Piper: Experimental Support for Parallel Pipelines in Intel® Cilk™ Plus

Reference Guide, Version 1.0

Jim Sukha

Intel Corporation

10/8/2013

This document describes a pipe-while loop, a proposed new parallel loop construct in Intel® Cilk™ Plus based on recent research into pipeline parallelism [2]. A pipe-while loop is a generalization of an ordinary while loop that allows for pipeline parallelism between iterations. Existing Intel Cilk Plus keywords are designed to encode only fork-join parallelism. Pipe-while loops extend the language by enabling programmers to code parallel computations with a pipeline grid dependency structure. This document (1) gives an overview of the proposed pipe-while loop construct using some motivating examples, (2) describes in greater detail the proposed semantics of a pipe-while loop, and (3) presents Piper, an experimental prototype of Intel Cilk Plus that provides runtime support for pipe-while loops.

Overview: Pipe-While Loops

As described in [2], a **pipe-while** loop is a generalization of an ordinary while loop that allows for pipeline parallelism between iterations. This document describes a proposed new pipe-while loop construct for Intel® Cilk™ Plus, which is used as follows:

1. A programmer specifies a pipe-while loop using the **`cilk_pipe_while`** keyword. Each **iteration** of the pipe-while loop is divided into **stages** using the **`cilk_stage`** and **`cilk_stage_wait`** keywords. Loop iterations may execute in parallel, possibly subject to additional dependency constraints between corresponding stages of consecutive iterations.
2. Each iteration of the pipe-while loop begins executing in stage 0. Stage 0 of an iteration cannot start until stage 0 of the previous iteration has completed.
3. A statement **`cilk_stage(s)`** in an iteration advances the iteration to stage **`s`**.
4. A statement **`cilk_stage_wait(s)`** in an iteration also advances the iteration to stage **`s`**, but this stage **`s`** can begin executing only after stage **`s`** of the previous iteration has completed. Thus, a **`cilk_stage_wait`** creates a dependency on the corresponding stage from the previous iteration.

Existing Intel Cilk Plus constructs, namely **`cilk_for`**, **`cilk_spawn`**, and **`cilk_sync`**, are designed to express only parallel computations with fork-join dependencies. Pipe-while loops extend the language and enable users to code parallel computations with a pipeline grid dependency structure.

One simple use for a pipe-while loop is for constructing simple static pipelines, similar to the parallel pipeline construct in Intel® Threading Building Blocks (Intel® TBB). For example, the pipe-while loop in Figure 1 illustrates a simple 5-stage SPSPS pipeline. Stages 0, 2, and 4 are serial stages (S-stages), meaning that these stages must process inputs from iterations in order. In contrast, stages 1 and 3 are parallel stages (P-stages), meaning that there is no dependency between corresponding stages of different iterations. Figure 2 shows the dag (directed acyclic graph) for the SPSPS pipeline in Figure 1. The pipeline loop in Figure 1 may be sped up by a parallel execution if the work of the parallel stages is generally much larger than the work of the serial stages.

```
bool done = false;
int iter_counter = 0;
cilk_pipe_while(!done) { // Each iteration starts executing in Stage 0.
    int i = iter_counter++;
    done = stage0(i);
    cilk_stage(1);       // Advance to Stage 1 (parallel stage)
    stage1(i);
    cilk_stage_wait(2);  // Advance to Stage 2 (serial stage)
    stage2(i);
    cilk_stage(3);       // Advance to Stage 3 (parallel stage)
    stage3(i);
    cilk_stage_wait(4);  // Advance to Stage 4 (serial stage)
    stage4(i);
}
```

Figure 1: A pipe-while loop for a simple 5-stage SPSPS pipeline. The `cilk_pipe_while` keyword indicates the loop is a pipe-while. In this example, the `cilk_stage` keyword is used to start each parallel stage. The `cilk_stage_wait` keyword is used to start each serial stage except stage 0, which starts when the iteration begins and is always serial.

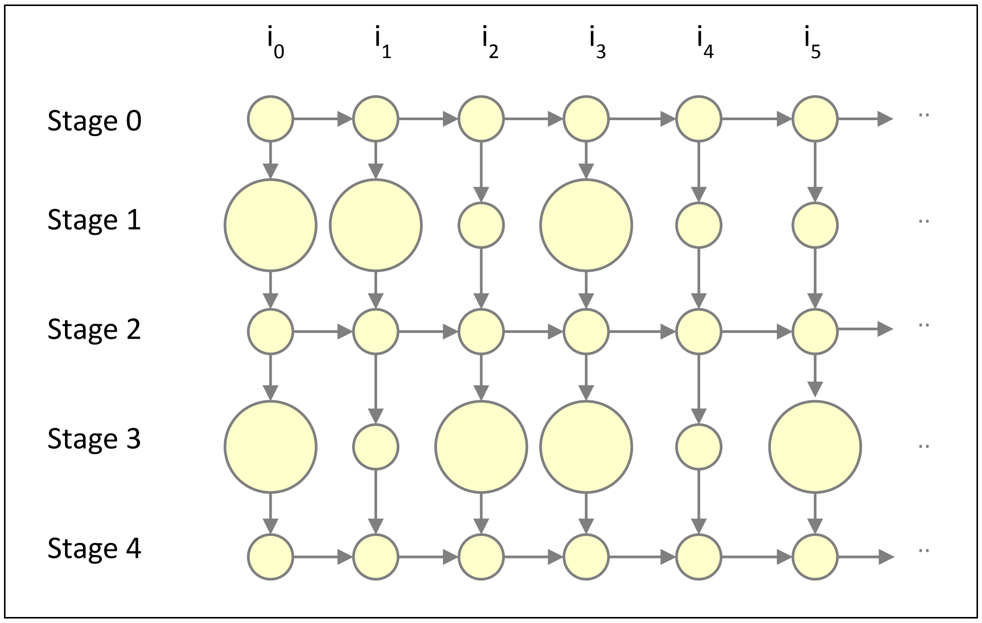

Figure 2: Pipeline dag for the SPSPS pipeline from Figure 1. Each column of circles in the grid represents a single iteration, and each circle represents a single stage in that iteration.

The PARSEC benchmark `dedup` [1] is another example of a static pipeline that can be coded using a pipe-while loop, as shown by the pseudocode in Figure 3. The dedup benchmark is an "SSPS" pipeline, where only stage 2 can execute in parallel.

```
int fd_out = open_output_file();
bool done = false;
cilk_pipe_while(!done) {  // Iteration starts executing in stage 0.
      chunk_t *chunk = get_next_chunk();
      if (chunk == NULL) { done = true; }
      else {
            // Advance to stage 1, but don't start stage 1 on
            // iteration i until stage 1 on iteration i-1 completes.
            // Stage 1 is "serial".
            cilk_stage_wait(1);
            chunk->is_dup = deduplicate(chunk);
            // Advance to stage 2, without having a dependency on
            // the previous iteration.  Stage 2 is "parallel".
            cilk_stage(2);
            if (!chunk->is_dup)
                 compress(chunk)
            // Stage 3.  Like stage 1, this stage is "serial".
            cilk_stage_wait(3);
            write_to_file(fd_out, chunk);
      }
}
```

Figure 3: Pseudocode for `dedup` PARSEC benchmark, written using a pipe-while loop.

Unlike pipelines in Intel TBB, however, the proposed pipe-while loop can be used to construct more complex pipelines where the number of stages or even the dependencies between stages may be specified dynamically. For example, a pipe-while loop can be used to pipeline a computation on a 2-d grid **`n`** by **`m`** grid, where the calculation of a cell **`[i, j]`** depends on the neighboring cells **`[i-1, j]`**, **`[i, j-1]`**, and **`[i-1, j-1].`** This computation has a dependency structure which is captured by the grid dag shown in Figure 4.

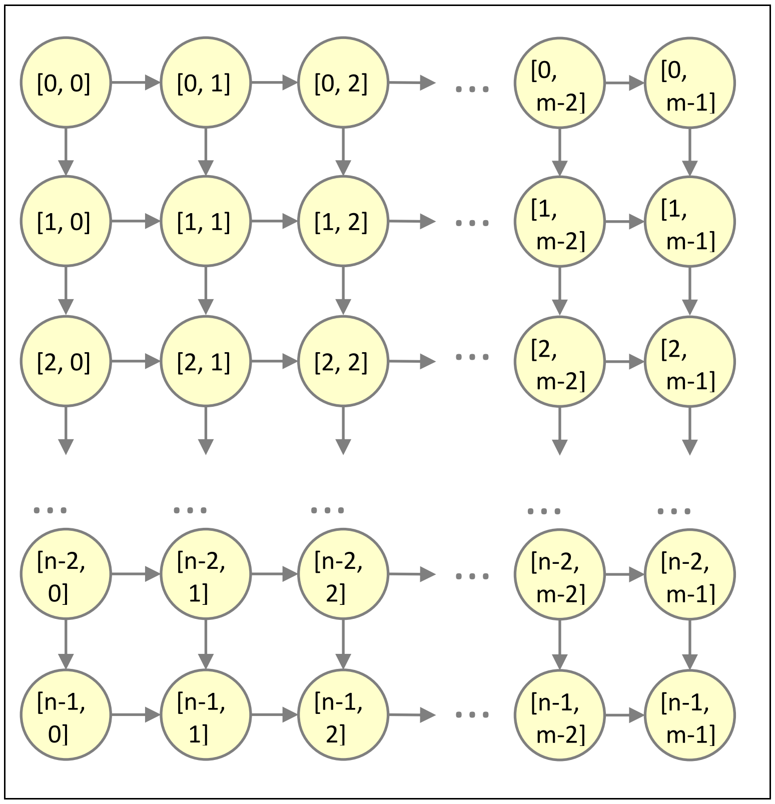

Figure 4: Grid dag for computing a dynamic program on a 2-d `n` by `m` grid.

One way to pipeline the computation shown in Figure 4 is to have each pipeline iteration process a column of the grid, and have the stages process individual cells in each column. The pipe-while loop for this computation is shown in Figure 5 (a). In this example, the number of stages in an iteration depends on **`n`**, the number of rows in the grid. Figure 5 (b) shows a blocked version of this computation, where each stage executes a block of the original grid. The blocked version in (b) has less theoretical parallelism than the simple loop in (a), but it is likely to be more efficient in practice than (a) because the parallelism is more coarse-grained and it incurs less overhead per pipeline iteration or stage.

```
// (a) Simple execution of an n by m grid.
int j_loop = 0;
cilk_pipe_while(j_loop < m) {
    int j = j_loop++;
    for (int i = 0; i < n; ++i) {
        compute(i, j);
        cilk_stage_wait(i+1);
    }
}

// (b) Blocked execution of the n by m grid.
in block_j_loop = 0;
cilk_pipe_while(block_j_loop < m) {
   int block_j = block_j_loop;
   int end_j = min(block_j+Bm, m);
   block_j_loop += Bm;
   for (int block_i=0; block_i<n; block_i+=Bn) {
      int end_i = min(block_i+Bn, n);
      // Process block.
      for (int j=block_j; j<end_j; ++j) {
         for (int i=block_i; i<end_i; ++i) {
            compute(i, j);
         }
      }
      // Stage numbers in an iteration need not be consecutive.
      cilk_stage_wait(block_i+Bn);
   }
}
```

Figure 5: Pipeline loops to execute the dynamic program with dag shown in Figure 4.

Finally, one can use pipe-while loops to construct complex pipeline dags where iterations have a variable number of stages, iterations skip some stages (i.e., have stages which are implicitly empty), and where iterations specify their dependencies on the previous iteration dynamically, as they are executing.

For example, Lee et al. in [2] describe the parallelization of an x264 encoder using a pipe-while loop, which utilizes the dynamic behavior of a pipe-while loop. The pipeline dag for this x264 encoder is shown in Figure 6. In this example, the number of (nonempty) stages in an iteration is proportional to the number of rows in the frame being encoded. An iteration processing a P-frame has dependencies on the previous iteration, and thus uses **`cilk_stage_wait`** statements. An I-frame, however, has no dependencies on the previous iteration, and thus uses **`cilk_stage`** statements. The decision of whether a frame is an I-frame or a P-frame can be made dynamically in each iteration, after the result of executing stage 0.

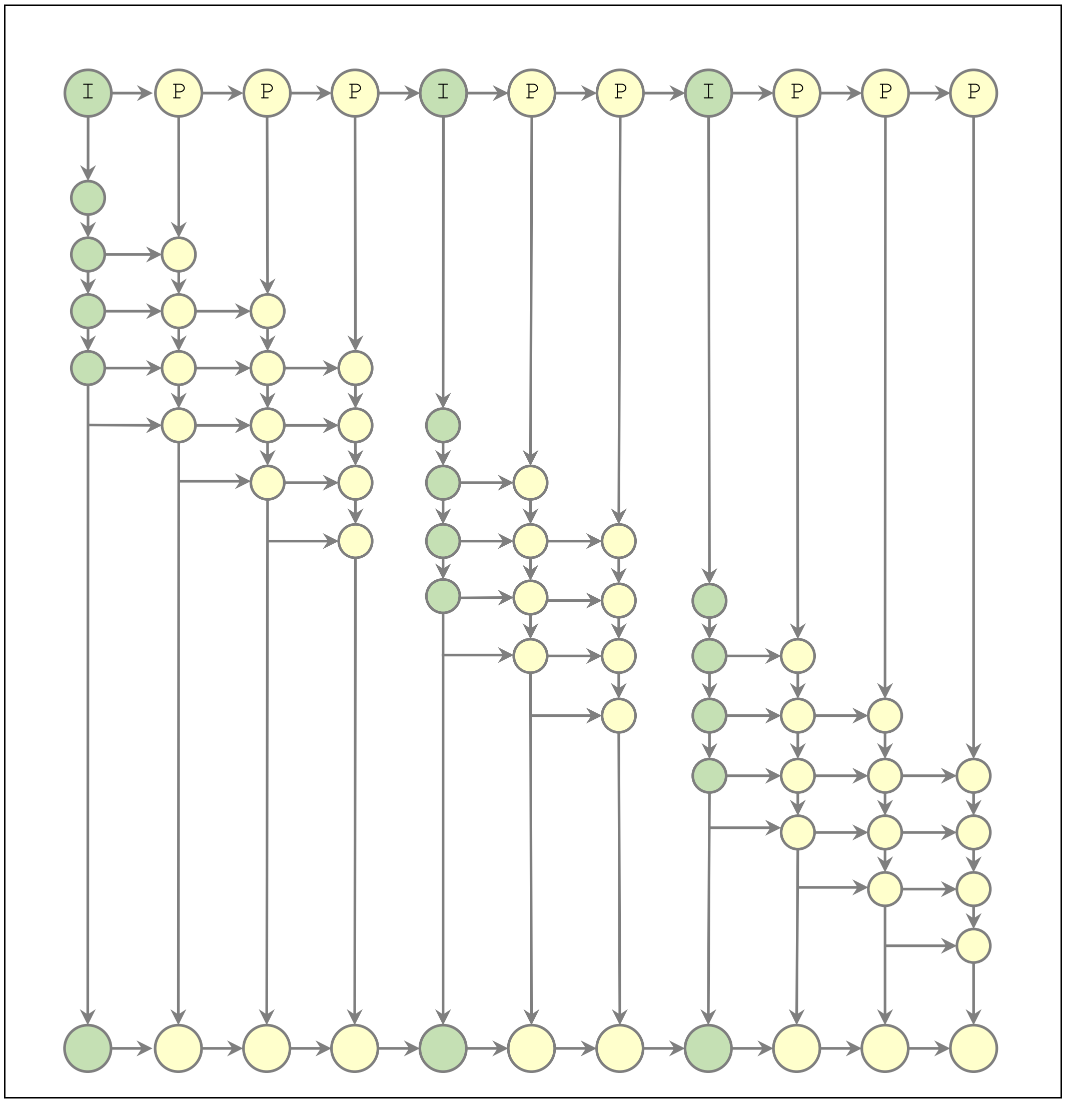

Figure 6: Pipeline dag for an x264 encoder. Each iteration processes a video frame. The first stage in each iteration decides whether the frame being processed is classified as an I-frame or P-frame.

Proposed Linguistics for a Pipe-While

This section summarizes the rules for the proposed pipe-while loop construct.

1. The `cilk_pipe_while` keyword is used to indicate a pipe-while loop. Like an ordinary while loop, the pipe-while loop has a **test condition** that determines whether there are any more iterations to execute. Conceptually, the test condition executes at the start of an iteration; once this condition is false, the pipe-while loop terminates.
2. The body of a `cilk_pipe_while` loop creates a separate lexical scope for each **iteration**, in the same fashion as an ordinary while loop.
3. Each iteration of the loop executes in **stages** which are named by nonnegative integers that increase monotonically as an iteration executes.

4. Stage boundaries are delineated by `cilk_stage` or `cilk_stage_wait` statements.
5. The `cilk_stage` and `cilk_stage_wait` statements must appear in the lexical scope of the `cilk_pipe_while` (similar to a **`break`** or **`continue`** in a normal while loop).
6. Each iteration of the loop begins executing in stage 0. The evaluation of the test condition for an iteration executes as part of stage 0 of the iteration.
7. The test condition for iteration **`i+1`** can be evaluated only after stage 0 of iteration **`i`** completes.
8. A `cilk_stage(s)` statement indicates that execution in the current iteration should advance to stage **`s`**. It is a runtime error if the argument **`s`** is not greater than the iteration's current stage number.
9. A `cilk_stage` statement takes as its argument any expression that evaluates to an integer **`s`** in the interval **`[1, std::numeric_limits<int64_t>::max()-1]`**.
10. The `cilk_stage_wait(s)` statement in iteration **`i+1`** has the same effect as a `cilk_stage(s)` statement, but also guarantees that stage **`s`** in iteration **`i+1`** does not begin executing until iteration **`i`** has completed all its stages **`s' <= s`**.
11. A call to `cilk_stage`, i.e., a call with no argument, is equivalent to `cilk_stage(s+1)`, where **`s`** is the current stage of the iteration.
12. Similarly, a call to `cilk_stage_wait` is equivalent to `cilk_stage_wait(s+1)`, where **`s`** is the current stage of the iteration.
13. The end of an iteration (e.g., the destructors of the variables declared in the scope of the iteration) execute as part of the last stage in the iteration. This last stage is either specified by the last `cilk_stage` or `cilk_stage_wait` statement to execute in the iteration, or it is stage 0 if the iteration executed no `cilk_stage` or `cilk_stage_wait` statements.
14. Nested fork-join parallelism (via **`cilk_spawn`**, **`cilk_sync`**, and **`cilk_for`**) is allowed within each stage. Each stage in an iteration is a separate task block. Said differently, each `cilk_stage` or `cilk_stage_wait` statement begins with an implicit **`cilk_sync`** (for the previous stage). This behavior is analogous to a **`cilk_for`** loop, except each stage in an iteration is a separate task block instead of the entire iteration being a separate task block.
15. As a pragmatic concern, we want to allow a user to specify a **throttling limit** on the pipeline loop, a limit on the number of loop iterations that can be active at once.
16. It may be useful to provide a `cilk_pipe_stage` keyword, that can be invoked in the lexical scope of a pipe-while loop, which returns the current stage number. This keyword would be particularly useful when using `cilk_stage` or `cilk_stage_wait` with the implicit "next stage" argument.

Piper User Guide

Piper is an experimental prototype of Intel® Cilk™ Plus that provides runtime support for pipe-while loops. Unfortunately, supporting pipe-while loops in the language requires compiler support to handle new keywords. Since compiler support is not yet available, Piper instead provides a header file with preprocessor macros that users can include to code their own pipe-while loops. This section describes how to install and use the Piper branch of Intel Cilk Plus.

Installing Piper

Piper can be installed either (1) via runtime binaries or (2) built from source.

1. **Runtime Binary Package**. On https://www.cilkplus.org/download, we provide a prebuilt Intel Cilk Plus runtime and the necessary header files for Piper.
   - On Linux* the provided runtime library `libcilkrts.so.5` is compatible with the existing Intel Cilk Plus compilers: ICC, GCC, and Clang/LLVM. This runtime can be used in place of an existing Intel Cilk Plus runtime library.
   - To compile programs using Piper, add the provided directory of header files to the appropriate include path, and set the path for the linker to find the Piper version of the runtime, e.g., by setting `LIBRARY_PATH`. Because Piper requires new runtime entry points, attempting to link a Piper program to an ordinary Intel Cilk Plus runtime will result in "undefined symbol" errors.
   - Note that the macros for pipe-while support require the use of lambda expressions from C++ 11. When compiling programs using Piper, one may need to specify `-std=c++0x` or some other appropriate compiler flag.
   - To run programs using Piper, set the appropriate path for shared libraries (e.g., `LD_LIBRARY_PATH`) to the directory containing the Piper version of the runtime.
2. **Building from source**. The source for the Piper is available as a branch of the Intel Cilk Plus runtime on Bitbucket. To checkout this branch:

   ```
   git clone https://intelcilkruntime@bitbucket.org/intelcilkruntime/intel-cilk-
   runtime.git
   git checkout origin/piper
   ```

   Regression tests for the Piper runtime are included in the `cilktest/piper` directory.

   After building the runtime binary with any existing Cilk Plus compiler, follow the previous instructions for the runtime binary package to install the newly built runtime binary.

Using Piper

The runtime binary package also comes with several sample programs that illustrate the use of the macros for Piper. These macros simulate the compiler support that one would eventually want for a full implementation of pipe-while loops. To use Piper, include the header file `cilk/piper.h`, and then replace the proposed keywords with macros as described by the table in Figure 7.

Description	**Proposed Keyword**	**Piper Cilk Plus Macro Equivalent**
Basic pipe-while loop.	`cilk_pipe_while(test())` `{` `…` `}`	`CILK_PIPE_WHILE_BEGIN(test())` `{` `…` `} CILK_PIPE_WHILE_END();`
Advance to stage `s`, no dependency.	`cilk_stage(s);`	`CILK_STAGE(s);`
Advance to next stage, no dependency.	`cilk_stage;`	`CILK_STAGE_NEXT();`
Advance to stage `s`, with dependency.	`cilk_stage_wait(s);`	`CILK_STAGE_WAIT(s);`
Advance to next stage with dependency.	`cilk_stage_wait;`	`CILK_STAGE_WAIT_NEXT();`
Throttled pipe-while loop. Limit of at most `K` active iterations.	`#pragma cilk throttle_limit=K;` `cilk_pipe_while(!done)` `{` `…` `}`	`CILK_PIPE_WHILE_BEGIN_THROTTLED (!done,K)` `{` `…` `} CILK_PIPE_WHILE_END();`
Current stage.	`cilk_pipe_stage;`	`CILK_PIPE_STAGE();`

Figure 7: Macro equivalents for the proposed keywords for pipe-while loops.

For example, to implement the pipe-while loop from Figure 1 in Piper, we can use the macros as shown in Figure 8.

```
bool done = false;
int iter_counter = 0;
CILK_PIPE_WHILE_BEGIN(!done) { // Each iteration starts in Stage 0.
    int i = iter_counter++;
    done = stage0(i);

    CILK_STAGE(1);      // Advance to Stage 1 (parallel stage)
    stage1(i);

    CILK_STAGE_WAIT(2); // Advance to Stage 2 (serial stage)
    stage2(i);

    CILK_STAGE(3);      // Advance to Stage 3 (parallel stage)
    stage3(i);

    CILK_STAGE_WAIT(4); // Advance to Stage 4 (serial stage)
    stage4(i);
} CILK_PIPE_WHILE_END();
```

Figure 8: Piper equivalent for the proposed pipe-while loop in Figure 1.

Other Notes

- Like the ordinary `cilk_spawn` and `cilk_sync` keywords in Intel Cilk Plus, the Piper prototype is designed for exploiting coarse-grained task parallelism. In particular, to execute pipeline iterations in parallel, the runtime incurs a steal overhead for starting each iteration. Thus, it may be inefficient to use a pipe-while loop if iterations do not have enough work to do.
- Because the prototype lacks compiler support, the error messages for Piper when a pipe-while loop fails to compile can be slightly tricky to debug. One simple way to debug the compilation of a pipe-while loop is to start by coding a normal serial while loop, and then adding in the pipeline keywords. To serialize all pipe-while loops, define the `CILK_PIPER_SERIALIZE` macro during compilation.
- The Intel Cilk Plus SDK, which includes the Intel Cilk screen and Intel Cilk view tools, do not yet support pipe-while loops.

Bibliography

1. Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. *PACT*, (pp. 72--81).

2. Lee, I.-T. A., Leiserson, C. E., Schardl, T. B., Sukha, J., & Zhang, Z. (2013). On-the-fly pipeline parallelism. *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13)* (pp. 140--151). New York: ACM.

Intel and Cilk are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.