Defining Synthesizable OpenMP Directives and Clauses

P. Dziurzanski and V. Beletskyy

Faculty of Computer Science and Information Systems,
Technical University of Szczecin, Zolnierska 49, 71-210 Szczecin, POLAND
{pdziurzanski,vbeletskyy}@wi.ps.pl

Abstract. Possibilities of synthesizing parallel C/C++ codes into hardware are presented provided that the code parallelism is represented by means of the directives of OpenMP, a de-facto standard that specifies portable implementation of shared memory parallel programs. The limitations of the hardware realizations of OpenMP directives are described and implementation details are stressed.

1 Introduction

Sources in different hardware description languages (HDLs) are used as input to behavioral synthesis. The most commonly used languages are VHDL and Verilog, but since designers often write system level models using programming languages, using software languages are of mounting importance. Applying software languages makes easier, performing SW/HW cosynthesis, which accelerates the design process and improves the flexibility of the software/hardware migration. Moreover, the system performance estimation and verification of the functional correctness is easier and software languages offer fast simulation and a sufficient amount of legacy code and libraries which facilitate the task of system modelling.

To implement parts of the design modelled in C/C++ in hardware using synthesis tools, designers must present these parts into a synthesizable subset of HDL, which then is synthesized into a logic netlist.

C/C++ defines sequential processes (procedures), whereas HDLs processes are run in parallel. Consequently, in order to synthesize hardware of high performance, there is the need of establishing groups of C/C++ functions which can be executed in parallel. To achieve the expected efficiency, each C/C++ function has to be treated as a separate process. This allows all functions to be executed concurrently taking into account that there exist no data dependences between functions. Otherwise, data synchronization techniques have to be utilized. Blocking actions before accessing to a shared variable can be implemented as a *wait* statement, which can be left when a synchronization signal from other module is set.

In order to control the calculations executed by the entities synthesized from C/C++ functions, distributed or centralized control schemes can be applied. They require that each entity includes both functional logic and a local controller. In the distributed scheme, this controller is connected with other blocks in a form of additional input and output ports. The input port determines whether all the preceding entities in a control flow have finished their tasks. After execution, the controller informs all the following entities

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 398–407, 2004.

in the control flow about finishing its task. In the centralized scheme, one controller determines which entities are executed at any given time.

More details on communication schemes are given in [7].

In this paper, we consider the following two-stage approach: (i) parallelizing an C/C++ code and presenting parallelism by means of OpenMP directives, (ii) transforming the C/C++ code with OpenMP pragmas into an HDL code.

We focus on the second stage, i.e., on transforming a parallelized C/C++ code into an HDL code and assume that an input code includes OpenMP directives representing C/C++ code parallelism.

2 Related Works

Recently, there has been a lot of work in the use of the C programming language to generate hardware implementations. However, as the C and C++ languages do not allow us to describe parallelism directly, the aspect of treating concurrency is of great importance in high-level synthesis approaches.

For example, in Transmogrifier C compiler [3], there exists only one thread of control. In order to compile multiple threads, one have to compile each thread separately into a netlist and then use the I/O ports for communication.

The PACT HDL compiler does not perform data dependency analysis and thus does not support function replication and parallel execution [4].

Despite the systems, described in [5,6], perform their own analysis of data dependency, they do not allow the designer to indicate parallel regions. Consequently, the information about parallelism known before compilation is ignored.

To our knowledge, there is no publications which describe hardware implementations of C/C++ code with standard OpenMP directives permitting us to present algorithm parallelism.

3 Principles of OpenMP

OpenMP [1] is a de-facto standard that specifies portable implementation of shared memory parallel programs. It includes a set of compiler directives, runtime library functions supporting shared memory parallelism in the C/C++ and Fortran languages.

OpenMP is based on the fork-and-join execution model in which a program is initialized as a single process (master thread). This thread is executed sequentially until the first parallel construct is encountered. Then, the master thread creates a team of threads that executes the statements concurrently. There is an implicit synchronization at the end of the parallel region, after which only the main thread continues its execution.

4 Influence of OpenMP Pragmas on Behavioral Synthesis

Similarly to C/C++ clauses, directives from the OpenMP standard [1] can be split into synthesizable and nonsynthesizable subsets. Certain OpenMP constructs have no equivalence in hardware realizations, whereas others can lead to hardware utilizing the enormous amount of resources.

```
if omp_get_dynamic() is equal to false {
  if num_thread clause is present
    return the parameter of the clause
  else if omp_set_num_threads function has been called
    return omp_get_num_threads()
  else if OMP_NUM_THREADS is defined
    return OMP_NUM_THREADS}
return omp_get_max_threads()
```

Fig. 1. Algorithm for determining the number of instances, NoOfInstances

In order to follow our results, the reader should be familiar with OpenMP attributes and environment variables (this knowledge is comprised in [1].)

4.1 Parallel Constructs

A parallel region is a region which can be executed by multiple threads concurrently.

The thread number, which corresponds to the number of multiple instances of elements in hardware implementations, can be determined statically or dynamically.

In the case of the statical thread number, to determine the number of requested threads, the **num_threads** clause, or the **omp_set_num_threads** OpenMP run-time function, or the **OMP_NUM_THREADS** environment variable can be used. In order to achieve an expected efficiency in hardware realizations, multiple threads should be implemented as multiple instances of the hardware realizing the same functionality. Consequently, it is requested that the number of threads is evaluated to a constant value during synthesis.

If a dynamic adjustment of the thread number is chosen, the number of threads is set to the maximum possible value, returned by the **omp_get_max_threads** OpenMP run-time function (described in section 4.5), unless the synthesizer determines that the maximum number of iterations is less than this value or a synthesized circuit does not fit into the destined chip.

The algorithm for determining the number of instances of parallel constructs is given in Fig. 1.

Although synthesis from nested parallel regions can lead to hardware with a very large area, an implementation of nested parallelism may improve the performance greatly [2]. Thus, nested parallelism should not be prohibited in hardware realizations.

The parallel clause can be modified with the following clauses **if**, **private**, **firstprivate**, **default**, **shared**, **copyin**, **reduction**, and **num_threads**. The majority of them are considered in corresponding sections below. In this subsection, only the clauses that are not used with work-sharing constructs are described.

In OpenMP, parallel execution can be conditional when an **if** clause is present. This clause usually defines a *breaking condition*. If this condition is not satisfied, a data dependency appears and the section cannot be run concurrently. If the condition can be evaluated during synthesis, then a synthesizer creates either single or multiple instances of hardware realizing the corresponding section body, according to the value of the condition. Otherwise, the decision on sequential or parallel execution is postponed up to runtime by the transformation which moves the condition from an OpenMP directive

```
#pragma omp parallel
if(condition)
// parallel section body
```

(a)

```
if(condition) {
  #pragma omp parallel
  // parallel section body}
else
  // sequential section body
```

(b)

Fig. 2. Replacing the OpenMP if clause (a) with the C code condition (b)

if **if(condition)** *clause is present in an OpenMP* **parallel** *directive* {
if the condition is evaluated to the zero value {
remove the corresponding OpenMP **parallel** *directive*}
else if the condition is evaluated to a nonzero value {
remove the **if(condition)** *clause from the corresponding OpenMP* **parallel** *directive* }
else if the condition cannot be evaluated during synthesis {
move the condition from an OpenMP directive to a regular C/C++ code
provide parallel and serial versions of the parallel section body (Fig. 2) } }

Fig. 3. Algorithm for the transformation of a conditional parallel construct

to a regular C/C++ code, as shown in Fig. 2. The algorithm for the conditional parallel construct transformation is given in Fig. 3.

The data sharing of variables is specified at the beginning of a parallel block using the **shared** or **private** clauses.

The **shared** clause indicates that the following variables which this clause defines are shared by each thread in a team. In the hardware realization, this may cause problems with global routing, as a number of hardware elements are to be connected to registers keeping one variable. On the other hand, this can decrease the number of cells in the hardware.

The **copyin** clause provides a mechanism for assigning the same value to multiple instances of one variable, each in one thread executing a parallel region. Each instance of the private variables are to be connected (by routing channels) with the hardware implementing the value of the variable instance from the instance which corresponds to the main thread.

The **default** clause informs whether the default data-sharing attributes of variables are **shared**. It is treated as a preprocessor command, which determines either shared or private variables and hence can be used in synthesis.

4.2 Work-Sharing Construct Clauses

OpenMP defines three work-sharing directives: **for**, **sections**, and **single**.

The **for** directive indicates that the iterations of the associated loop should be executed in parallel. In order to execute the loop iterations in parallel, the number of iterations has to be evaluated at the synthesis stage. The algorithm for the synthesizing of the **for** loops is presented in Fig. 4.

If the number of the loop iterations is not greater than the value $NumberOf\text{-}Instances$, returned by the NoOfInstances algorithm (Fig. 1), then all the loop iterations are executed in parallel. Otherwise, the loop either can be unrolled so as the

```
if a number of iterations can be evaluated during synthesis {
  NumberOfInstances = return value of the NoOfInstances algorithm
  if NumberOfInstances is greater than the number of iterations
    NumberOfInstances=number of iterations;
  if private clause is present
    create one instance of each private variable for each occurrence
    of parallel realization
  if firstprivate clause is present
    create one instance of each private variable for each occurrence
    of parallel realization which initialize their values
    with the value of the variable original object
  if master and synchronization directive is present
    create appropriate synchronization links between hardware
    modules (algorithm MasterSynchr)
  Compute IterPerInst, the number of iterations which should be executed
  by one instance by rounding up the number of iteration
  divided by NumberOfInstances and Synthesize the loop body
  with NumberOfInstances instances each executing IterPerInst iterations;
  if ordered clause is present
    create synchronization links between hardware modules
    to execute bodies in the appropriate order
  if lastprivate clause is present
    create hardware elements for coping the value of private variable in the last
    instance to the value of the variable original object
  if nowait clause is not present
    create hardware elements for waiting on finishing execution
    of each instance }
else synthesize the loop body with a single instance;
```

Fig. 4. Algorithm for synthesis of the **for** clause, MasterSynchr

number of the loop iterations is equal to $NumberOfInstances$ or the loop can be split into $NumberOfInstances$ independent loops.

According to the OpenMP specification, the execution of each iteration must not be terminated by a **break** statement and the value of the loop control expressions must be the same for all iterations.

With the **for** directive, the following clauses can be used:

- **private** indicates that the variables which this clause defines are private to each thread in a team. In the hardware realization, it usually results in multiple instances of hardware holding the variable.
- **firstprivate** (**lastprivate**) provides a superset of the **private** clause functionality, which causes that the initial (final) value of the new private variables is copied from (to) the value of the object existing prior to the section. Their hardware synthesis conditions are similar to those of the **private** clause.
- **reduction** which calculates one scalar value by performing a given operation on all elements of the vector. It is synthesizeable as the loop with a reduction variable can be split into $NumberOfVariables$ independent loops and an additional loop forming the final value of the reduction variable.

if **private** *clause is present*
create one instance of each private variable for each section
if **firstprivate** *clause is present*
create one instance of each private variable for each section
of parallel realization which initialize their values
with the value of the variable original object
Synthesize each section as a separate entity;
if **lastprivate** *clause is present*
create hardware elements for coping the value of private variable in the last
instance to the value of the variable original object
if **reduction** *clause is present*
create hardware elements for performing the reduction operation and copy its result
to the value of the variable original object
if **nowait** *clause is not present*
create hardware elements for waiting on finishing execution
of each instance

Fig. 5. Algorithm for synthesis of the **sections** clause

if **private** *clause is present*
create an instance of each private variable for the single region
if **firstprivate** *clause is present*
create an instance of each private variable for the single region
which initialize their values with the value of the variable original object
Synthesize a body as a single instance;
if **nowait** *clause is not present*
create hardware elements for waiting on finishing execution of each occurence
of parallel realization
if **copyprivate** *clause is present*
copy the value of the private variables to corresponding private
variables of other instances of the parallel region

Fig. 6. Algorithm for synthesis of the **single** clause

- **ordered** which causes the following region that it defines to be executed in the order in which iterations are executed in a sequential loop. It is considered in Section 4.4.
- **schedule** specifies dividing threads into teams. According to the OpenMP specification, the correctness of a program must not depend on scheduling, so it can be ignored in hardware realizations because it defines schedules for executing the loop iterations on a numbers of sequential processors and is not useful for hardware synthesis.
- **nowait** which causes that there is no barrier at the end of the parallel section. It is synthesizable by eliminating the need of adding the hardware for synchronization at the end of the parallel region.

The **sections** directive indicates that the corresponding region includes separate sections (declared with the **section** pragma) which can be executed in parallel. In hardware realization, for each section, separate instances should be synthesized. With this directive, the clauses **private**, **firstprivate**, **lastprivate**, **reduction**, and **nowait** can be used

with the same limitations as in the case of the **for** directive. The algorithm of the **sections** synthesis is presented in Fig. 5.

The **single** directive indicates that the following structured block should be executed by only one thread in the team. Consequently, it is realized by the single instantiation of hardware realizing the functionality of the block. The clauses **private**, **firstprivate**, and **nowait** can be used with the same limitations as in the case of the **for** directive. The **copyprivate** clause provides a mechanism for broadcasting values between threads of one team with their private variables. The **single** directive can be synthesized by implementing the hardware realizing the corresponding structured block in only one (arbitrary) instance of the hardware realizing the corresponding parallel region and synchronizing its execution with finishing the execution of functionality given prior to the single directive.

4.3 Data Environment Directives

In OpenMP, there are the following clauses for controlling the data environment in parallel regions

- the **threadprivate** directive for making file-scope, namespace-scope, or static-block scope enumerated variables local to a thread, so that each thread obtains its own copy of the common block. In the hardware realization, the common region is synthesized in multiple units.
- the **private**, **firstprivate**, **lastprivate**, **shared**, **default**, **reduction**, **copyin**, and **copyprivate**; their synthesis is described in the previous section.

4.4 Master and Synchronization Directives

Within the parallel section, the master and synchronization directives can be used. They change the standard execution flow and are as below:

- **master** specifies a block that is executed by the master thread of the team. In hardware, the functionality of this block is implemented in the only instance of the hardware, which corresponds to the master thread. Its implementation resembles the **single** construct one, synchronization is required only in the instance of the hardware which corresponds to the master thread.
- **critical** specifies a block that can be executed only by one process at a time. It is synthesizable by means of the hardware which allows only one entity to execute a given code at once.
- **barrier** synchronizes all the threads in a team. It is synthesizable by means of hardware which allows execution only when each instantiation of the parallel region terminates the functionality given prior to this clause.
- **atomic** specifies that a memory location is updated atomically. It can be synthesized by being treated as the **critical** construct.
- **flush** specifies a sequence point at which all the threads in a time are required to have a consistent view of specified objects in memory. In software realization, it results in copying the values from registers into memory or flushing write buffers. In hardware, this directive should be ignored, as such an incoherence is not permitted.

if **master** *clause is present*
create hardware implementation only in the first
occurrence of parallel realization
if **critical** *or* **atomic** *clause is present*
create hardware implementation of the mutual exclusion
and connect it with all the occurrences of parallel realization
if **barrier** *clause is present*
create hardware implementation of the barrier
and connect it with all the occurrences of parallel realization
if **ordered** *clause is present*
create synchronization links between hardware modules
to execute bodies in the appropriate order

Fig. 7. Algorithm for synthesis of the master and synchronization directives

– **ordered** causes that iterations in the parallel block are executed in the same order as in a sequential loop. The functionality is synthesized in each instance implementing the parallel region, but only execution in the first instance is not blocked. In each other instance, the execution is blocked until the previous instance finishes the execution of the corresponding block. This construct adds one synchronization wire between adjacent implementation and the synchronizing hardware.

The algorithm for parallelizing the mentioned above directives is presented in Fig. 7.

4.5 Run-Time Library Functions

In OpenMP, there exist the following execution environment functions:

– **omp_set_num_threads** sets the number of threads used during execution. In hardware realization, its value determines the maximal number of the hardware instances realizing parallel regions. This function is synthesizable as long as its parameter can be computed during the compilation process.
– **omp_get_num_threads** sets the number of threads used during execution. The value set by this function is used in the *NoOfInstances* algorithm.
– **omp_get_max_threads** returns an integer that is at least as large as the number of threads executing the parallel region. In hardware realization, it is equal to the maximal possible number of the instances realizing the parallel region. The return value of this function is substituted during the synthesis stage.
– **omp_get_thread_num**. This function is substituted during the synthesis stage with the value equal to the index of the instance realizing the parallel region.
– **omp_get_num_procs**. This function is substituted during the synthesis stage with the value equal to **omp_get_max_threads**.
– **omp_in_parallel** returns a nonzero value if it is called within the parallel region and 0 otherwise. This value is substituted during the synthesis stage.
– **omp_set_dynamic** enables or disables a dynamic adjustment of the number of threads executing a parallel region. In hardware realizations, this function should be treated as a preprocessor command which switches between the permission and the prohibition of the dynamic adjustment of the thread number.

- **omp_get_dynamic** returns a nonzero value if dynamic adjustment of threads is enabled and 0 otherwise. The return value of this function is substituted during the synthesis stage.
- **omp_set_nested** enables or disables the nested parallelism. In hardware realizations, this function should be treated a a preprocessor command, which switches between the permission and the prohibition of the nested parallelism during the synthesis state.
- **omp_get_nested** returns the value according to the actual state of nested parallelism enabled. The return value of this function is substituted during the synthesis stage.

In OpenMP, there exist the following lock functions: **omp_init_lock** initializes a lock, **omp_destroy_lock** uninitializes a lock, **omp_set_lock** blocks the thread as long as the given lock is available and then sets the lock, **omp_unset_lock** releases a lock, and **omp_test_lock** attempts to set a lock without blocking the thread. All of them can be synthesized in hardware.

Nested versions of the lock functions (**omp_init_nest_lock**, **omp_destroy_nest_lock**, **omp_unset_nest_lock**, **omp_test_nest_lock**, and **omp_set_nest_lock**) are similar to their plain versions, except that they are used for nested locks.

The usage of the timing routines **omp_get_wtime** and **omp_get_wtick** in hardware realizations are pointless and thus they should not be permitted.

Table 1. Synthesizable (a), ignored (b) and nonsynthesizable (c) OpenMP constructs

atomic construct	barrier directive	copyin attribute clause
copyprivate attribute clause	critical construct	default attribute clause
firstprivate attribute clause	for construct	lastprivate attribute clause
if clause which cannot be evaluated during synthesis	if clause evaluating to a zero value during synthesis	master construct
		nowait clause
omp_destroy_lock function	omp_destroy_nested_lock function	omp_get_dynamic function
omp_get_nested function	omp_get_num_procs function	omp_get_num_threads function
omp_get_thread_num function	omp_in_parallel function	omp_init_lock function
omp_init_nested_lock function	omp_set_dynamic function	omp_set_lock function
omp_set_max_threads function	omp_set_nested function	omp_set_nested_lock function
omp_set_num_threads function	omp_test_lock function	omp_test_nested_lock function
omp_unset_lock function	omp_unset_nested_lock function	ordered construct
parallel construct	parallel for construct	parallel sections construct
private attribute clause	reduction attribute clause	sections construct
shared attribute clause	single construct	threadprivate directive

(a)

if clause evaluating to a nonzero value during synthesis	flush directive	schedule clause

(b)

omp_get_wtick function	omp_get_wtime function

(c)

5 Conclusion

In this paper, we have presented the possibilities and limitations of the hardware realization of the OpenMP directives.

In Table 1a-c, we summarize the synthesizabilty of OpenMP directives. The constructs which can be synthesized into hardware, are given in Table 1a. The constructs enumerated in Table 1b are not relevant to synthesis and identified as **ignored**. These constructs result in displaying warnings during synthesis.

Finally, not synthesizable constructs are given in Table 1c. If a not supported construct is encountered in a program source, its synthesis is not possible.

We plan to build a compiler transforming C/C++ sources with OpenMP pragmas into synthesizable SystemC sources.

References

1. OpenMP C and C++ Application Program Interface, ver 2.0, OpenMP Architecture Review Board, 2002, *www.openmp.org*
2. R. Blikberg, T. Sorevik, Nested Parallelism: Allocation of Processors to Tasks and OpenMP Implementations, *Second European Workshop on OpenMP*, Edinburgh, Scotland, UK, 2000
3. D. Galloway, The Transmogrifier C hardware description language and compiler for FPGAs, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1995
4. A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, P. Banerjee, PACT HDL: A C Compiler with Power and Performance Optimizations, *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Grenoble, France, 2002
5. J.B. Peterson, R. Brendan O'Connor, P. M. Athanas, Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures, *IEEE Symposium on FPGAs for Custom Configurable Computing Machines*, Napa, California, 1996
6. J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe, Parallelizing Applications into Silicon, *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, USA, 1999, pp. 70-80
7. R. Kastner, M. Sarrafzadeh, Incorporating Hardware Synthesis into a System Compiler, Technical Report, *Department of Electrical and Computer Engineering University of California*, Santa Barbara, CA, USA