The Cilk Plus Extension

Brito, Rui
PG22781
Department of Informatics
University of Minho
ruibrito666@gmail.com

Alves, José
PG22765
Department of Informatics
University of Minho
zealves.080@gmail.com

Abstract

This report presents an analysis of the Cilk Plus extension for the C and C++ languages. Cilk Plus is not just a library, it is an extension implemented by the compiler and the Cilk Plus runtime, thus allowing for lower overhead than some of it's competitors. We compare Cilk Plus with a very well known alternative, OpenMP, and, although some results were positive (namely, recursive algorithms), overall, OpenMP wins.

1 Introduction

Cilk Plus provides an easy way to exploit parallelism inside an application, by using only three keywords, one can fully take advantage of both multicore and SIMD extensions. In Section 2 an introduction on what Cilk Plus is is given while Section 3 introduces its main features. Section 4 details what the language adds to parallel programming paradigms and provides some examples. Section 5 explains how Cilk fairs against OpenMp. Section 6 states Cilk Plus' advantages and disadvantages.

2 What is Cilk Plus

Intel Cilk is an extension to the C and C++ languages to support task and data parallelism. Unlike most of it's competitors, Cilk Plus is not just a library, it is a language extension that is implemented by the compiler and the Cilk Plus runtime, allowing lower overhead than library-only solutions. Cilk Plus is developed by Intel, and thus, through the Intel runtime, supports composability. Many applications are composed of independently written components. Typically, the components communicate with each other through higher-level interfaces rather than at a lower, implementation-oriented level. To allow that level of collaboration in a parallel program, the task scheduler has to allow composability (that is, when modules are combined to form the application, they continue to work and perform well). Cilk Plus achieves this with a work-stealing task scheduler.

3 Key Features

The main key features of Cilk Plus are as follows:

- Keywords: cilk adds three new keywords to the C/C++ languages to better express task parallelism in an application;
- Reducers: a mechanism to eliminate contention for shared variables;
- #pragma simd: a directive that tells the compiler a loop should be vectorized;
- Array Notations: a language addition to specify data paralellism for arrays or sections of arrays;

4 The Cilk Plus Extension

As seen in previous section, Cilk Plus as several of expressing parallelism, the following subsections will provide some insight into some of its core features and how it works.

4.1 Keywords

The are only three keywords in the Cilk Plus extensions and they provide a simple yet powerful way of expressing parallelism, they are as follows:

- cilk_spawn: tells the runtime environment that the statement following the cilk_spawn keyword can be run in parallel with other statements;
- cilk_sync: tells the runtime environment that all children of the spawning block must finish their execution before execution can continue;
- cilk_for: a replacement for the standard C/C++ for loop that lets iterations run in parallel;

The following example shows the typical use of the cilk_spawn and cilk_sync keywords:

```
double fib(double n) {
    if (n < 2)
        return n;

    if (n < 30)
        return seq_fib(n);

    double x = cilk_spawn fib(n-1);
    double y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Listing 1: Computation of the nth Fibonacci Number using Cilk Plus

The following example shows the use of the cilk_for keyword:

```
float* matrixDot (float *a, float *b, int n) {

    float value = 0;
    float *res = (float*) malloc(n * n *
        sizeof(float));

    cilk_for (int i = 0; i < n; ++i) {
        for(int j = 0; j < n; ++j) {
            for(int k = 0; k < n; ++k) {
                    value += a[i * n + k] * b[
                        k * n + j];
            }
            res[i * n + j] = value;
            value = 0;
        }
    }


    return res;
}
```

Listing 2: Matrix multiplication using Cilk Plus

4.1.1 More on cilk_for

A call to cilk_for isn't the same as spawning each iteration of the loop. What actually happens is, the compiler converts the loop body to a function that is called recursively using a divide-and-conquer strategy.

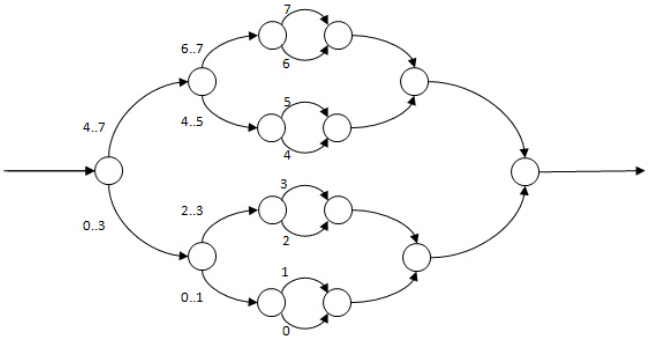

Figure 1: cilk_for with N=8 iterations

As it can be seen in the previous figure, the worker whose execution reached the cilk_for construct computes the trip count, it then takes the upper half of the iterations and puts it in its own work queue for later evaluation. It then continues to divide and conquer the lower trip count, until the trip count becomes sufficiently small (determined heuristically at run time). It then executes the iterations sequentially. Upon completion, the worker pops the next work item, which was posted last, from its own work queue. This will be the second set of iteration of the lower half. If another worker has an empty work queue, it might steal work from the core that created the list of work items corresponding to the loop iterations. This is how concurrent work is created in Cilk Plus.

4.2 Reducers

Reducers allow thread-safe access to shared objects by giving each parallel strand a separate instance of the object, called a view. The Cilk Plus Reducers are as follows:

- reducer_list_append: Creates a list by adding elements to the back;
- reducer_list_prepend: Creates a list by adding elements to the front;
- reducer_max: Calculates the maximum value of a set of values;
- reducer_max_index: Calculates the maximum value and index of that value of a set of values;
- reducer_min: Calculates the minimum value of a set of values;
- reducer_min_index: Calculates the minimum value and index of that value of a set of values;
- reducer_opadd: Calculates the sum of a set of values;
- reducer_opand: Calculates the binary AND of a set of values;
- reducer_opor: Calculate the binary OR of a set of values;
- reducer_opxor: Calculate the binary XOR of a set of values;
- reducer_string: Accumulates a string using append operations;
- reducer_wstring: Accumulates a "wide" string using append operations;
- reducer_ostream: An output stream that can be written in parallel;
- reducer_ostream: An output stream that can be written in parallel;

A small usage example follows:

```
void reducer_list_test() {
    cilk::reducer_list_append<char>
        letters_reducer;

    cilk_for(char ch = 'a'; ch <= 'z'; ch
        ++){
        simulated_work();
        letters_reducer.push_back(ch);
    }

    const std::list<char> &letters =
        letters_reducer.get_value();

    for(std::list<char>::const_iterator i
        = letters.begin(); i != letters.
        end(); i++) {
        std::cout << " " << *i;
    }
    std::cout << std::endl;
}
```

Listing 3: Example showing the usage of Cilk Plus Reducers

4.3 Array Notation

Array notation is intended to allow users to directly express high level parallel vector array operations in their code. This assists the compiler in performing vectorization and auto-parallelization. More predictable vectorization, improved performance and better hardware resource utilization are key goals.

This is a direct extension to the C/C++ languages. Here is an example:

```
// Copy elements 10->19 in A to elements 0->9
     in B.
B[0:10] = A[10:10];
// Transpose row 0, columns 0-9 of A, into
     column 0, rows 0-9 of B.
B[0:10][0] = A[0][0:10];
// Copy the specified array section in the 2nd
     and 3rd dimensions of A into the 1st and
     4th dimensions of B.
B[0:10][0][0][0:5] = A[3][0:10][0:5][5]
// Set all elements of A to 1.0.
A[:] = 1.0;
// Add elements 10->19 from A with elements
     0->9 from B and place in elements 20->29
     in C.
C[20:10] = A[10:10] + B[0:10];
// Element-wise equality test of B and C,
     resulting in an array of Boolean values,
     which are placed in A.
A[:] = B[:] == C[:];
```

Listing 4: Examples of Cilk Plus Array Notation

4.4 The Cilk Plus Scheduler

The Cilk Plus runtime uses a work-stealing scheduler to dynamically load-balance the tasks that are created by a Cilk Plus program. In abstract, the runtime uses worker threads. Each of these worker threads manipulates a stack, by removing task at the bottom. When a worker thread as an empty stack, it steals a task from the top of another worker thread's stack. If the flow of the program reaches a cilk_sync construct, then it tries to keep busy by popping tasks from the tail of its stack.

5 Cilk vs. OpenMP

5.1 Test Environment

In order to achieve precise and consistent results, all tests were run on a dedicated execution of the algorithm, with niceness set to -20, to ensure the tests execution was top priority to the OS's scheduler. Furthermore, all network connections were disabled. To decrease the chance of human error, all tests were hard-coded into a bash script, with another script traversing those tests and showing the results. Also, it should be noted that every test was run at least 3 times. A 5% error margin was checked between execution (the aforementioned script handles this part), and if this margin is not met, the tests are run again. It should also be noted that due to the nature of Cilk Plus the compiler used was icc (13.0.2).

Note: no tests were run on search because of problems with the icc licence.

	MacBookPro Intel Ivy-Bridge i7
# processors	1
# cores per processor	4
hyper-threading	yes
clock frequency(GHz)	2.3
L1 size	64KB
L2 size	256KB
L3 size	6MB
RAM size	16GB

Table 1: Test Machine

5.2 Results

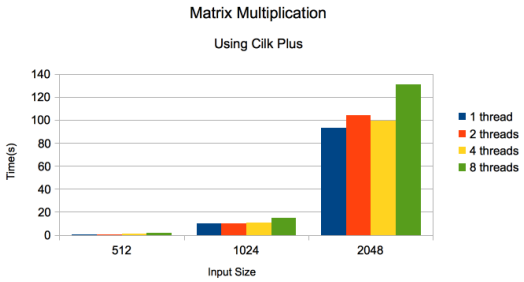

Figure 2: Results for matrix multiplication using Cilk Plus

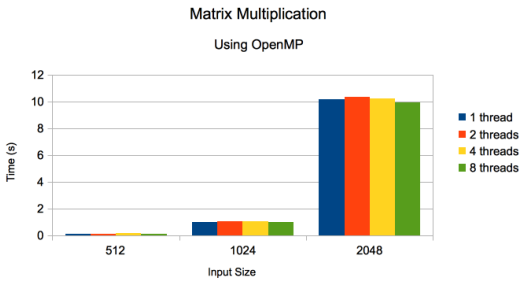

Figure 3: Results for matrix multiplication using OpenMP

As can be seen in the results above, Matrix Multiplication performs better in the OpenMP implementation, this is probably due to the way the cache is used.

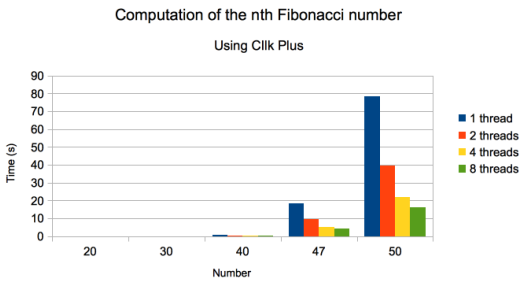

Figure 4: Results for fibonacci computation using Cilk Plus

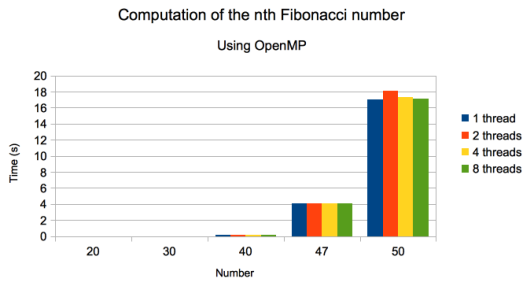

Figure 5: Results for fibonacci computation using OpenMP

Fibonacci is a recursive algorithm and so it was expected for it to perform better in the Cilk Plus implementation, and it did. This the kind of problem Cilk is good at.

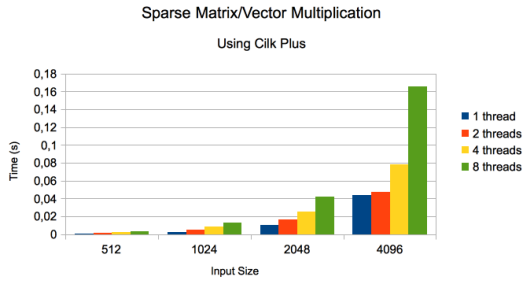

Figure 6: Results for sparse matrix/vector multiplication using Cilk Plus

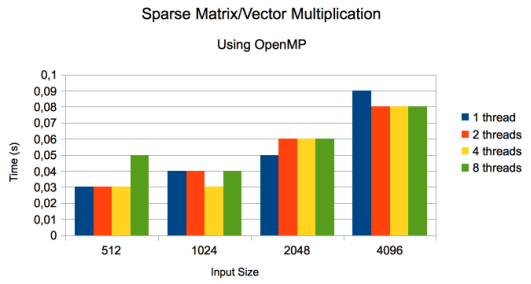

Figure 7: Results for sparse matrix/vector multiplication using OpenMP

Sparse Matrix/Vector multiplication was overall better in the Cilk implementation but not when the size of the matrix was 4096x4096. This is probably to cache alignment issues or because the problem doesn't fit in cache altogether, since each thread needs a stack and that stack, most probably, is in cache.

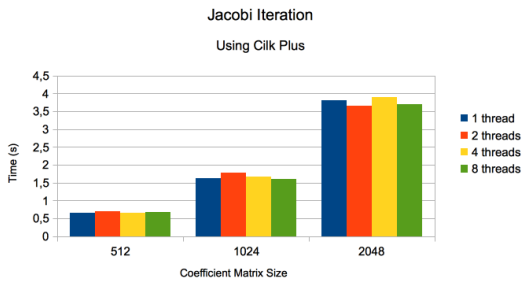

Figure 8: Results for jacobi iteration using Cilk Plus

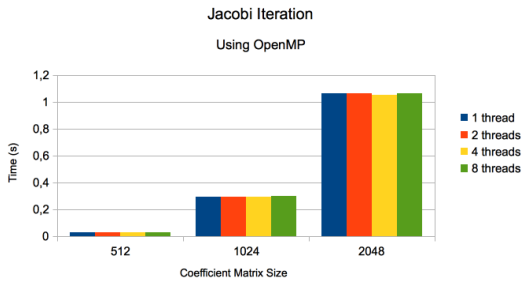

Figure 9: Results for jacobi iteration using OpenMP

Jacobi also performed better with openMP, this is mostly, we believe, due to high synchronization between worker threads.

6 Conclusion

Ease of use if definitely an advantage over Cilk Plus competitors. Although it is not as fine grained as OpenMP, the Intel runtime allows for it to be used with other tools, such as TBB (composability). We found it is good for recursive algorithms, or algorithms that can be divided in tasks. Automatic load balancing is one of the best things about Cilk Plus, in fact, Intel goes as far as claiming, guaranteed maximum memory usage scaling. It simplifies adding parallelism to existing serial programs. Multiple attributes of the extensions support this, including the non intrusive syntax, the ability to easily revert back to the serial program, the guarantee of serial semantics equivalence, and the low overhead of spawning a task.