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

Anonymous Author(s)

Abstract

An OpenMP program internally creates a team of threads, which share a given set of *activities* (for example, iterations of a parallel-for-loop). OpenMP allows synchronization among these threads. However, many classes of computations can be conveniently expressed by using synchronization among the parallel activities. Currently, OpenMP restricts any form of barriers among the parallel activities; otherwise, the behavior of the program can be unpredictable. While extensions like UWOmp++ (and its precursor UW-OpenMP) support all-to-all barriers among the activities, currently there is no support for efficiently performing point-to-point synchronization among them. In this paper, we present UWOmp$_{pro}$ as an extension to UWOmp++ to address these challenges to realize more expressive and efficient codes.

UWOmp$_{pro}$ allows point-to-point synchronizations among the activities of a parallel-for-loop and support reduction operations. We present a translation scheme to compile UWOmp$_{pro}$ code to efficient OpenMP code, such that the translated code does not invoke any synchronization operations 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 and wait operations, along with reduction operations and arbitrary schedules. 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

OpenMP is a widely used parallel-programming API, where a programmer can insert compiler directives and utilize runtime routines to enable parallelism in sequential code. OpenMP uses the efficient 'team of workers' model, where each worker (also interchangeably referred to as thread) is given a chunk of activities 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 different dependent 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 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 1a shows the classical 1D Jacobian kernel written in OpenMP. Note that OpenMP does not allow barriers inside work-sharing constructs (like parallel-for-loops), as the behavior of such programs can be unpredictable (may lead to incorrect output, correct output, or deadlock) [1]. Figure 1b shows an equivalent code written in UWOmp++ [2]. UWOmp++ supports the unique worker 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 have shown that such UWOmp++ versions are efficient and arguably more expressive compared to the OpenMP version. However, such codes still suffer from multiple drawbacks.

In the code shown in Figure 1b, each activity $X_{\mathtt{i}}$ (corresponding to iteration `i`) is waiting for all the remaining activities instead of only the 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 activity 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 [9], HJ [8], and so on, using explicit synchronization objects (like Clocks [9] and Phasers [8]). In the context of OpenMP, Shirako et al. [27] 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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
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 here
   #pragma omp single
      {x=A; A=B; B=x; t++;}
   /* implicit barrier here */ }}
```

(a) OpenMP Version

```
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
   } } }
```

(b) UW-OpenMP Version

```
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);
    } } }
```

(c) 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.

among the registered activities. However, their design has multiple restrictions (i) the synchronization can only be in one direction, that is, from left to right – can be limiting in terms of expressiveness; (ii) threads (not activities) block on wait operations – can limit parallelism and impact performance negatively; (iii) their scheme cannot work with dynamic/guided scheduling; and (iv) the activities do not have a way to perform reduction operations at the synchronization points.

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 1c shows a UWOmp$_{pro}$ version of the kernel shown in Figure 1b. Here, the first all-to-all barrier of Figure 1b 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), 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 all-to-all barrier of Figure 1b has been replaced with a command `signalAll`, followed by a `wait` command. 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 which require the creation, registration, and management of such synchronization objects by the individual activities, in UWOmp$_{pro}$ activities of a parallel-for-loop can synchronize between them without any need for the programmer to explicitly create (or pay the overheads of) synchronization objects. Unlike X10 and HJ, we also allow fine grain signalling operations (where the signal can be sent to specific iterations and not to a phaser object which can be accessed by any activity), along with the support of efficient reduction operations during the wait operation. Further, our design ensures that we continue to take advantage of the efficient 'team of workers' model of OpenMP to derive high performance.

To realize an efficient implementation of the point-to-point synchronization, we build a novel *postbox* based communication sub-system where activities can signal and exchange messages with each other. We present three variations of the *postbox* system, such that a compiler using static analysis, can use the appropriate type of the postbox, depending on the type of messages exchanged between the synchronizing activities. To support fast reduction operations, we propose two reduction algorithms termed *eager* and *lazy*, which support efficient reduction operations among a subset of activities and all activities, respectively.

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 synchronizations and reduction operations, between the activities of parallel-for-loop. In contrast to UWOmp++, UWOmp$_{pro}$ supports all the scheduling policies defined in OpenMP.
- We present a translation scheme to compile UWOmp$_{pro}$ code to efficient OpenMP code by using an IR that takes advantage of continuation-passing-style (CPS) to efficiently realize wait and continue operations.
- We present a runtime based on a novel communication subsystem using postboxes, to support efficient signal and wait functions, along with reduction operations and arbitrary schedules.
- 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.

commands	target activities	reduction?
signal, wait	1 or more	×
signalAll, waitAll	all	×
signalSend, waitRed	1 or more	✓
signalAllSend waitAllRed	all	✓

Figure 2. List of commands supported by UWOmp$_{pro}$.

2 Background

We now present some brief background needed for this paper.

OpenMP. We now briefly describe some OpenMP constructs; interested readers may see the OpenMP manual [11].

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

Parallel-For-Loop: A sequential for-loop can be annotated using `#pragma omp for nowait schedOpt` to distribute the iterations among the team of threads. The scheduling policy (static (default), dynamic, guided, or runtime) is mentioned using `schedOpt`. In the absence of the `nowait` clause, an implicit barrier is assumed after the for-loop.

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 [2].

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. An OpenMP 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 among the synchronizing activities) 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$_{pro}$.

3.1 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$_{pro}$, 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. As discussed in Section 1, Figure 1c shows the

```
#pragma omp parallel
{
 #pragma omp for
 for(i=1;i<N;i++) {
  while(d <= epsilon) {
  B[i]=(A[i-1]+A[i+1])*0.5;
  diff[i] = abs(A[i]-B[i]);
   #pragma omp barrier
   if(i==1){
    d=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(d <= epsilon) {
   B[i]=(A[i-1]+A[i+1])*0.5;
   diff[i] = abs(A[i]-B[i]);
   signalAllSend(1,diff[i]);
   waitAllRed(1,d,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.

UWOmp$_{pro}$ code (using point-to-point synchronization) to perform the Jacobian 1D stencil computation shown in Figure 1b. Note: a signal/wait command to/on a non-existing iteration is treated as *nop*s. That way, the programmers can simply write code of the form `#ompfor for (i=0;i<n;++i) {signal (i+1); wait (i-1);}` without having to worry about the boundary cases.

3.2 Reduction

Consider the example code snippet shown in Fig. 3a to perform iterated averaging on an array of N elements, 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 waits for X_1 to sequentially reduce the array `diff` to the shared variable `d`, which is used for checking 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 the code snippet Fig. 3b, after computing `diff[i]`, each activity X_i sends a signal to all the other activities with its value of `diff[i]`. Then, the code invokes a blocking reduction operation, specifying the variable (`d`) to hold the reduced value, and the reduction operation (`ADD`). Compared to the UW-OpenMP version, where a single activity performed the reduction operation in linear time, in the UWOmp$_{pro}$ version, all the threads together perform the reduction operation in parallel. Note that in contrast to OpenMP, for ease of readability, we use verbose reduction operator names (for example, ADD in place of '+').

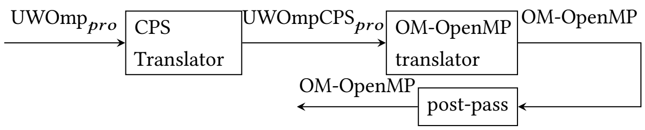


Figure 4. Block diagram of our proposed translation scheme.

3.3 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 that we use to convert a given input UWOmp$_{pro}$ code to efficient OM-OpenMP code. The main idea behind our translation is that in the generated OM-OpenMP code, the iterations of the for-loop (or the activities) 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. We now describe the different important modules of our translation scheme.

4.1 CPS Translator

Our translation scheme is inspired by that of Aloor and Nandivada [2], who translate an input program to an IR (called UWOmpCPS), before lowering it to OM-OpenMP. UWOmpCPS, which is an extension to CPS (Continuation Passing Style [18]); 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 observations, we first provide a modified version of the UWOmpCPS grammar and then discuss the changes to the translation rules of Aloor and Nandivada.

4.1.1 UWOmpCPS$_{pro}$: Modified CPS IR. We only CPS-transform the body of the parallel-for-loops (unlike UWOmpCPS). Thus, not all functions need to be in CPS form and incur the penalties thereof.

We use a modified IR called UWOmpCPS$_{pro}$; grammar shown in Figure 5. Some of the main differences between UWOmpCPS and UWOmpCPS$_{pro}$ are as follows: (i) A program may consist of both CPS and non-CPS functions.

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

Figure 5. Grammar for UWOmpCPS$_{pro}$

1.	⟦ **#ompparallel** `{ S }` ⟧	⇒	**#ompparallel** `{` ⟦ `S` ⟧ `}`
2.	⟦**#ompfor** `for(Header){ fun(args); } S` ⟧	⇒	**#ompfor** `for(Header){` $K = mkClsr(id, null, null)$; `funCPS(`$K$`, args)` `}` ⟦ `S` ⟧
3.	⟦ `S` ⟧ `//S contains no //par-for-loops`	⇒	`S`

Figure 6. CPS translation rules for the parallel constructs.

(ii) A `CPSParRegion` may only contain set of parallel-for-loops in CPS form (`CPSParForLoop`) or barriers. (iii) A `CPSParForLoop` can specify a schedule and related options (represented as `schedOpt`). (iv) As is standard in CPS translation, the continuation object is passed as an additional argument to each CPS function call (`CPSFunCall`). We pass this continuation as the first argument. Note that `Stmt` denotes any sequential statement, `FuncDecl` is any regular C function declaration, `FunCall` is any regular non-CPS function call statement; we skip the expansion of these non-terminals for brevity.

4.1.2 Generation of code in CPS form. The rules of our CPS-translator which translates the input UWOmp$_{pro}$ programs to UWOmpCPS$_{pro}$, are similar to that of Aloor and Nandivada [2], except for the rules shown in Figure 6. 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 right-hand side (Y) may contain further terms with ⟦ ⟧ indicating that those terms need to be further transformed. In the rules, we use `#ompparallel` as a shortcut for `#pragma omp parallel`, and `#ompfor` as a shortcut for `#pragma omp for nowait schedOpt`. Our CPS transformation starts by transforming the parallel-region (Rule 1).

Without any loss of generality and for the ease of translation, the rule to translate a parallel-for-loop (Rule 2)

```
#ompfor
for(Header){
 K=mkClsr(X);
 fCPS(K,args);
}
```

⇒

```
1  tid=thread-number();
2  sched=getSchedule(schedOpt);
3  chSize=getChunkSize(schedOpt);
4  if(sched is static){
5   scheduler=&scheduler-static;
6   WL[tid]=new WL(); //local Q
7  }else if(sched is dynamic){
8   scheduler=&scheduler-dynamic;
9   WL[tid]=globalWL; // global Q
10 }else{ // guided
11  scheduler=&scheduler-guided;
12  WL[tid]=globalWL;/*global Q*/}
13 #ompfor
14 for(Header) {
15   K=mkClsr(X);
16   C=mkClsr(fCPS,bEnv(args),K);
17   enqueue(WL[tid],C);}
18 (*scheduler)(chSize);
```

Figure 7. UWOmpCPS$_{pro}$ to OM-OpenMP Translation.

assumes that the body of the loop is a single function call. This can be easily obtained by applying a simplification step, like the ones used by Aloor and Nandivada [2]. Rule 2 has two substeps: (i) the function that is called in the body of the parallel-for-loop (`fun`) is translated to CPS form (using the standard CPS transformation rules, discussed by Aloor and Nandivada [2, Rules 2-10, Figure 4]), by passing the identify 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 a call to its CPS counterpart which passes the continuation as an additional argument.

During the translation of a parallel region, if any statement is encountered which contains no parallel-for-loops, we emit the statement as it is (Rule 3).

4.2 OM-OpenMP Translator

We now discuss how we translate code in UWOmpCPS$_{pro}$ format to OM-OpenMP code. The goal of the translation is to ensure 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 policy, all the closures are pushed to a global 'work queue'. Each thread takes some number of closures from the queue and executes the same. Figure 7 shows the rule to translate the parallel-for-loop.

We first emit code to identify the specified schedule (returned by `getSchedule`, which in turn derives it from the `schedOpt` string). Accordingly, we store the function pointer of the corresponding scheduler function (defined in Section 5.5) and remember the worklist to be used by each thread ($WL[tid]$). We emit a parallel-for-loop that pushes the closure for each activity to $WL[tid]$ (Lines 13-17). Finally, we invoke the appropriate scheduler (Line 18).

Note: if `schedOpt` is not set, then `getSchedule` sets the schedule to `static`. Similarly, if `schedOpt` is set to *runtime*, then `getSchedule` will obtain the schedule from the language-specified environment variable.

4.3 Post-Pass: Type Specific Reduction Operations

The final step in our translation process is to introduce type specific reduction operations for the reduction operations specified in the OpenMP specification [11]. As mentioned in Section 3.2, the reduction-related wait commands in the input UWOmp$_{pro}$ code take a reduction operation and a reduction variable *redVar* (which stores the reduced result), as additional arguments to the wait command. The compiler uses the declared type of *redVar* (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 of the primitive types T, our runtime provides functions for performing the copy operation (for example, '`COPYint (int *from, int *to)`') and the corresponding function 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`).

4.4 Example translation

For a better understanding of our translation scheme, in Figure 8, we describe the steps in transforming a sample UWOmp$_{pro}$ code to OM-OpenMP code. We now discuss the salient features in our translation. Figure 8a shows the input UWOmp$_{pro}$ code and Figure 8b shows the CPS transformed version. The standard set of CPS transformation rules are applied to the function `f` to convert it to `fCPS`, and generate other CPS functions (`pCPS1`, `pCPS2` and `pCPS3`). 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 the execution of the parallel-for-loop. It calls `fCPS` with closure `K` as an argument.

Figure 8c shows the OM-OpenMP translated code of the UWOmpCPS$_{pro}$ code. This step emits code to identify the appropriate scheduler (Lines 1-12 from Figure 7) and wraps the call to function `fCPS` inside the closure `C` before enqueuing the closure in the appropriate worklist $WL[tid]$.

```
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);
if(sched is static){
 scheduler=&scheduler-static;
 WL[tid]=new WL();
}else if(sched is dynamic){
 scheduler=&scheduler-dynamic;
 WL[tid]=globalWL;
}else{ // guided
 scheduler=&scheduler-guided;
 WL[tid]=globalWL; }
 #ompfor
  for(Header){
   K=mkClsr(id,null,null);
   C=mkClsr(fCPS,bEnv(args),K);
   enqueue(WL[tid],C); }
  (*scheduler)(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 8. Example Transformations.

Finally, Figure 8d 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 arbitrary schedules. Our support for these operations is based on our novel design of the communication sub-system between the activities of a parallel-for-loop. We will first describe that and then follow it up with a discussion on the runtime support required for implementing the key operations.

5.1 Shared Postbox System for Communication

We present a postbox based system for communication among 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. 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 (say k) 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 hash-map (of initial-size 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 reduce the overheads of this hash-map based postbox to a large extent for the common case of all-to-all communication.

The exact configuration of the slots of each postbox entry depend 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 any data-messages, then we simply represent each node in the hashmap using a counter (*sCounter*). When an activity wants to send a signal, we simply increment the corresponding counter. and the receiving activity atomically decrements it, if it is non-zero. Else, it returns -1 (indicating 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 node as a queue of data-messages. When an activity wants to send a data-message, we append the message to the appropriate queue and the receiving activity dequeues a message if the queue is non-empty. Else, 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 node as a queue, where each element of the queue is of the form shown in Figure 9. When an activity has to send a data-message (Figure 10), we create a new node (of type `QueueNode`), add the message to the node, and enqueue the node to the appropriate queue. When an activity has to send a signal, we first check if the queue is empty or the last node in the queue has a data-message (*message* field ≠ `NULL`). If so, we create a new node of type `QueueNode` with empty-message, initialize *sCounter* to 1, and enqueue the node to the appropriate queue. Else, we simply increment the *sCounter* field of the last node in the queue.

```
struct QueueNode{
 int sCounter; void * message; struct QueueNode* next; };
```

Figure 9. mixed-mode postbox: structure of the queue node.

```
void sendMsg(j, i, m, ctr) // send message m from X_j to X_i
begin
    Acquire Lock on P_i.C_j;
    Q = P_i.C_j.queue;
    if m ≠ NULL then                    // data-message
        n = create-qnode(m, ctr); Enqueue n to Q;
    else
        if Q is empty OR Q.lastNode().message ≠ NULL then
            n = create-qnode(NULL, ctr); Enqueue n to Q;
        else Q.lastNode().sCounter+ = ctr ;
    Release Lock P_i.C_j;
```

Figure 10. mixed mode: send message.

```
1  QueueNode* recvMsg(i, j, tp) // X_i receives a message of
   type tp from X_j; tp ∈ {signal, data-message}
2  begin
3      QueueNode* n = NULL;
4      Acquire Lock on P_i.C_j;
5      if P_i.C_j.queue not empty then
6          n = Peek P_i.C_j.queue;
7          if ((tp == data-message AND n.message == NULL)
           OR (tp == signal AND n.message ≠ NULL)) then
8              return NULL;   // message not available
9          if n.message == NULL then        // get signal
10             n.sCounter −−;
11         if n.signalCtr==0 then  n=Dequeue P_i.C_j.queue ;
12     Release Lock P_i.C_j;
13     return n;
```

Figure 11. mixed mode: Receive message, if available.

Figure 11 show the pseudo-code for receiving a message. An activity wanting to receive a message (signal or data-message), first checks if the queue is empty. If so, we return NULL. Else, we peek the first node (n) of the queue and check that the type of the requested message (signal or data-message) matches the type of n (Line 7). If not, we return NULL. Else, if the *message* field of n is NULL (indicating a signal type node), we decrement the counter. Then, if the value of *sCounter* is 0 (could be a data-message or the last signal), we dequeue the message and return it.

Note: (I) It is not the responsibility of the postbox to wait if the appropriate signal is not received. 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 details of the wrapper methods that are emitted by the CPS transformation (Section 4.1) to handle the signal commands: The CPS transformation rules

```
void signalCPS(K, e, ...)
begin
    if e == true then
        p=Iteration list from the variable list of arguments;
        from=K.iteration;
        for each i ∈ p do  sendMsg(from, i, NULL, 1) ;
    Invoke the continuation in K;
```

Figure 12. Signal a subset of activities

```
void signalAllCPS(K, e)
begin
    if e == true then
        s=getStartRangeOfParallelForLoop();
        e=getEndRangeOfParallelForLoop();
        N=e-s+1;
        from=K.iteration;
        sendMsg(from, from, NULL, N);
Invoke continuation in K;
```

Figure 13. Signal all activities

```
void signalSendCPS(K, e, m, ...)
begin
    if e == true then
        p=Iteration list from the variable list of arguments;
        from=K.iteration;
        for each i ∈ p do  sendMsg(from, i, m, 0) ;
    Invoke the continuation in K;
```

Figure 14. Signal a set of activities with message m.

may emit any of the four signalling wrapper `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` (Figure. 14) and `signalAllSendCPS` (Figure. 15) methods. 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 invoking `sendMsg` for 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.

5.3 Wait Algorithm

We now describe the details of the wrapper methods that are emitted by the CPS transformation (Section 4.1) to handle the wait commands: `waitCPS`, `waitRedCPS`, `waitAllCPS` and `waitAllRedCPS`. The first two methods take as arguments the list of activities from whom the signal/message is to

```
void signalAllSendCPS(K, e, m)
begin
    if e == true then
        s=getStartRangeOfParallelForLoop();
        e=getEndRangeOfParallelForLoop();
        N=e-s+1;
        sendMsg(from, from, m, N);
        msgArr[from]=m;
    Invoke continuation in K;
```

Figure 15. Signal all activities with message *m*.

```
void waitCPS(K, e, ...)
begin
    if e==true then
        p=Iteration list from the variable list of arguments;
        iter=K.iteration;
        struct QueueNode *qNode;
        for every iteration i in p do
            qNode=recvMsg(iter,i,signal);
            if qNode==NULL then
                qNode=recvMsg(i,i,typeMsg);
                if qNode==NULL then
                    wClsr=mkWaitClsr(K,iVal,rop′,redVar,copy,p);
                    wClsr.start=i;
                    acquireLock(lock);
                    enqueueClosure(WL[tid], wClsr);
                    releaseLock(lock); return;
    Invoke continuation in K
```

Figure 16. Wait for a subset of activities without reduction

```
void waitAllCPS(K, e)
begin
    if e==true then
        s=getStartRangeOfParallelForLoop();
        e=getEndRangeOfParallelForLoop();
        iter=K.iteration;
        for every iteration i in [s,e] do
            qNode=recvMsg(i,i,signal);
            if qNode==NULL then
                qNode=recvMsg(iter,i,signal);
                if qNode==NULL then
                    wClsr=mkWaitClsr(K,iVal,rop′,redVar,copy,p);
                    wClsr.start=i;
                    acquireLock(lock);
                    enqueueClosure(WL[tid], wClsr);
                    releaseLock(lock); return;
    Invoke continuation in K.
```

Figure 17. Wait for all activities without reduction

```
void waitRedCPS(K, e, rop′, redVar, copy, ...)
begin
    if e==true then
        p=Iteration list from the variable list of arguments;
        iter=K.iteration;
        if rop′==NULL then typeMsg=signal ;
        else typeMsg=data-message ;
        struct QueueNode *qNode;
        for every iteration i in p do
            qNode=recvMsg(iter, i, typeMsg);
            if qNode==NULL then
                qNode=recvMsg(i,i,typeMsg);
                if qNode==NULL then
                    wClsr=mkWaitClsr(K,iVal,rop′,redVar,copy,p);
                    wClsr.start=i;
                    acquireLock(lock);
                    enqueueClosure(WL[tid], wClsr);
                    releaseLock(lock); return;
            if typeMsg==data-message then
                Perform the reduction operation rop′ on
                  the message qNode.message and
                  reduction variable redVar;
    Invoke the continuation in K
```

Figure 18. Wait for a subset of activities with reduction

```
void waitAllRedCPS(K, e, rop′, redVar, copy)
begin
    if e==true then
        s=getStartRangeOfParallelForLoop();
        e=getEndRangeOfParallelForLoop();
        iter=K.iteration;
        if rop′==NULL then typeMsg=signal ;
        else typeMsg=data-message ;
        struct QueueNode *qNode;
        for every iteration i in p do
            qNode=recvMsg(i, i, typeMsg);
            if qNode==NULL then
                qNode=recvMsg(iter,i,typeMsg);
                if qNode==NULL then
                    wClsr=mkWaitClsr(K,iVal,rop′,redVar,copy,p);
                    wClsr.start=i;
                    acquireLock(lock);
                    enqueueClosure(WL[tid], wClsr);
                    releaseLock(lock); return;
        if typeMsg==data-message then
            lazyReduce(K);
        else
            Invoke Continuation in K
```

Figure 19. Wait for all activities with reduction

be received; we call these activities the target activities. The wrapper methods `waitCPS` and `waitAllCPS` simply call `waitRedCPS` and `waitAllRedCPS`, respectively, by passing the reduction specific arguments as NULL. We now describe the `waitRedCPS` and `waitAllRedCPS` methods. Similar to the signal wrapper methods, these methods are also in CPS form.

The method `waitRedCPS` (Figure. 18) first checks if the conditional-expression *e* is true. If so, it invokes the `recvMsg` function for each target activity (Line 10). In case any of the expected signal is not yet available, the function `recvMsg` returns NULL. Note that a thread executing an activity should not block, if there are other ready activities to be executed by that thread. Hence to ensure that the thread executing the wait-wrapper function does not block (or busy wait), we create a wait closure (using the macro `mkWaitClsr` by passing all the relevant information). The starting iteration number is saved indicating the information regarding already processed signals. This closure is marked to be executed later (Line 18). We execute the closure K, only if *e* is false or all the signals have been received. The `waitAllRedCPS` method works similarly, by considering the messages stored by all the iterations of the parallel-for-loop.

5.4 Reduction Operations

We now describe how we perform reduction operations in UWOmp$_{pro}$. Figure 18 shows a simple eager way of reduction which invokes the reduction operation (*rop′*) eagerly, on the value stored in $redVar$ and the message from the sender activity $qNode.message$(Line 19), as and when the message is read. By the time all the messages have been read, the final reduced value is available in the reduction variable $redVar$.

One main drawback of the eager method of reduction is that it is inherently serial in nature; hence it may take up to $O(N)$ steps for reduction, where N is the number of activities participating in reduction. While for small values of N this cost may be minimal, it can be prohibitively high, for large values of N; 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 propose an alternative reduction algorithm (termed lazy reduction) that collects the received messages and efficiently performs the reduction after all the messages have been received. We now discuss the changes to the `signalAllSendCPS` and `waitAllRedCPS` methods for realizing this lazy reduction algorithm.

5.4.1 Lazy reduction algorithm.

We maintain a global array called *msgArr* (for each parallel-for-loop), such that for iteration i, $msgArr[i]$ contains the message that has been sent by iteration i. The size of *msgArr* is set to N.

Changes to the `signalAllSendCPS` *method.* To enable lazy reduction algorithm, we make a minor change to the `signalAllSendCPS` function, just before invoking the

```
void scheduler-static(chSize)
begin // chSize unused; work already divided.
    executeWL(WL[tid]);
```

Figure 20. UWOmp$_{pro}$ static scheduling algorithm.

continuation: if $m \neq NULL$, we remember the message being sent by storing it in the message array ($msgArr[from] = m$).

Changes to the `waitAllRedCPS` *function.* To enable lazy reduction algorithm, we make two minor changes to the `waitAllRedCPS` function. (i) We remove the code that performs eager reduction (Lines 19-20). (ii) After the for-loop at Line 10, we invoke a special method `lazyReduce`, if *typeMsg == data-message*. This method reduces the array of messages *msgArr* to a single result. The algorithm works on the principle of the popular exchange based protocol [24] where an activity X_i exchanges messages with a neighbour X_{nb}. We utilize the point-to-point synchronization routines (discussed earlier in this paper) to synchronize with X_{nb}.

5.5 Supporting Different Scheduling Policies

UWOmp++ [2] 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.

As discussed in the Section 4.2, depending on the scheduling policy specified by the programmer (static / dynamic / guided), the translated code invokes the appropriate scheduler. Recall that the we support runtime-scheduling by emitting additional code immediately before each parallel-for-loop. We now discuss the details of the three schedulers.

static scheduler. The scheduler function `scheduler-static` (Figure 20) simply executes all the closures present in $WL[tid]$. We skip the definition of executeWL for brevity. If we are using static 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. As discussed in Section 4.2, for dynamic (and guided) scheduling we use the the global worklist. In the dynamic-scheduler function (Figure 21), each thread atomically dequeues (at most) *chSize* number of closures from the worklist and executes them.

guided-scheduler. Our `scheduler-guided` function works similar to the dynamic scheduling function, except that the chunk-size *chSize* is updated after each atomic dequeue. We skip the code for the same, for brevity.

6 Discussion

We now present a brief discussion on some of the salient features of our proposed extension.

• **Memory management.** Considering the overheads of CPS translation, we reuse many of the well known optimizations [2, 3, 26] to optimize the memory usage and malloc/free calls for closures. For simplicity, we also do not show the places where the closures are freed, though

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

Figure 21. UWOmp$_{pro}$ dynamic scheduler algorithm.

we do free them at the appropriate places in the actual implementation.

• **Optimization for Static Scheduling**: To minimize the memory and maintainance overhead of postbox in all-to-all based synchronization kernels and static scheduling policy, we use an optimized approach. For static scheduling policy, the work per thread is fixed apriori and thereby, the worklist is implemented as a single array of closures with two pointers, (*left* and *right*) per thread. In it, each thread executes the set of closures from left to right. When hit with a barrier, the thread only resumes executing the continuation once it finishes executing all the other activities in it's worklist (left != right) and then waits for other threads to finish executing their work (left == right).

• **Commands vs Pragmas.** Our current extension uses explicit commands for signal/wait, and so on. In future, for a release version of OpenMP, these can be specified using pragmas, to be consistent with the OpenMP philosophy.

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 [4] in the IMOP Compiler Framework [21] - approximately 8000 lines of code (ii) the runtime libraries are implemented in C [19] - 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 twelve benchmark kernels from various sources (details in Figure. 22). For each kernel, we indicate the type of synchronization needed and if it uses reduction operations. Note that the kernels that need point-to-point synchronization, 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

SN	Bench[Src]	I/P	A2A	P2P	reduction
1.	3MM [23]	8K	✓		
2.	LCS [22]	32K	✓		
3.	MCM [10]	32K	✓		
4.	WF [23]	32K	✓		
5.	LE-LCR [16]	128K	✓		
6.	GEMVER [23]	64K	✓		
7.	Jacobi1D [23]	128K		✓	
8.	Jacobi2D [23]	128K		✓	
9.	Stencil4D [28]	128K		✓	
10.	SOR [7]	128K		✓	
11.	Seidel2D [23]	128K		✓	
12.	IA [12]	4K	✓	✓	✓
13.	HP [6]	4K	✓	✓	✓

Figure 22. Benchmarks used in UWOmp$_{pro}$. Abbreviations: A2A = all-to-all, P2P = point-to-point.

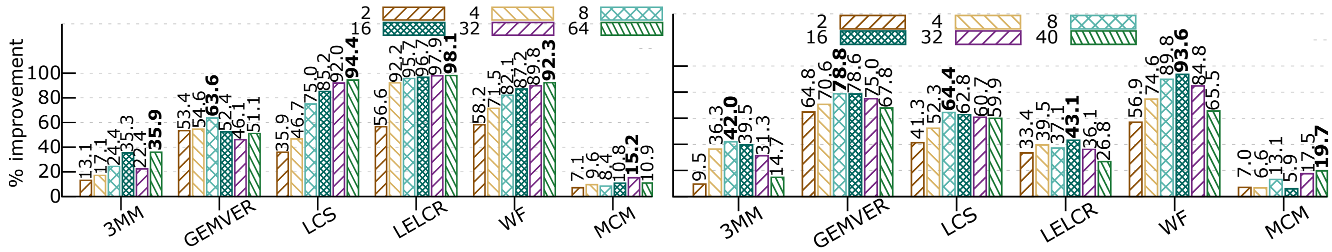
numbers reported in this section are obtained by taking a geometric mean over 10 runs (as suggested by this insightful paper by Georges et al. [15]). In this section, for a language L_x, we use the phrase "performance of an L_x program" to mean the performance of the code generated by the compiler for 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 1-6); we compare the performance of UWOmp$_{pro}$ codes against their UWOmp++ counterparts. (ii) UWOmp$_{pro}$ kernels that perform only point-to-point synchronization, with no reduction (kernels 7-11); we compared their performance with that of their all-to-all versions written in OpenMP. Note: we could not successfully run the code generated by the UWOmp++ compiler for the all-to-all UWOmp++ versions of these codes. (iii) UWOmp$_{pro}$ kernels that perform reduction operations (kernels 12-13); 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 original OpenMP implementations that have the OpenMP reduction clause. (iv) Impact of the scheduling policy; we present a comparative behavior of all the 13 kernels by varying the scheduling policy.

7.1 Evaluation of all-to-all synchronization

For the benchmark kernels 1-6, we show the percentage improvement of UWOmp$_{pro}$ codes over their UWOmp++ counterparts, for varying number of threads, in Figure 23. On HW64, we varied the threads from 2 to 64 (in powers of 2) and on HW40, we varied the threads from 2 to 40 (in powers of 2 and 40).

Our evaluation shows that the performance of UWOmp$_{pro}$ codes perform better than their UWOmp++ counterparts. The percentage improvements varied between 7.1% to 98.1% on the HW64 system and between 5.9% to 93.6% on the HW40 system. We believe that such significant performance gains are mainly due to our efficient handling of worklists (single



(a) HW64 System **(b)** HW40 System

Figure 23. Performance of UWOmp$_{pro}$ kernels with all-to-all synchronization (Vs. UWOmp++ kernels), for varying #threads.

local worklist vs two separate worklists in UWOmp++), and being conservative in converting only the essential parts of the code to CPS form.

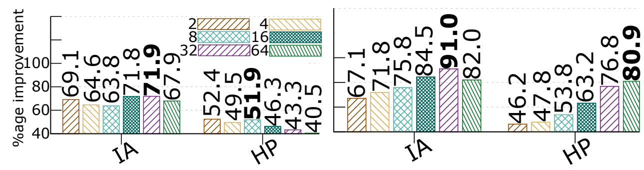
Note that we avoid showing a comparison with the OpenMP counterparts of these benchmarks as Aloor and Nandivada [2] have already shown that UWOmp++ programs run faster than the plain OpenMP programs, and in this evaluation we show that UWOmp$_{pro}$ programs fare significantly better than their UWOmp++ counterparts.

7.2 Evaluation of point-to-point synchronization

For the benchmark kernels 7-11, Figure 24 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 25 summarizes the percentage improvement of the point-to-point variants of the codes compared to All2All version for varying number of 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 All2All versions of UWOmp$_{pro}$.

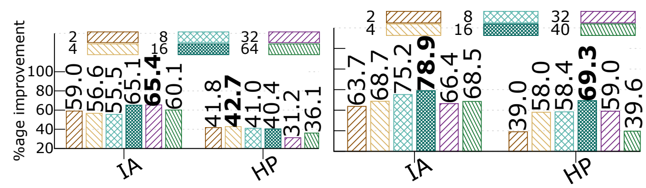
The main reason of this improvement is due to the lesser amount of communication in point-to-point compared to all-to-all synchronization in OpenMP – less waiting time. For most of the kernels we see that the performance improvement reduces gradually with the increasing number of threads. This is mainly 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.

7.3 Evaluation of reduction kernels



(a) HW64 System **(b)** HW40 System

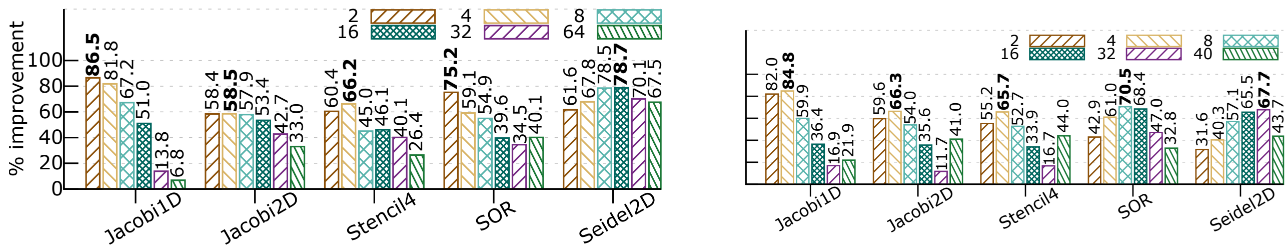
Figure 26. Performance of UWOmp$_{pro}$ kernels using the proposed reduction scheme (Vs. OpenMP), for varying #threads.



(a) HW64 System **(b)** HW40 System

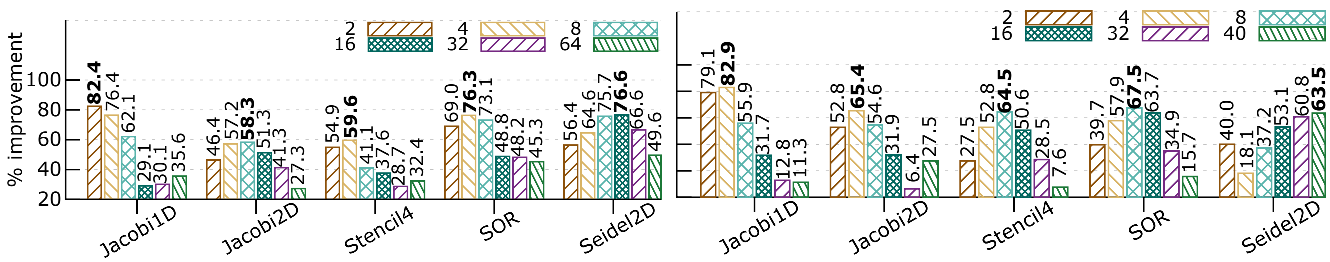
Figure 27. Performance of UWOmp$_{pro}$ kernels using the proposed reduction scheme (Vs. All2All), for varying #threads.

For the benchmark kernels 12-13, Figure 26 shows the percentage improvement obtained using our proposed reduction scheme against the standard OpenMP benchmarks. Figure 27 shows the percentage improvement obtained when comparing our generated codes using the techniques discussed in this paper which involve reduction operations in parallel over that generated using all-to-all barriers in which one of the activities X_1 performs the reduction operation in serial. We see that the proposed scheme performs better significantly. The percentage improvement varied between 40.5% to 71.9% on HW64, and between 46.2% to 91.0% on HW40 when compared with OpenMP. The percentage improvement varied between 31.2% to 65.4% on HW64, and between 39.0% to 78.9% on the HW40 system when compared with all-to-all versions with sequential reduction of UWOmp$_{pro}$. The main reason behind such significant gains is the way the two codes have been written. The UWOmp$_{pro}$ version performs the reduction operation in parallel in-place, while the activities are running. In contrast, the OpenMP reduction clause would only give values once



(a) HW64 System **(b)** HW40 System

Figure 24. Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. OpenMP), for varying #threads.



(a) HW64 System **(b)** HW40 System

Figure 25. Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. All2All), for varying #threads.

the parallel-for-loop is executed. Hence, the inner loop criterion for convergence is helpful in case of UWOmp$_{pro}$ as the values (performed by the reduction operation) are available as and when the synchronization is done.

7.4 Evaluation of different schedules

To show the importance of allowing different scheduling policies and the efficacy of our implemented dynamic and guided schedulers, we present an evaluation of all the kernels for different scheduling policies. Figure 28 shows the percentage improvement of dynamic and guided scheduling compared to static scheduling; due to the lack of space, we show this evaluation only for a fixed number of threads (set to the maximum available hardware cores in the system). For dynamic scheduling, the percentage improvement varied between -45% to 32% on the HW64 system, and between -47% to 30% on the HW40 system. Similarly, for guided scheduling, the percentage improvement varied between -39% to 31% on the HW64 system, and between -43% to 44% on the HW40 system. Such significant variance clearly attests to the importance of supporting different scheduling policies and the efficacy of our implementation of the schedulers. Note, this graph also summarizes the performance improvement of our optimized static scheduling kernels which have all-to-all synchronization as discussed in the Section 6. The optimization related to barrier synchronization kernels for static scheduling is the main reason of the performance of static scheduling being better than dynamic or guided scheduling.

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.

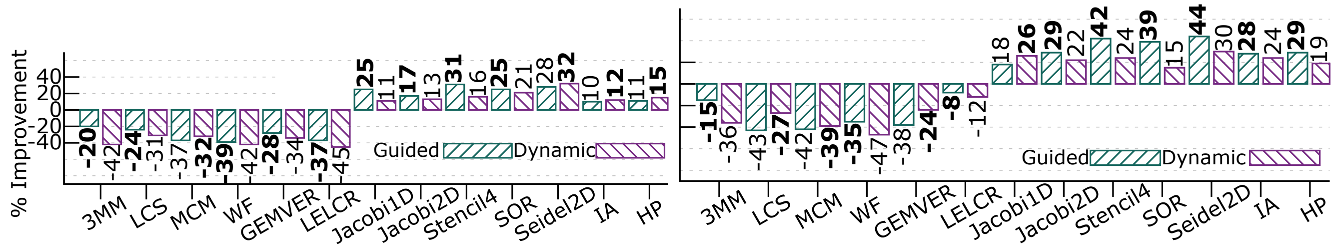
8 Related Work

There have been multiple efforts [14, 20, 25, 29] to utilize continuations to extend and translate parallel programs. Fischer et al. [13] provide a modular approach to do a CPS translation of event-driven programs in Java. For the Cilk language, Blumofe et al. [5] propose a C-based runtime with a work-stealing scheduler useful for multithreaded programming, which uses continuations to spawn and join tasks. Our translation scheme and the underlying runtime take advantage of CPS to efficiently perform wait and continue operations, and support different scheduling policies, along with efficient reduction operations.

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. Our translation takes inspiration from their approach, but generalizes 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



(a) HW64 System. #threads = 64.

(b) HW40 System. #threads = 40.

Figure 28. Comparison of dynamic and guided scheduling over static scheduling; #threads set to maximum #cores.

postbox based communication subsystem to support efficient signal and wait functions, along with reduction operations and arbitrary schedules. We have implemented our scheme in the IMOP compiler framework and performed a thorough evaluation. We argue that programmers can write more expressive and performant codes using UWOmp$_{pro}$.

References

[1] 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

[2] 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

[3] A. W. Appel and Z. Shao. 1994. *An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures*. Vol. 6. 47–74 pages.

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

[5] 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

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

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

[8] 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

[9] 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

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

[11] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[12] 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.

[13] 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

[14] 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.

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (oct 2007), 57–76. https://doi.org/10.1145/1297105.1297033

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

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

[20] 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.

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

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

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

[24] 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.

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

[26] Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. *SIGPLAN Lisp Pointers* VII, 3 (jul 1994), 150–161. https://doi.org/10.1145/182590.156783

[27] 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.

[28] 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

[29] 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