MetaFork: A Metalanguage for Concurrency Platforms Targeting Multicores

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

29 Ocotober 2013

1 Introduction

In the past decade the pervasive ubiquitous of multicore processors has stimulated a constantly increasing effort in the development of concurrency platforms (such as CilkPlus, OpenMP, Intel TBB) targeting those architectures. 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.

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 advantage of a combination of the DMPMC and BPAS libraries. Unfortunately, the run-time systems of CilkPlus and OpenMP cannot cooperate since their schedulers are based on different strategies, namely *work stealing* and *work sharing*, respectively. Therefore, an automatic translation mechanism is needed between CilkPlus and OpenMP.

Another motivation for such software tool is *comparative implementation* with the objective of narrowing performance bottleneck. 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).

In this note, we propose MetaFork, a metalanguage for multithreaded algorithms based on the fork-join parallelism model [5] and targeting multicore architectures. By its parallel programming constructs, this language is currently a super-set of CilkPlus [4, 7, 9] and OpenMP Tasks [1]. However, this language does not compromise itself in any scheduling strategies (work stealing [6], work sharing, etc.) Thus, it does not make any assumptions about the run-time system.

Based on the motivations stated above, a driving application of MetaFork is to facilitate automatic translations 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). Adding Intel TBB to this framework is work in progress. As a consequence, the MetaFork program examples given in this note were obtained either from original CilkPlus programs or from original OpenMP programs.

This report focuses on specifying the syntax and semantics of the parallel constructs of MetaFork. Indeed, MetaFork differs only from the C language by its parallel constructs. We also describe the main ideas underlying the implementation of our translators between MetaFork and, the CilkPlus and OpenMP concurrency platforms. We believe that this description helps understanding and justifying the design of MetaFork.

2 Basic Principles and Execution Model

We summarize in this section a few principles that guided the design of MetaFork. First of all, MetaFork is an extension of the C language 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 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*.

Similarly to CilkPlus, 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.

As mentioned above, MetaFork does not make any assumptions about the run-time system, in particular about task scheduling. Along the same idea, another design intention 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 one of CUDA's principles [8] which states that the execution of a given kernel should be independent of the order in which its thread blocks are executed.

To understand the implication of that idea in MetaFork, let us return to our concurrency platforms targeting multicore architectures: OpenMP offers several clauses which can be used to exchange information between 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 concurrently. MetaFork has both types of parallel constructs. But, for the latter, MetaFork does not offer counterparts to the above OpenMP data attribute clauses. Data attributes in MetaFork are discussed in Section 4.

3 Parallel constructs of MetaFork

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`. We stress the fact that `meta_fork` allows the programmer to spawn a function call (like in CilkPlus) as well as a block (like in OpenMP).

3.1 Spawning a function call or a block with `meta_fork`

As mentioned above, 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 is executing 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, for the function `f` called on the argument sequence `args` is done by

```
meta_fork f(args)
```

The semantics is similar to those 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: none of the `return` statements of the function `f` assumes an implicit barrier. 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* [10, 3].

Figure 1 illustrates how `meta_fork` can be used to define a function spawn. The underlying algorithm in this example is the classical divide and conquer *quicksort* procedure.

The other usage of the `meta_fork` construct is for spawning a basic block `B`, which is done as follows:

```
meta_fork { B }
```

Note that if `B` consists of a single instruction, then the surrounding curly braces can be omitted.

We also refer to the above as a *parallel region.* There is no equivalent in CilkPlus while it is offered by OpenMP. Similarly to a function call spawn, no implicit barrier is assumed at the end of the parallel region. Hence synchronization points have to be added explicitly, using `meta_join`; see the examples of Figures 2 and 6.

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 4.

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 due to Guy Blelloch [2].

3.2 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. In addition:

- the initialization expression initializes a variable, called the *control variable* which can be of type integer or pointer,

```
#include <algorithm>
#include <iostream>
using namespace std;
void parallel_qsort(int * begin, int * end)
{
        if (begin != end) {
        --end;  // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);    // move pivot to middle
        meta_fork parallel_qsort(begin, middle);
        parallel_qsort(++middle, ++end); // Exclude pivot and restore end
        meta_join;
    }
}
int main(int argc, char* argv[])
{
        int n = 10;
        int *a = (int *)malloc(sizeof(int)*n);
        srand( (unsigned)time( NULL ) );
        for (int i = 0; i < n; ++i)
              a[i] = rand();
        parallel_qsort(a, a + n);
        return 0;
}
```

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

```
long int parallel_scanup (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_scanup(x,t,i,k);
                }
                right = parallel_scanup (x,t, k+1, j);
                meta_join;
                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` loop.

- the condition expression compares the control variable with a compatible expression, using one of the relational operators `<, <=, >, >=, !=`,
- the stride uses one the unary operators `++, --, +=, -+` (or a statement of the form `cv = cv + incr` where `incr` evaluates to a compatible expression) in order to increase or decrease the value of the control variable `cv`,
- if `B` consists of a single instruction, then the surrounding curly braces can be omitted.

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 from each other in order to guarantee that this parallel loop is semantically equivalent to its serial version `for (I, C, S) B` .

A variable `v` which is not local to `B` may be either shared by both the parent and the children threads, or each child 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 4.

Figure 3 displays an example of `meta_for` loop, where the underlying algorithm is the naive (and cache-inefficient) matrix multiplication procedure.

3.3 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 the its children tasks but not for those of the subsequent descendant tasks.

4 Data Attribute Rules

In this section, we discuss the variables that are non-local to the block of a parallel region (resp. the body of a parallel for-loop). We start by reviewing a few standard definitions.

4.1 Terminology

4.1.1 Notions of 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.

4.1.2 Notions of 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` and `void`) or user-defined types (`enum`, `struct` and `union`). Reference-type variables are pointers, arrays and functions.

4.1.3 Notions of `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.

4.2 Data 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`,
(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 variables or file scope variables) are private to the child.

4.3 Data 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.

4.4 Examples

In the example of Figure 4, 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 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.

In Figure 7, the variables `a`, `c`, `d`, `f` and `g` are shared, meanwhile the variable `b` and `e` are still private.

```
/* To illustrate file scope variable,
3 files (incl. a headerfile) are used.
This file is a.cpp  */
#include<stdio.h>
extern int var;
void test(int *array)
{
 int basecase = 100;
 meta_for(int j = 0; j < 10; j++)
 {
     static int var1=0;
     int i = array[j];
     if( i < basecase )
         array[j]+=var;
 }
}
```

```
/* This file is b.cpp*/
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include"a.h"
int var = 100;
int main(int argc, char **argv)
{
 int *a=(int*)malloc(sizeof(int)*10);
 srand((unsigned)time(NULL));
 for(int i=0;i<10;i++)
  a[i]=rand();
 test(a);
 return 0;
}
/* This file is a.h*/
void test(int *a);
```

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

```
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 parallel region.

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
int a;
void subcall(int *a, int *b){
        for(int i=0;i<10;i++)
            printf("%d %d\n",a[i],b[i]);
}
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);
        }
}
int main(int argc, char **argv){
        long n=10;
        par_region(n);
        return 0;
}
```

Figure 7: Example of various variable attributes in a parallel region.

5 Semantics of the parallel constructs in MetaFork

The goal of this section is to formally define the semantics of each of the parallel constructs in MetaFork. To do so, we introduce the *serial C-elision* of a MetaFork program $\mathcal{M}$ as a C program $\mathcal{C}$ whose semantics define those of $\mathcal{M}$. The program $\mathcal{C}$ is obtained from the program $\mathcal{M}$ by a set of rewriting rules stated through Algorithm 1 to Algorithm 4.

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.

Next, we specify the semantics of the spawning of a block in MetaFork. To this end, we use Algorithms 2 and 3 which reduce the spawning of a block to that of a function call:

- Algorithm 2 takes care of the case where the spawned block consists of a single instruction of where a variable is assigned to the result of a function call,
- Algorithm 3 takes care of all other cases.

Note that, in the pseudo-code of those algorithms the **generate** keyword is used to indicate that a sequence of string literals and variables are written to the medium (file, screen, etc.) where the output program is being emitted.

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[1]

```
#pragma meta grainsize = expression
```

In order to obtain the *serial C-elision* a MetaFork for loops, it is sufficient to replace `meta_for` by `for`.

Algorithm 1: Fork_region(V, R)

```
Input: R is a statement of the form:
                    meta_fork [shared(Z)] B
   where Z is a sequence of variables, B is basic block of code, V is a list of all variables
         occuring in the parent region of R and which are not local to B.
Output: The serial C-elision of the above MetaFork statement.
1 $L \leftarrow$ [Z]  /* L is a list consisting of the items of the sequence Z */
2 if B consists of a single statement which is a function call without left-value then
3     generate(B);
4 else if B consists of a single statement which is a function call with left-value then
5     Function_with_lvalue($V$, $L$, B);
6 else
7     Block_call($V$, $L$, B);
```

6 Translation strategy from CilkPlus code to MetaFork code

Translating code from CilkPlus to MetaFork is easy in principles since CilkPlus is a subset of MetaFork. However, implicit CilkPlus barriers need to be explicitly inserted in the target

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

Algorithm 2: Function_with_lvalue(V, L, B)

```
Input: B is a statement of the form:
                    G = F(A);
  where G is a left-value, $A$ is an argument sequence, F is a function name and, $V, L$ are as
      in Algorithm 1.
Output: The serial C-elision of the MetaFork statement below:
                    meta_fork [shared(Z)] G = F(A)
  where $L$ is [Z].
1 if (G not in $L$) and (G is a value-type variable) and (G is not declared static or const)
  then
2     $(E, H, P)$ = Outlining_region($V$, $L$, B);
3     generate($E$);
4     generate($H$," ( ", $P$, " ) ");
5 else
6     generate(B);
```

Algorithm 3: Block_call(V, L, B)

```
Input: $V$, $L$ and B are as in Algorithm 1.
Output: The serial C function call of the above input.
1 $(E, H, P)$ = Outlining_region($V$, $L$, B);
2 generate($E$);
3 generate($H$," ( ", $P$, " ) ");
```

Algorithm 4: Outlining_region(V, L, B)

Input: V, L and B are as in Algorithm 1.

Output: (E, H, P) where E is a statement (actually a function definition) of the form

`void` H(T) {D}

H is a function name, T is a sequence of formal parameters, D is a function body and P is a sequence of actual parameters such that $H(P)$ is a valid function call.

Note that the output is obtained in 2 passes: H(T) in Pass 1 and D in Pass 2.

let M be the subset of V consisting of value type variables which are declared *static*;

```
1  Initialize $D$ to the empty string
2  Initialize $P$ to the empty string
3  Initialize $T$ to the empty string
4  /* PASS 1 */
5  foreach variable $v$ in $V$ do
6      if $v$ is redeclared in B and thus local to B then
7          do nothing
8      else if $v$ is in $L$ or in $M$ then
9          append & $v$.name to P
10         append $v$.type * $v$.name to T
11     else
12         append $v$.name to P
13         append $v$.type $v$.name to T
14 /* PASS 2 */
15 Translate B verbatim except
16 foreach right-value R within B do
17     if R is a function call G(A) then
18         Initialize A_new to the empty list
19         foreach variable $v$ in A do
20             if $v$ in $L$ or in $M$ then
21                 create a new variable name tmp
22                 insert $v$.type tmp = $v$.value at the beginning of D
23                 append tmp to A_new
24             else
25                 append $v$ to A_new
26         append G(A_new) to D
27     else if R is in $L$ or in $M$ then
28         append ("*" + R.name) to D
29     else
30         append R to D
```

MetaFork code, see Figure 9. This implies that, during translation, it is necessary to trace the structure of the CilkPlus parallel regions in order to properly insert barriers in the generated MetaFork code.

```
void function()                              void function()
{                                            {
   int i = 0, j = 0;                            int i = 0, j = 0;
   cilk_for ( int j = 0; j < ny; j++ )          meta_for ( int j = 0; j < ny; j++ )
   {                                            {
     int i;                                       int i;
     for ( i = 0; i < nx; i++ )                   for ( i = 0; i < nx; i++ )
     {                                            {
        u[i][j] = unew[i][j];                        u[i][j] = unew[i][j];
     }                                            }
   }                                            }
   ...                                          ...
}                                            }
```

Figure 8: Example of translating parallel for loop from CilkPlus to MetaFork

```
void test()                                  void test()
{                                            {
        int x;                                       int x;
        x = cilk_spawn test1();                      x = meta_fork test1();
}                                                    meta_join;
                                             }
```

Figure 9: Example of inserting barrier from CilkPlus to MetaFork.

7 Translation strategy from MetaFork code to CilkPlus code

Since CilkPlus has no constructs for spawning a parallel region (which is not a function call) we use the *outlining technique* (widely used in OpenMP) to wrap the parallel region as a function, and then call that function concurrently. In fact, we follow the algorithms of Section **??**hat is Algorithms 1, 2, 3 and 4. Indeed, the problem of translating code from MetaFork to CilkPlus is equivalent to that of defining the serial elision of a MetaFork program.

8 Translation strategy from OpenMP code to MetaFork code

8.1 Translation from OpenMP task to MetaFork

An OpenMP task statement can take one of the following forms:

- `#pragma omp task clause(..)`
  `var = function();`

- `#pragma omp task clause(..)`
  `function();`

- `#pragma omp task clause(..)`
  `block`

```
void function()
{
    int i, j;
    meta_fork shared(i)
    {
        i++;
    }

    meta_fork shared(j)
    {
        j++;
    }

    meta_join;
}
```

```
void fork_func0(int* i)
{
    (*i)++;
}
void fork_func1(int* j)
{
    (*j)++;
}

void function()
{
    int i, j;
    cilk_spawn fork_func0(&i);
    cilk_spawn fork_func1(&j);
    cilk_sync;
}
```

Figure 10: Example of translating parallel region from MetaFork to CilkPlus.

```
void function()
{
     int x;
     meta_fork shared(x)
     {
            func1(x);
     }
}
```

```
void function()
{
      int x;
      cilk_spawn func1(x);

}
```

Figure 11: Example of translating function spawn from MetaFork to CilkPlus.

```
void function()
{
   int i = 0, j = 0;
   meta_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

```
void function()
{
   int i = 0, j = 0;
   cilk_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

Figure 12: Example of translating parallel for loop from MetaFork to CilkPlus

8.2 Examples covering different cases of OpenMP Task

The following examples cover different forms of OpenMP tasks translation from original OpenMP code to MetaFork code.

- There are no shared variables(see Figure 13).

```
void function()
{
        int x;
        #pragma omp task
        {
         x = func1();
        }
        #pragma omp task:
        {
         func2();
        }
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        meta_fork
        {
         x = func1();
        }

        meta_fork func2();

        meta_join;
}
```

Figure 13: Example of translating task directive from OpenMP to MetaFork without shared variable

- Array is a local variable and used inside the task region(see Figure 14).

```
void function()
{
        int x[10];
        #pragma omp task
        {
         func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x[10];
        int temp[10];
        memcpy(temp,x,sizeof(x));
        meta_fork func1(temp);
        ...
        meta_join;
}
```

Figure 14: Example of translating task directive from OpenMP to MetaFork with private array

- Variable x stores the value of the function func1(see Figure 15).
- Variable x is passed as an argument to func1(see Figure 16).
- Translating a block[2] of the task(see Figure 17).

8.3 Translation from OpenMP Parallel for loop to MetaFork

The scheduling strategies that are in OpenMP parallel for loops are ignored in MetaFork.

The loop control variable is initialized inside the loop. Those variables that are declared as private are reinitialized inside the `parallel for loop` region in MetaFork(see Figure 18).

[2]The block can consists of a single intruction which is a non-function call.

```
void function()
{
        int x;
        #pragma omp task shared(x)
        {
         x = func1();
        }
       ...
       #pragma omp taskwait
}
```

```
void function()
{
        int x;
        x = meta_fork func1();
        ...
        meta_join;
}
```

Figure 15: Example of translating task directive from OpenMP tp MetaFork with shared lvalue

```
void function()
{
        int x;
        #pragma omp task shared(x)
        {
         func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        meta_fork func1(x);
        ...
        meta_join;
}
```

Figure 16: Example of translating task directive from OpenMP to MetaFork with shared variable as parameter

```
void function()
{
        int x;
        char y;
        #pragma omp task
        {
                x = 10;
                y = 'c';
                func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        char y;
        meta_fork
        {
                x = 10;
                y = 'c';
                func1(x);
        }
        ...
        meta_join;
}
```

Figure 17: Example of translating task directive from OpenMP to MetaFork with shared variable as parameter

```
void function()
{
   int i = 0, j = 0;
   #pragma omp parallel
   {
     #pragma omp for private(i, j)
      for ( j = 0; j < ny; j++ )
      {
        for ( i = 0; i < nx; i++ )
        {
          u[i][j] = unew[i][j];
        }
      }
    }
    ...
}
```

```
void function()
{
   int i = 0, j = 0;
   meta_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

Figure 18: Example of translating parallel for loop from OpenMP to MetaFork

8.4 Translation from OpenMP sections to MetaFork

OpenMP allows us to spawn a particular region. This is also supported in MetaFork. Since there is an implied barrier at the end of sections (unless there is a nowait clause specified), a barrier at the end of the translated MetaFork code is provided(see Figure 19).

```
void function()
{
        int i = 0, j = 0;
        #pragma omp parallel
        #pragma omp sections
        {
           #pragma omp section
               {
                  i++;
               }
           #pragma omp section
               {
                  j++;
               }
         }
}
```

```
void function()
{
        int i = 0, j = 0;
        meta_fork shared(i)
        {
           i++;
        }
        meta_fork shared(j)
        {
           j++;
        }
        meta_join;
}
```

Figure 19: Example of translating sections from OpenMP to MetaFork

9 Translation strategy from MetaFork code to OpenMP code

As of now, this translation is relatively easy because MetaFork constructs are very straightforward and design of the language is simple.

9.1 Translation from MetaFork to OpenMP Task

The meta_fork statement can have the following forms:

- `var = meta_fork function();`
- `meta_fork function();`

-
  ```
  meta_fork [shared(var)]
          block
  ```

In Figure 20 there is an example which shows how the above forms are translated to OpenMP tasks.

```
void function()
{
        int x = 0, i = 0;

        x = meta_fork y();

        meta_fork z();

        meta_fork shared(i)
        {
             i++;
        }
        meta_join;
}
```

```
void function()
{
        int x = 0, i = 0;

        #pragma omp task shared(x)
        {
              x = y();
        }

        #pragma omp task
              z();

        #pragma omp task shared(i)
        {
              i++;
        }
        #pragma omp taskwait
}
```

Figure 20: Example of translating meta_fork from MetaFork to OpenMP.

9.2 Translation from MetaFork parallel for loops to OpenMP parallel for loop

This translation is straightforward and simple as shown in Figure 21.

```
void main()
{
       int n = 10, j = 0;
       meta_for(int i = 0; i < n; i++)
       {
           j = j+1;
       }
}
```

```
void main()
{
        int n = 10, j = 0;
        #pragma omp parallel
        {
          #pragma omp for
          for(int i = 0; i < n; i++)
          {
              j = j+1;
          }
         }
}
```

Figure 21: Example of translating parallel for loop from MetaFork to OpenMP.

```
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 22: 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
ˆ	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.

10 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.

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.

10.1 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 8, 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.

10.2 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.

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

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

[4] 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.

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

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

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

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[9] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):0066–71, 2013.

[10] 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.