Accelerating OpenMP Applications Through Parallel Hardware Architecture

Atakan Doğan, İsmail San
Department of Electrical and Electronics Engineering
Eskişehir Technical University
Eskişehir, Turkey
email: atdogan@eskisehir.edu.tr,
email: isan@eskisehir.edu.tr

Kemal Ebcioğlu
Global Supercomputing Corporation
Yorktown Heights, NY, USA
email: kemal.ebcioglu@global-supercomputing.com

***Abstract*—It is a well-known fact that application-specific hardware has both performance and power advantages as compared to general-purpose CPUs and GPUs. Furthermore, in order to improve the computing performance leveraging available parallelism in software and hardware, high-level parallel programming paradigms, such as OpenMP and OpenCL, have been viable choices for designing application-specific hardware. In this study, an application-specific parallel hardware architecture with a specialized memory hierarchy is proposed for a class of fork-join applications that can be modeled by an OpenMP program. Furthermore, three different case studies are provided to show how this model can be employed for the hardware acceleration of such applications.**

Keywords-OpenMP applications; high-level synthesis; application-specific hardware; NoCs; system-on-chip.

I. INTRODUCTION

This article is based on our previous paper [1] and extends it in several dimension, which at least includes a revised parallel hardware architecture model.

The OpenMP Application Programming Interface is a well-established standard for parallel programming on shared-memory multiprocessors. OpenMP has adopted the fork-join model of parallel execution. According to this model, an OpenMP program begins as a single thread of execution, called an *initial thread*. When any thread encounters an OpenMP *parallel* construct, a team of master and slave threads is created to execute the code enclosed by the construct (this corresponds to the *fork*). At the end of the construct, only the master thread continues, while all slave threads are terminated (this corresponds to the *join*) [2][3].

In the literature, there are several approaches that attempt to generate parallel hardware from OpenMP applications. These studies may be broadly grouped into three classes: (i) OpenMP-based pure hardware-based acceleration [1][4], (ii) OpenMP-based system-on-chip design with a soft processor and a number of hardware accelerators [5][6][7][8], (iii) OpenMP-based device offloading [9][10].

In [4], OpenMP parallel directive and a few worksharing and synchronization directives are first translated to synthesizable VHDL, and then from VHDL to FPGA hardware. In [4], each OpenMP thread is implemented by a finite state machine. A crucial limitation of this study is that there is no memory hierarchy. That is, an OpenMP hardware thread is only enabled to access on-chip memory resources, which clearly hampers to provide a scalable shared memory system in an efficient manner.

Any task specified by OpenMP task directive is converted into a custom hardware unit that carries out the work within that particular task [5][6]. These hardware units are then combined together to form an accelerator component. Finally, a system-on-chip is created based on a Nios II soft-core processor and a number of such accelerator components. However, this created system-on-chip is not equipped with memory hierarchy. Furthermore, [5] and [6] neither provide any details about synchronization, nor support any nested parallelism.

In [7], on the other hand, the system-on-chip with a MIPS soft-core processor has a memory hierarchy that is composed of a local memory per accelerator unit and a shared L1 data cache, both of which are implemented on on-chip Block RAMs, and off-chip DDR memory. Two special-purpose IP cores, *hardware mutex core* and *hardware barrier core*, are further defined in order to support several OpenMP synchronization directives as well. In addition, [7] features two-level nested parallelism.

Different from the single MIPS processor in [7], the system-on-chip in [8] includes one or more Microblaze soft-core processors. An OpenMP thread can be run on either a processor or a hardware subsystem in [8]. Furthermore, the system-on-chip of [8] instantiates an application-specific synchronization network based on the Shared Memory Process Network model of the related OpenMP application.

The surveyed approaches [4][5][6][7][8] so far do not leverage the OpenMP `target` directive for creating hardware accelerators. The OpenMP `target` directive [2], on the other hand, enables programmers to mark regions of an application that should be offloaded to an FPGA (or GPU or DSP) device. Additionally, the data mapping clauses of the OpenMP `target` directive help programmers specify what and how data should be mapped to the target device. Two different tool chains are introduced in [9][10]. These tool chains aim to offload OpenMP-based applications annotated with the OpenMP `target` directive to FPGA-based hardware accelerators.

A few High Level Synthesis (HLS) tools, such as [11][12] have support to produce parallel hardware from OpenCL. Finally, fork-join like hardware constructs that are automatically generated from single-threaded sequential code using compiler dependence analysis is described in [13]. The present work focuses on converting explicitly parallel OpenMP programs to parallel hardware, as opposed to converting single threaded sequential programs as in [13], to parallel hardware.

This study makes the following contributions to the literature as compared to [1], [4]-[10]: (i) A parallel hardware architecture with explicit support for the OpenMP synchronization directives is introduced. That is, it provides specialized components and networks for the hardware implementations of the OpenMP barrier, atomic, and critical directives, which are not found in [1]. (ii) The memory hierarchy with L1 and L2 caches is presented in detail to support the OpenMP memory model. In [1], however, there are a few untouched crucial issues related to OpenMP memory model. In the other studies [4]-[10], either there is no data cache, or there is a single data cache shared by all hardware threads. (iii) The model proposed here features dynamic scheduling of OpenMP for-loop iterations, while [1] supports only static scheduling. (iv) Finally, the nested parallelism in [1] is refined here to make it fully conform to the OpenMP semantics.

The rest of the paper is organized as follows: Section II summarizes a few features of OpenMP pertaining to this study. Section III introduces the proposed parallel hardware architecture. Section IV shows how this architecture provides support for the fork-join applications using three different case studies. Finally, Section V concludes the paper.

II. PARALLEL PROGRAMMING IN OPENMP

OpenMP is briefly introduced in this section to show how it can be used to express parallelism in applications. The execution model of OpenMP is based on the creation and management of threads, which requires the execution of at least one parallel region. In order to better explain the execution model, consider the following OpenMP code fragment:

```
void main_prog () {
   ......
   sequential_part-1
   ......
   #pragma omp parallel {
      ......
      parallel_region-1
      ......
   }
   ......
   sequential_part-2
   ......
}
```

According to the semantics of OpenMP, an initial thread starts with executing `sequential_part-1`. The sequential execution of the initial thread continues until it encounters `#pragma omp paralel`, which results in spawning (forking) a team consisting of itself (master thread) and additional other slave threads. Each thread in the team executes an implicit task that will be generated by the code according to `parallel_region-1`. At the end of the parallel construct, there is always an implicit barrier. Once all threads reach to this implicit barrier point, only the master thread continues its execution with `sequential_part-2`, while all slave threads are terminated, which corresponds to a join event [2][3]. There are a few points to emphasize related to the `parallel` directive:

- Any part of a program that is not enclosed by a parallel construct will be executed serially, including OpenMP worksharing constructs.
- The work of a parallel region will not be distributed among the threads in a team unless a worksharing construct is used.
- Although a parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution.

OpenMP allows any number of parallel constructs to be specified in a single program. For example, right after the end of `sequential_part-2`, there could be another `#pragma omp paralel` that encloses `parallel_region-2`. It is possible in OpenMP that each parallel region can be executed by a different number of threads.

OpenMP also supports *nested parallelism* that enables a parallel region to be nested within another one. For example, `parallel_region-1` above can include a second `#pragma omp parallel` with an additional `parallel_region-2` nested inside `parallel region-1`. Any thread of `parallel region-1` that encounters this nested parallel construct can start a new team of threads and become the master of its own team.

A. Worksharing

A worksharing construct distributes the execution of the related worksharing region among the members of the team that encounters it. Each thread executes a portion of the worksharing region in the context of its implicit task. A worksharing region has no barrier upon entry, but an implied barrier upon exit, unless a `nowait` clause is specified. Note also that a worksharing construct does not launch any new threads and it is effective only in a parallel region [2][3].

The `#pragma omp for` directive is the most important worksharing construct of OpenMP since loops are the most common source of parallelism in many applications. Here is an example OpenMP code fragment with the for directive:

```
#pragma omp parallel num_threads(4)
{
 #pragma omp for schedule(static) {
   for (i=0; i<1000; i++)
     a[i]=(b[i]+b[i+1])/2.0;
 }
}
```

The for directive causes the iterations of the loop immediately following it to be distributed across the threads and executed in parallel. The most relevant clause supported by the for directive is `schedule`, which determines how the iteration space should be distributed among the team of threads. The schedule clause accepts one of the five different scheduling choices, namely *static*, *dynamic*, *guided*, *runtime*, and *auto*. Thus, the user, the compiler, or the runtime is allowed to decide about the load balancing of threads for achieving the best application performance. In the case of static scheduling, for example, iterations are equally divided among threads as specified by the OpenMP standard. As a result, in the example given above, each thread will be assigned a task composed of 250 i-loop iterations. In the case of dynamic and guided scheduling, however, a thread is assigned a new chunk of iterations only if it completes the execution of the current task and is ready for the next one.

The `#pragma omp sections` directive allows a set of structured code blocks (e.g., several independent subroutines) to be executed in parallel by a team of threads, where each thread executes one code block at a time, and each code block will be executed exactly once. Note that all threads must finish their corresponding sections before any thread can proceed [2][3].

The `#pragma omp single` directive specifies that the associated structured block must be executed by only one of the encountering threads among in the team, while the other threads wait at an implicit barrier at the end of the single construct if the barrier is not eliminated by a `nowait` clause [2][3].

`#pragma omp paralel for` and `#pragma omp paralel sections` are parallel worksharing constructs that can be used when a parallel region is composed of only one worksharing construct. That is, the worksharing region includes all the code in the parallel region.

B. Synchronization

OpenMP does not guarantee atomicity when accessing and/or modifying shared data by multiple threads running in parallel. Consequently, the user is responsible for avoiding data race conditions among multiple threads. In order to make it easier for the user to orchestrate the access to shared data by multiple threads, OpenMP supports a few synchronization constructs, such as *critical*, *atomic*, and *barrier* [2][3].

The `#pragma omp critical` directive restricts the associated critical region of an application to be executed atomically by a single thread at a time. Suppose that a thread is currently executing inside a critical region. When another thread reaches that same critical region and attempts to execute it, it will be blocked at least until the first thread exits that critical region.

In contrast to the critical construct, the `#pragma omp atomic` directive provides that a single memory location is accessed atomically by multiple threads without interference. The atomic construct is similar to the atomic read-modify-write types of instructions in an instruction set architecture.

In OpenMP parallelism model, there are both implicit and explicit barriers. Remember that there is an implicit barrier at the entry to or exit from parallel regions and at the end of worksharing regions without the `nowait` clause. OpenMP further allows users to explicitly add a barrier to its parallel application by means of `#pragma omp barrier` directive, which ensures that no thread of a team is allowed to proceed beyond a barrier until all threads in the team have reached that point.

C. Memory Model

OpenMP is based on the relaxed-consistency shared memory model. According to this model, there is a *global shared memory* which any thread may read from or write to data; each OpenMP thread is allowed to have a *local*, *temporary view* of the global shared memory that is accessible to only the reads and writes from that thread [2][3]. Here are more details about the OpenMP memory model:

- A thread's temporary view of memory is not required to be consistent with the shared memory at all times.
- A read from a variable by a thread may not reflect all prior writes from other threads to this variable.
- A write to a variable by a thread is not immediately observable by another one.
- Both reads and writes by a thread may be completed with respect to only that thread's temporary view of memory without any access to shared memory.
- All modifications to the shared data objects by a thread must be written (flushed) back to the shared memory at the synchronization points of the program.

In order to make a thread's temporary view of memory consistent with the global shared memory, OpenMP provides users with `#pragma omp flush` directive. Executing the flush directive causes to write the whole thread-visible data state of the program, as defined by the base language, back to memory and then invalidate it in its temporary view.

III. PARALLEL HARDWARE ARCHITECTURE

Motivated by related studies in the literature, a generic parallel hardware architecture that can be instantiated by an OpenMP program for a class of fork-join parallel applications is proposed in this study and illustrated in Figure 1.

Inside an FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit) chip in Figure 1, there are a few types of components, which include

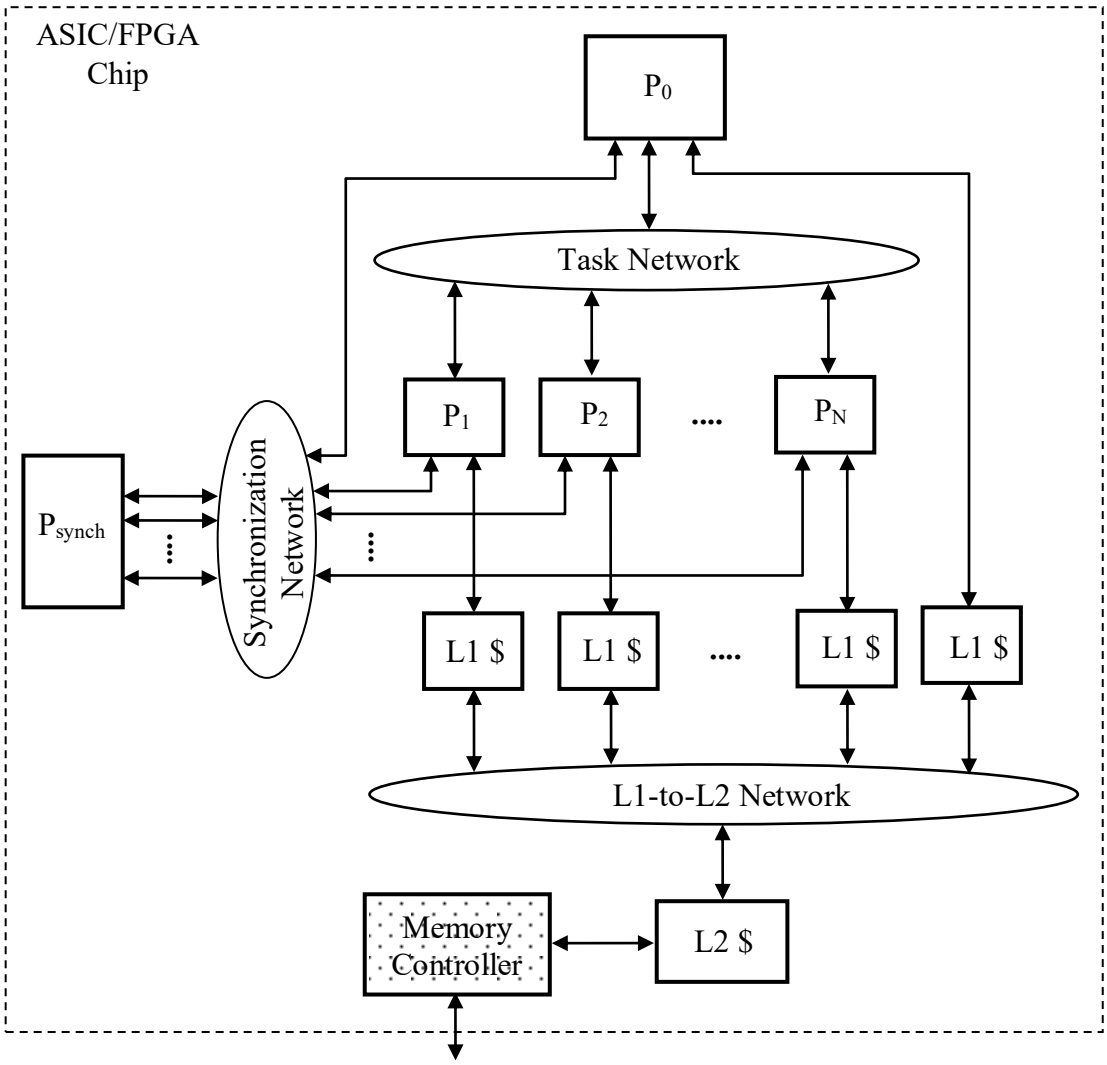

Figure 1. A parallel hardware architecture for fork-join applications

hardware threads, L1 caches (L1 $), a single L2 cache (L2 $), and interconnection networks. A two-way arrow in Figure 1 represents a bidirectional message communication port with sending FIFO (First-In First-Out) and receiving FIFO interfaces, where it can be either master or slave port. That is, a master port has a master sending FIFO to send requests and a master receiving FIFO to receive the corresponding responses; a slave port has a slave receiving FIFO to receive requests and a slave sending FIFO to send the related responses. Thus, a master port of a component is connected to a slave port of another one.

A. Hardware Threads

A hardware thread component is a finite state machine that performs either coordination (P_0 in Figure 1) or computation (P_i, $i>0$), or in some cases, both.

P_0 is the master hardware thread that coordinates/synchronizes the execution of a parallel application among the slave hardware threads P_i, $1 \leq i \leq N$. In Figure 1, P_0 has the following bidirectional ports: a master port to the task network, a master port to its L1 cache, and a master port to the synchronization network, if P_0 also contains a critical or atomic region.

P_0 implements the parallel directive as follows: P_0 forks a team of M slave threads by sending a `start request` with all initial input parameters to every slave thread of the current parallel region through its master send port to the task network. Depending on the number of threads that will be used in different parallel regions, the number of slave threads M is in general a varying number ($M \leq N$ or $M > N$ are both possible), which basically corresponds to `omp_set/get_num_threads` function of OpenMP. In order to perform a join event, P_0 waits until it receives a `finish response` over the task network from each one of the M slave threads of the team at the end of the parallel region. The `finish response` message will be received in the master receive port of P_0. Note that P_0 relies on the number of `finish response` messages received being equal to the number of `start request` messages sent, in order to implement the implicit barrier required at the end of the `parallel` directive.

According to [2], each worksharing region and barrier region must be encountered by all threads in a team or by none at all; the sequence of worksharing regions and barrier

regions encountered must be the same for every thread in a team. Otherwise, such an OpenMP program is considered to be non-conforming, which will lead to unspecified behavior. For a conforming OpenMP program, P_0 plays a role during the implementation of both implicit and explicit barriers (`#pragma omp barrier`).

P_i, $1 \leq i \leq N$, are a team of slave hardware threads that really implements the execution of a parallel application. Each slave thread in Figure 1 has the following bidirectional ports: a slave port to the task network, a master port to its L1 cache, and a master port to the synchronization network. P_i will be designed based on the following principles:

- P_i will be initially in an idle state, waiting for a `start request` message from its master hardware thread P_0.
- Upon receiving the `start request`, each slave thread immediately starts executing its associated task, which is generally realized by a deeply pipelined datapath with a finite state machine.
- The execution of a task may require one or more implicit or explicit barrier requests and one or more other synchronization operations. The details of those synchronization operations will be given shortly.
- A slave task is coupled with a private L1 cache through which it accesses its private and/or shared variables, which will be fetched from the L2 cache on demand.
- Once the computation of a slave thread is completed, the slave thread sends a `finish response` message to P_0 and starts waiting for its next `start request` message.

The static scheduling of any worksharing for loop inside a parallel region will be predetermined at compile time, by means of assigning an approximately equal chunk of loop iterations to each of the slave hardware threads.

In the case of dynamic scheduling of worksharing for a loop, however, P_0 will dynamically assign multiple chunks of loop iterations to slave threads one after another. Such dynamic scheduling can be done with a *load balancing* task network which sends `start request` messages from P_0 to *any* slave thread unit $P_1,...P_N$ that is currently free (whose slave receive port FIFO is not full). For example, implementing the task network as a 1-dimensional or 2-dimensional torus network through which task start requests sent by P_0 travel until they encounter a currently free slave thread, can accomplish such load balancing [13]. When the number M of dynamic tasks is greater than the number N of (physical) slave threads, and the task network is flooded with start requests from P_0, so that it can no longer accept further messages, P_0 will stall until the slave threads complete some of their ongoing work, so that the task network is able to accept start requests again.

On the other hand, *n-level* nested parallelism is possible by enhancing the hardware in Figure 1 as follows:

- A second level nested parallelism can be created by replacing each slave thread P_i by a copy of the part of Figure 1 consisting of the components P_0, the *task_network*, slave threads $P_1,...,P_N$ and their attached L1 caches $L1_1$,…, $L1_N$, while preserving the interconnections between these components. Components in this copy will be renamed respectively as P_i-P_0, P_i-*task_network,* P_i-P_1 … P_i-$P_{N'}$ and P_i-$L1_1$ … Pi-$L1_{N'}$ (the number of slave threads N' within the copy may be different than the original number of slave threads N). In this case, P_i-P_0 (which is the specific component that replaced P_i) will remain connected to the original *task_network* with a slave port and will remain connected to the original L1 cache $L1_i$ of P_i with a master port and will also have an additional master port attached to the new P_i-*task_network*. P_i-P_0 will perform its own computation work. It will also send subthread `start request` messages to the pool of subthread hardware units P_i-P_1 … P_i-$P_{N'}$ and will wait for all invoked subthreads to send back `finish response` messages over the new P_i-*task_network*, before P_i-P_0 itself finally sends back a `finish response` message back to P_0. The components P_i-P_0, P_i-P_1, …, P_i-$P_{N'}$ will also be connected to the synchronization network, so that a subthread running on them can execute critical or atomic regions. Finally, the P_i-$L1_j$ caches for $j \geq 1$ will be directly connected with master ports to the original L1-to-L2 network as well.
- One can repeat the previous step for each level of the nested parallelism. In case the resulting hardware does not fit on a single chip, the hardware can be partitioned into hardware modules interconnected by a scalable network and a semi-reconfigurable ASIC "union module", which can act as any of the partitions based on configuration parameters, can be created, to reduce ASIC NRE costs, as described in [13].

B. Synchronization in Hardware

As explained above, the hardware thread P_0 is used for the realization of an implicit barrier. An OpenMP explicit barrier `#pragma omp barrier` can also be implemented either with any barrier network known in the literature (e.g., for the simpler case where the number of tasks equals the number of physical hardware slave threads), or by using the mechanism already given in Figure 1. That is, a hardware thread P_0' can spawn and wait for completion of task parts before an explicit barrier and then a hardware thread P_0'' can spawn and wait for completion of task parts after the explicit barrier, and a top level hardware thread P_0 can invoke P_0', wait for its completion, and then invoke P_0'', therefore accomplishing the desired barrier synchronization.

In order to implement both critical and atomic directives of OpenMP, on the other hand, a specialized synchronization hardware unit P_{synch} is included in Figure 1. P_{synch} has a number of bidirectional slave ports to the synchronization network, where the number of slave ports n_{synch} depends on the unique synchronization identifiers in application. Note that $n_{synch} = 0$ if P_{synch} is not needed in an application without any synchronization requirement at all; $n_{synch} \leq N$ and $n_{synch} > N$ if there are less than or equal to or more unique synchronization identifiers than the number of slave threads, respectively.

Suppose that every `critical` or `atomic` directive in the application program text is assigned a synchronization identifier *synchID* between 0 and n_{synch}-1. In general, the mapping from atomic or critical constructs to *synchID*'s will be a many-to-one function. However, to enhance concurrency, two distinct critical or atomic constructs in the program text which are known not to access overlapping memory areas can be assigned different synchronization identifiers either by compiler dependence analysis or expressly by the programmer, when the programmer uses different *name* parameters in the related `critical` constructs. In order to get exclusive access to a contended structured block uniquely identified by its *synchID*, a slave thread with *threadID* is first required to send a `synchronization request` with *threadID* and *synchID* to P_{synch} through its master port to the synchronization network. The synchronization network will then route any synchronization request message by using its *synchID* as the destination network output port. P_{synch} will be receiving simultaneous multiple synchronization request messages through its slave ports to the synchronization network. In order to respond these synchronization requests as soon as possible, P_{synch} can also be designed as a parallel hardware architecture as follows:

- A finite state machine with a synchronization status register R_{synch} is assigned to manage each slave port of P_{synch}.
- Initially, R_{synch} is NULL and no thread is granted for the contended structured block.
- If there is a `synchronization request` with *synchID*, one of the following choices is applied:
  - If $R_{synch} = NULL$, let $R_{synch} = threadID$, which corresponds to an acquire event.
  - If $R_{synch} \neq NULL$ and $R_{synch} = threadID$, let $R_{synch} = NULL$, which corresponds to a release event.
  - If $R_{synch} \neq NULL$ and $R_{synch} \neq threadID$, keep R_{synch} unmodified, which ensures that only one thread is granted at any time.
- P_{synch} sends a `synchronization response` with the current value of R_{synch} to the sender thread of the respective synchronization request.

Upon receiving a synchronization response, a slave thread checks if its thread ID is equal to the R_{synch} field of the received response message. If they are equal, it means that its exclusive access has been granted by R_{synch}. At the end of the synchronization point, such a thread must send another `synchronization request` with *threadID* and *synchID* in order to end the period of its exclusive access. Otherwise, a slave thread whose request has been rejected is expected to retry after waiting for a random amount of time.

C. Memory Hierarchy

A two-level on-chip memory hierarchy as shown in Figure 1 is proposed to support the parallel hardware acceleration.

Each hardware thread in Figure 1 is associated with a dedicated, private L1 cache (L1 $) where it keeps its temporary view of the global shared memory. L1 cache has a slave port to its hardware thread and a master port to L1-to-L2 network. L1 cache is a write-back cache that supports conventional *load* and *store* requests coming from slave hardware threads. Furthermore, L1 cache has the following main features:

- L1 cache does not implement any cache coherence protocol, therefore its hardware is simplified.
- L1 cache has a *dirty bit* for each byte of a cache line in order to overcome a false sharing[1] error [15].
- In order to maintain coherence between L1 caches and globally shared L2 cache according to the respective memory model, L1 cache supports `flush_list` and `flush_all` requests.
- A `flush_list address_list request` forces L1 cache to send the cache lines containing any of the given addresses along with their line dirty bits to the L2 cache, invalidate these lines, and return an acknowledgement. Note that the address list may include a single address or multiple addresses.
- A `flush_all request` forces L1 cache to send all dirty cache lines together with their line dirty bits to the L2 cache, invalidate all cache lines, and return an acknowledgement.

The L2 cache is a write-back cache that receives *line read* and *line write* requests from L1 caches and responds to these requests accordingly. All initial and final data of the parallel application are assumed to be kept in the L2 cache. Furthermore, according to Figure 1, the L2 cache state data is held in an on-chip memory, whereas the application data are kept in an off-chip memory accessed through a memory controller. Note that L2 cache has a slave port to L1-to-L2 network and master port to its memory controller.

According to the semantics of OpenMP programs, in addition to the explicit flushes due to the flush directive, a flush operation is implied at several locations in the program as well. The implicit flushes per OpenMP requirement are supported by the proposed hardware accelerator with the help of L1 and L2 caches as follows [2]:

- *Entry to a parallel region*: Before starting the parallel region, P_0 sends a `flush_all request` to its L1 cache. Upon receiving the related acknowledgement, P_0 starts to send a `start request` to each slave thread.

[1] With non-coherent caches, a false sharing error may occur even when two slave threads access non-overlapping memory areas. Assume that a line in L2 contains two data items *a* and *b*. L1 Cache A loads the initial contents of the line and stores *a* into the line. L1 Cache B loads the initial contents of the line and stores *b* into the line. If L1 Cache A flushes the line last, it will incorrectly store the old stale value of *b*. Similarly, if L1 Cache B flushes the line last, it will incorrectly store the stale value of *a*. However, when only the bytes corresponding to the dirty bits of a line are stored back (only *a* from the line from L1 cache A and only *b* from the line from L1 cache B) into L2, this false sharing error is eliminated. Dirty bits per byte can be replaced by dirty bits per 4-byte (or 8-byte) word, when a compiler can determine that a group of L1 caches is accessed with only word accesses.

- *Exit from a parallel region*: Just before a slave thread ends (i.e., reaches the implicit barrier ending the parallel region), it sends a `flush_all request` to its L1 cache. After receiving the respective acknowledgement, the slave thread sends a `finish response` to P_0.
- *Explicit or implicit barrier region*: An explicit barrier can be implemented using an implicit one as described in Section III B first paragraph. Therefore, the L1 cache of a slave thread is flushed just before it reaches any kind of barrier.
- *Exit from a worksharing region without the nowait clause*: Remember that there is an implicit barrier at the end of a worksharing region if there is no `nowait` clause. Thus, this case will also be implemented as an implicit barrier.
- *Entry to an atomic region*: After a slave thread acquires the exclusive right for updating a single shared variable through `synchronization response`, it first sends a `flush_list request` including only the shared variable address to its L1 cache. Upon the acknowledgement of this request, it sends a load request to the cache to obtain the latest value of this variable.
- *Exit from an atomic region*: This will be achieved by first sending a `flush_list request` including the related shared variable to L1 cache, followed by a `synchronization request` with *threadID* and *synchID* to finish its atomic operation.
- *Entry to or exit from a critical region*: These two events are the same as the entry to or exit from an atomic region, except the `flush_list request` will include multiple shared variables instead of a single one.

D. Interconnection Network

In Figure 1, there are three different interconnection networks, namely *task network*, *synchronization network*, and *L1-to-L2 network*. Each of these networks is a packet-based network-on-chip network (NoC) [14] that interconnects various components of the architecture as shown in the figure. For example, the task network can be realized by a 1-to-N forward and N-to-1 backward butterfly networks, whereas the L1-to-L2 network can be implemented by a N-to-1 forward and a 1-to-N backward butterfly networks.

IV. CASE STUDIES

In this section, how different OpenMP code fragments can be compiled into application-specific parallel hardware architectures with respect to Figure 1 will be demonstrated.

A. Matrix-Vector Multiplication

The first case study considers the matrix-vector multiplication of y = A×x, where A is an n×m matrix, x and y denote m×1 and n×1 vectors, respectively. The parallel implementation of the matrix-vector multiplication in OpenMP is given below:

```
#pragma omp parallel num_threads(N) \
   default(shared) private (i,j)
{
 #pragma omp for schedule(static) {
   for (i=0; i<n; i++) {
     y[i] = 0.0;
     for (j=0; j<m; j++)
       y[i] += A[i*m+j]*x[j];
   }
 }
}
```

In Figure 1, this matrix-vector computation will be carried out as follows:

- Each hardware thread P_i, $1 \leq i \leq N$, starts its computation upon receiving a start request from P_0. Note that in this case the number of physical hardware slave threads N is specified by the clause `num_threads(N)`.
- Since OpenMP is directed to assign the iterations of the i-loop to threads in an equal fashion due to the clause `schedule(static)`, each P_i, $1 \leq i \leq N$, computes n/N vector elements y[i], where computing y[i]=A[i,:]×x requires a complete row A[i,:] of the matrix A and the whole vector x.
- The L1 cache ($L1_i$) directly attached to every P_i will be loaded with n/N rows of the matrix and the vector x from the L2 cache on demand during the computation.
- Each P_i computes its n/N part of the y vector and stores this part into its L1 cache.
- At the end of the computation, each P_i sends a flush_all request to $L1_i$ so that all dirty lines of of the y vector in $L1_i$ are written back to the L2 cache.
- Each P_i waits for a flush acknowledgement from $L1_i$, and then sends a `finish response` to P_0. Once P_0 receives N finish responses, the matrix-vector multiplication is completed.
- According to the semantics of OpenMP, there are two implicit barriers, one of which is for the end of the parallel directive, and the other one is for the end of the worksharing for directive. However, a single implicit barrier would be enough for the correct execution of the algorithm. That is why only one implicit barrier that corresponds to the end of the parallel directive is implemented.

Note that the synchronization networks and P_{synch} will not be needed for this example. Thus, the matrix-vector multiplication is realized as a single fork-join paradigm.

B. Vector Inner-Product

The second case study considers the vector inner-product of r = b×x, where b is a 1×n row vector, x denotes an n×1 column vector, and r is a resulting scalar value.

```
r=0.0;
#pragma omp parallel num_threads(N) \
   default(shared)
{
 #pragma omp for reduction(+:r)
   for (i=0; i<n; i++)
     r += b[i]*x[i];
}
```

The parallelization of the vector inner-product can be accomplished within the framework of Figure 1 as follows:

- Upon receiving a start request from P_0, each P_i, $1 \le i \le N$, computes a partial sum scalar value r_i by means of multiplying its exclusive part of n/N elements of vectors b and x, and then performing n/N sums, in pipelined fashion.
- Since each thread needs n/N elements of both vectors, $L1_i$ is loaded with n/N columns of b and n/N rows of x from the L2 cache.
- After the computation of the local r_i is over, each P_i, $1 \le i \le N$, sends a special finish response with the local value of r_i of r (`finish_reduction response`) to P_0.
- Note that P_0 initially sets as r=0.0. For each received `finish_reduction response`, P_0 updates the global value of r with the local one. After P_0 receives N finish responses, P_0 stores the final reduction sum r in cache $L1_0$.
- Finally, P_0 sends a flush request to $L1_0$. With the reception of the respective flush acknowledgement from $L1_0$, the vector inner-product computation is finished.

Once again the synchronization network and P_{synch} component in Figure 1 will not be needed for case B either. Thus, the implementation of a vector-inner product requires a fork-join type of parallel execution with a final reduction operation.

C. Gauss-Seidel Algorithm

Finally, the Gauss-Seidel algorithm is used to iteratively solve differential equations, which based on the finite difference method. A baseline OpenMP implementation of the Gauss-Seidel algorithm [16] is provided in this section:

The parallel implementation of the Gauss-Seidel algorithm is supported by Figure 1 as follows:

- P_0 executes the do-while loop as long as the loop condition is true. During each iteration of the loop, P_0 forks N slave threads in order to simply update the u[i][j] matrix and calculate the new value of dmax.
- Each P_i, $1 \le i \le N$, starts its computation upon receiving a start request from P_0. Each slave thread is assigned a task to update n/N rows of u[i][j] and computes its dmaxL value based on new u[i][j] matrix values.
- $L1_i$ is loaded with the respective n/N rows of the u[i][j] matrix from the L2 cache on demand during computation.
- At the end of each iteration of the i-loop, all N slave threads will contend for the critical section, which requires sending/receiving synchronization requests/responses through the synchronization network.
- P_{synch} will grant access to only one of the slave threads at a given time to update the shared variable dmax. Meanwhile, each slave thread reads the latest value of dmax from the main memory upon entering the critical section and writes the updated value of dmax back to main memory before exiting the critical section.
- At the end of the computation, each P_i sends a `flush_all` request to $L1_i$ so that all dirty lines of u[i][j] in $L1_i$ are written back to the L2 cache. After receiving its flush acknowledgement from $L1_i$, slave thread sends a `finish response` to P_0.
- Once P_0 receives N finish responses, P_0 checks if the loop condition is true. If it is true, it will repeat the loop as explained above. Otherwise, the Gauss-Seidel algorithm has converged, and the algorithm is completed.

```
do {
 dmax=0.0;
 #pragma omp parallel num_threads(N)\
   default(shared)
 {
  #pragma omp for private(temp,d,dmaxL){
    for(i=1; i<n+1; i++) {
      dmaxL=0.0;
      for(j=1; j<n+1; j++) {
        temp=u[i][j];
        u[i][j]= ....
        d=fabs(temp-u[i][j]);
        if(dmaxL<d) dmaxL=d;
      }
      #pragma omp critical
        if (dmax<dmaxL) dmax=dmaxL;
    }
  }
 }
} while (dmax>eps);
```

V. CONCLUSIONS

A parallel hardware architecture for a class of parallel applications that can be modeled by a fork-join programming model adopted by OpenMP is introduced. Its features are further highlighted on three different case studies. The proposed parallel hardware architecture has several important features implemented purely on hardware that are not typically supported by other studies in the literature, such as an L1 data cache for each hardware thread, n-level nested parallelism, and dynamic scheduling of worksharing for loops.

Future work involves devising a compiler to generate such parallel hardware from regular OpenMP applications; measuring and reporting the performance that can be attainable by the generated parallel hardware using a set of benchmark OpenMP applications, and making this compiler to support most of OpenMP constructs.

REFERENCES

[1] A. Doğan, İ. San, and K.Ebcioğlu, "A parallel hardware architecture for fork-join parallel applications," The Eighth International Conference on Advanced Communications and Computations, (INFOCOMP 2018), IARIA Press, July 2018, pp. 57-59.

[2] OpenMP Application Programming Interface, Version 5.0, November 2018.

[3] B. Chapman, G. Jost, R. van der Pas, Using OpenMP Portable Shared Memory Parallel Programming. London, UK: The MIT Press, 2008.

[4] Y. Y. Leow, C. Y. Ng, and W.F. Wong, "Generating hardware from OpenMP programs," IEEE International Conference on Field Programmable Technology, (FPT 2006), IEEE Press, Dec. 2006, pp. 73-80, doi: 10.1109/FPT.2006.270297.

[5] A. Podobas, "Accelerating Parallel Computations with OpenMP-driven System-on-Chip Generation for FPGAs," IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, IEEE Press, Sept. 2014, pp 149-156, doi: 10.1109/MCSoC.2014.30.

[6] A. Podobas and M. Brorsson, "Empowering OpenMP with automatically generated hardware," International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), IEEE Press, Jul. 2016, pp. 201-205, doi: 10.1109/SAMOS.2016.7818354.

[7] J. Choi, St. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," International Conference on Field-Programmable Technology (FPT'13), IEEE Press, Dec. 2013, pp. 270-277, doi: 10.1109/FPT.2013.6718365.

[8] A. Cilardo, L. Gallo, and N. Mazzocca, "Design space exploration for high-level synthesis of multi-threaded applications," Journal of Systems Architecture, vol. 59, pp. 1171-1183, Nov. 2013, doi: 10.1016/j.sysarc.2013.08.005.

[9] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators," 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017), IEEE Press, Jul. 2017, pp. 201-205, doi: 10.1109/ASAP.2017.7995280.

[10] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, D. J.-Gonzalez, "OpenMP extensions for FPGA Accelerators," International Symposium on Systems, Architectures, Modeling, and Simulation, IEEE Press, Jul. 2009, pp. 17-24, doi: 10.1109/ICSAMOS.2009.5289237.

[11] Xilinx SDAccel Programmers Guide. [Online]. Available fromhttps://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1277-sdaccel-programmers-guide.pdf 2019/02/22.

[12] Intel® FPGA SDK for OpenCLTM Pro Edition Programming Guide [Online]. Available from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf 2019/02/22.

[13] K. Ebcioglu, E. Kultursay, and M. T. Kandemir, "Method and system for converting a single-threaded software program into an application-specific supercomputer," US patent 8,966,457, filed 2011/11/15 issued 2015/02/24.

[14] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," ACM Computing Surveys, vol. 38, pp. Jun. 2006, doi: 10.1145/1132952.1132953.

[15] E. Kultursay, K. Ebcioglu, "Storage Unsharing", US patent 8,825,982, filed 2011/06/09 issued 2014/09/02.

[16] Parallel Methods for Partial Differential Equations [Online]. Available from http://www.hpcc.unn.ru/mskurs/ENG/PPT/pp12.pdf 2019/02/22.