Experimenting with the MetaFork Framework Targeting Multicores

Xiaohui Chen, Marc Moreno Maza & Sushek Shekar
University of Western Ontario

26 January 2014

1 Introduction

The work reported in this report evaluates the correctness, performance and usefulness of the four MetaFork translators (MetaFork to CilkPlus, CilkPlus to MetaFork, MetaFork to OpenMP, OpenMP to MetaFork). To this end, we run these translates on various input programs written either in CilkPlus or OpenMP, or both.

We stress the fact that our purpose is not to compare the performance of the CilkPlus or OpenMP run-time systems and programming environments. The reader should notice that the codes used in this experimental study were written by different persons with different levels of expertise. In addition, the reported experimentation is essentially limited to one architecture (Intel Xeon) and one compiler (GCC). Therefore, it is delicate to draw any clear conclusions that would compare CilkPlus or OpenMP. For this reason, this questions is not addressed in this thesis And, once again, this is not the purpose of this work.

2 Experimentation set up

We conducted two experiments. In the first one, we compared the performance of hand-written codes. The motivation, recalled from the introduction, is *comparative implementation*. The underlying observation is that the same multithreaded algorithm, based on the fork-join parallelism model, implemented with two different concurrency platforms, say CilkPlus and OpenMP, could result in very different performance, often very hard to analyze and compare. If one code scales well while the other does not, one may suspect an efficient implementation of the latter as well as other possible causes such as higher parallelism overheads. Translating the inefficient code to the other language can help narrowing the problem. Indeed, if the translated code still does not scale one can suspect an implementation issue (say the programmer missed to parallelize one portion of the algorithm) whereas if it does scale, then one can suspect a parallelism overhead issue in the original code (say the grain-size of the parallel for-loop is too small).

For this experience, we use a series of test-cases consisting of a pair of programs, one hand-written OpenMP program and one, hand-written CilkPlus program. We observe that one program (written by a student) has a performance bottleneck while its counterpart (written by an expert programmer) does not. We translate the inefficient program to the other language, then check whether the performance bottleneck remains or not, so as to narrow the performance bottleneck in the inefficient program.

Our second experience, also motivated in the introduction, is dedicated to automatic translation of highly optimized code. Now, for each test-case, we have either a hand-written-and-optimized CilkPlus program or a hand-written-and-optimized OpenMP. Our goal is to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts.

For both experiences, apart from student's code, the code that we use comes from the following the sources:

- John Burkardt's Home Page (Florida State University) `http://people.sc.fsu.edu/~%20jburkardt/c_src/`
- Barcelona OpenMP Tasks Suite (BOTS) [6]
- CilkPlus distribution examples `http://sourceforge.net/projects/cilk/`

The source code of those test case was compiled as follows:

- CilkPlus code with GCC 4.8 using `-O2 -g -lcilkrts -fcilkplus`
- CilkPlus code with GCC 4.8 using `-O2 -g -fopenmp`

All our compiled programs were tested on

- AMD Opteron 6168 48core nodes with 256GB RAM and 12MB L3
- Intel Xeon 2.66GHz/6.4GT with 12 physical cores and hyper-threading,

The two main quantities that we measure are:

- *scalability* by running our compiled OpenMP and CilkPlus programs on $p = 1, 2, 4, 6, 8, \ldots$ processors; speedup curves data are shown in Figures 4, 3, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 and 17 is mainly collected on Intel Xeon's nodes (for convenience) and repeated/verified on AMD Opteron nodes.
- *parallelism overheads* by running our compiled OpenMP and CilkPlus programs on $p = 1$ against their serial elisions. This is shown in Table 1

As mentioned in the introduction, validating the correctness of our translators was a major requirement of our work. Depending on the test-case, we could use one or the other following strategy.

- Assume that the original program, say $\mathcal{P}$, contains both a parallel code and its serial elision (manually written). When program $\mathcal{P}$ is executed, both codes run and compare their results. Let us call $\mathcal{Q}$ the translated version of $\mathcal{P}$. Since serial elisions are unchanged by our translation procedures, then $\mathcal{Q}$ can be verified by the same process used for program $\mathcal{P}$. This first strategy applies to the `Cilk++` distribution examples and the BOTS (Barcelona OpenMP Tasks Suite) examples
- If the original program $\mathcal{P}$ does not include a serial elision of the parallel code, then the translated program $\mathcal{Q}$ is verified by comparing the output of $\mathcal{P}$ and $\mathcal{Q}$. This second strategy had to be applied to the FSU (Florida State University) examples.

3 Comparing hand-written codes

3.1 Matrix transpose

In this example, the two original parallel programs are based on different algorithms for matrix transposition which is a challenging operation on multi-core architectures. Without doing complexity analysis, discovering that the OpenMP code (written by a student) runs in $O(n^2 \log(n))$ bit operations instead of $O(n^2)$ as the CilkPlus (written by Matteo Frigo) is very subtle.

Figure 1 shows code snippet for the matrix transpose program for both original CilkPlus and translated OpenMP. Figure 2 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code and it suggests that the code translated from CilkPlus to OpenMP is more appropriate for fork-join multi-threaded languages targeting multicores because of the algorithm used in CilkPlus code.

```
              /*---------------Original CilkPlus code-------------*/
template <typename T>
void transpose(T *A, int lda, T *B, int ldb,
               int i0, int i1, int j0, int j1)
{
 tail:
     int di = i1 - i0, dj = j1 - j0;
     if (di >= dj && di > THRESHOLD) {
          int im = (i0 + i1) / 2;
          cilk_spawn transpose(A, lda, B, ldb, i0, im, j0, j1);
          i0 = im; goto tail;
     } else if (dj > THRESHOLD) {
          int jm = (j0 + j1) / 2;
          cilk_spawn transpose(A, lda, B, ldb, i0, i1, j0, jm);
          j0 = jm; goto tail;
     } else {
          for (int i = i0; i < i1; ++i)
               for (int j = j0; j < j1; ++j)
                    B[j * ldb + i] = A[i * lda + j];
     }
}

              /*---------------Translated OpenMP code-------------*/
template <typename T>
void transpose(T *A, int lda, T *B, int ldb,
               int i0, int i1, int j0, int j1)
{
 tail:
     int di = i1 - i0, dj = j1 - j0;
     if (di >= dj && di > THRESHOLD) {
          int im =(i0 + i1) / 2;
           #pragma omp task
 transpose(A, lda, B, ldb, i0, im, j0, j1);
          i0 = im; goto tail;
     } else if (dj > THRESHOLD) {
          int jm =(j0 + j1) / 2;
           #pragma omp task
 transpose(A, lda, B, ldb, i0, i1, j0, jm);
          j0 = jm; goto tail;
     } else {
          for (int i = i0; i < i1; ++i)
               for (int j = j0; j < j1; ++j)
                    B[j * ldb + i] = A[i * lda + j];
     }
#pragma omp taskwait
 }
```

Figure 1: Code snippet for the matrix transpose program

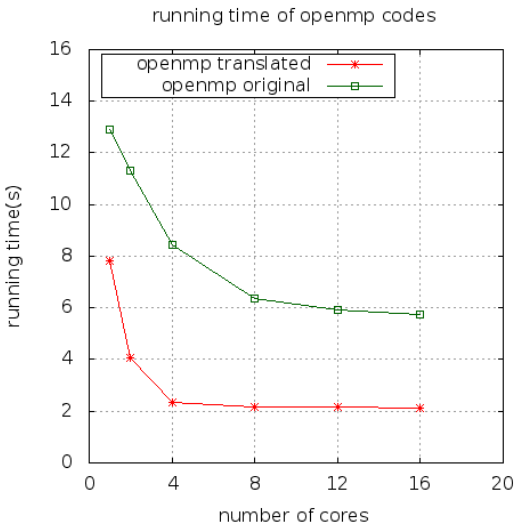


Figure 2: Matrix_transpose : $n = 32768$

3.2 Matrix inversion

In this example, the two original parallel programs are based on different serial algorithms for matrix inversion. The original OpenMP code uses Gauss-Jordan elimination algorithm while the original CilkPlus code uses a divide-and-conquer approach based on Schur's complement. Figure 3 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code.

As in the previous example, the translation narrows the algorithmic issue in this example as well.

3.3 Mergesort

There are two different versions of this example. The original OpenMP code (written by a student) misses to parallelize the merge phase and simply spawns the two recursive calls whereas the original CilkPlus code (written by an expert) does both. Figure 4 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code.

As you can see in Figure 4 the speedup curve of the translated OpenMP code is as theoretically expected while the speedup curve of the original OpenMP code shows a limited scalability.

Hence, the translated OpenMP (and the original CilkPlus program) exposes more parallelism, thus narrowing the performance bottleneck in the original OpenMP code.

3.4 Naive matrix multiplication

This is the naive three-nested-loops matrix multiplication algorithm, where two loops have been parallelized. The parallelism is $O(n^2)$ as for DnC MM. However, the ratio work-to-memory-access is essentially equal to 2, which is much less that for DnC MM. This limits the ability to scale and reduces performance overall on multicore processors. Figure 5 shows the speed-up curve for naive matrix multiplication.

As you can see from Figure 5 both CilkPlus and OpenMP scale poorly because of algorithmic issues.

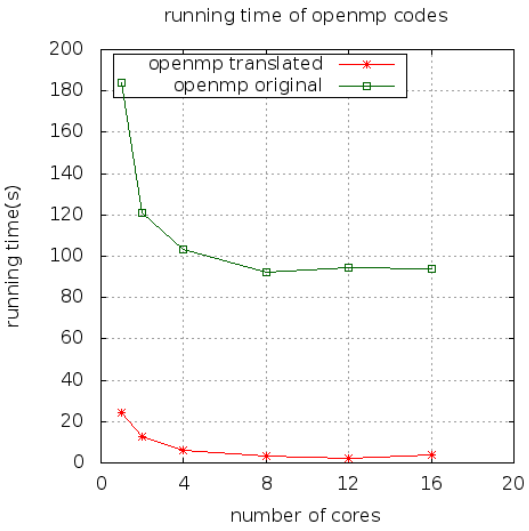


Figure 3: Matrix inversion : $n = 4096$

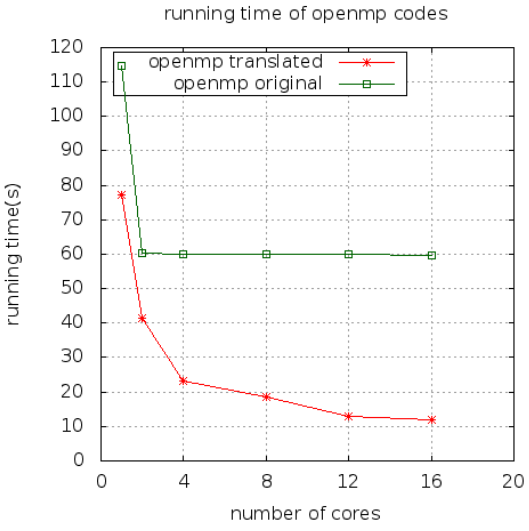


Figure 4: Mergesort (Student's code) : $n = 5 \cdot 10^8$

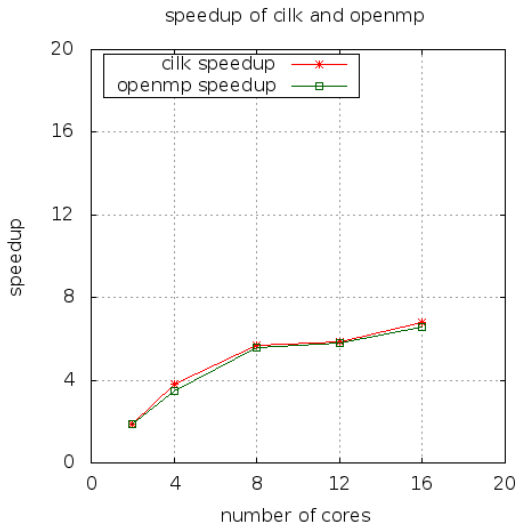


Figure 5: Naive Matrix Multiplication : $n = 4096$

4 Automatic translation of highly optimized code

4.1 Fibonacci number computation

In this example, we have translated the original CilkPlus code to OpenMP. The algorithm used in this example has high parallelism and no data traversal. The speed-up curve for computing Fibonacci with inputs 45 and 50 are shown in Figure 6(a) and Figure 6(b) respectively.

As you can see from Figure 6 CilkPlus (original) and OpenMP (translated) codes scale well.

4.2 Divide-and-conquer matrix multiplication

In this example, we have translated the original CilkPlus code to OpenMP code. The divide-and-conquer algorithm of [8] is used to compute matrix multiplication. This algorithm has high parallelism and optimal cache complexity. It is also data-and-compute-intensive. The speed-up curve after computing matrix multiplication with inputs 4096 and 8192 are shown in Figure 7(a) and Figure 7(b) respectively.

As you can see from Figure 7 CilkPlus (original) and OpenMP (translated) codes scale well.

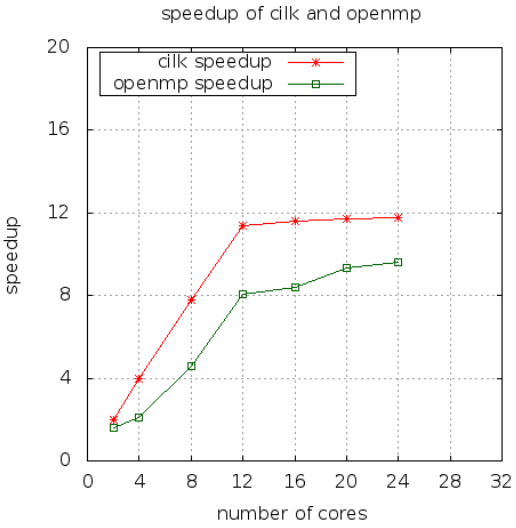
4.3 Parallel prefix sum

In this example, we have translated the original CilkPlus code to OpenMP code. The algorithm [3] used in this example has high parallelism, low work-to-memory-access ratio which is $(O(log(n))$ traversals for a $O(n)$ work. The speed-up curves with inputs $5 \cdot 10^8$ and 10^9 are shown in Figure 8(a) and Figure 8(b) respectively.

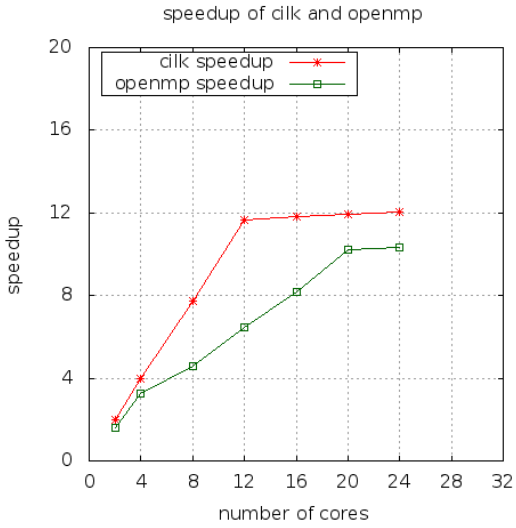
As you can see from Figure 7 CilkPlus (original) and OpenMP (translated) codes scale well at almost the same rate.

4.4 Quick sort

This is the classical quick-sort where the division phase has not been parallelized, on purpose. Consequently, the theoretical parallelism drops to $(log(n))$. The ratio of work-to-memory-access is constant. The speed-up curve for quick sort is shown in Figure 9(a) and Figure 9(b).

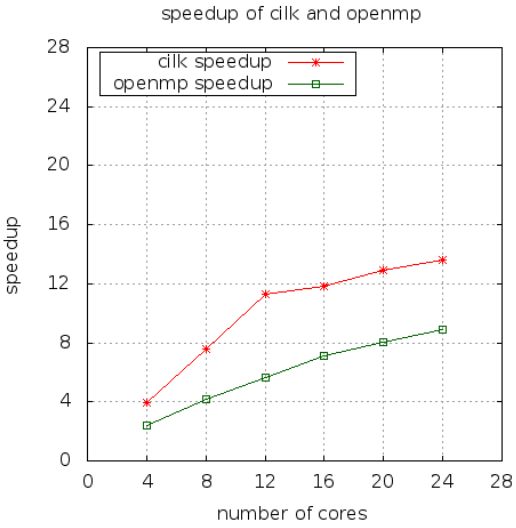


(a) Fibonacci : 45

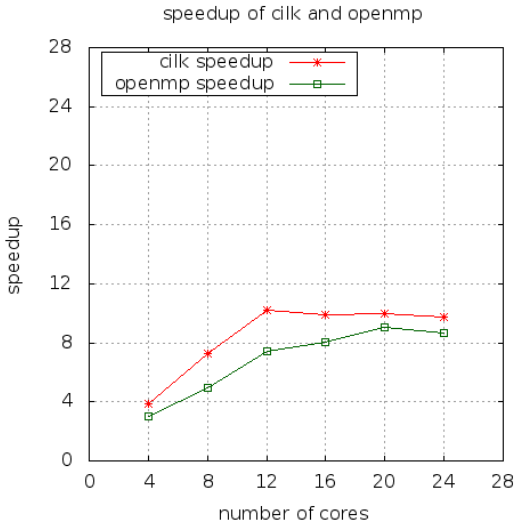


(b) Fibonacci : 50

Figure 6: Speedup curve on Intel node

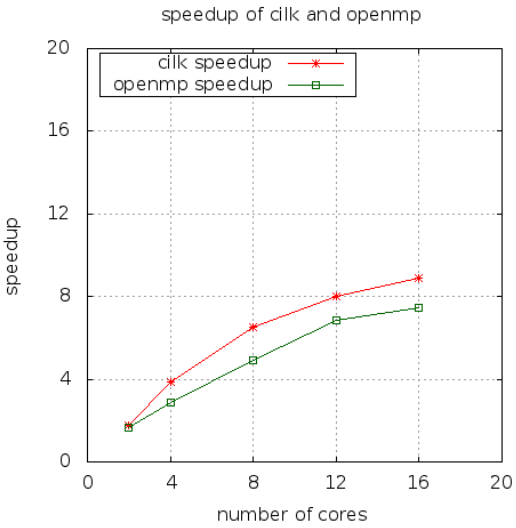


(a) DnC MM : 4096

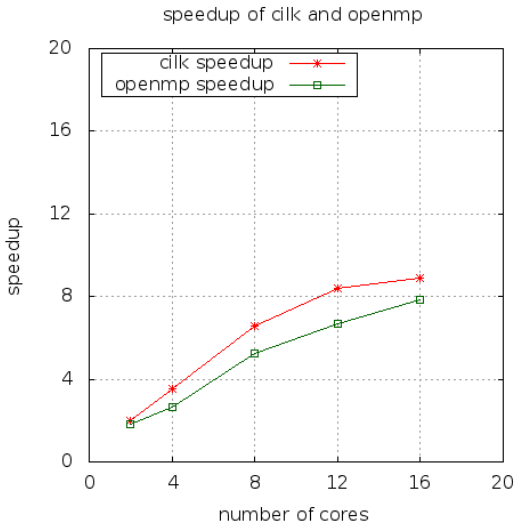


(b) DnC MM : 8192

Figure 7: Speedup curve on intel node

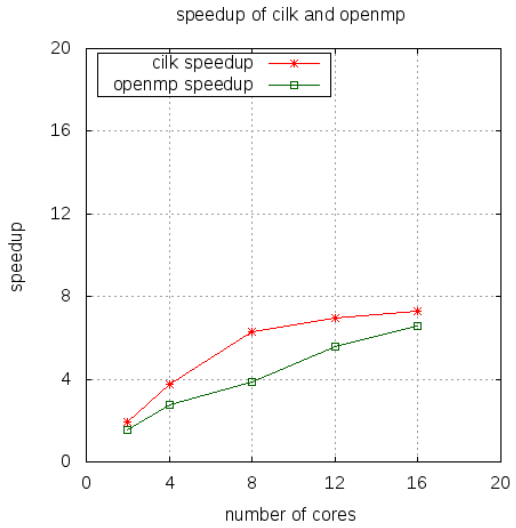


(a) Prefix_sum : $n = 5 \cdot 10^8$

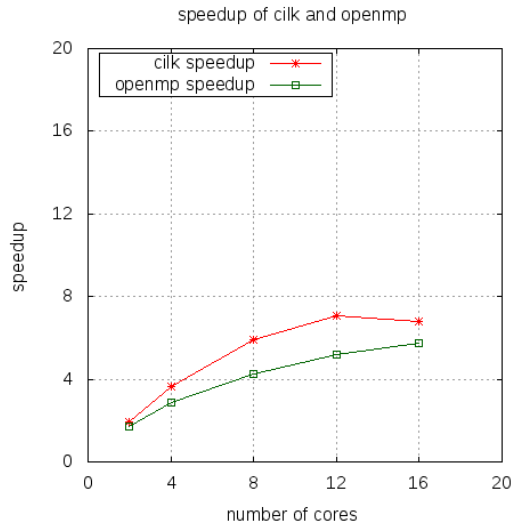


(b) Prefix_sum : $n = 10^9$

Figure 8: Speedup curve on Intel node



(a) Quick Sort: $n = 2 \cdot 10^8$



(b) Quick Sort: $n = 5 \cdot 10^8$

Figure 9: Speedup curve on Intel node

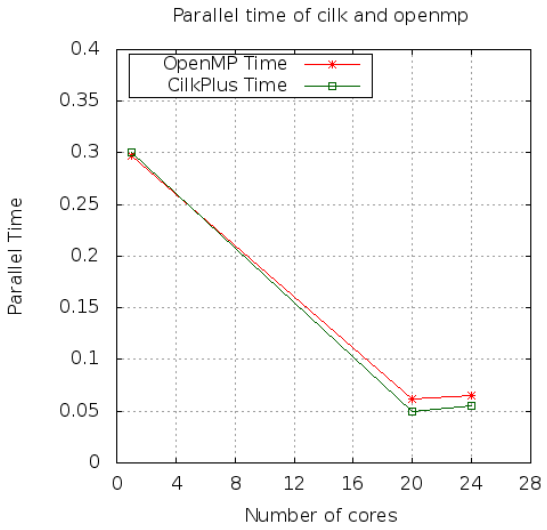


Figure 10: Mandelbrot set for a 500×500 grid and 2000 iterations.

As you can see from Figure 9 CilkPlus (original) and OpenMP (translated) codes scale at almost the same rate.

4.5 Mandelbrot set

The Mandelbrot set [1] is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape.

Mandelbrot set images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all.

In this example, algorithm is compute-intensive and does not traverse large data. The running time after computing the Mandelbrot set with grid size of 500x500 and 2000 iterations is shown in Figure 10.

As you can see from Figure 10 the speed-up is close to linear since this application is embarrassingly parallel. Both OpenMP (original) and CilkPlus (translated) codes scale at almost the same rate.

4.6 Linear system solving (dense method)

In this example, different methods of solving the linear system A*x=b are compared. In this example there is a standard sequential code and slightly modified sequential code to take advantage of OpenMP. The algorithm in this example uses Gaussian elimination.

This algorithm has lots of parallelism. However, minimizing parallelism overheads and memory traffic is a challenge for this operation. The running time of this example is shown in Figure 11.

As you can see from the Figure 11 both OpenMP (original) and CilkPlus (translated) codes scale well up to 12 cores. Note that that we are experimenting on a Xeon node with 12 physical cores with hyper-threading turned on.

[1] `http://en.wikipedia.org/wiki/Mandelbrot_set`

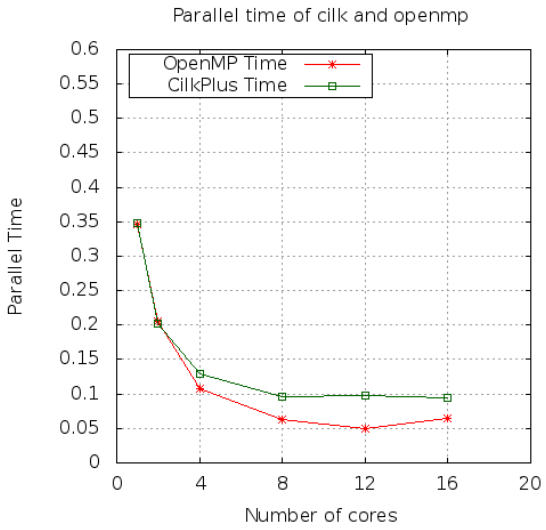


Figure 11: Linear system solving (dense method)

Test	input size	CilkPlus Serial	CilkPlus T_1	OpenMP serial	OpenMP T_1
FFT (BOTS)	33554432	7.50	8.12	7.54	7.82
MergeSort (BOTS)	33554432	3.55	3.56	3.57	3.54
Strassen	4096	17.08	17.18	16.94	17.11
SparseLU	128	568.07	566.10	568.79	568.16

Table 1: Running times on Intel Xeon 12-physical-core nodes (with hyperthreading turned on).

4.7 FFT (FSU version)

This example demonstrates the computation of a Fast Fourier Transform in parallel. The algorithm used in this example has low work-to-memory-access ratio which is challenging to implement efficiently on multi-cores. The running time of this example is shown in Figure 12.

As you can see from the Figure 12 both OpenMP (original) and CilkPlus (translated) codes scale well up to 8 cores.

Table 1 shows the running time of the serial version v/s single core for some of the examples.

4.8 Barcelona OpenMP Tasks Suite

Barcelona OpenMP Tasks Suite (BOTS) project [6] is a set of applications exploiting regular and irregular parallelism, based on OpenMP tasks.

4.9 Protein alignment

Alignment application aligns all protein sequences from an input file against every other sequence using the Myers and Miller [9] algorithm.

The alignments are scored and the best score for each pair is provided as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. The output is the best score for each pair of them. The speed-up curve for this example is shown in Figure 13.

This algorithm is compute-intensive and has few coarse grained tasks which is a challenge for the implementation to avoid load balance situations. As you can see from the Figure 13 both

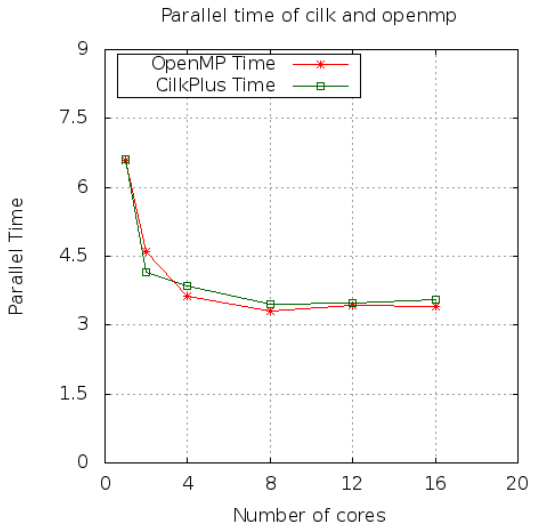


Figure 12: FFT (FSU) over the complex in size 2^{25}.

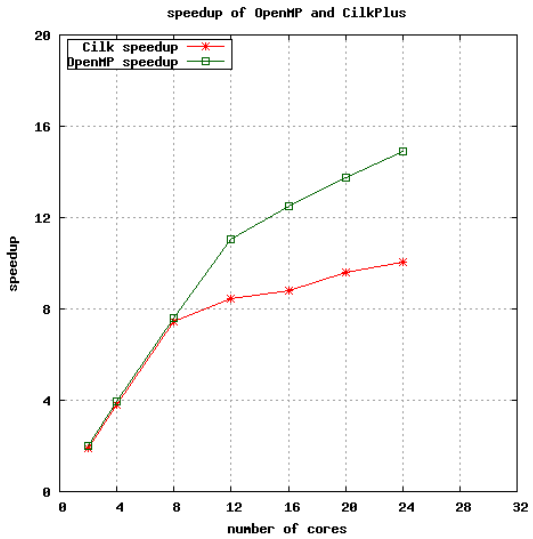


Figure 13: Protein alignment - 100 Proteins.

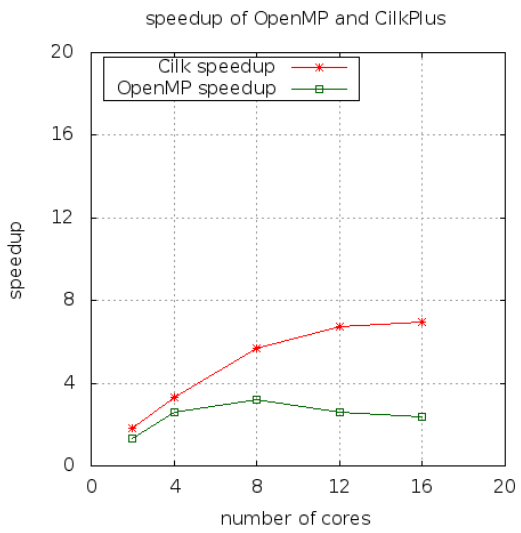


Figure 14: FFT (BOTS).

OpenMP (original) and CilkPlus (translated) codes scale almost same up to 8 cores.

4.10 FFT (BOTS version)

In this example, FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey [5] algorithm. It's a divide and conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFTs. The speed-up curve for this example is shown in Figure 14.

4.11 Merge sort (BOTS version)

Sort example sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation [1] of the ordinary merge sort. First, it divides an array of elements in two halves, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. The speed-up curve for this example is shown in Figure 15.

Hence, the translated CilkPlus (and the original OpenMP program) exposes more parallelism, thus narrowing the performance bottleneck in the original OpenMP code.

4.12 Sparse LU matrix factorization

Sparse LU computes an LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to small submatrices that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists.

Matrix size and submatrix size can be set at execution time which can reduce the imbalance, a solution with tasks parallelism seems to obtain better results [2]. The speed-up curve for this example is shown in Figure 16

This algorithm is compute-intensive and has few coarse grained tasks which is a challenge for the implementation to avoid load balance situations. As you can see from the Figure 16 both OpenMP (original) and CilkPlus (translated) codes scale almost same. Note that that we are experimenting on a Xeon node with 12 physical cores with hyperthreading turned on.

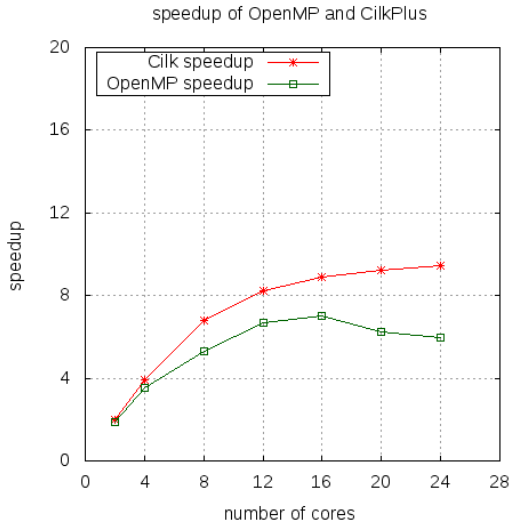


Figure 15: Mergesort (BOTS).

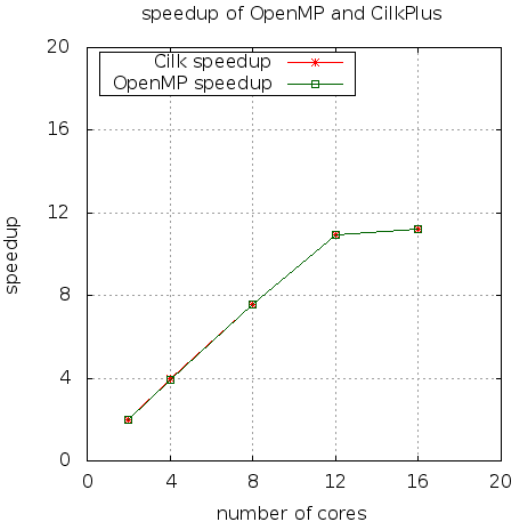


Figure 16: Sparse LU matrix factorization.

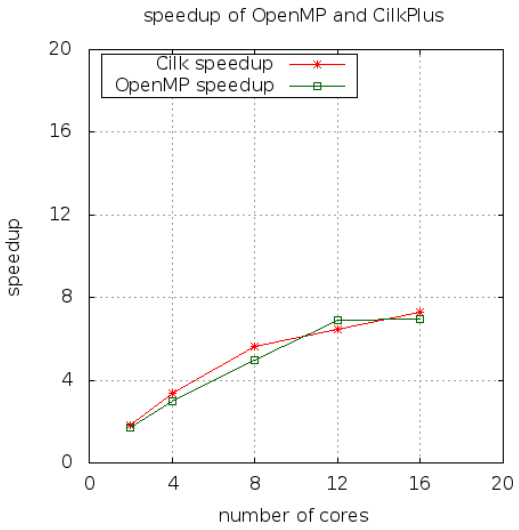


Figure 17: Strassen matrix multiplication.

4.13 Strassen matrix multiplication

Strassen algorithm [2] uses hierarchical decomposition of a matrix for multiplication of large dense matrices [7]. Decomposition is done by dividing each dimension of the matrix into two sections of equal size. The speed-up curve for this example is shown in Figure 17

As you can see from the Figure 17 both OpenMP (original) and CilkPlus (translated) codes scale almost same.

5 Concluding remarks

Examples in Section 3 suggest that our translators can be used to narrow performance bottlenecks. By translating a parallel program with low performance, we could suspect the cause of inefficiency whether this cause was a poor implementation (in the case of mergesort, where not enough parallelism was exposed) or an algorithm inefficient in terms of data locality (in the case of matrix inversion) or an algorithm inefficient in terms of work (in the case of matrix transposition).

For Section 4, we observe that, in most cases, the speed-up curves of the original and translated codes either match or have similar shape. Nevertheless in some cases, either the original or the translated program outperforms its counterpart. For instance the original CilkPlus programs for Fibonacci and the divide-and-conquer matrix multiplication perform better than their translated OpenMP counterparts, see Figure 6 and Figure 7 respectively.

For the original OpenMP programs from FSU, (Mandelbrot Set, Linear solving system and FFT (FSU version) the speed-up curves of the original programs are close to those of the translated CilkPlus programs. This is shown by Figures 10, 11 and 12 respectively.

The original OpenMP programs from BOTS offer different scenarios. First, for the sparse LU and Strassen matrix multiplication examples, original and translated programs scale in a similar way, see Figure 16 and Figure 17 respectively. Secondly, for the mergesort (BOTS) and FFT (BOTS) examples, translated programs outperform their original counterparts., see Figure 15 and Figure 14 respectively. For the protein alignment example, the scalability of the translated CilkPlus program and the original OpenMP program are the same from 2 to 8 cores while after 8 cores the latter outperform the former, as shown in Figure 13.

[2] `http://en.wikipedia.org/wiki/Strassen_algorithm`

References

[1] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, Nov. 1987.

[2] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. Languages and compilers for parallel computing. chapter An Experimental Evaluation of the New OpenMP Tasking Model, pages 63–77. Springer-Verlag, Berlin, Heidelberg, 2008.

[3] G. E. Blelloch and J. J. Little. Parallel solutions to geometric problems in the scan model of computation. *J. Comput. Syst. Sci.*, 48(1):90–115, 1994.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:356 – 368, 1994.

[5] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[6] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[7] P. C. Fischer and R. L. Probert. Efficient procedures for using matrix algorithms. In J. Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1974.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, USA, October 1999.

[9] G. Myers, S. Selznick, Z. Zhang, and W. Miller. Progressive multiple alignment with constraints. In *Proceedings of the First Annual International Conference on Computational Molecular Biology*, RECOMB '97, pages 220–225, New York, NY, USA, 1997. ACM.