MetaFork: A Framework 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 property 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 and 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 combining 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[1] provides sparse polynomial arithmetic and is entirely written in OpenMP, meanwhile the BPAS library[2] provides dense polynomial arithmetic and 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

---

[1] From the TRIP project `www.imcce.fr/trip` developed at the *Observatoire de Paris*.
[2] From the *Basic Polynomial Algebra Subprograms* `www.bpaslib.org` developed at the University of Western Ontario.

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 [5] and targeting multi-core architectures. By its parallel programming constructs, the METAFORK language is currently a super-set of CILKPLUS [4, 13, 11] and offers counterparts for the following widely used parallel constructs of OPENMP [16, 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 make any assumptions about the run-time system, in particular about scheduling strategies (work sharing, work stealing [6]). In fact, METAFORK is not designed to be a target language, but rather as the internal intermediate representation (IR) of a source-to-source compiler framework for multithreaded languages.

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 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 perform program translations between CILKPLUS and OPENMP (both ways). Integrating TBB in this framework is work in progress.

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) [10] and translate faithfully to CILKPLUS each of the BOTS test cases. In the other direction, we could translate the BPAS library to OPENMP.

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 the METAFORK translators. Simple examples of code translation are provided.

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. 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. We tested more than 20 examples in total for which experimental results can be found in the technical report [8] and for which code can be found on the web site of the METAFORK project. Moreover, the source code of the METAFORK translators can be downloaded from the same web site at `http://www.metafork.org`.

Related work. While the well-developed source-to-source compiler framework ROSE³ has been used to support many programming languages, including OPENMP and UPC, we are not aware of a ROSE-based platform similar to METAFORK, that is, providing source-to-source translation between multithreaded languages. On the other hand, several projects offer automatic one-way translation from a concurrency platform running on one hardware architecture to another concurrency platform running on another hardware architecture, e.g. OPENMP shared-memory code to MPI distributed-memory code as in the papers [2] [9] (HOMPI Project) or [15] (OPENMP Accelerator Model). Other projects offer extension of a concurrency platform from one hardware architecture to another hardware architecture, like HOMP [15] or OPENMPC [12] which allow extended OPENMP code to run on NVIDIA GPUs. In contrast to these two types of projects, METAFORK is currently dedicated to a single type of hardware architecture, namely multicore processors. However, METAFORK offers automatic *two-way* translations. Moreover, the generated code is human-readable, as illustrated by the examples available on the METAFORK web site.

2   Parallel Constructs and Execution Model 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 the fact 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 5, 6, 7, 8, 9, 10, 11, 12 and 13.

As mentioned, the keyword `meta_fork` is used to express the fact that a function call or a block is executed by a child thread, concurrently to the execution

---

³ `http://en.wikibooks.org/wiki/ROSE_Compiler_Framework`

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[4] 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`, with 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 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 feature. While there is an implicit `cilk_sync` at the end of the `Cilk` block [11] surrounding this latter `cilk_spawn`, no such implicit barriers are assumed with `meta_fork`. This feature is motivated by the fact that, in addition to the fork-join parallelism, we plan to extend the METAFORK language to other forms of parallelism such as *parallel futures* [17, 3].

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 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. The parent thread will share the work of executing the iterations of the loop with

---

[4] In fact, the parent thread does not participate to the execution of a function call (or block) spawn, but will participate to the execution of the iterations of a parallel for-loop.

the child threads. 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 paths 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 (or a parallel for-loop with loop body Y). X denotes 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 to be *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. In particular, if Y is a parallel for-loop we say that a local variable w is *shared* by Y whenever the name of w gives access to the same block of storage in any loop iteration of Y, which means that all the threads that execute this parallel for-loop share the same variable w; otherwise we say that w is *private* to 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 ... */          /* ... and  continues here ... */
#include<stdio.h>                         void subcall(int *a,int *b){
#include<time.h>                                 for(int i=0;i<10;i++)
#include<stdlib.h>                                    printf("%d %d\n",a[i],b[i]);
int a;                                    }
long par_region(long n){                  int main(int argc,char **argv){
      int b;                                     long n=10;
      int *c = (int *)malloc(sizeof(int)*10);    par_region(n);
      int d[10];                                 return 0;
      const int f=0;                       }
      static int g=0;                      /* ... and finishes here. */
      meta_fork{
         int e = b;
         subcall(c,d);
      }
}
```

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 private.

```
/* To illustrate variable attributes, three    /* This file is b.cpp*/
   files (a headerfile "a.h" and two source     #include<stdio.h>
   files "a.cpp" and "b.cpp") are used.         #include<stdlib.h>
This file is a.cpp  */                          #include<time.h>
#include<stdio.h>                               #include"a.h"
extern int var;                                 int var = 100;
void test(int *array)                           int main(int argc,char **argv)
{                                               {
  int basecase = 100;                             int *a=(int*)malloc(sizeof(int)*10);
  meta_for(int j = 0; j < 10; j++)                srand((unsigned)time(NULL));
  {                                               for(int i=0;i<10;i++)
        static int var1=0;                          a[i]=rand();
        int i = array[j];                         test(a);
        if( i < basecase )                        return 0;
                array[j]+=var;                   }
  }                                               /* This file is a.h*/
}                                                 void test(int *a);
```

Fig. 2. Example of shared and private variables with `meta_for`

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 by* Y whenever it is declared `static`, otherwise it is private to Y. In particular, loop control variables are private to Y. In the example of Figure 2, the variables `array`, `basecase`, `var` and `var1`, are shared by all threads while the variables i and j are private. In the example of Figure 9, the variable

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

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

Fig. 3. Parallel fib code using a function spawn

Fig. 4. Parallel fib code using a block spawn

b is private, thus the OpenMP, MetaFork, CilkPlus codes of Figures 8, 9 and 10 are semantically equivalent.

The `shared` *keyword.* Programmers can explicitly qualify a given variable as shared by using the `shared` keyword. In the example of Figure 3, the variable `n` is private to `fib_parallel(n-1)`. In Figure 4, we specify the variable `x` as shared and the variable `n` is still private. Notice that the programs in Figures 3 and 4 are semantically equivalent. In the parallel regions of the example of Figure 12, the variables `sum_a` and `sum_b` are qualified shared. Hence the OpenMP, MetaFork and CilkPlus programs of Figure 11, 12 and 13 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 program $\mathcal{C}$ expressed in the C-language and with the 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. Due to space consideration, we cannot include this algorithmic definition here. However, we believe that sketching its 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[5]: `#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

---

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

by the C-language `for` - whatever is the grainsize of this METAFORK for loop - without changing the initialization expression, condition expression and stride. (Of course, the loop-body must be replaced 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 [14]. To have a taste of that transformation, one should observe how the METAFORK code of Figure 12 is transformed into the CILKPLUS code of Figure 13. Obtaining the serial elision of that latter code is easy and one can finally derive a serial C-elision for our input METAFORK code.

5   Translation

In this section, we briefly explain how the translators of the METAFORK compilation framework are implemented. Obviously, for each translator, the semantics of each input program are preserved into the output program. However, scheduling strategies (like an OPENMP clause `schedule(static, chuksize)`) are ignored by our translators. Retaining them (at least as structured comments) will be explored in a future release of METAFORK.

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 7, 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.

```
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. 5. 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. 6. 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. 7. 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. 8. 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. 9. 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. 10. CILKPLUS code

6   Experimentation

In this section, 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.

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

```
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];
      } } }
}
```

```
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;
}
```

```
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. 11. OPENMP code

Fig. 12. METAFORK code

Fig. 13. CILKPLUS code

of expertise. In addition, the reported experimentation is essentially limited to one architecture (AMD Opteron) and one compiler (GCC). Therefore, it would be delicate to draw any clear conclusions comparing CILKPLUS and OPENMP. We conducted three sets of experiments:

- In the first one, we compared the performance of hand-written codes. The motivation, specified in the introduction, is *comparative implementation.*
- In the second one, we translated large portions of the BPAS library from CILKPLUS to OPENMP, motivated by the *interoperability* question raised in the introduction.
- In the last experiment, we compared the *parallelism overheads* measured the original codes (either CILKPLUS or OPENMP) and their translated counterparts.

Before reporting on these three sets of experiments:

- we describe the *setup* (hardware, software) in which they were conducted and,
- we explain how we verified the *correctness* of the multithreaded code generated by our translators.

Experimentation setup. For all experiments, apart from student's code, we use codes from the following sources:

- The BPAS library http://www.bpaslib.org,
- John Burkardt's Home Page
  http://people.sc.fsu.edu/~%20jburkardt/c_src/openmp/openmp.html,
- the BOTS [10] 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
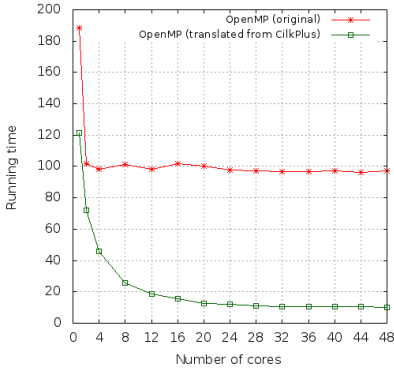
Fig. 14. Parallel mergesort in size 5×10^8
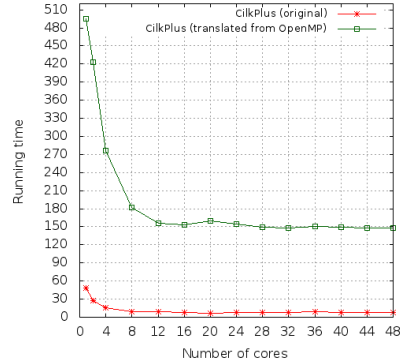
Fig. 15. Matrix inversion of order 4096

We run all our programs on AMD Opteron 6168 48-core nodes (with 256GB RAM and 12MB L3) and Intel Xeon 2.66GHz/6.4GT with 12-cores nodes.

Correctness. Validating the correctness of our translators was a major requirement of our work. Depending on the test-case, we could use one of the following strategies.

- 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.

Comparative implementation. For this first purpose, we use a series of test-cases, each of them consisting of a pair of hand-written programs: one written in OPENMP and the other in CILKPLUS. Within each pair, a program S, written by a student, has a performance bottleneck; meanwhile its counterpart E, written by an expert does not. For each pair, we translate one program (either S or E) to the other language. For these two programs (expressed in the same concurrency platform) we measure the running time on p processors, for $1 \leq p \leq 48$, and compare the resulting data so as to narrow down the performance bottleneck in the inefficient program. Figures 14 and 15 illustrate two test-cases: *Parallel mergesort*, *Matrix inversion*. More test-cases can be found in this technical report [8].

Table 1. BPAS timings with 1 and 16 workers: original CILKPLUS code and translated OPENMP code

Test	Input size	CILKPLUS		OPENMP	
		T_1	T_{16}	T_1	T_{16}
8-way	2048	0.423	0.231	0.421	0.213
Toom-Cook	4096	1.849	0.76	1.831	0.644
	8192	9.646	2.742	9.241	2.774
	16384	39.597	9.477	39.051	8.805
	32768	174.365	34.863	172.562	33.032
DnC	2048	0.874	0.259	0.867	0.299
Plain	4096	3.95	1.264	3.925	1.123
Polynomial	8192	18.196	3.335	18.154	4.428
Multiplication	16384	77.867	12.778	75.885	12.674
	32768	331.351	55.841	332.126	55.925

Table 2. 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>

– 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 parallelize the merge phase. On Figure 14, the running time curve of the translated OPENMP code is as theoretically expected while the curve of the original OPENMP code shows a limited scalability. This suggests that the original hand-written OPENMP code should expose more parallelism.

– For *Matrix inversion*, the two original parallel programs are based on different serial algorithms for inverting a dense matrix. 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 15 shows that the code translated from CILKPLUS to OPENMP is more appropriate

for fork-join multithreaded languages targeting multicores. In other words the Schur's complement approach should be prefered in this context.

Interoperability. Our second experiment is dedicated to automatic translation of highly optimized libraries. The motivation, presented in the introduction, is to facilitate interoperability between libraries developed for different concurrency platforms, namely CILKPLUS and OPENMP. For this question, we want to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts. For this experiment, we have used the BPAS library which counts more than 150,000 lines of CILKPLUS code. Half of those lines are dedicated to polynomial multiplication and we translated those to OPENMP. In Table 1, we report on timings of two of the main algorithms for polynomial multiplication, namely 8-way Toom-Cook and divide-and-conquer plain multiplication. One can see that the original and translated codes have similar running times on 1 and 16 cores, for all input data sizes that we tested. Therefore, the OPENMP version of the BPAS library retains the good performance of the original version written in CILKPLUS.

Parallelism overheads. Our third experiment is devoted to the following question: do the METAFORK translators add extra parallelism overheads to the generated code w.r.t. the original code? We focus here on *work overhead*. By work overhead, we mean the time ratio between a multithreaded program run on one core and its serial elision. For this experiment, we have considered original programs using different parallelism patterns (divide-and-conquer, parallel for-loops) and written in both OPENMP and CILKPLUS. Our results are collected in Table 2. For all the examples that we tested, we could observe that, if the original program has little work overhead, then the same holds for the translated program.

7   Concluding Remarks

METAFORK allows for rapidly mapping 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.

Last but not least, we think that a great benefit of METAFORK is the abstraction that it provides. It can be useful for parallel language design (for example in designing parallel extensions to C/C++) as well as a good tool to teach parallel programming.

In the future work, as discussed in the METAFORK specifications document [7], we will consider parallel reduction as an important extension and include other parallel computing models like pipelining.

www.metafork.org
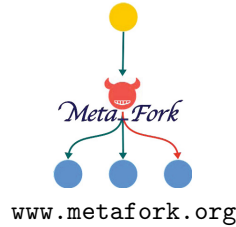
References

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

[2] Basumallik, A., Eigenmann, R.: Towards automatic translation of OpenMP to MPI. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 189–198. ACM, New York (2005)

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

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

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

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

[7] Chen, X., Moreno Maza, M.: MetaFork: A metalanguage for concurrency platforms targeting multicores. Technical report, U. of Western Ontario (2013)

[8] Chen, X., Moreno Maza, M., Shekar, S.: Experimenting with the MetaFork framework targeting multicores. Technical report, U. of Western Ontario (2013)

[9] Dimakopoulos, V.V., Hadjidoukas, P.E.: HOMPI: A hybrid programming framework for expressing and deploying task-based parallelism. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 14–26. Springer, Heidelberg (2011)

[10] Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: 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 2009, pp. 124–131. IEEE Computer Society, Washington, DC (2009)

[11] Intel Corporation. Intel CilkPlus language specification, version 0.9 (2013)

[12] Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society (2010)

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

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

[15] Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013)

[16] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0 (2013)

[17] Spoonhower, D., Blelloch, G.E., Gibbons, P.B., Harper, R.: Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In: Meyer auf der Heide, F., Bender, M.A., (ed.) SPAA, pp. 91–100. ACM (2009)