MetaFork: A Metalanguage for Concurrency Platforms Targeting Multicores

Xiaohui Chen[1], Marc Moreno Maza[1], Sushek Shekar[1] and Priya Unnikrishnan[2]

[1] Department of Computer Science, University of Western Ontario
[2] Compiler Development Team, IBM Toronto Lab

Abstract. We present MetaFork, a metalanguage for multithreaded algorithms based on the fork-join concurrency model and targeting multicore architectures. MetaFork is implemented as a source-to-source compilation framework allowing automatic translation of programs from one concurrency platform to another. The current version of this framework supports CilkPlus and OpenMP. We evaluate the benefits of the MetaFork framework through a series of experiments, such as narrowing performance bottlenecks in multithreaded programs. Our experiments show also that, if a native program, written either in CilkPlus or OpenMP, has little parallelism overhead, then the same holds for its OpenMP or CilkPlus counterpart translated by MetaFork.

1 Introduction

In the past decade the pervasive ubiquity of multicore processors has stimulated a constantly increasing effort in the development of concurrency platforms, such as CilkPlus, OpenMP, Intel TBB. While those programming languages are all based on the fork-join concurrency model, they largely differ in their way of expressing parallel algorithms and scheduling the corresponding tasks. Therefore, developing software code involving libraries written with several of those languages is a challenge.

Nevertheless there is a real need for facilitating interoperability between concurrency platforms. Consider for instance the field of symbolic computation. The DMPMC library (from the TRIP project `www.imcce.fr/trip` developed at the *Observatoire de Paris*) provides sparse polynomial arithmetic and is entirely written in OpenMP meanwhile the BPAS library (from the *Basic Polynomial Algebra Subprograms* `www.bpaslib.org` developed at the University of Western Ontario) provides dense polynomial arithmetic is entirely written in CilkPlus. Polynomial system solvers require both sparse and dense polynomial arithmetic and thus could take advantage of a combination of the DMPMC and BPAS libraries. However, CilkPlus and OpenMP have different run-time systems. In order to achieve interoperability between them, we propose an automatic source-to-source translation mechanism.

Another motivation for such a software tool is *comparative implementation* with the objective of narrowing performance bottlenecks. 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 inefficient 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 the translated code does scale, then one can suspect a parallelism overhead issue in the original code (say the grain-size of a parallel for-loop is too small).

In this paper, we propose MetaFork, a metalanguage for multithreaded algorithms based on the fork-join parallelism model [4] and targeting multicore architectures. By its parallel programming constructs, the MetaFork language is currently a super-set of CilkPlus [3, 11, 10] and includes the following widely used parallel constructs of OpenMP [13, 1]: `#pragma omp parallel`, `#pragma omp task`, `#pragma omp sections`, `#pragma omp section`, `#pragma omp for`, `#pragma omp taskwait`, `#pragma omp barrier`, `#pragma omp single` and `#pragma omp master`. However, MetaFork does not bind itself to any scheduling strategies (work stealing [5], work sharing, etc.) thus ignores for-loop scheduling policies. More generally, MetaFork does not make any assumptions about the run-time system.

The syntax and the semantics of MetaFork's parallel constructs are specified in Sections 2, 3 and 4. Since MetaFork is a faithful extension of the C/C++ language, this is actually sufficient to completely define MetaFork.

Recall that a driving motivation of the MetaFork project is to facilitate automatic translation of programs between the above mentioned concurrency platforms. To date, our experimental framework includes translators between CilkPlus and MetaFork (both ways) and, between OpenMP and MetaFork (both ways). Hence, through MetaFork, we are able to perform program translations between CilkPlus and OpenMP (both ways). Despite of the fact that it does not support all features of OpenMP, the MetaFork language is rich enough to capture the semantics of large bodies of OpenMP code, such as the *Barcelona OpenMP Tasks Suite* (BOTS) [8] and translate faithfully to CilkPlus each of the BOTS test cases.

In Section 5, we briefly explain how the translators of the MetaFork compilation framework are implemented. In particular, we specify which OpenMP data-sharing clauses are captured by MetaFork translators. Simple examples of code translation are provided through Figures 2, 3, 4, 5, 6, 7, 8, 9 and 10.

In Section 6, we evaluate the benefits of the MetaFork framework through a series of experiments. First, we show that MetaFork can help narrow down performance bottlenecks in multithreaded programs by means of comparative implementation, as discussed above. See the running time comparisons displayed on Figures 11, 12, 13. Secondly, we observe that, if a native CilkPlus (resp. OpenMP) program has little parallelism overhead, then the same holds for its

OpenMP (resp. CilkPlus) counterpart translated by MetaFork, see Table 1. We tested more than 20 examples in total for which experimental results can be found in the technical report [6] and for which code can be found on the web site of MetaFork project. Moreover, the source code of the MetaFork translators can be downloaded from the same web site at `http://www.metafork.org`.

2 Parallel Constructs of MetaFork

MetaFork extends both the C and C++ languages into a multithreaded language based on the fork-join concurrency model. Thus, concurrent execution is obtained by a parent thread creating and launching one or more child threads so that, the parent and its children execute a so-called *parallel region*. An important example of parallel regions are for-loop bodies. MetaFork has the following natural requirement regarding parallel regions: control flow cannot branch into or out of a *parallel region*.

MetaFork has four parallel constructs: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword `meta_fork` while the other two use respectively the keywords `meta_for` and `meta_join`. The parallel constructs of MetaFork grant permission for concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine. We emphasize that `meta_fork` allows the programmer to spawn a function call (like in CilkPlus) as well as a block (like in OpenMP). Examples of MetaFork code with CilkPlus and OpenMP, can be found through Figures 2, 3, 4, 5, 6, 7, 8, 9 and 10.

As mentioned, the `meta_fork` keyword is used to express the fact that a function call or a block is executed by a child thread, concurrently to the execution of the parent thread. If the program is run by a single processor, the parent thread is suspended during the execution of the child thread; when this latter terminates, the parent thread resumes its execution after the function call (or block) spawn. If the program is run by multiple processors, the parent thread may continue its execution after the function call (or block) spawn, without being suspended, meanwhile the child thread executes the function call (or block) spawn. In this latter scenario, the parent thread waits for the completion of the execution of the child thread, as soon as the parent thread reaches a synchronization point.

Spawning a function call with `meta_fork`. Spawning a call to the function `f`, called on the argument sequence `args`, is done by `meta_fork f(args)`. The semantics is similar to that of the CilkPlus counterpart `cilk_spawn` f(args). In particular, all the arguments in the argument sequence `args` are evaluated before spawning the function call `f(args)`. However, the execution of `meta_fork f(args)` differs from that of `cilk_spawn f(args)` on one aspect. There is an implicit `cilk_sync` at the end of the `Cilk` block [10] surrounding this latter `cilk_spawn`, while no such implicit barriers are assumed with `meta_fork`. This design decision is motivated by the fact that, in addition to fork-join parallelism, the MetaFork language may be extended to other forms of parallelism such as *parallel futures* [14, 2].

Spawning a block with `meta_fork`. The other usage of the `meta_fork` construct is for spawning a basic block `B`, which is done as follows: `meta_fork { B }`. If `B` consists of a single instruction, then the surrounding curly braces can be omitted. We also refer to this construction as a *parallel region*. There is no equivalent in CilkPlus while it is offered by OpenMP. Similarly to a function call spawn, this parallel region is executed by a child thread (once the parent thread reaches the `meta_fork` construct) meanwhile the parent thread continues its execution after the parallel region. Similarly also to a function call spawn, no implicit barrier is assumed at the end of the surrounding region. Hence synchronization points have to be added explicitly, using `meta_join`. A variable `v` which is not local to `B` may be either shared by both the parent and child threads; alternatively, the child thread may be granted a private copy of `v`. Precise rules about data attributes, for both parallel regions and parallel for-loops, are stated in Section 3.

Parallel for-loops with `meta_for`. Parallel for-loops in MetaFork have the following format `meta_for (I, C, S) { B }` where `I` is the *initialization expression* of the loop, `C` is the *condition expression* of the loop, `S` is the *stride* of the loop and `B` is the loop body. The specifications of `C`, `S`, `B` are standard and similar to the initialization expression, condition expression and stride of a CilkPlus for-loop. We refer to the MetaFork specifications document [7] for details. An implicit synchronization point is assumed after the loop body. That is, the execution of the parent thread is suspended when it reaches `meta_for` and resumes when all children threads (executing the loop body iterations) have completed their execution. As one can expect, the iterations of the parallel loop `meta_for (I, C, S) { B }` must execute independently of each other in order to guarantee that this parallel loop is semantically equivalent to its serial version `for (I, C, S) { B }`.

Synchronization point with `meta_join`. The construct `meta_join` indicates a *synchronization point* (or *barrier*) for a parent thread and its children tasks. More precisely, a parent thread reaching this point must wait for the completion of its children tasks but not for those of the subsequent descendant tasks.

3 Variable Attribute Rules

Variables that are *non-local* to the block of a parallel region may be either *shared* by or *private* to the threads executing the code pathes where those variables are defined. After a terminology review, we specify the rules that MetaFork uses in order to decide whether such a non-local variable is shared or private.

Shared and private variables. Consider a parallel region with block Y (resp. a parallel for-loop with loop body Y). We denote by X the immediate outer scope of Y. We say that X is the *parent region* of Y and that Y is a *child region* of X. A

variable v which is defined in Y is said *local* to Y; otherwise we call v a *non-local* variable for Y. Let v be a non-local variable for Y. Assume v gives access to a block of storage before reaching Y. (Thus, v cannot be a non-initialized pointer.) We say that v is *shared* by X and Y if its name gives access to the same block of storage in both X and Y; otherwise we say that v is *private* to Y. If Y is a parallel for-loop, we say that a local variable w is *shared* within Y whenever the name of w gives access to the same block of storage in any loop iteration of Y; otherwise we say that w is *private* within Y.

Value-type and reference-type variables. In the C programming language, a *value-type variable* contains its data directly as opposed to a *reference-type variable*, which contains a reference to its data. Value-type variables are either of primitive types (`char`, `float`, `int`, `double`, `void`) or user-defined types (`enum`, `struct`, `union`). Reference-type variables are pointers, arrays and functions.

`static` *and* `const` *type variables.* In the C programming language, a *static* variable is a variable that has been allocated statically and whose lifetime extends across the entire run of the program. This is in contrast to *automatic* variables (*local* variables are generally *automatic*) whose storage is allocated and deallocated on the call stack and, other variables (such as objects) whose storage is dynamically allocated in heap memory. When a variable is declared with the qualifier *const*, the value of that variable cannot typically be altered by the program during its execution.

```
/* This file starts here ... */
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
int a;
long par_region(long n){
        int b;
        int *c = (int *)malloc(sizeof(int)*10);
        int d[10];
        const int f=0;
        static int g=0;
        meta_fork{
           int e = b;
           subcall(c,d);
        }
}
```

```
/* ... and  continues here ... */
void subcall(int *a, int *b){
        for(int i=0;i<10;i++)
            printf("%d %d\n",a[i],b[i]);
}
int main(int argc, char **argv){
        long n=10;
        par_region(n);
        return 0;
}
/* ... and finishes here. */
```

Fig. 1: Various variable attributes in a parallel region.

Variable attribute rules of `meta_fork`. A non-local variable v which gives access to a block of storage before reaching Y is shared between the parent X and the child Y whenever v is: (1) a global variable, (2) a file scope variable, (3) a reference-type variable, (4) declared `static` or `const`, or (5) qualified `shared`. In all other cases, the variable v is private to the child. In particular, value-type variables (that are not declared `static` or `const`, or qualified `shared` and, that

are not global or file scope variables) are private to the child. In Figure 1, the variables `a`, `c`, `d`, `f` and `g` are shared, meanwhile the `b` and `e` are still private.

Variable attribute rules of `meta_for`. A non-local variable which gives access to a block of storage before reaching Y is *shared between parent and child*. A variable local to Y is *shared within* Y whenever it is declared `static`, otherwise it is private within Y. In particular, loop control variables are private within Y. In the example of Figure 6, the variable `b` is private, thus the OpenMP, MetaFork, CilkPlus codes of Figures 5, 6 and 7 are semantically equivalent.

The `shared` *qualifier.* Programmers can explicitly qualify a given variable as shared by using the directive `shared`. In the parallel regions of the example of Figure 9, the variables `sum_a` and `sum_b` are qualified shared. Hence the OpenMP, MetaFork and CilkPlus programs of Figure 8, 9 and 10 are semantically equivalent.

4 Semantics of the Parallel Constructs in MetaFork

In order to formally define the semantics of each of the parallel constructs in MetaFork, we introduce the *serial C-elision* of a MetaFork program $\mathcal{M}$: this is a C-language program $\mathcal{C}$ with same semantics as $\mathcal{M}$. In [7], we obtain such a serial C-elision $\mathcal{C}$ from the program $\mathcal{M}$ by means of a series of rewriting rules stated as Algorithms 1, 2, 3 and 4. We emphasize the fact that this algorithmic construction of such a $\mathcal{C}$ is a definition, not the statement of a property. Due to space consideration, we cannot include those rules here. However, we believe that sketching their principle is sufficient for understanding the rest of this paper.

As mentioned before, spawning a function call in MetaFork has the same semantics as spawning a function call in CilkPlus. More precisely: `meta_fork f(args)` and `cilk_spawn f(args)` are semantically equivalent.

A `meta_for` loop allows iterations of the loop body to be executed in parallel. By default, each iteration of the loop body is executed by a separate thread. However, using the `grainsize` compilation directive, one can specify the number of loop iterations executed per thread[3]: `#pragma meta grainsize = expression.` Nevertheless, in order to obtain the *serial C-elision* of a MetaFork for-loop, we require that the `meta_for` construct could be replaced by the C-language `for`, whatever is the grainsize of this MetaFork for loop, and without changing the initialization expression, condition expression and stride, and by replacing the loop-body with its serial C-elision.

Specifying the semantics of the spawning of a block in MetaFork is the difficult part. We do it in [7] in an algorithmic fashion, using rewriting rules, that are similar to a LEX-YACC program. The main idea is to use *outlining*, a widely used technique in the OpenMP community, see [12]. To have a taste of that transformation, one should observe how the MetaFork code of Figure 9 is transformed into the CilkPlus code of Figure 10. Obtaining the serial elision

[3] The loop iterations of a thread are then executed one after another by that thread.

of that latter code is easy and one can finally derive a serial C-elision for our input MetaFork code.

5 Translation

From CilkPlus *code to* MetaFork *code.* Translating code from CilkPlus to MetaFork is easy in principle since, up to the vectorization constructs of CilkPlus, the MetaFork language is a superset of CilkPlus. However, implicit CilkPlus barriers need to be explicitly inserted in the target MetaFork code. This implies that, during translation, it is necessary to trace the instruction stream DAG of the CilkPlus program in order to properly insert barriers in the generated MetaFork code.

From MetaFork *code to* CilkPlus *code.* Since CilkPlus has no constructs for spawning a block of code, we naturally use the *outlining technique* to: (1) wrap the parallel region as a function, and then (2) call that function concurrently. In fact, the problem of translating code from MetaFork to CilkPlus is equivalent to that of defining the serial elision of a MetaFork program.

From OpenMP *code to* MetaFork *code.* We first consider the translation of an OpenMP task directive: if it is a function call spawn, as in Figure 4, we use the MetaFork construct for spawning a function call. Otherwise, we use the MetaFork construct for spawning a block. Currently, we translate faithfully the following OpenMP optional clause directives: `shared`, `private` and `firstprivate`. For the translation of OpenMP sections to the MetaFork parallel regions we only support the default variable attribute and note that this case leads us to insert extra synchronization points. Finally, for the translation of an OpenMP parallel for-loop to MetaFork, we note that: (1) the `private` and `firstprivate` optional clause directives are faithfully translated, (2) every variable specified `private` is re-declared in the parallel for-loop of the MetaFork translation, (3) the loop control variables are initialized inside the loop, and (4) scheduling strategies of OpenMP parallel for loops are ignored,

From MetaFork *code to* OpenMP *code.* This is easy in principle, since the MetaFork language can be regarded as a subset of the OpenMP language. We note that function calls spawned with the `meta_fork` construct are translated using the task constructs of OpenMP.

6 Experimentation

We evaluate the performance and the usefulness of the four MetaFork translators (MetaFork to CilkPlus, CilkPlus to MetaFork, MetaFork to OpenMP, OpenMP to MetaFork). To this end, we run these translators on various input programs written either in CilkPlus or OpenMP, or both.

```
long fib(long n)
{
    long x, y;
    if (n<2) return n;
    else if (n<BASE)
        return fib_serial(n);
    else
    {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x+y);
    }
}
```

Fig. 2: CilkPlus code

```
long fib(long n)
{
    long x, y;
    if (n<2) return n;
    else if (n<BASE)
        return fib_serial(n);
    else
    {
        x = meta_fork fib(n-1);
        y = fib(n-2);
        meta_join;
        return (x+y);
    }
}
```

Fig. 3: MetaFork code

```
long fib(long n)
{
    long x, y;
    if (n<2) return n;
    else if (n<BASE)
        return fib_serial(n);
    else
    {
        #pragma omp task shared(x)
        x = fib(n-1);
        y = fib(n-2);
        #pragma omp taskwait
        return (x+y);
    }
}
```

Fig. 4: OpenMP code

```
int main()
{
  int a[N];
  int b = 0;
  #pragma omp parallel
  #pragma omp for private(b)
  for(int i = 0; i < N; i++)
  {
    b = i ;
    a[i] = b;
  }
}
```

Fig. 5: OpenMP code

```
int main()
{
  int a[N];
  int b = 0;

  meta_for(int i = 0; i < N; i++)
  {
    int b;
    b = i ;
    a[i] = b;
  }
}
```

Fig. 6: MetaFork code

```
int main()
{
  int a[N];
  int b = 0;

  cilk_for(int i = 0; i < N; i++)
  {
    int b;
    b = i ;
    a[i] = b;
  }
}
```

Fig. 7: CilkPlus code

We emphasize the fact that our purpose is not to compare the performance of the CilkPlus or OpenMP run-time systems. The reader should notice that the codes used in this study were written by different persons with different levels of expertise. In addition, the reported experimentation is essentially limited to one architecture (AMD Opteron) and one compiler (GCC). Therefore, it is delicate to draw any clear conclusions comparing CilkPlus and OpenMP.

We conducted two experiments. In the first one, we compared the performance of hand-written codes. The motivation, specified in the introduction, is *comparative implementation*. For this first purpose, we use a series of test-cases, each of them consisting of a pair of programs, one hand-written OpenMP program and one hand-written CilkPlus program. Within each pair, one program (written by a student) has a performance bottleneck while its counterpart (written by an expert) does not. We translate the inefficient program to the other language, then check whether the performance bottleneck remains or not, so as to narrow down the performance bottleneck in the inefficient program.

Our second experiment, 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

```
int main(){
  int sum_a=0, sum_b=0;
  int a[5] = {0,1,2,3,4};
  int b[5] = {0,1,2,3,4};
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        for(int i=0; i<5; i++)
          sum_a += a[i];
      }
      #pragma omp section
      {
        for(int i=0; i<5; i++)
           sum_b += b[i];
      }  }  }
}
```

Fig. 8: OpenMP code

```
int main()
{
  int sum_a=0, sum_b=0;
  int a[5] = {0,1,2,3,4};
  int b[5] = {0,1,2,3,4};

  meta_fork shared(sum_a){
        for(int i=0; i<5; i++)
          sum_a += a[i];
      }

  meta_fork shared(sum_b){
        for(int i=0; i<5; i++)
           sum_b += b[i];
      }

  meta_join;
}
```

Fig. 9: MetaFork code

```
void fork_func0(int* sum_a, int* a)
{
        for(int i=0; i<5; i++)
          (*sum_a) += a[i];
}
void fork_func1(int* sum_b, int* b)
{
        for(int i=0; i<5; i++)
           (*sum_b) += b[i];
}
int main()
{
  int sum_a=0, sum_b=0;
  int a[5] = {0,1,2,3,4};
  int b[5] = {0,1,2,3,4};
  cilk_spawn fork_func0(&sum_a, a);
  cilk_spawn fork_func1(&sum_b, b);
  cilk_sync;
}
```

Fig. 10: CilkPlus code

Table 1: Timings on AMD 48-core: underlined timings refer to original code and non-underlined timings to translated code.

Test	Input size	CilkPlus		OpenMP	
		Serial	T_1	Serial	T_1
Protein alignment (for)	100	568.07	566.10	<u>568.79</u>	<u>568.16</u>
quicksort	$5 \cdot 10^8$	<u>94.42</u>	<u>96.23</u>	94.15	97.20
prefixsum	$1 \cdot 10^9$	<u>27.06</u>	<u>28.48</u>	27.14	28.42
Fibonacci	$1 \cdot 10^9$	<u>96.24</u>	<u>96.26</u>	97.56	97.69
DnC_MM	$1 \cdot 10^9$	<u>752.04</u>	<u>752.74</u>	751.79	750.34
Mandelbrot	500×500	0.64	0.64	<u>0.64</u>	<u>0.65</u>

programs have similar serial and parallel running times as their hand-written-and-optimized counterparts.

Experimentation setup. For both experiments, apart from student's code, we use codes from the following sources: John Burkardt's Home Page`http://people.sc.fsu.edu/~%20jburkardt/c_src/openmp/openmp.html`, `http://hpc.mines.edu/examples/examples/openmp/index.html`, the BOTS [8] and the Cilk distribution examples `http://sourceforge.net/projects/cilk/`. The source code of those test cases was compiled as follows:

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

All our compiled programs were tested on AMD Opteron 6168 48-core nodes (with 256GB RAM and 12MB L3) and Intel Xeon 2.66GHz/6.4GT with 12 physical cores and hyper-threading,. For the first (resp. second) experiment, we measure *running time* (resp. *scalability*) by running our compiled OpenMP and CilkPlus programs on $1 \leq p \leq 48$ processors, as displayed on Figures 11,

12, 13 (resp. Figures 14, 15, 16). These measurements were repeated on Intel Xeon nodes, where we obtained results coherent with those obtained on the AMD Opteron nodes. We also measured *parallelism overheads* by running our compiled OpenMP and CilkPlus programs on $p = 1$ processor against their serial elisions. This is shown in Table 1.

For each experiment, we report below on three test cases. More test cases can be found in this technical report [6]. For our comparative implementation study, we consider three problems: *Parallel mergesort*, *Matrix inversion*, *Matrix transposition*. For *Parallel mergesort*, the original OpenMP code (written by a student) misses to parallelize the merge phase (and simply spawns the two recursive calls using OpenMP sections) while the original CilkPlus code (written by an expert) does both. On Figure 11, the running time 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 hand-written OpenMP code.

For *Matrix inversion*, the two original parallel programs are based on different serial algorithms for matrix inversion. The original OpenMP code uses Gauss-Jordan elimination while the original CilkPlus code uses a divide-and-conquer approach based on Schur's complement. Figure 12 shows that the code translated from CilkPlus to OpenMP is more appropriate for fork-join multithreaded languages targeting multicores.

Matrix transposition is a challenging operation on multicore architectures. Without doing complexity analysis, discovering the fact 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. Here again, the translation narrows the algorithmic issue, as shown on Figure 13.

For the automatic translation of highly optimized code. we consider: *Divide-and-conquer matrix multiplication* (*DnC MM*), *Protein alignment*, *Mandelbrot set computation*. For *DnC MM*, we have translated the original CilkPlus code to OpenMP. This algorithm has high parallelism and optimal cache complexity [9]. The speedup curves with inputs of order 8192 are shown in Figure 14. For *Protein alignment*, the underlying algorithm uses dynamic programming, and runs in quadratic time w.r.t. input size. Load balancing is challenging for this application. The speedup curves with 100 input proteins are shown in Figure 15. *Mandelbrot set computation* is a compute-intensive application with simpler data traversal than the previous two. The speedup curves for a 500-by-500 grid and 2000 iterations is shown in Figure 16.

7 Conclusion

MetaFork allows for rapidly prototyping algorithms written for one concurrency platform to another. As we have seen in Section 6, MetaFork can be applied for (1) comparing algorithms written with different concurrency platforms and (2) porting more programs to systems that may have a highly optimized

run-time for one paradigm (say divide-and-conquer algorithms, or producer-consumer). The MetaFork translation framework may also avoid the negative interferences of having multiple interfaces between the different components of a large solver written with various concurrency platforms. Along the same idea, the MetaFork translators can be used to transform legacy code into a more adequate concurrency platform.

Acknowledgments. This work was supported in part by NSERC of Canada and in part by an IBM CAS Fellowship in 2013 and 2014. We are also grateful to Abdoul-Kader Keita (IBM Toronto Lab) for his advice and technical support.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[2] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory Comput. Syst.*, 32(3):213–239, 1999.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[4] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[6] X. Chen, M. Moreno Maza, and S. Shekar. Experimenting with the MetaFork framework targeting multicores. Technical report, Univ. of Western Ontario, 2013.

[7] X. Chen, M. Moreno Maza, and S. Shekar. Metafork: A metalanguage for concurrency platforms targeting multicores. Technical report, Univ. of Western Ontario, 2013.

[8] 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 *Proc. of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[9] 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.

[10] Intel Corporation. Intel CilkPlus language specification, version 0.9, 2013.

[11] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[12] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(18):2317–2332, Dec. 2007.

[13] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0, 2013.

[14] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In F. Meyer auf der Heide and M. A. Bender, editors, *SPAA*, pages 91–100. ACM, 2009.

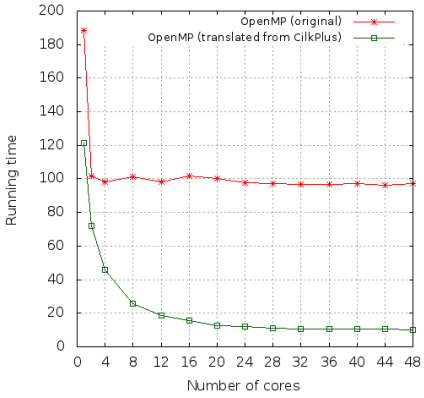

Fig. 11: Parallel mergesort of size 5×10^8

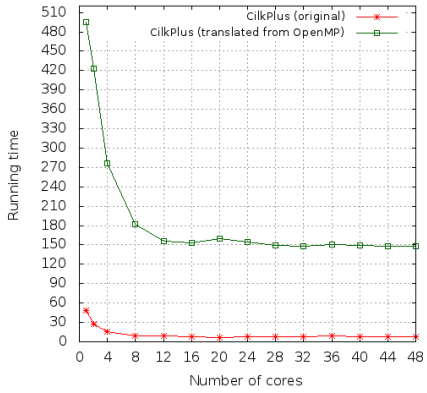

Fig. 12: Matrix inversion of order 4096

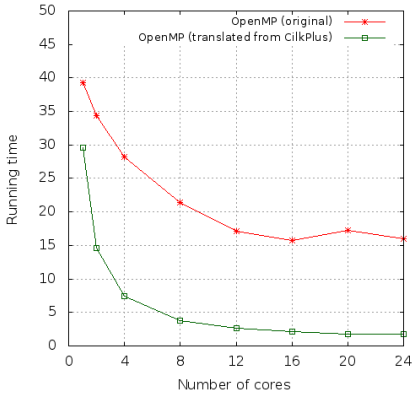

Fig. 13: Matrix transposition of order 32768

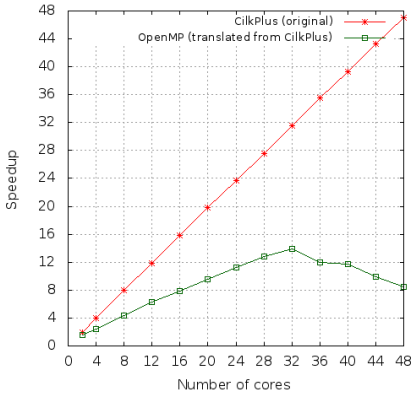

Fig. 14: DnC MM of order 8192

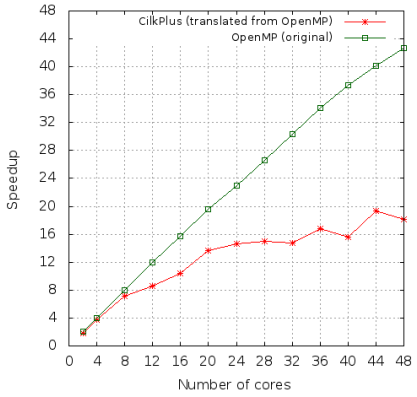

Fig. 15: Protein alignment - 100 Proteins.

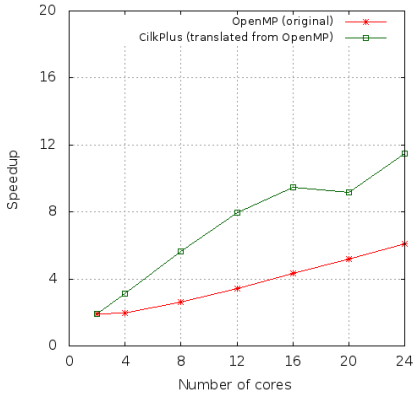

Fig. 16: Mandelbrot set computation: 500×500 grid and 2000 iterations.