UWOmp$_{pro}$: UWOmp++ with Point-to-Point Synchronization, Reduction and Schedules

Aditya Agrawal
adityaag@alumni.iitm.ac.in
Department of CSE, IIT Madras
Chennai, TN, India

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

Abstract

OpenMP is one of the most popular APIs widely used to realize parallelism in C/C++ and FORTRAN programs. For efficient execution, an OpenMP program internally creates a team of threads, which share a given set of *activities* (for example, iterations of a parallel-for-loop). While OpenMP allows synchronization among these threads, many classes of computations can be conveniently expressed by specifying synchronization among the parallel activities. However, OpenMP currently restricts arbitrary synchronization among the parallel activities; otherwise, the behavior of the program can be unpredictable. While extensions like UWOmp++ (and UW-OpenMP) support all-to-all barriers among the activities, currently there is very limited support for performing point-to-point synchronization among them. In this paper, we present UWOmp$_{pro}$ as an extension to UWOmp++ (and OpenMP) to address these challenges and realize more expressive and efficient codes.

UWOmp$_{pro}$ allows point-to-point synchronization among the activities of a parallel-for-loop and supports reduction operations (during synchronization). We present a translation scheme to compile UWOmp$_{pro}$ code to efficient OpenMP code, such that the translated code does not invoke any synchronization operation(s) within parallel-for-loops. Our translation takes advantage of continuation-passing-style (CPS) to efficiently realize wait and continue operations. We also present a runtime, based on a novel communication subsystem to support efficient signal, wait, and reduction operations. We have implemented our scheme in the IMOP compiler framework and performed a thorough evaluation. We show that our approach leads to highly performant codes.

1 Introduction

The emergence of multi-core systems has brought forth many different parallel languages like X10 [11], Chapel [18], HJ [9], OpenMP [25], and so on to the mainstream. These languages provide means to express parallel logic conveniently. Besides supporting different ways of expressing parallelism, these languages support varied forms of synchronizations.

For example, OpenMP uses the efficient 'team of workers' model, where each worker (also interchangeably referred to as thread) is given a chunk of activities (for example, iterations of a parallel-for loop) to execute. An important facet of this model is that workers (and not activities) synchronize among themselves using barriers. However, certain computations (for example, stencil computations, graph analytics, and so on) are specified, arguably more conveniently, by expressing the synchronization among all the dependent activities. Languages like X10, HJ, and so on, support such notions of asynchronous activities and synchronization among the activities. Further, in contrast to global barriers (that perform all-to-all synchronization) among the parallel activities of a program, it may be more expressive and efficient (fewer number of communications) to synchronize only the inter-dependent activities. We refer to the latter as the point-to-point mode of synchronization.

We first use a motivating example to illustrate the expressiveness due to point-to-point synchronization and the scope of improved performance therein. Figure 1 shows five different versions of the classical 1D Jacobian kernel (source [3]). Figure 1a shows the kernel in OpenMP (source [4, 26]). OpenMP prohibits the use of barrier statements inside the parallel-for-loop as the behaviour of the program can be unpredictable (may lead to incorrect output, correct output, or deadlock) [2]). In light of such a restriction, at the end of the parallel-for-loop, an implicit barrier is present, which allows all threads to synchronize after computing the respective average values (`B[i]`). The 'single' block performs the pointer swapping and incrementing the time-step variable `t`. The code snippet in Figure 1b, shows the HJ version using all-to-all barriers among all the asynchronous activities. The `next` statement synchronizes all the activities before proceeding to the next phase. To support such all-to-all barriers in OpenMP, Aloor and Nandivada [2] proposed UW-OpenMP that introduces the Unique Worker (UW) model in OpenMP, in which the programmer gets an impression that each iteration (a.k.a. activity) of the parallel-for-loop is run by a unique worker and thus the model allows all-to-all barriers to be specified among the activities. Aloor and Nandivada [4] extended this idea to derive UWOmp++, and show that UWOmp++ codes are efficient and arguably more expressive compared to the OpenMP codes. Figure 1c shows UWOmp++ version of the 1D Jacobian Kernel. However, such codes still suffer from multiple drawbacks, as discussed below.

In the code shown in Figure 1c (and in Figure 1b), each activity $X_{\mathtt{i}}$ (corresponding to iteration `i`) is waiting for all remaining activities instead of only the activity X_1 waiting for all the other activities (to complete their computation), before swapping the pointers. Similarly, each activity waits for every other activity at the second barrier, even though each activity $X_{\mathtt{i}}$ ($\mathtt{i} \neq 1$) needs to wait only for X_1. This leads to significant communication overheads.

To address such issues of communication overheads and improve the expressiveness, there have been many prior efforts to support point-to-point synchronization in task parallel languages like X10 [11], HJ [9], and so on. These languages use explicit synchronization objects (like Clocks [11] and Phasers [9]) to realize the synchronization. Figure 1d shows the HJ version of the 1D Jacobian kernel. The code snippet first allocates two phaser objects and registers the phaser objects with each activity. These phasers are used to perform only the required communication (signal or wait) among the activities. A similar computation can also be encoded using an array of phasers (one unique phaser per activity).

2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT) | 979-8-3503-4254-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/PACT58117.2023.00011

979-8-3503-4254-3/23/$31.00 ©2023 IEEE
DOI 10.1109/PACT58117.2023.00011

```
t = 0;
#pragma omp parallel
{
while(t <= T) {
#pragma omp for
for(i=1;i<N-1;i++)
 B[i]=0.3*(A[i-1]+
         A[i]+A[i+1]);
 // implicit barrier
#pragma omp single
 {x=A; A=B; B=x; t++;}
 // implicit barrier
} }
```

(a) OpenMP Version

```
t = 0;
finish {
for(i=1..N){
 async phased {
  while (t <= T) {
  B[i]=0.3*(A[i-1]+A[i]+A[i+1]);
   next;
   if(i==1) {x=A;A=B;B=x;t++;}
   next; } } }
```

(b) HJ version: all-to-all barriers.

```
t = 0;
#pragma omp parallel
{
#pragma omp for
for(i=1;i<N-1;i++){
while(t <= T){
 B[i]=0.3*(A[i-1]+A[i]+A[i+1]);
 #pragma omp barrier
 if (i==1) {x=A;A=B;B=x;t++;}
 #pragma omp barrier
} } }
```

(c) UWOmp++ version

```
t = 0;
finish {
phaser p1=new phaser(SIG_WAIT);
phaser p2=new phaser(SIG_WAIT);
for(i=1..N){
 async phased (p1, p2) {
  while (t <= T) {
   B[i]=0.3*(A[i-1]+A[i]+A[i+1]);
   if(i==1){
    // wait for signal from
    // activities [2-n]
    for(j=2..n) p1.wait();
   }else{// signal task [1]
    p1.signal();
   }
   if(i==1) {x=A;A=B;B=x;t++;}
   if(i==1) // signal activities [2-n]
    for(j=2..n) p2.signal();
   else{//wait for signal from activity [1]
    p2.wait();
   } } } } }
```

(d) HJ version: point-to-point barriers

```
t = 0;
#pragma omp parallel
{
#pragma omp for
for(i=1;i<N-1;i++){
 while(t <= T) {
  B[i]=0.3*(A[i-1]+
     A[i]+A[i+1]);
  signal(i!=1,1);
  waitAll(i==1);
  if (i==1)
   {x=A;A=B;B=x;t++;}
  signalAll(i==1);
  wait(i!=1,1);
} } }
```

(e) UWOmp$_{pro}$ Version

Figure 1: 1D Jacobian computation. Here `A` and `B` are shared arrays of `N` elements and `T` indicates the number of timesteps.

In the context of OpenMP, Shirako et al. [29] present a promising approach to adapt HJ phasers to OpenMP. They allow activities to explicitly register/deregister themselves with phaser objects and these phaser objects are used to perform the synchronization among the registered activities. However, their design has multiple restrictions: (i) the phaser objects have to be explicitly allocated – leads to cumbersome code, (ii) the synchronization can only be in one direction, that is, from iteration with lower index to iteration with higher index – can be limit expressiveness; (iii) threads (not activities) block on wait operations – can limit parallelism and impact performance negatively; (iv) their scheme cannot work with dynamic/guided scheduling of OpenMP; and (v) the activities cannot perform reduction operations at the synchronization points. OpenMP 5.2 [25] provides support for expressing dependencies between the iterations of parallel-for-loops with the doacross clause [30, 31]. Although this feature allows expressing some form of point to point synchronization, it also suffers from two of the restrictions ((ii) and (v)) discussed above.

In this paper, we address all these issues and propose a generic scheme to allow synchronization among the activities of each parallel-for-loop of OpenMP. We call our extension UWOmp$_{pro}$. Figure 1e shows a UWOmp$_{pro}$ version of the kernel shown in Figure 1c. Here, the first all-to-all barrier of Figure 1c has been replaced with two commands, where all activities (except the first activity) signal X_1, and X_1 in turn waits for the signals from them. A convenient feature of UWOmp$_{pro}$ is that it supports conditional signal/wait commands. The first argument passed to the corresponding commands, evaluates to 1 (true) or 0 (false) and determines if the command should be executed by that activity or not. Further, the `signal` (`wait`) commands can signal to (wait for) multiple activities that are specified by a comma-separated list of iterations. Example: `signal(1,i-1,i+1)` sends a signal to X_{i-1} and X_{i+1}.

The second barrier of Figure 1c is replaced by `signalAll`, followed by `wait`. The given condition in the `signalAll` command ensures that signalling is done only by X_1 to all the remaining activities. These activities (X_i, $i \neq 1$) in turn wait for that signal.

In contrast to X10 and HJ, in UWOmp$_{pro}$, activities of a parallel-for-loop can synchronize among themselves without any need for the programmer to explicitly create (or pay the overheads of) clock/phaser objects. Further, the UWOmp$_{pro}$ code performing communication is arguably more readable than that of the HJ code involving multiple phaser objects performing point-to-point communication (for example, Figure 1e vs. Figure 1d). An important aspect of our design is that we continue to take advantage of the efficient 'team of workers' model of OpenMP to derive high performance.

In addition, UWOmp$_{pro}$ optionally supports efficient reduction operations at the synchronization (wait) points, a feature not supported by languages like X10, HJ, or even OpenMP. Note: though OpenMP supports reduction operations in parallel-for-loops, the final reduced value is only available at the end of the parallel-region (and not immediately after the reduction operation).

UWOmp$_{pro}$ can help effectively and efficiently code wide classes of problems involving point-to-point synchronizations and reductions. Note: We do not claim that using point-to-point synchronization among the activities of parallel-for-loops is the only/best way to encode such computations. Instead, our proposed extension (common in modern languages like X10, HJ, and so on) provides additional ways to encode task parallelism, which is otherwise missing in OpenMP (and UWOmp++), while not missing out on the advantage of the efficient 'team of workers' model of OpenMP.

Our Contributions

- We propose UWOmp$_{pro}$ to allow point-to-point synchronization and reduction operations, among the activities of a parallel-for-loop. In contrast to UWOmp++, UWOmp$_{pro}$ supports all the scheduling policies defined in OpenMP.
- We present a scheme to compile UWOmp$_{pro}$ code to efficient OpenMP code by taking advantage of continuation-passing-style

(CPS) to efficiently realize wait and continue operations.
• We present a runtime based on a novel communication subsystem to support efficient signal, wait, and reduction operations.
• To support fast reduction operations, we propose two reduction algorithms termed *eager* and *lazy*, to support efficient reduction operations among a subset of activities and all activities, respectively.
• We have implemented our scheme in the IMOP compiler framework and performed a thorough evaluation. We show that our generated code scales well and is highly performant.

2 Background

We now present some brief background needed for this paper.

OpenMP. We summarize three popular constructs of OpenMP.

Parallel Region: `#pragma omp parallel S` creates a team of threads where each thread executes the statement S in parallel.

Parallel-For-Loop: A sequential for-loop can be annotated using `#pragma omp for [nowait] [schedOpt]` to share the iterations among the team of threads. The scheduling policy (static, dynamic, guided, or runtime) is mentioned using `schedOpt`. If the `nowait` clause is omitted, OpenMP provides an implicit barrier.

Barrier: `#pragma omp barrier` construct is used to synchronize the workers in the team.

Unique Worker Model for OpenMP. We now restate two relevant definitions given by Aloor and Nandivada [4], for OpenMP.

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

Definition 2.2. A parallel-for-loop is said to be executing in (One-to-Many model) or OM-OpenMP model if a worker may execute one or more iterations of a parallel-for-loop. OM-OpenMP model is the default execution model in OpenMP. A program executing in OM-OpenMP model cannot invoke barriers (or wait commands) inside work-sharing constructs.

3 UWOmp$_{pro}$: Extending UWOmp++

We now describe three new extensions to UWOmp++ that can improve the expressiveness and lead to efficient code. Two of these extensions (support for point-to-point synchronization among the activities, and performing reduction at the point of synchronization) are novel to OpenMP as well. The third extension admits powerful scheduling policies (*dynamic*, *guided*, and *runtime*) of OpenMP, apart from the *static* scheduling policy that was already supported by UWOmp++. We call this extended language UWOmp$_{pro}$.

Point-to-Point Synchronization. UWOmp$_{pro}$ proposes an extension to UW-OpenMP, where a programmer can specify point-to-point synchronization among the activities of a parallel for-loop. Figure 2 summarizes the list of commands supported by UWOmp$_{pro}$ (along with brief syntax), for easy reference. All these commands are conditional in nature and support (i) signal and wait operations to a subset of activities or all of them, and (ii) (optionally) reduction operations. Note: a signal/wait commands to/on a non-existing iteration are treated as *nop*s.

Reduction. Consider the example code snippet shown in Fig. 3a (Source [3, 10]) to perform iterated averaging on an `N` element array, written in UWOmp++. Here, each activity X_i first computes a new value for the `i`th element using `A[i-1]` and `A[i+1]` and then computes the absolute difference compared to the older value. Towards the end of each iteration of the while-loop, each activity

commands	syntax
signal $^{+}$	`signal`(int *e*, int *act*, ...);
wait $^{+}$	`wait`(int *e*, int *act*, ...);
signalAll *	`signalAll`(int *e*);
waitAll *	`waitAll`(int *e*);
signalSend $^{+,r}$	`signalSend`(int *e*, void *m, int *act*, ...);
waitRed $^{+,r}$	`waitRed`(int *e*, `FptrT` *rOp*, void *$rVar$, int *act*, ...);
signalAllSend ^{*,r}	`signalAllSend`(int *e*, void * m);
waitAllRed ^{*,r}	`waitAllRed`(int *e*, `FptrT` *rOp*, void *$rVar$);

Figure 2: List of commands supported by UWOmp$_{pro}$. The first argument *e* is a predicate expression; the signal/wait operation is invoked only if *e* evaluates to true. `FptrT` specifies a function pointer type, used to pass the reduction operator function. Varargs are used when we have to specify more than one activity. Brief description of the remaining arguments: *act*: target activity, *m*: message, *rOp*: reduction function, *rVar*: reduction variable. A superscript of '$*$' indicates the command interacts with all the activities, '+' indicates that the command interacts with one or more activities, and '*r*' indicates that the command supports reduction operation.

```
#pragma omp parallel
{
 #pragma omp for
 for(i=1;i<N;i++) {
  while(diffSum <= epsilon) {
   B[i]=(A[i-1]+A[i+1])*0.5;
   diff[i] = abs(A[i]-B[i]);
   #pragma omp barrier
   if(i==1){
   diffSum=computeSum(diff,N);
    x=A; A=B; B=x; }
   #pragma omp barrier
   } /*while*/ } /*for*/ }
```

(a) UW-OpenMP Version

```
#pragma omp parallel
{
 #pragma omp for
 for(i=1;i<N;i++){
  while(diffSum <= epsilon) {
    B[i]=(A[i-1]+A[i+1])*0.5;
    diff[i] = abs(A[i]-B[i]);
    signalAllSend(1,diff[i]);
    waitAllRed(1,diffSum,ADD);
    if(i==1){x=A; A=B; B=x;}
    signalAll(i==1);
    wait(i!=1,1);
  } /*while*/ } /*for*/ }
```

(b) UWOmp$_{pro}$ Version

Figure 3: Iterated Averaging. Here `A` and `B` are shared arrays of `N` elements and epsilon specifies the tolerance limit.

waits for X_1 to sequentially reduce the array `diff` to the shared variable `diffSum`, which is used to check the convergence condition specified in the while-loop predicate. The sequential reduction operation can pose serious performance overheads. Note that, we cannot use the OpenMP reduction operation to perform the reduction here, as the reduced value would only be available after the end of the parallel for-loop. To address these issues, UWOmp$_{pro}$ supports a blocking reduction operation within the activities of a parallel for-loop. For example, in Fig. 3b, after computing `diff[i]`, each activity X_i sends a signal to all the other activities with the value of `diff[i]`. Then, the code invokes a blocking reduction operation, specifying the variable (`diffSum`) to hold the reduced value, and the reduction operation (`ADD`). In contrast to the UW-OpenMP version, in UWOmp$_{pro}$, all threads together perform the reduction operation in parallel. Further, to reduce the number of message exchanges, Section 5.4 presents an optimization such that messages, linear (not quadratic) in the number of activities are exchanged to perform the reduction. Note: (1) For readability, we use verbose reduction operator names (e.g., ADD in place of '+'). (2) Like the regular reduction operations in OpenMP, UWOmp$_{pro}$ also supports user-defined reduction operations; details skipped for brevity.

Schedules. Due to its design decisions, UW-OpenMP supports only static scheduling. Considering the importance of other scheduling

policies of OpenMP, UWOmp$_{pro}$ supports all of them by using a runtime extension. Details in Section 5.5.

4 UWOmp$_{pro}$ to Efficient OM-OpenMP

We now present the translation rules used to convert input UWOmp$_{pro}$ code to efficient OM-OpenMP code. The main idea behind our translation is that in the generated OM-OpenMP code, the activities of the parallel-for-loop are stored as closure (in one or more work-queues) to be executed by different workers. When an activity encounters a wait operation, it enqueues the continuation to the work-queue of the parent activity and continues executing other activities in the work-queue. Figure 4 shows the block diagram of our translation. The input UWOmp$_{pro}$ code is fed to the Simplifier module which converts the given input code to a representative subset of UWOmp$_{pro}$ code called mUWOmp$_{pro}$. The mUWOmp$_{pro}$ code is input to the 'CPS Translator' module which converts the code to CPS form called UWOmpCPS$_{pro}$. The UWOmpCPS$_{pro}$ code is input to the 'OM-OpenMP Translator' module which translates the code to conforming OpenMP code such that it does not invoke barriers inside work-sharing constructs (parallel-for-loops). Finally, a post-pass step introduces type-specific reduction operations. We now describe these important modules of our translation scheme.

4.1 Simplifier

For the ease of explaining the translation scheme, like Aloor and Nandivada [4], we use a representative subset of the input UWOmp$_{pro}$ language called miniUWOmp$_{pro}$ (mUWOmp$_{pro}$); Section 6 discusses how any general UWOmp$_{pro}$ program can be translated to mUWOmp$_{pro}$ code. Figure 5 shows the grammar of mUWOmp$_{pro}$. A mUWOmp$_{pro}$ program consists of a sequence of function declarations (`FuncDecl`) followed by the `MainFunc`. `FuncDecl` can have an assignment statement, a function call, return statement, barrier statement or a statement generated by Seq(X): the program formed from X closed under sequential constructs. `MainFunc` consists of a parallel region which in turn consists of a sequence of parallel-for-loops or barrier statements. Each parallel-for-loop is a normalized loop [22] whose body is a function call.

4.2 CPS Translator

Our translation scheme is inspired by that of Aloor and Nandivada [4], who translate an input program to an IR (called UWOmpCPS), before lowering it to OM-OpenMP. UWOmpCPS is an extension to CPS (Continuation Passing Style [19]); its choice was inspired by the fact that CPS naturally provides support for operations like wait and continue. A UWOmpCPS program is similar to a program in CPS form, except that the former may include parallel-for-loops and barriers. One of the sources of overheads of the scheme of Aloor and Nandivada was that all the methods were converted to CPS form. We observe that since only the activities of parallel-for-loops can synchronize with each other (point-to-point or all-to-all), we need to CPS transform only those functions that may be invoked by the iterations of the parallel-for-loop. Further, in the input UWOmp$_{pro}$ program, thread-level barriers (invoked via `#pragma omp barrier`), may appear outside the work-sharing constructs.

Based on these points, we first present a modified UWOmpCPS grammar and then discuss the modified translation rules.

4.2.1 UWOmpCPS$_{pro}$: Modified CPS IR. The grammar for the modified IR (called UWOmpCPS$_{pro}$) is shown in Figure 6. Some of the main differences between UWOmpCPS and UWOmpCPS$_{pro}$ are as follows: (i) A program may consist of both CPS (`CPSFuncDecl`) and non-CPS (`FuncDecl`) functions. (ii) A `CPSParRegion` may contain a set of parallel-for-loops in CPS form (`CPSParLoop`) or barriers (`BarrierStmt`). (iii) A `CPSParLoop` can specify a schedule and related options (represented as `schedOpt`). Note that `Stmt` denotes any sequential statement, `FuncDecl` is any regular C function declaration, `FunCall` is any regular non-CPS function call statement. As is standard in CPS translation, the continuation object is passed as an additional argument to each CPS function call (`CPSFunCall`).

4.2.2 Generation of code in CPS form. Figure 7 shows the translation rules. Here, a rule of the form $[\![X]\!] \Rightarrow Y$ is used to denote that input code X is transformed to the output code Y in UWOmpCPS$_{pro}$. The RHS (Y) may contain further terms with $[\![\ ]\!]$ indicating that those terms need to be further transformed. Here, we use `#ompparallel` as a shortcut for `#pragma omp parallel`, and `#ompfor` for `#pragma omp for nowait schedOpt`. Our CPS transformation starts by transforming the parallel-region (Rule 9). Rules 1-8 are the standard CPS translation rules used to convert a given input program to CPS Form. Note: We only handle code that is reachable from a parallel-for-loop and if any function is called from outside the parallel-region then it is left as it is. Rule 10 has two substeps: (i) the function (`fun`) called in the body of the parallel-for-loop is translated to CPS form (using the standard CPS transformation rules, [Rules 1-8, Figure 7], by passing the identity function *id* as the continuation. Here, *mkClsr* is a macro that creates a closure by taking three arguments: a function pointer, the list of arguments required for the function (obtained by invoking a compiler-internal routine *bEnv*), and a continuation to be executed after executing the function. (ii) The call to `fun` is replaced by its CPS counterpart by passing the continuation as an additional argument. If a barrier is encountered, it is left as it is (Rule 11).

4.3 OM-OpenMP Translator

We now discuss how we translate code in UWOmpCPS$_{pro}$ format to OM-OpenMP code. We emit code such that each iteration of the parallel-for-loop creates a closure object and enqueues to a work-queue. The details of the work-queue depend on the scheduling policy of the parallel-for-loop. For static scheduling policy, the activities to be executed by each thread is fixed a priori and thus we maintain a local worklist for each thread. For guided or dynamic scheduling, all closures are pushed to a global 'work queue'. Each thread dequeues closures from the queue and executes the same. Figure 8a shows the rule to translate the parallel-for-loop.

Line 4 in Figure 8a calls the function `getScheduler` that takes the scheduling policy string *sched* and threadID *tid* as parameter. This string is obtained using a call to the function `getSchedule` using the *schedOpt* string as parameter. Figure 8b shows the pseudo-code of the `getScheduler` function that returns a pointer to the corresponding scheduler function and assigns the worklist to be used by each thread ($WL[tid]$). The function assigns the *schedPtr* to the corresponding scheduler function, depending on the value of *sched*. We emit a parallel-for-loop that pushes the closure for each activity to $WL[tid]$ (Lines 5-9 in Figure 8a). Finally, we invoke the appropriate scheduler (Line 10 in Figure 8a).

Figure 4: Block diagram of our proposed translation scheme.

Program	::=	(FuncDecl)* MainFunc
FuncDecl	::=	Type ID(Args){ (Stmt)* RetStmt}
MainFunc	::=	int main() { ParRegion }
ParRegion	::=	**#pragma omp parallel** { (ParLoop | BarrierStmt)* }
BarrierStmt	::=	**#pragma omp barrier**
ParLoop	::=	**#pragma omp for nowait schedOpt** for(ID=0;ID<ID;ID++){ FunCall }
Stmt	::=	SimpleStmt | FunCall | RetStmt | Seq (Stmt)
SimpleStmt	::=	AssignStmt | IfStmt
AssignStmt	::=	ID = SimpleExpr;
FunCall	::=	ID(ActualParamList);
SimpleExpr	::=	ID | Op ID | ID Op ID
IfStmt	::=	if (SimpleExpr) { (Stmt)* }

Figure 5: Grammar for mUWOmp$_{pro}$

Program	::=	(CPSFuncDecl)* (FuncDecl)* MainFunc
CPSFuncDecl	::=	void ID(Clsr K,Args){(SimpleStmt)* TailCallStmt}
MainFunc	::=	int main() { CPSParRegion }
CPSParRegion	::=	**#pragma omp parallel** { (CPSParLoop | BarrierStmt)* }
CPSParLoop	::=	**#pragma omp for nowait schedOpt** for(ID=0;ID<ID;ID++){ (SimpleStmt)* CPSFunCall }
TailCallStmt	::=	CPSFunCall | CPSIfStmt | CPSParLoop
CPSFunCall	::=	ID(ID,ActualParamList);
CPSIfStmt	::=	if (SimpleExpr) { (SimpleStmt)*CPSFunCall }

Figure 6: Grammar for UWOmpCPS$_{pro}$

1.	⟦ K *T* fun(args) { S } ⟧ //fun is called from within parallel-**for**-loop	⇒ *void* funCPS(Clsr K,args) { ⟦ K S ⟧ }
2.	⟦ K fun(a_1,...,a_n) ⟧	⇒ funCPS(K,a_1,...,a_n)
3.	⟦ K S1; S2 ⟧ //S1 has no call, return //or pragmas inside.	⇒ S1; ⟦ K S2 ⟧
4.	⟦ K { S } ⟧	⇒ { ⟦ K S ⟧ }
5.	⟦K S⟧//S: an AssignStmt	⇒ S
6.	⟦ K **return** x ⟧	⇒ K x
7.	⟦ K α ⟧ //α is a RetStmt or //AssignStmt. //α=X fun(args) Y //X has no calls	⇒ mkProc(void pCPS(Clsr K, T_1,V_1) {⟦ K X V_1 Y ⟧}); C=mkClsr(pCPS,bEnv(X,Y),K); funCPS(C,args)
8.	⟦ K **if**(e) { S1 } **else** { S2 } Y⟧ //e and S1 have no calls.	⇒ if(e) { ⟦ K S1 ⟧ } else { ⟦ K S2 ⟧} ⟦ K Y ⟧
9.	⟦ K #**ompparallel** { S } ⟧	⇒ #**ompparallel** { ⟦ K S ⟧ }
10.	⟦K #**ompfor** **for**(Header){ fun(args); } S ⟧	⇒ #**ompfor** **for**(Header){ K= *mkClsr*(*id*, *null*, *null*); funCPS(K, args); } ⟦K S ⟧
11.	⟦ K S ⟧//S is a BarrierStmt	⇒ S

Figure 7: CPS translation rules for the parallel constructs.

If schedOpt is omitted, then getSchedule sets the schedule to static. Similarly, if schedOpt=*runtime*, then getSchedule obtains the schedule from the language-specified environment variable.

4.4 Post-Pass: Type Specific Reduction Ops

The final step in our translation process introduces type specific reduction operations for the operations specified in the OpenMP specification [25]. As mentioned in Section 3, the reduction-related

```
#ompfor                                  1  tid=thread-number();
for(Header){                             2  sched=getSchedule(schedOpt);
 K=mkClsr(X);              ⇒             3  chSize=getChunkSize(schedOpt);
 fCPS(K,args);                           4  schedPtr=getScheduler(sched,tid);
}                                        5  #ompfor
                                         6  for(Header) {
                                         7    K=mkClsr(X);
                                         8    C=mkClsr(fCPS,bEnv(args),K);
                                         9    enqueue(WL[tid],C);}
                                        10  (*schedPtr)(chSize);
```

(a) Rules to translate UWOmpCPS$_{pro}$ to OM-OpenMP.

```
1  funcPtr getScheduler(sched,tid){
2    if(sched is static){
3      schedPtr=&scheduler-static; WL[tid]=new WL(); /*local Q*/
4    }else if(sched is dynamic){
5      schedPtr=&scheduler-dynamic; WL[tid]=gWL; /*global Q*/
6    }else{ // guided
7      schedPtr=&scheduler-guided; WL[tid]=gWL;/*global Q*/}
8    return schedPtr; }
```

(b) Helper function to set the task worklist and return the scheduler.

Figure 8: UWOmpCPS$_{pro}$ to OM-OpenMP Translation.

wait commands (waitRed and waitAllRed, Figure 2) in the input UWOmp$_{pro}$ code take a reduction operation and a reduction variable *rVar* (which stores the reduced result), as additional arguments to the wait command. The compiler uses the declared type of *rVar* (say, int) to replace the user specified reduction operation (say, ADD representing the '+') with the actual reduction function (say, ADDint) in the wait commands. For each of the primitive types T, our runtime provides functions for performing the reduction (for example, ADDint, ADDdouble, and so on).

In addition to introducing the type-specific reduction operation, the reduction procedure needs a method to copy values from one variable to the other (for example, to copy the final computed value to the reduction variable). Similar to the type-specific reduction operations, for each primitive type T, our runtime provides functions for performing the copy operation (e.g., 'COPYint(int *from, int *to)'), which is passed as an additional argument to the wait method calls. For example, the command waitAllRedCPS (K, i==1, ADD, x), where x is the reduction variable of type int, gets replaced by waitAllRedCPS (K, i==1, ADDint, x, COPYint).

4.5 Example translation

For a better understanding of our translation scheme, Figure 9 describes the steps to transform a sample UWOmp$_{pro}$ code to OM-OpenMP code. Figure 9a shows the input UWOmp$_{pro}$ code and Figure 9b shows the CPS transformed version. The standard set of CPS transformation rules is applied to the function f to convert it to fCPS, and generate other CPS functions (pCPS1 and pCPS2). We avoid showing the second argument to mkClsr as it depends on the actual statements following the call (for example, S2 and S3). The parallel-for-loop body creates an identity closure K to denote the continuation after executing the parallel-for-loop. It calls fCPS with closure K as an argument.

```
void f(args){
 S1; signalSend(1,m,i+1);
 S2; waitRed(1,ADD,x,i-1);
 S3; }
int main(){
#ompparallel
{
  #ompfor
   for(Header){f(args);} } }
```

(a) Input UWOmp$_{pro}$ code.

```
void fCPS(K,args){
 S1;
 C1=mkClsr(pCPS1,...,K);
 signalSendCPS(C1,1,m,i+1);}
void pCPS1(K){
 S2;
 C2=mkClsr(pCPS2,...,K);
 waitRedCPS(C2,1,ADD,x,i-1);}
void pCPS2(K){
 S3;
 Invoke Continuation in K;}
int main(){
#ompparallel
{
#ompfor
  for(Header){
   K=mkClsr(id,null,null);
   fCPS(K,args); } } }
```

(b) UWOmpCPS$_{pro}$ code

```
int main(){
#ompparallel
{
tid=thread-number();
sched=getSchedule(schedOpt);
chSize=getChunkSize(schedOpt);
schedPtr=getScheduler(sched,tid);
 #ompfor
  for(Header){
   K=mkClsr(id,null,null);
   C=mkClsr(fCPS,bEnv(args),K);
   enqueue(WL[tid],C); }
  (schedPtr)(chSize); } }
```

(c) Translated OM-OpenMP Code. Only the changes are shown.

```
void pCPS1(K){
 S2;
 C2=mkClsr(pCPS2,bEnv(S3),K);
 waitRedCPS(C2,1,ADDint,x,
   COPYint,i-1); }
```

(d) OM-OpenMP Code with postpass. Only the changes are shown.

Figure 9: Example Transformations.

Figure 9c shows the OM-OpenMP translated code of the UWOmpCPS$_{pro}$ code. This step emits code to identify the appropriate scheduler (Lines 2-7 from Figure 8b), and wraps the call to function `fCPS` inside the closure `C` before enqueuing the closure in the appropriate worklist $WL[tid]$. Finally, Figure 9d shows the OpenMP translated code with the postpass translation rules applied on the `waitRedCPS` method.

5 Runtime Support

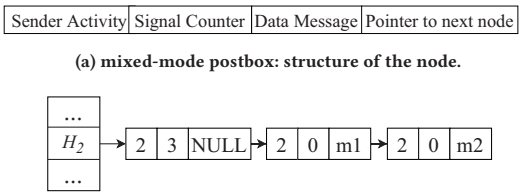
We now describe the extensions to the OpenMP runtime that we made to support the key operations supported by the language extensions defined in UWOmp$_{pro}$: signalling, waiting, performing reduction and supporting the different scheduling policies of OpenMP. We will start by describing our novel design of the communication sub-system between the activities of each parallel-for-loop that forms the basis for these key operations.

5.1 Shared Postbox System for Communication

We present a postbox based system for communication between the activities of a parallel-for-loop. We discuss the design of three types of postboxes: *signal-only*, *data-messages-only*, or mixed signals and data-messages (*mixed-mode*).

5.1.1 Design of the Postbox. Each activity X_i of a parallel-for-loop, may receive one or more signals/data-messages from other activities. To avoid contention among the communicating activities, we associate a postbox with each activity X_i. Thus, the postbox P is an array of N elements (where N is the total number of activities), such that each element P_i represents the postbox of X_i.

We observed that for most of the parallel-for-loops using point-to-point synchronization, the number of activities that an activity communicates with, in a phase, is small. Based upon this observation, for such loops we set each postbox P_i to be a hashmap (of initial-size set to a constant k, with load factor set to a constant $M\%$). Note that there are two straightforward alternatives to our proposed scheme: (i) each post-box entry P_i, is an array of N slots - no locking required among the activities communicating with any particular activity X_i, but leads to high space wastage. (ii) each post-box entry P_i is represented as a linked-list - low space overhead, but may lead to significant performance overheads due to the locking contention among the activities communicating with any particular activity X_i. We use the hash-maps as a middle ground for supporting communication among the activities. In Section 6 we discuss an optimization where we can further reduce the overheads of this hash-map based postbox to a large extent for the common case of all-to-all communication (with static scheduling policy).



(a) mixed-mode postbox: structure of the node.

(b) Postbox example: P_1 is the postbox of activity X_1 and H_2 is the hashed-index of activity X_2 in P_1. X_2 sends 3 signals and 2 data-messages to X_1.

Figure 10: Mixed-Mode Postbox: Structure and Example.

The exact configuration of the slots of each postbox entry depends on the type of communication: *signal-only*, *data-only*, or *mixed-mode*. We briefly explain the first two modes and then explain the *mixed-mode* type of postbox, in more detail.

Signal Only Postbox If the communicating activities are guaranteed to never send/receive any data-messages, then we simply represent each slot in the hashmap as a list of pairs of the form (sender, counter). When an activity X_i wants to send a signal to X_j, we simply increment the counter of X_j in P_i At the receiving activity, we atomically decrements the counter, if non-zero we return 1. Else, we return 0 (indicates that the signal is not yet available).

Data Only Postbox If the communicating activities are guaranteed to send/receive only data-messages, then we represent each slot as a list of pairs of the form (sender, data-message). When an activity wants to send a data-message, we append the message to the appropriate list, and for the receiving activity we take out and return the first message of the sender available in the list. If no such message is available, we return NULL.

Mixed-mode Postbox. We use this type of postbox, when the communicating activities may send either type of messages. We implement each slot as a list, where each element of the list is of the form shown in Figure 10a. Consider an element e of the form (j, `ctr`, `m`, `next`) in one of the lists of P_i. If `ctr` is non-zero then e represents `ctr` number of contiguous signals sent from X_j to X_i. Else, e represents a data-message `m` sent from X_j to X_i. For example, Figure 10b shows an example list, on receiving the following signals/data-messages from X_2 to X_1: signal, signal, signal, and data-messages m1 and m2.

The postbox supports two routines: `sendMsg` and `recvMsg`. The `sendMsg` routine updates the appropriate list, and the `recvMsg` routine returns the appropriate signal/data-message, if available. We skip the details of these routines for brevity. The details can be found in the extended report [1].

signalCPS(ClsrT K, int e, int act, ...);
waitCPS(ClsrT K, int e, int act, ...);
signalAllCPS(ClsrT K, int e);
waitAllCPS(ClsrT K, int e);
signalSendCPS(ClsrT K, int e, void *m, int act, ...);
waitRedCPS(ClsrT K,int e,FptrT rOp',void* $rVar$,FptrT $copy$,int act, ...);
signalAllSendCPS(ClsrT K, int e, void *m);
waitAllRedCPS(ClsrT K, int e, FptrT rOp', void *$rVar$, FptrT $copy$);

Figure 11: Signatures of the signal/wait methods in CPS form; derived from signatures shown in Figure 2. `FptrT` **specifies a function pointer type, used to pass the reduction operator function.** `ClsrT` **specifies the continuation type. Brief description of the arguments:** K**: continuation,** e**: predicate,** act**: target activity,** m**: message,** rOp'**: reduction function,** $rVar$**: reduction variable,** $copy$**: copy function.**

Note: (I) The `recvMsg` routine is non-blocking in nature. The actual waiting, if at all, is performed by the wait-call invoking the `recvMsg` of the postbox. (II) We use a static analysis to decide which type of postbox is to be used, based on the signal/wait commands specified in the input program.

5.2 Signal Algorithm

We now describe the wrapper methods emitted by the CPS transformation (Section 4.2) to handle the signal commands: `signalCPS`, `signalSendCPS`, `signalAllCPS` and `signalAllSendCPS`. The first two methods take variable number of arguments, corresponding to the list of activities to whom the signal/message is to be sent. The wrapper methods `signalCPS` and `signalAllCPS` simply call `signalSendCPS` and `signalAllSendCPS`, respectively, by passing the message argument m as NULL. We now describe the `signalSendCPS` and `signalAllSendCPS` methods (signatures shown in Figure 11). An interesting point about these wrapper methods is that they are in CPS form and take the continuation K as an argument.

The method `signalSendCPS` first checks the predicate e. If true, it does the actual signalling by sending the message to each receiving iteration. Finally, it invokes the continuation. The design of `signalAllSendCPS` is similar, except that it stores the message of each sender i at the i^{th} element of a shared array. The details of these algorithms can be found in the extended report [1].

5.3 Wait Algorithm

We now describe the wrapper methods emitted by the CPS transformation (Section 4.2) to handle the wait commands: `waitCPS`, `waitRedCPS`, `waitAllCPS` and `waitAllRedCPS`. The first two methods take as arguments the list of (target) activities from whom the signal/message is to be received. The wrapper methods `waitCPS` and `waitAllCPS` simply call `waitRedCPS` and `waitAllRedCPS`, respectively, by passing the reduction specific arguments as NULL. Similar to the signal wrapper methods, these methods are also in CPS form. For brevity, we only describe the `waitRedCPS` and `waitAllRedCPS` methods.

The method `waitRedCPS` (signature in Figure 11) first checks if the conditional-expression e is true. If so, it first checks if the signal/message has been received from all of the target activities. For each received data-message, it performs the reduction operation. If all the signals/messages have been received, then it invokes the continuation K. Else, it creates a closure remembering the set of activities whose signals/messages are yet to be processed and

```
void scheduler-static(chSize) // chSize unused here
begin // work already divided during enqueing in Figure 8a.
  executeWL(WL[tid]);
```

Figure 12: UWOmp$_{pro}$ static scheduling algorithm.

```
void scheduler-dynamic(chSize)
begin
  WorkList rdyWL=empty-worklist;
  while true do
    begin Atomic
      if !gWL.isEmpty() then
        rdyWL=gWL.dequeue(chSize) else break ;
    executeWL(rdyWL);
```

Figure 13: UWOmp$_{pro}$ dynamic scheduler algorithm.

pushes the closure to the appropriate work-queue, before returning from the function. This ensures that the thread executing the wait-wrapper function does not block (or busy wait). The `waitAllRedCPS` method works similarly, by waiting for all the messages to be available before performing the reduction. The details of these algorithms can be found in the extended report [1].

5.4 Reduction Operations

We now highlight some salient points about our reduction strategy. As discussed in Section 5.3, the reduction operation is invoked eagerly for point-to-point synchronization (`waitRedCPS`), as and when the message from any target activity is processed. However, for the all-to-all synchronization (`waitAllRedCPS`), we efficiently perform the reduction after all the messages have been received, in a *lazy* manner. We now describe the intuition behind this design.

One main drawback of the eager method of reduction is that it is inherently serial in nature; hence each activity may take up to $O(N_a)$ steps for reduction, where N_a is the number of activities participating in reduction. While for small values of N_a this cost may be minimal, it can be prohibitively high, for large values of N_a; a common use-case being performing all-to-all reduction (realized by consecutive calls to `signalAllSend` and `waitAllRed` commands of UWOmp$_{pro}$). To address this issue in case of all-to-all reduction we use the lazy mode of reduction. The algorithm works on the principle of the popular parallel message-exchange based protocol [27] that leads to each activity performing $O(log_2(N_a))$ steps; in this scheme, after every time step t, each activity holds a reduced value over the messages of 2^t activities. However, for small values of N_a, we continue to use the eager mode and avoid the storage overhead of the shared array.

5.5 Supporting Different Scheduling Policies

UWOmp++ [4] could not handle any scheduling policies of OpenMP except static scheduling. Considering the importance of scheduling policies beyond static, we also provide support for *dynamic*, *guided* and *runtime* scheduling.

In Section 4.3, we discuss how the `getSchedule` function handles the *runtime* schedule option during execution. We now discuss the details of the remaining three schedulers.

static scheduler. The scheduler function `scheduler-static` (Figure 12) simply executes all the closures present in $WL[tid]$. We skip the definition of executeWL for brevity. In this scheduling, each thread maintains its own local worklist and as a result, in the

`waitRedCPS` function (described earlier in Section 5.3), the locking mechanism before and after the enqueue operation is not required.

dynamic-scheduler and guided-scheduler. As discussed in Section 4.3, for dynamic scheduling we use the global worklist. In scheduler-dynamic (Figure 13), each thread atomically dequeues (at most) *chSize* number of closures from the worklist and executes them. The `scheduler-guided` function works similar, except that *chSize* is updated after each atomic dequeue. We skip the code for the same, for brevity.

6 Discussion

We now discuss three salient features of our proposal.

• **Translating input UWOmp$_{pro}$ programs to mUWOmp$_{pro}$ code.** We use the following simplification steps (similar to that of Aloor and Nandivada [4]), to convert any general UWOmp$_{pro}$ code to mUWOmp$_{pro}$, before we invoke our CPS translator. We apply these 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 replaced with the body of 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.*Set of Statements inside the parallel-region and not a parallel-for-loop or barrier statement.* Similar to Step 1, we first move the set of statements to a separate function (say foo). The statements include the set of sequential statements until we hit a barrier statement or parallel-for-loop. Then, since the code has to be executed by all the workers, we replace the sequence of statements with the following code:

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

Note: The arguments to `foo` are the list of free variables and T denotes the number of workers executing this code.

• **Optimization for Static Scheduling.** We optimize the postbox for all-to-all based synchronization kernels and static scheduling policies by using a worklist implemented as a single array of closures with two indices (`left` and `right`) per thread. When a thread hits a barrier, it resumes executing the continuation only after (i) the thread has finished executing all the other closures in its worklist (between `left` and `right`), and (ii) the remaining activities have also reached the barrier. This approach simplifies maintenance and reduces memory overhead.

• **Maintaining the thread-id.** The UW model gives a guarantee that each iteration is executed by a unique worker. Thus, querying the thread-id at any point in the iteration should return the same value (consistent thread-id requirement). However, in our proposed solution, an iteration is divided into one or more closures executed by different threads. To satisfy the consistent thread-id requirement, we store the expected thread-id of each iteration in the closure, and modify the `omp_get_thread_num` function to access this closure.

• **Compiling UWOmp$_{pro}$ code with OpenMP disabled.** Unlike regular OpenMP codes, as is usual with codes using point-to-point synchronization, the semantics of UWOmp$_{pro}$ may not match with their serial counter-parts (obtained by compiling the code by disabling OpenMP).

Sl	Bench[Src]	Brief description	I/P	a2a	p2p	Redn
1	3MM [26]	3 Matrix Multiplication	8K	✓		
2	LCS [24]	Longest Common Subseq.	32K	✓		
3	MCM [12]	Matrix Chain Mult.	32K	✓		
4	WF [26]	Wave Front Simulation	32K	✓		
5	LELCR [16]	Leader Election	128K	✓		
6	GEMVER [26]	Vect. Mult. and Mat. Add	64K	✓		
7	KPDP [26]	0/1 Knapsack	128K	✓		
8	Jacobi1D [26]	1 dim Jacobian	128K		✓	
9	Jacobi2D [26]	2 dim Jacobian	128K		✓	
10	Stencil4D [32]	4 dim Stencil	128K		✓	
11	SOR [8]	Successive-Over Relax	128K		✓	
12	Seidel2D [26]	2 dim Gauss Seidel	128K		✓	
13	IA [13]	1 dim Iterated Avg.	4K	✓	✓	✓
14	HP [7]	4 dim Heated Plate	4K	✓	✓	✓

Figure 14: Benchmarks used in UWOmp$_{pro}$. Abbreviations: a2a = all-to-all, p2p = point-to-point, Redn = Reduction.

• **Signal/wait outside parallel-for-loops.** UWOmp$_{pro}$ assumes that during execution, signal/wait functions are never invoked from outside parallel-for-loops. To handle invocation of signal/wait outside parallel-for-loops, we ensure that signal/wait (and their CPS counterparts) will abort if not invoked inside a parallel-for-loop.

• **Possibility of deadlocks.** Similar to clocks (X10), and phasers (HJ), programs written in UWOmp$_{pro}$ can also deadlock. For example, iterations may wait for each other, without sending signals. But if a UWOmp$_{pro}$ program has no such dependencies (using signal/wait commands) causing circular-wait, then the translated code will not lead to circular-waits (and hence no deadlocks).

• **Multi-file compilation.** For ease of presentation, the paper discussed the concepts assuming that there is a single file. To support multi-file compilation, we require that all files are compiled with a suitable option (e.g., -uwpro) or none are compiled with the option.

7 Implementation and Evaluation

We implemented our proposed language extension, translation and the runtime support for UWOmp$_{pro}$ in two parts: (i) the translator has been written in Java [5] in the IMOP Compiler Framework [23] - approximately 8000 lines of code (ii) the runtime libraries are implemented in C [20] - approximately 2000 lines of code. IMOP is a source-to-source compiler framework for analyzing and compiling OpenMP programs. To compile the generated OpenMP codes, we used GCC with -O3 switch (includes tail-call optimization).

We evaluate our proposed translation scheme and the runtime using 14 benchmark kernels from various sources (details in Figure 14). These include all the kernels used by Aloor and Nandivada [4] (except FDTD-2D, which we could not compile/run using the baseline compiler of Aloor and Nandivada) and a few additional kernels: WF, Jacobi1D, Stencil4D, and HP. For each kernel, we indicate the type of synchronization needed and if it uses reduction operations. Note that point-to-point kernels, can also be written using all-to-all synchronization.

To demonstrate the versatility of our proposed techniques, we performed our evaluation on two systems: (i) Dell Precision 7920 server, a 2.3 GHz Intel system with 64 hardware threads, and 64 GB memory, referred to as *HW64*. (ii) HPE Apollo XL170rGen10 Server, a 2.5 GHz Intel 40-core system, and 192GB memory, referred to as *HW40*. All numbers reported in this section are obtained by taking a geometric mean over 10 runs. For each benchmark kernel we chose the largest input such that the 10 runs of the UWOmp$_{pro}$ kernel would complete within one hour on HW64. In this section, for a

language L_x, we use the phrase "performance of an L_x program" to mean the performance of the code generated by the compiler for L_x, for the program written in L_x.

We show our comparative evaluation across four dimensions: (i) UWOmp$_{pro}$ kernels that perform all-to-all synchronization with no reduction operations (kernels 1-7); we compare the performance of these UWOmp$_{pro}$ codes against their UWOmp++ counterparts. (ii) UWOmp$_{pro}$ kernels that perform only point-to-point synchronization, with no reduction (kernels 8-12); we compared their performance with that of their all-to-all versions written in UWOmp$_{pro}$ and standard OpenMP. Note: we could not successfully run the code generated by the UWOmp++ compiler for the all-to-all UWOmp++ versions of these codes and hence we do not show a comparison against these codes. (iii) UWOmp$_{pro}$ kernels performing reduction operations (kernels 13-14); we compare the performance of these kernels with their OpenMP original benchmarks. We first rewrote these kernels to use our reduction algorithm and compare them with their standard OpenMP benchmarks. (iv) Impact of the scheduling policy; we present a comparative behavior of all the kernels by varying the scheduling policy. We also found that the UWOmp$_{pro}$ codes scale well with the increasing number of threads. Due to lack of space, the details are made available in the extended report [1].

Evaluation of all-to-all synchronization. For the benchmark kernels 1-7, Figure 15 shows the percentage improvement of UWOmp$_{pro}$ codes over their UWOmp++ counterparts, for varying number of threads.

Our evaluation shows that except for KPDP in one particular configuration (64 cores on HW64 and 40 cores on HW40), the UWOmp$_{pro}$ codes perform better than their UWOmp++ counterparts. Even for that particular configuration the performance degradation is minimal (<7%). One common pattern we find is that if a kernel has a lot of computation (for example, LELCR, LCS and WF) UWOmp$_{pro}$ outperforms UWOmp++ significantly, in contrast to kernels with very low computation (for example, KPDP and MCM) where our comparative gains are less. Overall, we find that the percentage improvements varied between −4.0% to +98.1% on the HW64 system and between −6.6% to +89.5% on the HW40 system. We believe that such significant performance gains are mainly due to efficient handling of worklists (single local worklist vs two separate worklists in UWOmp++; see Section 6), and being conservative in converting only the essential parts of the code to CPS form.

Note: we avoid showing a comparison with the OpenMP counterparts of these benchmarks as Aloor and Nandivada [4] have already shown that UWOmp++ programs run faster than the plain OpenMP programs, and we show that UWOmp$_{pro}$ programs fare significantly better than their UWOmp++ counterparts.

Evaluation of point-to-point synchronization. For the benchmark kernels 8-12, Figure 16 summarizes the percentage improvement of the point-to-point variants of the codes compared to OpenMP, for varying number of threads, on both HW64 and HW40 systems. Figure 17 summarizes the percentage improvement of the point-to-point variants of the codes compared to the all-to-all UWOmp$_{pro}$ versions, for varying #threads. We see a significant performance improvement obtained when using point-to-point synchronization routines over that of OpenMP. The percentage improvement varied between 6.8% to 86.5% on HW64, and between 6.9% to 84.8% on HW40 when compared with OpenMP. The percentage improvement varied between 27.3% to 82.4% on HW64, and between 6.4% to 82.9% on the HW40 system when compared with the all-to-all versions of UWOmp$_{pro}$.

The main reason of this improvement is due to the lesser amount of communication (and faster execution) in point-to-point synchronization compared to all-to-all synchronization in OpenMP. For most of the kernels we see that the performance improvement reduces gradually with the increasing number of threads. This is mainly because the main overhead in all-to-all synchronization is the waiting time incurred by all the activities. As the number of threads increase, the overall waiting time gets amortized better and leads to a reduction in the overhead.

Evaluation of reduction kernels. For the benchmark kernels 13-14, Figure 19 shows the percentage improvement obtained using our proposed reduction scheme against the standard OpenMP benchmarks (using the OpenMP reduction clause wherever possible). We see that the proposed scheme performs significantly better. The percentage improvement varied between 26.1% to 52.4% on HW64, and between 31.4% to 82.7% on HW40, compared to OpenMP.

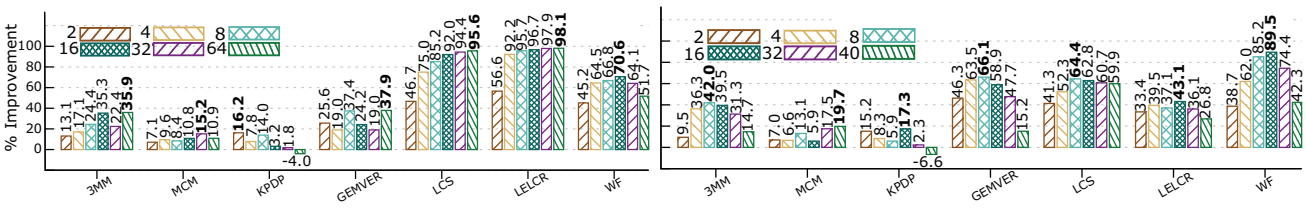
For reference, we also compared our generated codes using the techniques discussed in this paper (use parallel reduction operation) against that in which one of the activities X_1 performs the reduction operation in serial. We have found that the parallel reduction operation clearly outperforms (31% to 78%) the serial one; the graphs for the same have been moved to the extended report [1].

Evaluation of different schedules. We evaluate all kernels for different scheduling policies to demonstrate the importance of supporting diverse scheduling policies and the effectiveness of our dynamic and guided schedulers. Figure 18 shows the percentage improvement of dynamic and guided scheduling compared to static scheduling; due to lack of space, we show this evaluation only for a fixed number of threads (set to the maximum available hardware cores in the system). We see that for kernels that only have all-to-all synchronization, the static schedule performed much better; we believe this is mainly due to our proposed optimization for all-to-all synchronization in the context of static-scheduling; see Section 6. In the case of kernels with point-to-point synchronization, since the set of tasks waiting for each other was not predictable, the dynamic/guided scheme performed better.

Further, we observe that for IA and HP kernels, the gains due to dynamic and guided schedules is less. We believe that it is due to the presence of all-to-all reduction operations in those kernels that seem to work better with static scheduling. For most kernels that do not use reduction operations, we find that the dynamic and guided policies work better.

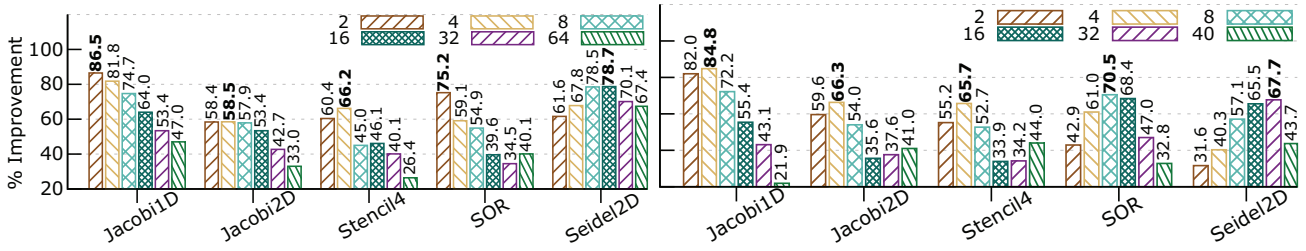
Overall, for dynamic scheduling, the percentage improvement varied between −45% to +32% on the HW64 system, and between −43% to +44% on the HW40 system. Similarly, for guided scheduling, the percentage improvement varied between −39% to +31% on the HW64 system, and between −47% to +30% on the HW40 system. Such significant variance clearly attests to the importance of supporting different scheduling policies and the efficacy of our implemented schedulers.

Note: We compared our kernels with the baseline OpenMP kernels with dynamic or guided scheduling. We observed that the



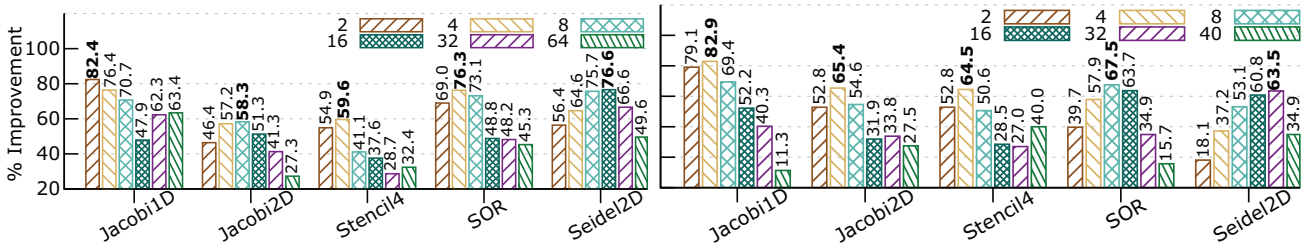
(a) HW64 System (b) HW40 System

Figure 15: Performance of UWOmp$_{pro}$ kernels with all-to-all synchronization (Vs. UWOmp++ kernels), for varying #threads.



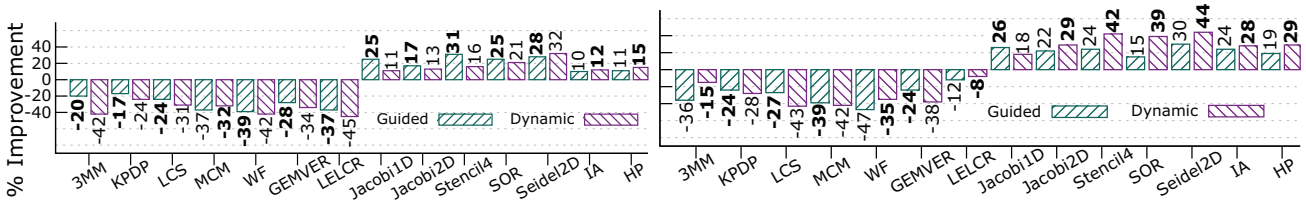
(a) HW64 System (b) HW40 System

Figure 16: Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. OpenMP), for varying #threads.



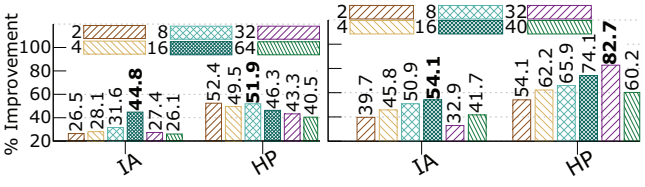
(a) HW64 System (b) HW40 System

Figure 17: Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. All2All), for varying #threads.



(a) HW64 System. #threads = 64. (b) HW40 System. #threads = 40.

Figure 18: Comparison of dynamic and guided scheduling over static scheduling; #threads set to maximum #cores.



(a) HW64 System (b) HW40 System

Figure 19: Performance of UWOmp$_{pro}$ kernels that use reduction (Vs. OpenMP), for varying #threads.

performance of our dynamic/guided scheduling scheme is comparable to that of the baseline (-2 to 2%); not much overhead. We skip the details due to lack of space.

8 Related Work

Aloor and Nandivada [2] proposed the novel idea of a unique worker model (UWOpenMP), which gives the programmer an impression that each iteration of a parallel-for-loop is executed by a unique thread (worker). Such a model allowed barrier statements to be inserted inside work-sharing constructs like parallel-for-loops. Aloor and Nandivada [4] work on UWOmp++ extended the above idea further and provided support for recursive functions (with barriers) to be invoked within parallel-for-loops. In contrast, UWOmp$_{pro}$ extends this idea further to allow point-to-point synchronization, reduction operations between the activities (iterations) of a parallel-for-loop and allow arbitrary scheduling policies of OpenMP.

OpenMP [25] supports a taskwait command with depend clause which can be used synchronize between the tasks; the dependencies are specified in terms of shared variables (for example, `depend (out:x)`, or `depend (in:x)`) and not individual tasks. In contrast, UWOmp$_{pro}$ supports point-to-point synchronization among the iterations (activities) of parallel-for-loops. Further, UWOmp$_{pro}$ supports reduction operations in the middle of the activities, such that the final reduced value is available immediately after the reduction operation (do not have to wait for the end of the parallel region).

There have been multiple efforts [15, 21, 28, 33] to utilize continuations to extend and translate parallel programs. [15] use the idea of continuations to explicitly maintain activation records for all the activities, and use these activation records at the time of pausing (store the activation records) and resuming (restore the activation record) the activities. Maintaining activation records for all the activities creates unnecessary memory overhead In contrast, our approach only saves the information that needs to be executed by each activity in its corresponding closure data structure; further, we reutilize memory in order to avoid unnecessary `malloc` calls. Fischer et al. [14] provide a modular approach to do a CPS translation of event-driven programs in Java. For the Cilk language, Blumofe et al. [6] propose a C-based runtime with a work-stealing scheduler useful for multithreaded programming, which uses continuations to spawn and join tasks. Our approach takes advantage of CPS to efficiently perform wait and continue operations, and supports different scheduling policies, along with reduction operations.

White [34] describes an implementation for OpenMP-tasks (created using `#pragma omp task` directive) using continuations. UWOmp$_{pro}$ uses continuations to efficiently handle activities in parallel-for-loops, which may contain synchronization points (point-to-point or all-to-all) even within recursive functions.

For HJ, Imam and Sarkar [17] propose the idea of one-shot delimited continuations (OSDeCont) to support cooperative scheduling and event-driven controls. One main restriction in their approach is that it works only for help-first and work-first approaches of work-stealing. We take inspiration from their approach, but generalize the techniques so that we are not limited to specific scheduling policies and our scheme works in the context of OpenMP parallel-for-loops.

9 Conclusion

In this paper, we present UWOmp$_{pro}$ that allows point-to-point synchronizations and reduction operations, among the activities of parallel-for-loops of OpenMP. We present a scheme to compile UWOmp$_{pro}$ codes to efficient OpenMP code. We have also designed a runtime, based on a novel postbox based communication subsystem to support efficient signal and wait functions, along with reduction operations and arbitrary schedules of OpenMP. We have implemented our scheme in the IMOP compiler framework and performed a thorough evaluation. We argue that programmers can write expressive and performant codes using UWOmp$_{pro}$.

A Artifact Description: UWOmp$_{pro}$: UWOmp++ with Point-to-Point Synchronization, Reduction and Schedules

A.1 Abstract

This artifact contains the description to write a UWOmp$_{pro}$ program and how to invoke the translator to convert it to the final machine code. It also includes a script to compile and run the sample benchmarks.

A.2 Description

The UWOmp$_{pro}$ compiler takes as input a C program. It preprocesses the C code and inputs the preprocessed code to the IMOP Translator (written in Java) that converts the given code to CPS form. Finally, the converted CPS code is compiled using any C compiler (with OpenMP support) that generates the final machine code.

A.2.1 Check-list (artifact meta information)

- **Program: C**
- **Compilation: javac (1.7 or higher) for the translator and gcc (any version) for the compiler**
- **Hardware: Any hardware**
- **Output: Code Output**
- **Publicly available?: Yes**

A.2.2 How Software can be Obtained? UWOmp$_{pro}$ source code and supporting script files are hosted on GitHub. (https://github.com/adityaag24/artifact-pact)

A.2.3 Hardware dependencies. The source code should run on any general purpose computer that supports GCC, Java and OpenMP.

A.2.4 Software dependencies. To run the artifacts successfully, one would require the latest Java JDK (version 17 or higher), gcc, ant (for building the translator). Steps to install the same are given below:

- JDK: sudo apt-get install openjdk-17-jdk openjdk-17-jre
- GCC: sudo apt-get install build-essential
- ANT: sudo apt-get install ant

The following steps work on a Linux Based System(tested on Ubuntu 22.04). For mac, we can use homebrew to install the same.

- JDK: brew install openjdk@17
- GCC: brew install gcc
- ANT: brew install ant

A.3 Installation

The translator and instructions to use it can be downloaded from the github repository.

A.4 Experiment workflow

Clone the source code to local machine and compile: This should clone the repository to the local machine and build the translator.

- Clone the Repository: **git clone https://github.com/adityaag24/artifact-pact.git**
- Enter the folder: **cd artifact-pact**
- Clean any existing build files: **ant clean**
- Build the Java Translator: **ant -v**

After this, to translate and compile any <file>.c file,

- Preprocess the file using a C Compiler: **gcc -P -E -o <file>.i <file>.c**

- The following command invokes the IMOP translator to translate the code: **java -da -Xms2048M -Xmx4096M -cp third-party-tools/ com.microsoft.z3.jar:. imop.Translator -nru -f <file>.i**
- Compile the translated benchmark using any OpenMP C Compiler: **gcc -fopenmp -O3 -o <file> output-dump/<file>-processed.i**
- Run the translated benchmark: **./<file> [Optional Arguments]**

A.5 Evaluation and expected result

After running the translator, the translated output will be present in a output-dump folder created by the IMOP translator. The filename will contain a postfix -processed denoting the final processed output.

References

[1] Aditya Agrawal and V. Krishna Nandivada. 2023. Extended Report on UWOmp$_{pro}$. https://bit.ly/3DWiNoN

[2] Raghesh Aloor and V. Krishna Nandivada. 2015. Unique Worker Model for OpenMP. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/2751205.2751238

[3] R. Aloor and V. K. Nandivada. 2019. Applicability of UWOmp++ and Reference Codes. Supplementary Material. http://www.cse.iitm.ac.in/~krishna/uwompp-master.zip.

[4] Raghesh Aloor and V. Krishna Nandivada. 2019. Efficiency and Expressiveness in UW-OpenMP. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 182–192. https://doi.org/10.1145/3302516.3307360

[5] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language*. Addison Wesley Professional.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

[7] John Burkardt. 2020. Heated Plate. https://people.sc.fsu.edu/~jburkardt/c_src/heated_plate_openmp/heated_plate_openmp.html

[8] John Burkardt. 2020. SOR. https://people.sc.fsu.edu/~jburkardt/cpp_src/sor/sor.html

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[10] Bradford L. Chamberlain, Sung eun Choi, Steven J. Deitz, and Lawrence Snyder. 2004. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*. IEEE, 66–75.

[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

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

[14] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. 2007. Tasks: Language Support for Event-Driven Programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Nice, France) *(PEPM '07)*. Association for Computing Machinery, New York, NY, USA, 134–143. https://doi.org/10.1145/1244381.1244403

[15] Dennis Gannon, Vincent A. Guarna, and Jenq Kuen Lee. 1990. Static Analysis and Runtime Support for Parallel Execution of C. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing* (Urbana, Illinois, USA). Pitman Publishing, Inc., USA, 254–274.

[16] S Gupta and V K Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *JPDC* 75 (2015), 1–19.

[17] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP*. 618–643.

[18] Cray Inc. 2013. *The Chapel Language Specification*. Technical Report. http://chapel.cray.com.

[19] Andrew Kennedy. 2007. Compiling with Continuations, Continued. *SIGPLAN Not.* 42, 9 (oct 2007), 177–190. https://doi.org/10.1145/1291220.1291179

[20] Brian W Kernighan and Dennis M Ritchie. 2006. *The C programming language*.

[21] Olin Shivers Mit and Olin Shivers. 1997. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *In Proceedings of the Second ACM SIGPLAN Workshop on Continuations*. ACM Press, 2–1.

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

[23] Krishna Nandivada and Aman Nougrahiya. 2019. IMOP. http://cse.iitm.ac.in/~amannoug/imop

[24] Thao Nguyen. 2015. LCS In Parallel. https://github.com/taoito/lcs-parallel

[25] OpenMP Architecture Review Board. 2020. OpenMP Application Programming Interface Version 5.2. https://www.openmp.org/specifications/.

[26] Louis-Noël Pouchet. 2010. PolyBench/C Suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[27] Rolf Rabenseifner and Jesper Larsson Träff. 2004. More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–46.

[28] John H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press, USA.

[29] Jun Shirako, Kamal Sharma, and Vivek Sarkar. 2011. Unifying Barrier and Point-to-Point Synchronization in OpenMP with Phasers. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era* (Chicago, IL) *(IWOMP'11)*. Springer-Verlag, Berlin, Heidelberg, 122–137.

[30] Jun Shirako, Priya Unnikrishnan, Sanjay Chatterjee, Kelvin Li, and Vivek Sarkar. 2013. Expressing DOACROSS Loop Dependences in OpenMP. In *OpenMP in the Era of Low Power Devices and Accelerators*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 30–44.

[31] Priya Unnikrishnan, Jun Shirako, Kit Barton, Sanjay Chatterjee, Raul Silvera, and Vivek Sarkar. 2012. A Practical Approach to DOACROSS Parallelization. In *Euro-Par 2012 Parallel Processing*, Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 219–231.

[32] Hasitha Waidyasooriya and Masanori Hariyama. 2019. Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability. *IEEE Access* 7 (04 2019), 53188–53201. https://doi.org/10.1109/ACCESS.2019.2910824

[33] Mitchell Wand. 1980. Continuation-Based Multiprocessing. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming* (Stanford University, California, USA) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/800087.802786

[34] L. White. 2014. *Extending old languages for new architectures*. Ph. D. Dissertation. University of Cambridge, UK.