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

ADITYA AGRAWAL, Department of CSE, IIT Madras, India
V. KRISHNA NANDIVADA, Department of CSE, IIT Madras, India

OpenMP is one of the most popular APIs widely used in industry and academia 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 any form of synchronization 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++ (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. 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 and wait operations, reduction operations, and all the scheduling policies of OpenMP. 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.

CCS Concepts: • **Software and its engineering → Compilers**; **Parallel programming languages**.

1 INTRODUCTION

The emergence of multi-core systems has brought forth many different parallel languages like X10 [11], Chapel [19], HJ [9], OpenMP [13], and so on to the mainstream. These languages encourage programmers to think in parallel and provide means to express parallel logic in a convenient manner. 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 tasks/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[1]. 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

[1] In this manuscript, we use the terms tasks and activities interchangeably.

Authors' addresses: Aditya Agrawal, cs20s026@cse.iitm.ac.in, Department of CSE, IIT Madras, India; V. Krishna Nandivada, nvk@iitm.ac.in, Department of CSE, IIT Madras, India.

XXXX-XXXX/2023/2-ART $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
#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 with 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) UW-OpenMP 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 with 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

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

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 the classical 1D Jacobian kernel (source [2]) written in five different versions. Figure 1a shows the implementation of the kernel using Vanilla OpenMP (source [3, 26]). Vanilla OpenMP prohibited 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) [1]). 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 of the code 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 [1] proposed UW-OpenMP that introduces 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 [3] 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 that in Figure 1b), each activity $X_\texttt{i}$ (corresponding to iteration `i`) is waiting for all the 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 $X_\texttt{i}$ ($\texttt{i} \neq 1$) 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 [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). In the context of OpenMP, Shirako et al. [30] 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 left to right – can be limiting in terms of 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.

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 $X_{\texttt{i-1}}$ and $X_{\texttt{i+1}}$.

The second all-to-all barrier of Figure 1c has been replaced with two commands: `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 ($X_\texttt{i}$, $\texttt{i} \neq 1$) 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) synchronization objects. Further, the code in UWOmp$_{pro}$ performing the communication is arguably more readable than that of the HJ version which involves multiple phaser objects performing point-to-point communication (for example, Figure 1e vs Figure 1d). In addition, UWOmp$_{pro}$ optionally supports efficient reduction operations at the synchronization (wait) points. 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.

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 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 of OpenMP.

• 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. Along with support for all reduction operations for all primitive types supported in OpenMP, UWOmp$_{pro}$ allows custom reduction operations.

• 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 now briefly describe some OpenMP constructs; interested readers may see the OpenMP manual [13].

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 [3].

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$_{pro}$ (along with brief syntax), for easy reference. All these commands are conditional in nature and

commands	target activities	reduction?	syntax
signal	1 or more	×	`signal`(int *e*, int *act*, ...);
wait	1 or more	×	`wait`(int *e*, int *act*, ...);
signalAll	all	×	`signalAll`(int *e*);
waitAll	all	×	`waitAll`(int *e*);
signalSend	1 or more	✓	`signalSend`(int *e*, void **m*, int *act*, ...);
waitRed	1 or more	✓	`waitRed`(int *e*, `FptrT` *rOp*, void **rVar*, int *act*, ...);
signalAllSend	all	✓	`signalAllSend`(int *e*, void * *m*);
waitAllRed	all	✓	`waitAllRed`(int *e*, `FptrT` *rOp*, void **rVar*);

Fig. 2. List of commands supported by UWOmp$_{pro}$. The first argument is a predicate expression; the signal/wait operation is invoked only if the predicate 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 arguments: *e*: predicate, *act*: target activity, *m*: message, *rOp*: reduction function, *rVar*: reduction variable.

```
#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

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

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 1e shows the UWOmp$_{pro}$ code (using point-to-point synchronization) to perform the Jacobian 1D stencil computation shown in Figure 1c. 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 (Source [2, 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 $\mathtt{i}^{th}$ 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

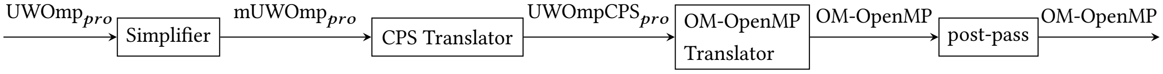


Fig. 4. Block diagram of our proposed translation scheme.

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 the 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`). In contrast to the UW-OpenMP version, in the UWOmp$_{pro}$ version, all threads together perform the reduction operation in parallel. In addition to the standard set of reduction operations supported by OpenMP, UWOmp$_{pro}$ supports any arbitrary reduction operations, via user-specified reduction functions, as long as the function is associative and commutative (not checked). Compared to the OpenMP syntax, for the OpenMP supported reduction operators, for ease of readability, we use verbose reduction operator names (for example, ADD in place of '+'). See Section 6 for a discussion on the same.

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. The input UWOmp$_{pro}$ code is input to the Simplifier module which converts the given input code to a representative subset of UWOmp$_{pro}$ code which we call mUWOmp$_{pro}$. The mUWOmp$_{pro}$ code is input to the 'CPS Translator' module which converts the given mUWOmp$_{pro}$ code to CPS form, which we call UWOmpCPS$_{pro}$. The UWOmpCPS$_{pro}$ code is input to the 'OM-OpenMP Translator' module which translates the CPS code to conforming OpenMP code such that it does not invoke barriers inside work-sharing constructs (parallel-for-loops). Finally we invoke a post-pass step step to introduce type-specific reduction operations. We now describe these important modules of our translation scheme.

4.1 Simplifier

For the ease of explaining the transformations, like Aloor and Nandivada [3], we use a representative subset of the input UWOmp$_{pro}$ language called miniUWOmp$_{pro}$ (in short, mUWOmp$_{pro}$). Figure 5a shows the grammar of the mUWOmp$_{pro}$ language. The program accepts a set of Function Declarations (`FuncDecl`) followed by the `MainFunc`. The body of the function declaration 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. The body of the main function consists of a parallel region which in turn consists of a set of parallel-for-loops or barrier statements. Each parallel-for-loop is a normalized loop [23] whose body is a function call. In Section 6, we discuss how any general UWOmp$_{pro}$ program can be translated to mUWOmp$_{pro}$ code.

```
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)* }
```

(a) Grammar for mUWOmp$_{pro}$

```
Program       ::=  (CPSFuncDecl)* (FuncDecl)*
                   MainFunc
CPSFuncDecl   ::=  void ID(Closure 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 }
```

(b) Grammar for UWOmpCPS$_{pro}$

Fig. 5. Grammar for mUWOmp$_{pro}$ and UWOmpCPS$_{pro}$ language.

4.2 CPS Translator

Our translation scheme is inspired by that of Aloor and Nandivada [3], 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 [20]); 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. This observation eliminates the unnecessary overhead induced if we were to translate the entire program to CPS form. 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.

4.2.1 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 5b. 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`). (iv) As is standard in CPS translation, the continuation object is passed as an additional argument to each CPS function call (`CPSFunCall`). Note that `Stmt` denotes any sequential statement, `FuncDecl` is any regular C function declaration, `FunCall` is any regular non-CPS function call statement.

4.2.2 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 [3], except for some modifications. 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

1.	⟦ K *T* fun(TypeArgLst) { S } ⟧ //fun is called from within parallel-**for**-loop	⇒	*void* funCPS(Closure K,TypeArgList) { ⟦ 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 is 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(Closure K, T_1,V_1)
			{⟦ K X V_1 Y ⟧});
			C=mkClsr(pCPS,FV(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.	⟦ **#ompparallel** { S } ⟧	⇒	**#ompparallel** { ⟦ S ⟧ }

10.	⟦**#ompfor**
	for(Header){
	fun(args);
	} S ⟧
		⇒	**#ompfor**
			for(Header){
			$K = mkClsr(id, null, null)$;
			funCPS(K, args)
			} ⟦ S ⟧

11.	⟦ S ⟧
	//S contains no
	//par-for-loops
		⇒	S

Fig. 6. CPS translation rules for the parallel constructs.

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 9). Rules 1-8 are standard CPS Translation Rules that are used to convert a given input program to CPS Form. Note: We only handle functions/statements that are reachable from a parallel-for-loop and if any function is called from outside the parallel-region is left as it is. Rule 10 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, [Rules 1-8, Figure 6], 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 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 11).

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

⇒

```
1  tid=thread-number();
2  sched=getSchedule(schedOpt);
3  chSize=getChunkSize(schedOpt);
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

.

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

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

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

4.3 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 7a shows the rule to translate the parallel-for-loop.

Line 4 in Figure 7a 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 7b 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 function pointer *schedPtr* to the corresponding scheduler function that needs to be called depending on the schedule policy string *sched*. We emit a parallel-for-loop that pushes the closure for each activity to $WL[tid]$ (Lines 5-9 in Figure 7a). Finally, we invoke the appropriate scheduler (Line 10 in Figure 7a).

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.

4.4 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 [13]. As mentioned in Section 3.2, the reduction-related 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 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`).

4.5 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 is 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 2-10 from Figure 7b) and wraps the call to function `fCPS` inside the closure `C` before enqueuing the closure in the appropriate worklist $WL[tid]$. 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 the different scheduling policies of OpenMP. Our support for these operations is based on our novel design of the communication sub-system between the activities of each 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 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

```
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.

Fig. 8. Example Transformations.

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).

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.

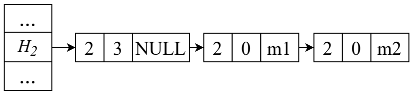
Signal Only Postbox If the communicating activities are guaranteed to never send/receive any data-messages, then we simply represent each slot in the hashmap as a list of pairs of the form (sender, counter). When an activity X_i wants to send a signal to X_j, we simply increment the counter of X_j in P_i At the receiving activity, we atomically decrements the counter, if it is non-zero, and return 1. Else, we return 0 (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 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 9a.

```
struct Node{
int senderId;
int sCounter;
void * message;
struct Node* next; };
```

(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.

Fig. 9. Mixed-Mode Postbox: Structure of a node and example postbox to illustrate the design of the postbox.

```
void sendMsg($j$, $i$, $m$, $ctr$) // send message $m$ from $X_j$ to $X_i$
begin
    $H_j$=hashFunction($j$);
    Acquire Lock on $P_i.H_j$;
    $L$ = $P_i.H_j$.list;
    if $m \neq NULL$ then ; // data-message

        $n$ = create-node($i$, $ctr$, $m$); Insert $n$ to $L$;
    else
        $n_s$=search($i$,$j$,$signal$); // search signal-node
        if $n_s$==$NULL$ then
            $n$ = create-node($i$, $ctr$, $m$);
            Insert $n$ to $L$;
        else
            $n_s.sCounter$++;
    Release Lock $P_i.H_j$;
```

Fig. 10. mixed mode: send message.

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 9b 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.

When an activity has to send a data-message (Figure 10), we create a new node (of type Node), add the message to the node, and insert the node to the appropriate list. When an activity has to send a signal, we first search for the first node corresponding to activity X_j in the list L. If the node n_s is not found (n_s==NULL), we create a new node with the counter field initialized with ctr and insert it to L, otherwise, we increment the corresponding counter.

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 list is empty. If so, we return NULL. Else, we search for the first node n of the required type sent from the sender X_j. If the node is not found, we return NULL. Else, if the type of the requested message is signal, we decrement the corresponding counter $sCounter$. Finally, if the node is a data-message ($sCounter$==0) or the last signal message, we delete the element from the list L.

```
Node* recvMsg($i$, $j$, $tp$) // $X_i$ receives a message of type $tp$ from $X_j$; $tp \in$ {signal, data-message}
begin
    Node* $n$ = NULL;
    $H_j$=hashFunction($j$);
    Acquire Lock on $P_i.H_j$;
    $L$=$P_i.H_j$.list;
    if $L$ not empty then
        $n$ = search($i$,$j$,$tp$);
        if $n$ == NULL then return NULL ;
        if $tp$ == signal then
            $n.sCounter--$;
        if $n.sCounter$==0 then $n$=Delete $n$ from $L$;
    Release Lock $P_i.H_j$;
    return $n$;
```

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

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

Fig. 12. Signal a set of activities with message m.

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 that are 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 (Figure. 12) and signalAllSendCPS (Figure. 13) 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 *msgArr* (unique for each parallel-for-loop). This helps in performing the reduction operation.

```
void signalAllSendCPS(K, e, m)
begin
    if e == true then
        N=endIdx-startIdx+1;
        sendMsg(from, from, m, N);
        if m != NULL then msgArr[from]=m;
    Invoke continuation in K;
```

Fig. 13. Signal all activities with message *m*. Here, *startIdx* and *endIdx* denote the starting and ending index number of the parallel-for-loop.

```
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 Node *aNode;
        for every iteration i in p do
            aNode=recvMsg(iter, i, typeMsg);
            if aNode==NULL then
                aNode=recvMsg(i,i,typeMsg);
                if aNode==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 aNode.message and reduction variable redVar;
    Invoke the continuation in K
```

Fig. 14. Wait for a subset of activities with reduction

5.3 Wait Algorithm

We now describe the wrapper methods that are 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. We now describe the `waitRedCPS` and `waitAllRedCPS` methods. Similar to the signal wrapper methods, these methods are also in CPS form.

```
1  void waitAllRedCPS(K, e, rop′, redVar, copy)
2  begin
3      if e==true then
4          iter=K.iteration;
5          if rop′==NULL then typeMsg=signal ;
6          else typeMsg=data-message ;
7          struct Node *aNode;
8          for every iteration i in [startIdx,endIdx] do
9              aNode=recvMsg(i, i, typeMsg);
10             if aNode==NULL then
11                 aNode=recvMsg(iter,i,typeMsg);
12                 if aNode==NULL then
13                     wClsr=mkWaitClsr(K,iVal,rop′,redVar,copy,p);
14                     wClsr.start=i;
15                     acquireLock(lock);
16                     enqueueClosure(WL[tid], wClsr);
17                     releaseLock(lock); return;
18         if typeMsg==data-message then
19             lazyReduce(K);
20         else
21             Invoke Continuation in K
```

Fig. 15. Wait for all activities with reduction. *startIdx* and *endIdx* denote the starting and ending indices of the parallel for loop.

The method `waitRedCPS` (Figure. 14) first checks if the conditional-expression e is true. If so, it invokes the `recvMsg` function for each target activity (Line 9). 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 17). We execute the closure K, only if e is false or all the signals have been received. The `waitAllRedCPS` method works similarly, by waiting for all the messages to be available before performing the reduction. The differences can be observed by considering the Lines 19-21 in Figure 14, and Lines 18-21 in Figure 15.

5.4 Reduction Operations

We now highlight some salient points about our reduction strategy. Figure 14 shows a simple eager way of reduction which invokes the reduction operation (*rOp′*) eagerly, on the value stored in $rVar$ and the message from the sender activity $aNode.message$(Line 19), as and when the message is read. In contrast for the all-to-all synchronization (`waitAllRedCPS`), we efficiently perform the reduction after all the messages have been received (in a *lazy* manner, Line 18 in Figure 15). We describe the intuition behind this design decision below.

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

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

Fig. 16. 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);
            break;
        executeWL(rdyWL);
```

Fig. 17. UWOmp$_{pro}$ dynamic scheduler algorithm.

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 $O(log(N_a))$ steps. 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++ [3] 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 4.3, if the programmer specified scheduling policy is one of static, dynamic, or guided, then the translated code invokes the appropriate scheduler. Recall that for runtime-scheduling, we emit additional code immediately before each parallel-for-loop. We now discuss the details of the former three schedulers.

static scheduler. The scheduler function `scheduler-static` (Figure 16) 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 4.3, for dynamic (and guided) scheduling we use the global worklist. In the dynamic-scheduler function (Figure 17), 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 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 [3]), 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<omp_num_threads();++i) {foo(···);}
```

Note: The arguments to foo are the list of free variables and omp_num_threads() returns the number of threads executing this code.

• **Memory management.** Considering the overheads of CPS translation, we reuse many well known optimizations [3, 4, 29] to optimize the memory usage and malloc/free calls for closures. For simplicity, we do not show the places where the closures are freed, though we do free them at the appropriate places in the actual implementation.

• **Optimization for Static Scheduling.** To minimize the memory and maintenance 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 a priori 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 its worklist (*left* != *right*) and then waits for other threads to finish.

• **User defined reduction operations.** Besides the reduction operations supported by OpenMP, UWOmp$_{pro}$ also supports user-defined reduction operators (say OP, e.g., Intersect) over user-defined type (say, Type, e.g., Set) of data , as long as the user provides the implementation of two methods OPType and COPYType. The type signature of any reduction function OPType must be of the form `void fun(T* a, T* b)`, where `T` is the type of the values being reduced. This function should reduce the two values `*a` and `*b` and should store the result back in `*a`. Similarly, along with the reduction function, the user must specify a copy-back method COPYType of the form `void copy(T *a,T *b)`, which copies the value pointed to by `*b` to `*a`. This copy-back method is only required if `T` is a user-defined data-type (and not one of the primitive ones). We will use this copy method in the post-pass 4.4 of the translation.

7 IMPLEMENTATION AND EVALUATION

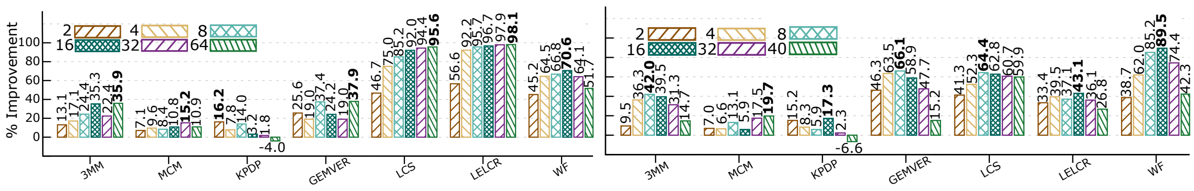
We implemented our proposed language extension, translation and the runtime support for UWOmp$_{pro}$ in two parts: (i) the translator has been written in Java [5] in the IMOP Compiler Framework [24] - approximately 8000 lines of code (ii) the runtime libraries are implemented in C [21] - approximately 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 18). These include all the kernels used by Aloor and Nandivada [3] (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

SN	Bench[Src]	I/P	A2A	P2P	reduction
1.	3MM [26]	8K	✓		
2.	LCS [25]	32K	✓		
3.	MCM [12]	32K	✓		
4.	WF [26]	32K	✓		
5.	LELCR [17]	128K	✓		
6.	GEMVER [26]	64K	✓		
7.	KPDP [26]	128K	✓		
8.	Jacobi1D [26]	128K		✓	
9.	Jacobi2D [26]	128K		✓	
10.	Stencil4D [31]	128K		✓	
11.	SOR [8]	128K		✓	
12.	Seidel2D [26]	128K		✓	
13.	IA [14]	4K	✓	✓	✓
14.	HP [7]	4K	✓	✓	✓

Fig. 18. Benchmarks used in UWOmp$_{pro}$. Abbreviations: A2A = all-to-all, P2P = point-to-point.



(a) HW64 System

(b) HW40 System

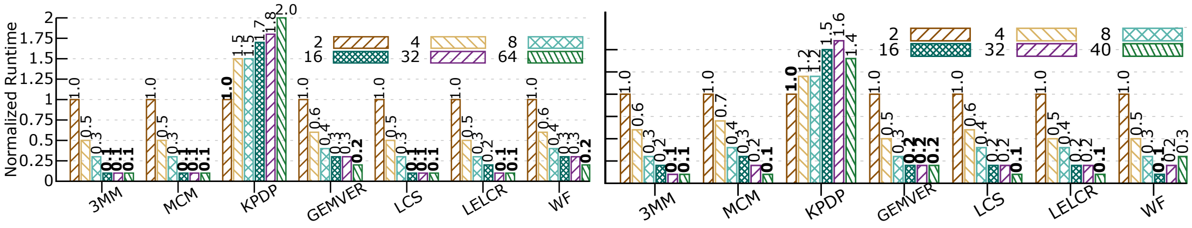
Fig. 19. Performance of UWOmp$_{pro}$ kernels with all-to-all synchronization (Vs. UWOmp++ kernels), for varying #threads.

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 that perform 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 14 kernels by varying the scheduling policy.

7.1 Evaluation of all-to-all synchronization

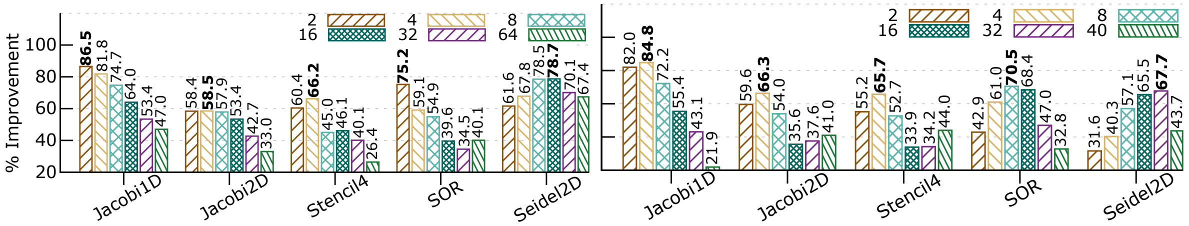
For the benchmark kernels 1-7, Figure 19 shows the percentage improvement of UWOmp$_{pro}$ codes over their UWOmp++ counterparts, for varying number of threads. On HW64, we varied the threads from 2 to 64 (in powers of 2) and on HW40, we varied the threads from 2 to 40 (in powers of 2 and 40).



(a) HW64 System

(b) HW40 System

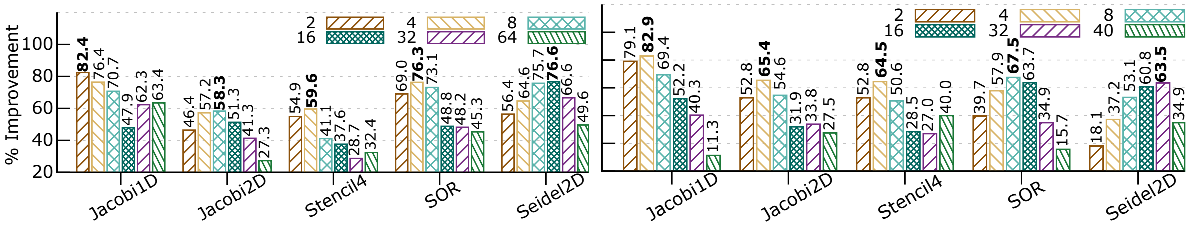
Fig. 20. Scaling: Normalized execution times of UWOmp$_{pro}$ kernels (w.r.t. execution time for two threads), with all-to-all synchronization, for varying #threads.



(a) HW64 System

(b) HW40 System

Fig. 21. Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. OpenMP), for varying #threads.

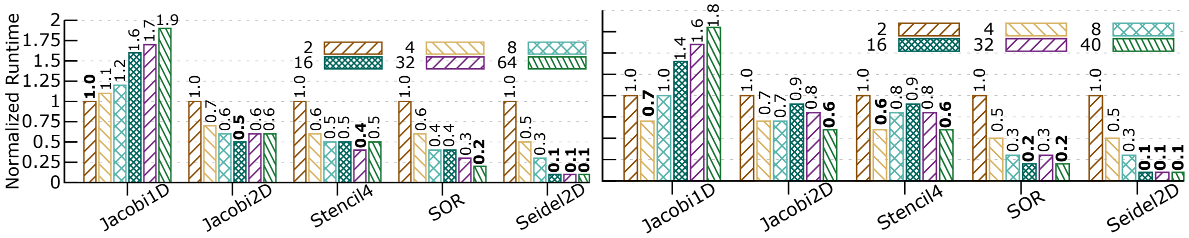


(a) HW64 System

(b) HW40 System

Fig. 22. Performance of UWOmp$_{pro}$ kernels with point-to-point synchronization (Vs. All2All), for varying #threads.

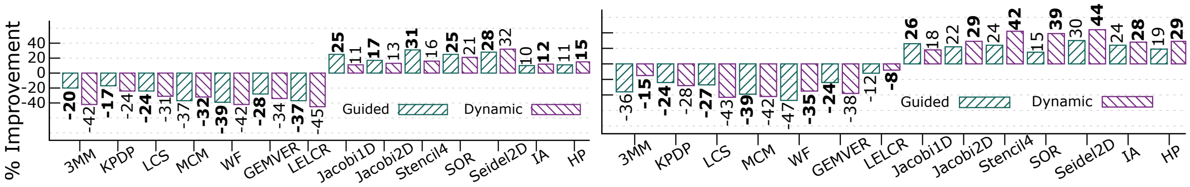
Our evaluation shows that except for KPDP in one particular configuration (64 cores on HW64 and 40 cores on HW40), the UWOmp$_{pro}$ codes perform better than their UWOmp++ counterparts. Even for that particular configuration the performance degradation is minimal (<7%). One common pattern we find is 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 our efficient handling of worklists (single local worklist vs two separate worklists in UWOmp++), and being conservative in converting only the essential parts of the code to CPS form.



(a) HW64 System

(b) HW40 System

Fig. 23. Scaling: Normalized execution times of UWOmp$_{pro}$ kernels (w.r.t. execution time for two threads), with point-to-point synchronization, for varying #threads.



(a) HW64 System. #threads = 64.

(b) HW40 System. #threads = 40.

Fig. 24. Comparison of dynamic and guided scheduling over static scheduling; #threads set to maximum #cores.

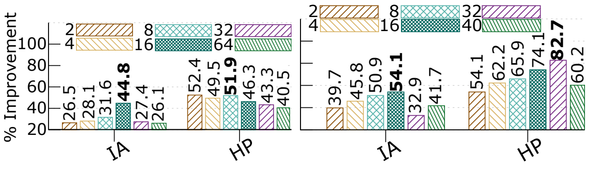
We also found that the UWOmp$_{pro}$ codes scale well with the increasing number of threads, for most kernels; Figure 20 shows the normalized execution time of the UWOmp$_{pro}$ kernels (with respect to the execution time for two threads). Except for KPDP, we find that the generated UWOmp$_{pro}$ code scales well with increase in the number of threads. In case of KPDP, we observed that neither of the UWOmp++ kernel and OpenMP kernels scale well and this behavior carried over to the UWOmp$_{pro}$ kernels (though more performant than the UWOmp++ and OpenMP kernels).

Note that we avoid showing a comparison with the OpenMP counterparts of these benchmarks as Aloor and Nandivada [3] 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

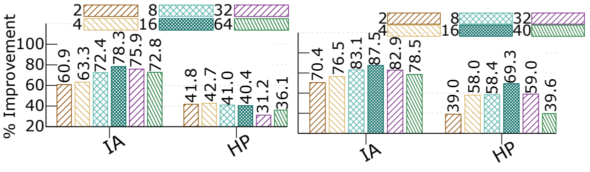
For the benchmark kernels 8-12, Figure 21 summarizes the percentage improvement of the point-to-point variants of the codes compared to OpenMP, for varying number of threads, on both HW64 and HW40 systems. Figure 22 summarizes the percentage improvement of the point-to-point variants of the codes compared 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



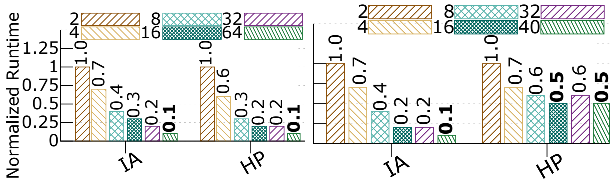
(a) HW64 System (b) HW40 System

Fig. 25. Performance of UWOmp$_{pro}$ kernels that use reduction (Vs. OpenMP), for varying #threads.



(a) HW64 System (b) HW40 System

Fig. 26. Performance of UWOmp$_{pro}$ kernels that use reduction (Vs. All2All), for varying #threads.



(a) HW64 System (b) HW40 System

Fig. 27. Scaling: Normalized execution times of UWOmp$_{pro}$ kernels (w.r.t. execution time for two threads), using the proposed reduction scheme, for varying #threads.

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.

Figure 23 shows the normalized execution time of the UWOmp$_{pro}$ kernels (with respect to execution of two threads). Except for Jacobi1D, we find that the generated UWOmp$_{pro}$ codes scales well with increase in the number of threads. We find that Jacobi1D, similar to KPDP (discussed in the evaluation of all-to-all synchronization kernels), did not scale well as a kernel (even for UWOmp++ and OpenMP) and that behavior is reflected even in UWOmp$_{pro}$; though the latter is much more performant than the UWOmp++ and OpenMP kernels.

7.3 Evaluation of reduction kernels

For the benchmark kernels 13-14, Figure 25 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 when compared with OpenMP.

As a point of 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 the serial one: 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$_{pro}$. Figure 26 shows the detailed graphs for this comparison.

Figure 27 shows the normalized execution time of the UWOmp$_{pro}$ kernels (with respect to the execution time for two threads). We find that the generated UWOmp$_{pro}$ codes scale well with increase in the number of threads.

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 24 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). 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.

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

Aloor and Nandivada [1] 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 [3] 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.

There have been multiple efforts [16, 22, 28, 32] to utilize continuations to extend and translate parallel programs. [16] 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. [15] 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 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.

White [33] 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 [18] 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 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$_{pro}$.

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] 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.

[3] 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

[4] 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.

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

[14] 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.

[15] 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

[16] 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.

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

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

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

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

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

[22] 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.

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

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

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

[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] 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

[30] 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.

[31] 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

[32] 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

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