MetaFork: A Metalanguage for Concurrency Platforms Targeting Multicores

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

September 1, 2013

Document number: N1746
Date: 2013-09-01
Project: Programming Language C, WG14 CPLEX Study Group
Authors:
Xiaohui Chen, xchen422@csd.uwo.ca
Marc Moreno Maza, moreno@csd.uwo.ca
Sushek Shekar, sshekar@uwo.ca
Reply-to: Michael Wong, michaelw@ca.ibm.com
Revision: 1

1 Introduction

Today most computers are equipped with multicore processors leading to a constantly increasing effort in the development of the concurrency platforms which target those architectures, such as CilkPlus, OpenMP, Intel TBB. While those programming languages are all based on the fork-join parallelism model, they largely differ on their way of expressing parallel algorithms and scheduling the corresponding tasks. Therefore, developing programs involving libraries written with several of those languages is a challenge.

In this note, we propose MetaFork, a metalanguage for multithreaded algorithms based on the fork-join parallelism model and targeting multicore architectures. By its parallel programming constructs, this language is currently a common denominator of CilkPlus and OpenMP. However, this language does not compromise itself in any scheduling strategies (work stealing, work sharing, etc.) Thus, it does not make any assumptions about the run-time system.

The purpose of this metalanguage is to facilitate automatic translations of programs between the above 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). Adding Intel TBB to this framework is a work in progress. As a consequence, the MetaFork program examples given in this note were obtained either from original CilkPlus programs or original OpenMP programs. See Appendices A, B, C for an example.

This report focuses on the syntax and semantics of MetaFork. A presentation of our experimental framework will appear elsewhere.

2 Basic Principles

We summarize in this section a few principles that guided the design of MetaFork. First of all, MetaFork is an extension of C/C++ and a multithreaded language based on the fork-join parallelism 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 examples of parallel regions are for-loop bodies.

Similarly to CilkPlus, the parallel constructs of MetaFork allow concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine. Moreover, the semantics of a MetaFork program (assumed to be free of data-races) is defined by its serial elision. This latter is obtained by erasing the keywords `meta_fork` and `meta_join` and by replacing the keyword `meta_for` by `for`.

As mentioned above, MetaFork does not make any assumptions about the run-time system, in particular about task scheduling. Another design principle is to encourage a programming style limiting thread communication to a minimum so as to

- prevent from data-races while preserving a satisfactory level of expressiveness and,
- minimize parallelism overheads.

To some sense, this principle is similar to that of CUDA which states the execution of a given kernel should be independent of the other in which its thread blocks are executed. Returning to our concurrency platforms targeting multicore architectures, OpenMP offers several clauses which can be used to exchange information between the threads (like `threadprivate`, `copyin` and `copyprivate`) while no such mechanism exists in CilkPlus. Of course, this difference follows from the fact that, in CilkPlus, one can only fork a function call while OpenMP allows other code regions to be executed by several threads. MetaFork has both parallel constructs. For the latter, being able to explicitly qualify variables *shared* or *private* is a clear need and MetaFork offers this feature. However, this is the only mechanism for communication among threads that MetaFork provides explicitly.

3 MetaFork Syntax

Essentially, MetaFork has three keywords, namely `meta_fork`, `meta_for` and `meta_join`, described below. We stress the fact that the first one allows for spawning function calls (like in CilkPlus) as well as concurrent execution of a region (like in OpenMP).

1. `meta_fork`

   Description: the `meta_fork` keyword is used to express a function call or a piece of code which could be run concurrently with the parent thread.

   - `meta_fork` function (...)
     - we call this construct a *function spawn*,

     Figure 1 below illustrates how `meta_fork` can be used to define a function spawn. The underlying algorithm is the cache oblivious divide and conquer matrix multiplication.
   - `meta_fork` { ...code ... }
     - we call this construct a *parallel region*,
     - no equivalent in `CilkPlus`.

     Figure 2 below, illustrates how `meta_fork` can be used to define a parallel region. The underlying algorithm is one of the two-subroutines in the work-efficient parallel prefix sum.

   On the contrary of CilkPlus, no implicit barrier is assumed at the end of a function spawn or a parallel region. Hence synchronization points have to be added explicitly, using `meta_sync`, as in the examples of Figures 2 and 6.

2. `meta_for (`*`initialization expression, condition expression, stride,`*`[ chunksize ])` { *`loop body`* }

   Description: the `meta_for` keyword allows the for loop to run in parallel.

   - the *initialization expression* initializes a variable, called the *control variable* which can be of type integer, iterator or pointer.
   - *condition expression* compares the control variable with a compatible expression, using of the following operators:

     $<$
     $<=$
     $>$
     $>=$
     $!=$
   - the *stride* uses one the following operation to increase or decrease the value of the control variable:

```
template<typename T> void DnC_matrix_multiply(int i0, int i1, int j0,
    int j1, int k0, int k1, T* A, int lda, T* B, int ldb, T* C,int ldc)
{
    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= RECURSION_THRESHOLD) {
        int mi = i0 + di / 2;
        meta_fork DnC_matrix_multiply(i0,mi,j0,j1,k0,k1,A,lda,B,ldb,C,ldc);
        DnC_matrix_multiply(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc);
        meta_sync;
    } else if (dj >= dk && dj >= RECURSION_THRESHOLD) {
        int mj = j0 + dj / 2;
        meta_fork DnC_matrix_multiply(i0,i1,j0,mj,k0,k1,A,lda,B,ldb,C,ldc);
        DnC_matrix_multiply(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc);
        meta_sync;
    } else if (dk >= RECURSION_THRESHOLD) {
        int mk = k0 + dk / 2;
        DnC_matrix_multiply(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc);
        DnC_matrix_multiply(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc);
    } else {
        for (int i = i0; i < i1; ++i)
            for (int k = k0; k < k1; ++k)
                for (int j = j0; j < j1; ++j)
                    C[i * ldc + j] += A[i * lda + k] * B[k * ldb + j];
    }
}
```

Figure 1: Example of a MetaFork program with function spawns.

```
long int parallel_pscanup (long int x [], long int t [], int i, int j)
{
        if (i == j) {
                return x[i];
        }
        else{
                int k = (i + j)/2;
                int right;
                meta_fork {
                        t[k] = parallel_pscanup(x,t,i,k);
                }
                right = parallel_pscanup (x,t, k+1, j);
                meta_sync;
                return t[k] + right;}
}
```

Figure 2: Example of a MetaFork program with a parallel region.

```
template<typename T> void multiply_iter_par(int ii, int jj, int kk, T* A, T* B,
    T* C)
{
    meta_for(int i = 0; i < ii; ++i)
        for (int k = 0; k < kk; ++k)
            for(int j = 0; j < jj; ++j)
                C[i * jj + j] += A[i * kk + k] + B[k * jj + j];
}
```

Figure 3: Example of `meta_for`

$++$
$--$
$+=$
$-+$
$var = var$ +/- incr

- the *chunksize* specifies the number of loop iterations executed per thread; the default value is one.

Following the principles stated in Section 2, the iterations must be independent. Note that there is an implicit barrier after the loop body, since this is expected in most practical examples, and MetaFork should not puzzle programmers on this. Figure 3 displays an example for `meta_fork`, where the underlying algorithm is the naive (and cache-inefficient) matrix multiplication algorithm.

3. `meta_join`

```
void test(int *array)
{
        int basecase = 100;
        meta_for(int j = 0; j < 10; j++)
        {
                int i = array[j];
                if( i < basecase )
                        array[j]++;
        }
}
```

Figure 4: Example of shared and private variables with `meta_for`.

Description: this directive indicates a *synchronization point*. It only waits for the completion of the children tasks but not for those of the subsequent descendant tasks.

4 Variable Attribute

Following principles implemented in other multithreaded languages, in particular OpenMP, the variables of a MetaFork program, that are involved within a parallel region or a parallel for loop, can be either private or shared among threads. In a MetaFork parallel for-loop, variables that are defined for the first time in the body of that loop (i.e not defined before reaching the loop body) are considered private, hence each thread participating to the for-loop execution has a private copy. Meanwhile, variables that are defined before reaching the for-loop are considered shared. Consequently, in the example of Figure 4, the variables `array` and `basecase` are shared by all threads while the variable `i` is private.

For the `meta_fork` directives, by default, the variables passed to a function spawn or occurring in a parallel region are private. However, programmers can qualify a given variable as shared by using the directive `shared` explicitly. In the example of Figure 5, the variable `n` is private to `fib_parallel(n-1)`. In Figure 6, we specify the variable `x` as shared and the variable `n` is still private. Notice that the programs in Figures 5 and 6 are equivalent, in the sense that they compute the same thing.

5 Extensions of MetaFork

The MetaFork language as defined in the previous sections serves well as a metalanguage for multithreaded programs, or equivalently, as a pseudo-code language for multithreaded algorithms.

```
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);}
}
```

Figure 5: Example of private variables in a function spawn.

```
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);}
}
```

Figure 6: Example of a shared variable attribute in a function spawn.

```
void reduction(int* a)
{
        int max = 0;
        meta_for (int i = 0; i < MAX; i++)
        {
                reduction:MAX max;
                if(a[i] > max)
                        max = a[i];
        }
}
```

Figure 7: Reduction example in MetaFork

Operator	Initial value	Description
+	0	performs a sum
-	0	performs a subtraction
*	1	performs a multiplication
&	0	performs bitwise AND
|	0	performs bitwise OR
^	0	performs bitwise XOR
&&	1	performs logical AND
||	0	performs logical OR
MAX	0	largest number
MIN	0	least number

Table 1: Typical binary operations involved in reducers.

In order to serve also as an actual programming language, we propose two extensions. The first one provides support for reducers, a very popular parallel construct. The second one controls and queries workers (i.e. working cores) at run-time.

5.1 Reduction

A reduction operation combines values into a single accumulation variable when there is a true dependence between loop iterations that cannot be trivially removed. Support for reduction operations is included in most parallel programming languages. In Figure 7, the MetaFork program computes the maximum element of an array.
Reduction variables (also called *reducers*) are defined as follows:

reduction : ***op var_list***

where *op* stands for an associative binary operation. Typical such operations are listed in Table 1.

5.2 Run-time API

In order to conveniently run an actual MetaFork, we propose the following run-time support functions:

1. **void meta_set_nworks(int arg)**

   Description: sets the number of requested cores (also called workers in this context).

2. **int meta_get_nworks(void)**

   Description: gets the number of available cores.

3. **int meta_get_worker_self(void)**

   Description: obtains the ID of the calling thread.

6 Conclusion

In this proposal, we have presented the three fundamental constructs of the MetaFork language, namely `meta_fork`, `meta_for`, `meta_join` which can be used to express *multithreaded algorithms* based on task level parallelism. In the last section, we have also proposed additional constructs which are meant to facilitate the translation of *multithreaded programs* from one concurrency platform targeting multicore architectures to another.

We believe that programmers may benefit from MetaFork to develop efficient multithreaded code because the constructs are easy to understand while being sufficient to express many desirable parallel algorithms. Moreover, the infrastructure of MetaFork is compatible with popular concurrency platforms targeting multicores: we could verify this fact experimentally thanks to our translators to and from OpenMP and CilkPlus. In addition, we could use this infrastructure to understand performance issues in a program originally written with one of these languages by comparing it with its counterpart automatically generated to the other language.

In a sum, to ease programmers' activity who need to work with several concurrency platforms, we beleive that it is really necessary to have a well-designed unified parallel language like MetaFork.

A Original OpenMP code

```
long int parallel_pscanup (long int x[], long int t[], int i, int j)
{
        if ((j-i)<=base)
        {
                int re = serial_pscanup(x,t,i,j);
                return re;
        }
    else if (i == j) {
        return x[i];
     }
     else
     {
        int k = (i + j)/2;
        int right;
        #pragma omp task
        {
        t[k] = parallel_pscanup(x,t,i,k);
        }
        right = parallel_pscanup (x,t, k+1, j);
        #pragma omp taskwait
        return t[k] + right  ;
     }
}

int parallel_pscandown (long int u, long int x[], long int t[], long int y[], int i, int j)
{
        if ((j-i)<=base)
        {
                serial_pscandown(u,x,t,y,i,j);
                return 0;
        }
    else if (j != i ) {
         int k = (i + j)/2;
         y[j] = t[k] + u;
         y[k] = u;
        #pragma omp task
        {
      parallel_pscandown (y[j], x, t, y, k+1, j);
        }
     parallel_pscandown (y[k], x, t, y, i, k);
     }
return 0;
}
```

B Translated MetaFork code from OpenMP code

```
long int parallel_pscanup (long int x[], long int t[], int i, int j)
{
        if ((j-i)<=base)
        {
                int re = serial_pscanup(x,t,i,j);
                return re;
        }
    else if (i == j) {
        return x[i];
     }
     else
     {
        int k = (i + j)/2;
        int right;

        {
        t[k] =meta_fork parallel_pscanup(x,t,i,k);
        }
        right = parallel_pscanup (x,t, k+1, j);
        meta_join;
        return t[k] + right  ;
     }
}

int parallel_pscandown (long int u, long int x[], long int t[], long int y[], int i, int j)
{
        if ((j-i)<=base)
        {
                serial_pscandown(u,x,t,y,i,j);
                return 0;
        }
    else if (j != i ) {
         int k = (i + j)/2;
         y[j] = t[k] + u;
         y[k] = u;

        {
meta_fork       parallel_pscandown (y[j], x, t, y, k+1, j);
        }
     parallel_pscandown (y[k], x, t, y, i, k);
     }
return 0;
}
```

C Translated CilkPlus code from MetaFork code

```
long int parallel_pscanup (long int x[], long int t[], int i, int j)
{
        if ((j-i)<=base)
        {
                int re = serial_pscanup(x,t,i,j);
                return re;
        }
    else if (i == j) {
        return x[i];
     }
     else
     {
        int k = (i + j)/2;
        int right;

        {
        t[k] =cilk_spawn parallel_pscanup(x,t,i,k);
        }
        right = parallel_pscanup (x,t, k+1, j);
        cilk_sync ;
        return t[k] + right  ;
     }
}

int parallel_pscandown (long int u, long int x[], long int t[], long int y[], int i, int j)
{
        if ((j-i)<=base)
        {
                serial_pscandown(u,x,t,y,i,j);
                return 0;
        }
    else if (j != i ) {
         int k = (i + j)/2;
         y[j] = t[k] + u;
         y[k] = u;

        {
cilk_spawn parallel_pscandown (y[j], x, t, y, k+1, j);
        }
     parallel_pscandown (y[k], x, t, y, i, k);
     }
return 0;
}
```