K);
 fCPS(args,C2);
}
void fCPS(args,Closure *K){
 if (cond) K 1;
 S1;
 C1=mkClsr(pCPS1,T_seq,FV,K);
 uwBarrier(C1);
}
```

Figure 8: Example Transformation

and $S^{uw}_{i,n}$, which in turn, implies that dependency between them is preserved. □

4 GENERATING EFFICIENT OM-OPENMP CODE

The transformation scheme in Section 3 may generate inefficient OM-OpenMP code due to extensive use of `malloc/free` calls. In this section, we present a series of optimizations for reusing the allocated heap memory to generate efficient OM-OpenMP code. To explain our optimizations, we first introduce the notion of *strands* for OM-OpenMP programs, which like UW-groups and agents (see Section 2), are runtime entities. Note: The definitions of UW-group and agent defined for UW-OpenMP programs (Section 2) are also applicable for UWOmp++ and OM-OpenMP programs.

Definition 5.1. A strand is a UW-group with no explicit return statements. A UW-group in a UWOmp++ program corresponds to a strand in the OM-OpenMP program. Note that a strand may include one or more *substrands*.

Definition 5.2. A substrand is a maximal sequence of statements within a strand such that the substrand ends with a tail-call and has no other tail-calls in it. A strand is composed of a sequence of one or more substrands.

Let `f` be a function invoked within a UW-group G in the input UWOmp++ program. Let S be the corresponding strand executed in the transformed OM-OpenMP program.

Definition 5.3. We define a projection θ which maps an instance of a call-statement in G to a set of substrands in S. Let T be a substrand in S. For a given call-statement R_f (to a function f) in G, we say $T \in \theta(R_f)$, if (i) T is of the form '`S1; tail-call(..);`' and (ii) `S1` is present in the execution trace of f, when invoked by R_f.

4.1 Reusing Allocated Memory for Closures

Consider the example in Figure 9. Here, function `h` is invoked at call-statement `L1` in `fun` within a UW-group G in the input UWOmp++ program. In the transformed OM-OpenMP code, say strand S corresponds to the UW-group G in the input program. Here, θ(`L1`) = {T_1, T_2, T_3}. Note that within S, these substrands are interleaved with the substrands corresponding to the call-statements in `h`.

Within T_1, when `iCPS` is invoked, a closure C_1 is passed as an additional argument to `iCPS`; the closure will be used for executing T_2 after all substrands in θ(`L1`) are executed. The macro *mkClsr* (Section 3) allocates memory for C_1 in the heap. In general, before executing the tail-call in any substrand, a `malloc` call will be performed. As the number of substrands increases, the number of `malloc/free` calls increases – a possible performance bottleneck. To optimize this, we propose a scheme to allow a substrand to reuse the memory allocated for the closure of another substrand. Our scheme depends on the notion of the *live range* of a closure.

Definition 5.4. A closure C is said to be live at a program point p if the values stored in the memory locations pointed by C are needed at a statement reachable from p. The live range of C is the set of statements at which C is live.

Similar to the idea of register allocation [?], a substrand T_2 can reuse the memory allocated for a closure of substrand T_1, if the live ranges of the closures used by T_1 and T_2 do not overlap. Figure 10 shows three cases under which two given closures do not overlap. Case I: In the input UWOmp++ program, for any function f invoked (at R_f) within a UW-group G, live ranges of closures of any two substrands in $\theta(R_f)$ do not overlap. In the example shown in Figure 9, the heap memory allocated for the closure C_1, can be reused by T_2 for storing C_2. Case II: In the input UWOmp++ program, if function g is invoked (at R_g) as a tail call in function f

```
void h(){
        S0; Li: i();
        S1; Lj: j();
        S2; Lk: k()
}
void fun(){
        S3; L1: h();
        S4; L2: h(); S5;
}
```

```
T1: S0; iCPS(C1);
T2: S1; jCPS(C2);
T3: S2; kCPS(C3);
```

Figure 9: Example code to depict reuse cases. C_1,C_2,C_3 **are closures.** θ(L1)={T_1,T_2,T_3}

I.	For any invocation R_f, within a UW-group G, $((T_1 \in \theta(R_f)) \wedge (T_2 \in \theta(R_f)) \wedge (T_1 \neq T_2)) \Rightarrow \neg overlap(T_1, T_2)$
II.	For any function invocation R_g, invoked within a function f, which in turn is called at R_f $((T_1 \in \theta(R_f)) \wedge (T_2 \in \theta(R_g)) \wedge tailcall(g, f)) \Rightarrow \neg overlap(T_1, T_2)$
III.	For any two invocations R_{f_1} and R_{f_2} of the same function f within a UW-group G, $((T_1 \in \theta(R_{f_1})) \wedge (T_2 \in \theta(R_{f_2})) \wedge \neg recursive(f)) \Rightarrow \neg overlap(T_1, T_2)$

Figure 10: Conditions under which two substrands T_1 **and** T_2 **do not overlap, and hence can share the memory allocated for the closure. Used predicates: (i)** $overlap(x, y)$**: live ranges of closures of** x **and** y **overlap (ii)** $tailcall(x, y)$**:** x **is a tail call in** y **(iii)** $recursive(x)$**:** x **is a recursive function.**

(at R_f) then the live ranges of closures of $\theta(R_f)$ and $\theta(R_g)$ do not overlap. In Figure 9, when kCPS is invoked, k being a tail call in h, the live ranges of closures of θ(L1) and θ(Lk) do not overlap and all the substrands in θ(Lk) can reuse the memory allocated in h, for C_1.
Case III: In the input UWOmp++ program, in a UW-group G, if R_{f_1} and R_{f_2} are two function invocations of a non-recursive function f, then the live ranges of the closures of $\theta(R_{f_1})$ and $\theta(R_{f_2})$ do not overlap. Since f is non-recursive, R_{f_1} and R_{f_2} will be invoked one after the other and hence no overlap. In Figure 9, where the non-recursive function h is invoked twice from fun in the UW-group G, it can be seen that live ranges of θ(L1) and θ(L2) do not overlap. Hence, the invocation of h at L2 can reuse the memory allocated for the invocation at L1.

4.2 Reusing Memory for Passing Free Variables

Instead of allocating memory every time for passing free-variables (stored in the closure) to a continuation call, we allocate memory once per function f to store the free-variables in each of the substrands in $\bigcup_{\forall i}(\theta(R_{f_i}))$, where R_{f_i} indicates the call-statement that invokes f at different program points. Such a reuse strategy requires that each such free-variable has a unique index in the one-time-allocated free-variable array for f. The benefit that we obtain with such a strategy is that we avoid frequent calls to expensive mallocs/frees for storing free variables while creating closures for substrands. This strategy comes at the cost of allocating and maintaining space for all the variables for each function for the entire period of program execution. For programs, for which such a strategy is impractical, we can avoid this optimization. However, in our evaluation, we did not face such a requirement.

4.3 Optimizing Code for `Static` Scheduling

In the generated OM-OpenMP code, the linked-list implementation of worklists induces some overheads. If the parallel-for-loop uses static scheduling (one of the most popular scheduling policies), the maximum chunk given to each worker can be known at the beginning of the parallel-for-loop. We exploit this property of static scheduling and use an array based worklist implementation. One array is used per parallel-for-loop per worker. This scheme is far more efficient than the linked-list based implementation, as we avoid expensive linked-list operations, and improve cache locality. Further, this allocated memory is re-used for the subsequent parallel-for-loops, if the size of the existing dynamic array ≥ the size of the dynamic array required at the later parallel-for-loop.

4.4 Tail-call Optimization

Our transformation scheme relies on the tail-call optimization performed by compilers like GCC (using `-foptimize-sibling-calls`), to avoid the costs (such as, allocating/deallocating stack frames at each call/return, saving/restoring caller and callee save registers, return jumps, and so on) that are typically incurred during procedure calls, which can otherwise make CPS transformation very expensive. We transform all the functions invoked from the parallel-region to CPS form, such that they have the same signature – required by compilers like GCC and LLVM to perform tail-call optimization [?].

Note: our transformation scheme can also be applied on parallel-for-loops which use dynamic or guided scheduling. But, we add all the tasks to the worklist without considering their work load, which is necessary for dynamic scheduling to be effective. Handling dynamic scheduling effectively is left as an interesting future work.

4.5 CPS and Its Impact

As briefly alluded to in Section 3, CPS is a well established intermediate representation (IR) with inherent support to wait and continue, which is needed in the presence of the barriers inside tasks. Thus CPS makes for a natural choice for our implementation. It is thus not surprising that a similar solution was used by Imam and Sarkar [?] to implement cooperative scheduling. Some of the other possible alternatives could be (1) designing compiler translation to translate such recursive task-parallel programs to iterative form – quite non-trivial (2) redesigning the underlying task implementation to change its standard behavior of running to completion without context-switching and allow for context-switching in between – may lead to "heavier" tasks and consequently significant overheads.

As an additional optimization, to limit the impact of CPS transformation, we can limit our CPS conversion to to only those parts that include recursive tasks with barriers.

5 EVALUATION

We have implemented our translation in the ROSE [?] compilation framework and present an evaluation on two different multi-core systems: an Intel 32 core system (two E5-2670 processors, 64GB

	Bench [Source]	Description	I/P	Steps
Recursive Kernels				
1	IA [?]	Distributed Iterative averaging	512	NA
2	LE-LCR [?]	Leader election	128K	NA
3	KPDP [?]	Distributed Knapsack	128K	NA
4	LCS[?]	Longest common subsequence	2048	NA
5	MCM[?]	Matrix chain multiplication	16K	NA
Iterative kernels				
6	3MM [?]	3 Matrix multiplication	8K	NA
7	GEMVER [?]	Vector mult, matrix addition	64K	NA
8	JACOBI2D [?]	2D Jacobi	32K	1024
9	FDTD2D [?]	Finite difference time-domain	16K	1024
10	SOR [?]	Successive over-relaxation	32K	1024

Figure 11: Details of the Benchmarks.

RAM, hyper threaded, Cache: L1-32KB, L2-256KB, L3-20MB), and an AMD 64 core system (four AMD 6376 processors, 512GB RAM, no hyper threading support, Cache: L1-16KB, L2-2048KB, L3-6MB).

The evaluation is performed on ten benchmark kernels from various sources (details in Figure 11; source code available online [?]). These kernels have been categorized into two groups: recursive (the parallel-for-loop has a recursive function that may invoke a barrier), and iterative. The later ones are those kernels for which we did not find intuitive recursive versions. For the recursive kernels, we have coded two versions: the recursive UWOmp++ code and the iterative UW-OpenMP code. For the iterative kernels, we have used the same code for both UW-OpenMP and UWOmp++. The figure also lists the input sizes and the number of 'Steps'; for the kernels that iteratively perform computation till convergence, the column named 'Steps' shows the number of steps used for convergence.

We now present an evaluation to compare the performance of the code generated by the UWOmp++ compiler compiling UWOmp++ code against that of the code generated by the UW-OpenMP compiler [?] compiling equivalent UW-OpenMP code. We divide the comparison into two parts: evaluation against the recursive kernels and iterative kernels. We show that in the context of recursive kernels, the proposed UWOmp++ compilation scheme leads to significant improvements compared to the corresponding UW-OpenMP compiled code. We also show that in case of iterative kernels the performance of both UWOmp++ kernels match the performance of UW-OpenMP kernel and importantly we do not encounter any noticeable deterioration. To calculate the improvements we use the standard formula %Improvement=100×(timeOld−timeNew)/timeOld.

5.1 Evaluation Using the Recursive Kernels

OMOmp++Opt Vs OM-OpenMP. Using the recursive kernels, we first compare the performance of our generated codes using the techniques discussed in this paper (input UWOmp++ codes) against that generated by the UW-OpenMP compiler [?] (input UW-OpenMP codes). We refer to the former codes as OMOmp++Opt and the latter as OMOmp. The OM prefix indicates that even though the input is in UW (Unique-Worker) mode, the generated code can be compiled and run as normal OpenMP code (One worker may be mapped to Many tasks).

Figure 12a and 12b show the percentage improvement in execution times of the OMOmp++Opt codes with respect to the OMOmp,

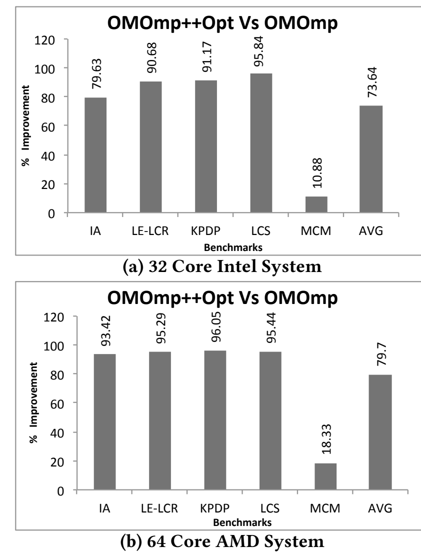


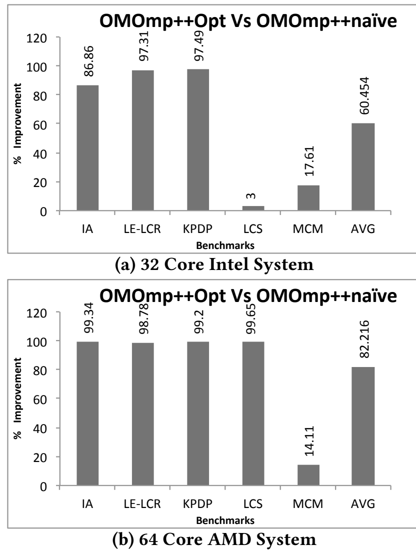
Figure 12: Recursive kernels: Performance improvement of OMOmp++Opt codes with respect to the OMOmp codes.

on the Intel and AMD systems, respectively. It can be seen that with respect to the OMOmp codes, the OMOmp++Opt codes run much faster (on average 73.64% on the Intel system and 79.7% on the AMD system). It can be seen that except for MCM, the performance gains are significantly high for the other recursive kernels. We realize these improvements because the overheads in our translated OMOmp++Opt codes are far less as compared to that in the OMOmp codes. For example: (i) unlike the OMOmp codes, the generated OMOmp++Opt codes do not include any additional parallel-for-loops and barriers. (ii) unlike the OMOmp codes, the generated OMOmp++Opt codes do not use too many space-consuming auxiliary variables, data structures and the time-consuming auxiliary maps for keeping track of the phase information and statement execution history. (iii) our proposed optimizations are able to overcome the typical overheads associated with the CPS transformation.

In case of MCM the gains are slightly lower (10.88% and 18.33% on the Intel and AMD system, respectively), but not insignificant. We investigated the reasons for the relatively lowers gains and found that for the MCM kernel, the relative overheads in the OMOmp codes, compared to the actual computation are quite less – less scope for improvement.

Summary: the UWOmp++ compiler not only admits the recursive kernels but also generates more performant codes compared to the UW-OpenMP compiler.

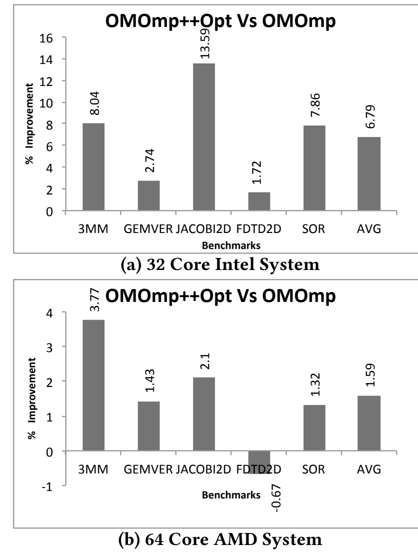
Impact of the proposed optimizations. To study the overall impact of the optimizations presented in Section 4, we now compare the behavior of the OMOmp++Opt codes against the unoptimized codes (generated using the scheme discussed in Section 3, and referred to as OMOmp++$_{naive}$ code). Figures 13a and 13b show the performance improvement of OMOmp++Opt codes over the OMOmp++$_{naive}$ codes, for the Intel and AMD systems, respectively.



(a) 32 Core Intel System

(b) 64 Core AMD System

Figure 13: Recursive kernels: Performance improvement of OMOmp++Opt code with respect to OMOmp++$_{naive}$ code



(a) 32 Core Intel System

(b) 64 Core AMD System

Figure 14: Iterative kernels: Performance improvement of OMOmp++Opt codes with respect to the OMOmp codes.

For IA, LE-LCR and KPDP, the OMOmp++Opt codes show significant improvement. In these kernels, our naive scheme introduces too many `malloc`/`free` calls, along with linked-list manipulation instructions and our proposed optimizations are able to elide them. And since, in these kernels, these (reduced) overheads are significant compared to the rest of the computation, the impact is high. In case of LCS the improvement in the AMD system is much higher than that of the Intel system. Here, though the static schedule optimization improves the cache locality, the wavefront access pattern in LCS and the smaller cache size in the Intel system diminish the advantages. However, the large cache size in the AMD system acts favorably. For the MCM kernel, the overheads resulting from OMOmp++$_{naive}$ are not that significant compared to the rest of the computation and hence OMOmp++Opt has less scope to improve (17.61% and 14.11% gains on the Intel and AMD system, respectively). Overall, on average, over the OMOmp++$_{naive}$ codes, the OMOmp++Opt codes show 60.5% and 82.2% improvement on the Intel and AMD systems, respectively. These high improvements attest to the importance of the proposed optimizations.

5.2 Evaluation against the Iterative Kernels

We have also compared the performance of UWOmp++ generated code against that of the UW-OpenMP, by considering the iterative kernels. Figures 14a and 14b show the performance gains of the OMOmp++Opt codes in comparison with the OMOmp, for the Intel and AMD system, respectively. As it can be seen, compared to the recursive kernels, in the context of iterative kernels, the gains are less (average 6.79% and 1.59% on the Intel and AMD system, respectively). The lower gains in the iterative kernels are because the relative overheads in the OMOmp codes, compared to the actual computation are quite less. Hence, there is less scope for improvement for the OMOmp++ codes.

On the Intel system 3MM, JACOBI2D, and SOR showed non-trivial amount of gains (7.8% to 13.6%) for OMOmp++Opt. The rest of the numbers shown in Figure 14 indicate that the performance of OMOmp++Opt codes matches that of OMOmp. This suggests that our approach is suitable for both recursive and iterative kernels.

Aloor and Nandivada [?] compare the OM-OpenMP code (that contain barriers within parallel-for-loops) against the GCC compiled code executed in UW-mode (by setting the environment variable `OMP_NUM_THREADS` set to the number of tasks in the loop). Overall they show that OM-OpenMP code runs faster than the UW-mode GCC compiled code (GeoMean 86% faster). While the recursive kernels listed in listed in Figure 11 cannot be run in UW-mode, we do not show a comparison even for the iterative kernels (except for 3MM), as the UW-mode GCC compiled codes fails to execute - runs out resources, for the chosen input sizes. For 3MM, the OMOmp++Opt code runs 10.7% and 6.1% faster, on the Intel and AMD system, respectively.

We have also compared the iterative OMOmp++Opt codes against their base OpenMP versions (compiled with GCC, and executed by setting `OMP_NUM_THREADS`=#H/W cores). We found that the average performance difference between the OMOmp++Opt and OpenMP programs was around 6%. We see that UWOmp++ admits expressive iterative codes, without incurring much performance penalty.

6 RELATED WORK

Many prior works extend OpenMP with different types of synchronization constructs. For example, Shirako et al. [?] present a runtime approach to adopt phasers in HJ to OpenMP. UW-OpenMP [?] is an extension of OpenMP that admits barriers within parallel-for-loops. In this paper, we extend UW-OpenMP to admit barriers inside recursive functions called within parallel-for-loops. In addition, our translation leads to more performant codes.

Nandivada et al. [?] do semantics-preserving transformation of task parallel loops (in languages like X10 and HJ) in the presence of synchronization primitives (like HJ Phasers). Aloor and Nandivada [?] present a scheme to translate UW-OpenMP codes to equivalent OM-OpenMP codes. In contrast, we present a systematic approach to translate UWOmp++ programs (barriers may be present anywhere within parallel-for-loops) to OM-OpenMP; this ensures that the generated code takes advantage of the efficient 'team of workers' model of OpenMP.

There have been efforts to use continuations to extend and translate parallel programs. Fischer et al. [?] translate an event-driven input program with method annotations to CPS format and achieve event-driven synchronization. Cilk [?] runtime uses continuations to spawn and synchronize workers. Continuation Passing C [?] allows the programmers to spawn new workers to execute a particular function, put a worker to sleep and wake it up later using library functions. Li et al. [?] present an alternative way to implement concurrency in the Glasgow Haskell Compiler, by polling the pool of suspended continuations to see if blocking can be resolved. To the best of our knowledge, ours is the first work which uses CPS to achieve task synchronization within parallel-for-loop in OpenMP.

For HJ programs, Imam et al. [?] present a novel cooperative scheduling technique (using a help-first policy), based on delimited continuations [?], to design efficient synchronization constructs. The cooperative scheduler wraps each spawned task inside a subcomputation having a well defined boundary, called as one-shot delimited continuation (OSDeCont). In contrast, we use general continuations which never need to return and are not bound by any parallel-for-loop boundary, which allows a much more general synchronization pattern, where barriers can be used to synchronize across multiple parallel-for-loops. Unlike a cooperative scheduler, which converts only a subset of function calls ("suspendable" methods) to CPS, our approach converts all the function calls to CPS, while ensuring that the overheads due to CPS are minimal. Importantly, unlike the scheme of OSDeCont, our scheme is independent of the runtime scheduling policies like help-first and work-first.

White [?] describes a continuation based implementation of OpenMP 3.0 tasks. However, UWOmp++ deals with tasks in parallel-for-loops, which may contain barriers even within recursive functions. White uses liveness analysis to reduce the number of free-variables shared across continuations. In contrast, we use a fixed size allocation for storing (live) free variables across functions.

Shao and Appel [?] and Appel and Shao [?] have presented techniques to optimize memory for closures, in the domain of functional programming. They reduce heap usage for nested functions by maintaining a shareable record for common variables among closures. In contrast, we reuse the memory associated with closures (for reducing mallocs/free calls). Moreover, their scheme relies on the underlying efficient garbage collector, which is absent in C.

7 CONCLUSION

We proposed an extension (UWOmp++) to UW-OpenMP to admit barriers in recursive functions called from the iterations (tasks) of OpenMP parallel-for-loops. This helps programmers use recursion (an important programming methodology) to specify task parallelism with inter-task synchronization. We also present a transformation scheme to generate efficient OpenMP code, so the generated code can continue to take advantage of the underlying efficient 'team of workers' model of OpenMP. Our translation is based on a novel extension of the popular CPS form (termed UWOmpCPS), which not only helps compile UWOmp++ programs but also in the process realizes more efficient programs.