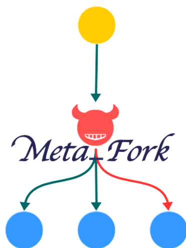
MetaFork: A Compilation Framework for Concurrency Platforms Targeting Multicores

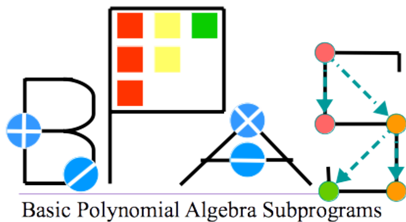
Xiaohui Chen, Marc Moreno Maza & Sushek Shekar

University of Western Ontario, Canada

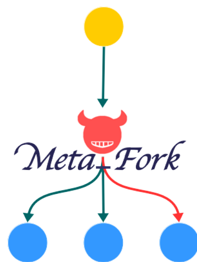
IBM CAS University Days 2014



High-performance computing and symbolic computation



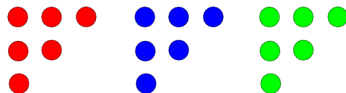
www.bpaslib.org



www.metafork.org

CUDA MODular Polynomial $\in \mathbb{F}_p[X_1 \ldots X_s]$

www.cumodp.org



www.regularchains.org

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

Background

Fork-join model

- The *fork-join execution model* is a model of computations where concurrency is expressed as follows.
- A parent gives birth to child tasks. Then all tasks (parent and children) execute code paths concurrently and synchronize at the point where the child tasks terminate.
- On a single core, a child task preempts its parent which resumes its execution when the child terminates.

CilkPlus *and* OpenMP

- CilkPlus and OpenMP are multithreaded extensions of C/C++, based on the fork-Join model and primarily targeting shared memory architectures.
- CilkPlus uses random work-stealing scheduler whereas OpenMP uses work-sharing scheduler.

Motivation: interoperability

Challenge

- Different concurrency platforms (e.g: Cilk and OpenMP) can hardly cooperate at run-time since their schedulers are based on different strategies (work stealing vs work sharing).
- This is unfortunate: there is, indeed, a real need for interoperability.

Example:

- In the field of symbolic computation:
  - the DMPMC (TRIP project) library provides sparse polynomial arithmetic and is entirely written in OpenMP,
  - the BPAS (UWO) library provides dense polynomial arithmetic is entirely written in Cilk.

  We know that polynomial system solvers require both sparse and dense polynomial arithmetic and thus could take advantage of a combination of the DMPMC and BPAS libraries.

Motivation: comparative implementation

Challenge

- Performance bottlenecks in multithreaded programs are very hard to detect:
  - algorithm issues: low parallelism, high cache complexity
  - hardware issues: memory traffic limitation
  - implementation issues: true/false sharing, etc.
  - scheduling costs: thread/task management, etc.
  - communication costs: thread/task migration, etc.
- We propose to use comparative implementation. for narrowing performance bottlenecks.

Code Translation:

- Of course, writing code for two concurrency platforms, say P_1, P_2, is clearly more difficult than writing code for P_1 only.
- Thus, we propose automatic code translation between P_1 and P_2.

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

MetaFork

Definition

- MetaFork is an extension of C/C++ and a multithreaded language based on the fork-join concurrency model.
- MetaFork differs from the C language only by its parallel constructs.
- By its parallel programming constructs, the MetaFork language is currently
  - a super-set of CilkPlus and,
  - includes the following widely used parallel constructs of OpenMP `#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, this language does not compromise itself in any scheduling strategies (work-stealing, work-sharing) and thus makes no assumptions about the run-time system.

MetaFork constructs for parallelism

MetaFork has four parallel constructs:

- **meta_fork** $\langle \text{function} - \text{call} \rangle$
  - we call this construct a function spawn,
  - it is used to express the fact that a function call is executed by a child thread, concurrently to the execution of the parent thread,

  Example:

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

- **meta_join**
  - this indicates a synchronization point.

MetaFork constructs for parallelism

- **meta_for** (start, end, stride) $\langle \text{loop} - \text{body} \rangle$
  - we call this construct a parallel for-loop,
  - the execution of the parent thread is suspended when it reaches meta_for and resumes when all children threads have completed their execution,
  - there is an implicit barrier at the end of the parallel area;

Example:

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

MetaFork constructs for parallelism

- **meta_fork** [**shared**(variable)] ⟨body⟩
  - we call this construct a parallel region,
  - is used to express the fact that a block is executed by a child thread, concurrently to the execution of the parent
  - no equivalent in CilkPlus.

Example:

```
int main()
{
  int sum_a=0;
  int a[ 5 ] = {0,1,2,3,4};
  meta_fork shared(sum_a){
      for(int i=0; i<5; i++)
        sum_a += a[ i ];
      }
}
```

Expressing fork-join concurrency in CilkPlus & OpenMP

platform	function spawn	parallel for-loop	parallel region	sync
CilkPlus	$\surd$	$\surd$	$\times$	$\surd$
OpenMP	$\times$	$\surd$	$\surd$	$\surd$
MetaFork	$\surd$	$\surd$	$\surd$	$\surd$

MetaFork data attribute rules (1/3)

MetaFork terminology:

Local and non-local variables

- Consider a parallel region or parallel for-loop Y and its immediate outer scope X. We say that X is the *parent region* of Y and that Y is a *child region* of X.
- A variable v defined in Y is said *local* to Y otherwise we call it an *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.)

Shared and private variables

- 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.

MetaFork data attribute rules (2/3)

Recall:

Value-type and reference-type variables

- In C, 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.

`static` and `const` qualified variables

- In C, a `static` variable is a variable that has been allocated statically and whose lifetime extends across the entire run of the program.
- a `const` variable is a variable which cannot be altered by the program during its execution.

MetaFork data attribute rules (3/3)

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 or (2) a file scope variable or (3) a reference-type variable or (4) declared `static` or `const`, or (5) qualified `shared`.
  - otherwise 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.

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.

MetaFork semantics of parallel constructs

Semantics of MetaFork

- To formally define the semantics of each of the parallel constructs in MetaFork, we introduce the *serial C-elision* of a MetaFork program *M* as a C program whose semantics define those of *M*.
- For spawning a function call or executing a parallel for-loop, MetaFork has the same semantics as CilkPlus. In these cases, the serial C-elision is obtained by replacing
  - **meta_fork** with the empty string,
  - **meta_for** with `for`.
- The non-trivial part is to define the serial C-elision of a parallel region in MetaFork, that is, when the **meta_fork** keyword is followed by a block of code.
- We formally define the serial C elision of the `meta_fork` construct when applied to a code block. This is done essentially by wrapping this code block into a function which is, then, called.

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

Original MetaFork code and translated OpenMP code

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

Original OpenMP code and translated MetaFork 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;
  }
}
```

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

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

Original MetaFork code and translated CilkPlus code

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

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

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

  meta_join;
}
```

```
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;
}
```

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

Experimentation: set up

Source of code

- John Burkardt's Home Page (Florida State University)
  `http://people.sc.fsu.edu/ %20jburkardt/c_src/openmp/openmp.html`
- Barcelona OpenMP Tasks Suite (BOTS)
- Cilk++ distribution examples
- Students' code

Compiler options

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

Architecture

Running time on $p = 1, 2, 4, 6, 8, \ldots$ processors. All our compiled programs were tested on :

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

Validation

Verifying the correctness of our translators was a major requirement. Depending on the test-case, we could use one or the other following strategy.

For Cilk++ distribution examples and the BOTS (Barcelona OpenMP Tasks Suite) examples:

- both a parallel code and its serial elision were executed and the results were compared,
- since serial elisions remain unchanged by our translators, the translated programs could be verified by the same procedire.

For FSU (Florida State University) examples:

- Since these examples do not include a serial elision of the parallel code, they are verified by comparing the result between the original program and translated program.

Experimentation: two experiences

Comparing two hand-written codes via translation

- For each test-case, we have a hand-written OpenMP program and a hand-written CilkPlus program
- For each test-case, we observe that one program (written by a student) has a performance bottleneck while its counterpart (written by an expert programmer) does not.
- We translate the efficient program to the other language, then check whether it incurs the same performance bottleneck as the inefficient program. This generally help narrowing the issue.

Automatic translation of highly optimized code

- For each test-case, we have either a hand-written-and-optimized CilkPlus program or a hand-written-and-optimized OpenMP program.
- We want to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts.

Comparing hand-written codes (1/2)

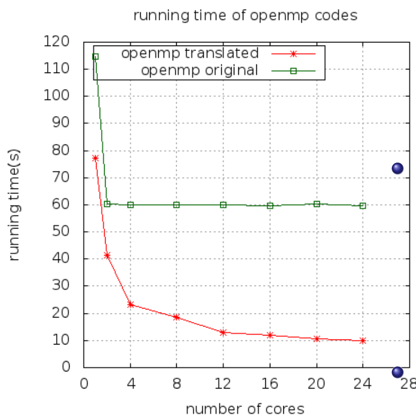


Figure: Mergesort: $n = 5 \cdot 10^8$

- Different parallelizations of the same serial algorithm (merge sort).
- The original OpenMP code (written by a student) misses to parallelize the merge phase (and simply spawns the two recursive calls) while the original CilkPlus code (written by an expert) does both.
- On the figure, the speedup curve of the translated OpenMP code is as theoretically expected while the speedup curve of the original OpenMP code shows a limited scalability.
- Hence, the translated OpenMP (and the original CilkPlus program) exposes more parallelism, thus narrowing the performance bottleneck

Comparing two hand-written codes (2/2)

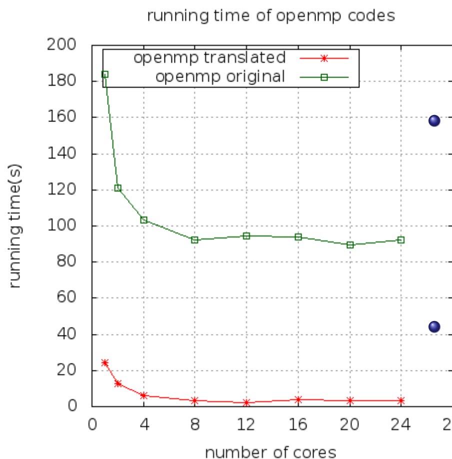
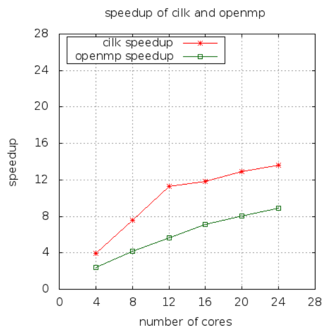


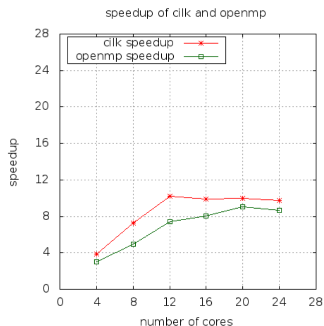
Figure: Matrix inversion: $n = 4096$

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

Automatic translation of highly optimized code (1/2)



(a) DnC MM: 4096



(b) DnC MM: 8192

Figure: Speedup curve on intel node

- About the algorithm (divide-and-conquer matrix multiplication): high parallelism, data-and-compute-intensive, optimal cache complexity
- CilkPlus (original) and OpenMP (translated) codes scale well

Automatic translation of highly optimized code (2/2)

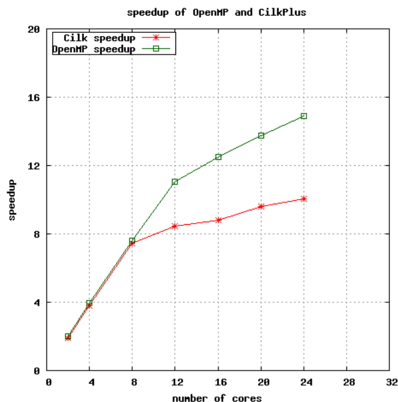


Figure: Protein alignment sequence: speedup curves.

- Dynamic programming typical example: relatively high parallelism but high communication /synchronization costs. The original code was heavily tuned to address these latter costs.
- OpenMP (original) and CilkPlus (translated) codes scale well up to 8 cores.

Parallelism overhead measurements

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

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

Table: Running time of the serial version v/s single core.

Plan

1 Motivation

2 MetaFork: fork-join constructs and semantics

3 MetaFork: code translation examples

4 MetaFork: interoperability between CilkPlus and OpenMP

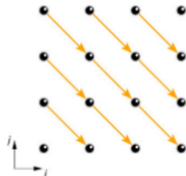
5 The MetaFork framework: generation of parametric code

6 Concluding remarks

Parallelizing polynomial multiplication

Serial dense univariate polynomial multiplication

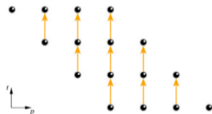
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



Dependence analysis suggests to set $t(i,j) = n - j$ and $p(i,j) = i + j$.

Synchronous parallel dense univariate polynomial multiplication

```
for (p=0, p<=2*n, p++) c[p]=0;

for (t=0, t=n, t++)
  meta_for (p=n-t; p<=2*n -t; p++)
    c[p] = c[p] + a[t+p-n] * b[n-t];
}
```



Switching from synchronous to asynchronous (1/2)

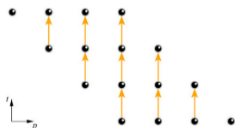
The synchronous and asynchronous inequality systems

$$\begin{array}{c} 0 \le t \le n \\ n - t \le p \le 2n - t \end{array}$$

$$\begin{array}{c} 0 \le p \le 2n \\ 0 \le t \le n \\ n - p \le t \le 2n - p \end{array}$$

Asynchronous parallel dense univariate polynomial multiplication

```
meta_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p
    c[p] = c[p] + a[t+p-n] * b[n-t];
}
```



Switching from synchronous to asynchronous (2/2)

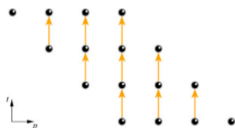
```
> ff := &E([i,j]), (0 <= i) &and (i <= n) &and
 (0 <= j) &and (j <= n) &and  (t = n - j) &and  (p = i + j):

>    R :=  PolynomialRing([i,j,t,p,n]):

>    sols := QuantifierElimination(ff, R);

  sols := ((((0 <= n) &and (0 <= p)) &and

   (p <= 2 n)) &and (Max(-p + n, 0) <= t)) &and

    (t <= Min(n, -p + 2 n))
```

- Our QE tools generate a case discussion (disjunction of mutually exclusive clauses, defined by a triangular system).
- A post-processing algorithm merges cells to simplify the generated program, solving a concern posed by Grösslinger, Griebl and Lengauer.

Generating parametric code & use of tiling techniques (1/4)

Non-practical parallel dense polynomial multiplication (recall)

```
meta_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p
    C[p] = C[p] + A[t+p-n] * B[n-t];
}
```



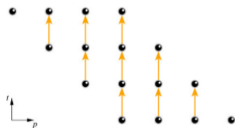
Issues with the asynchronous schedule:

The above generated code is not practical for multicore implementation:

1. the number of processors is in $\Theta(n)$. (Not to mention poor locality!)
2. the work is unevenly distributed among the workers.

Generating parametric code & use of tiling techniques (2/4)

```
meta_for (p=0; p<=2*n; p++){
  c[p]=0;
  for (t=max(0,n-p); t<= min(n,2*n-p
    C[p] = C[p] + A[t+p-n] * B[n-t];
}
```



Improving the parallelization

- Make the number of processors a parameter N; let r be a *real* processor.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property. The total number of blocks may exceed N so blocks are processed in a cyclic manner; the cycle index is s.
- We have: $0 \le r \le N-1,\ b = sN + r,\ 0 \le u < B,\ p = bB + u$.

Generating parametric code: using tiles (3/4)

Let us first generate CUDA-like code. Hence we do not need to worry about scheduling the blocks and we just schedule the threads within each block. Thus we only consider the following relations on the left to which we apply our QE tools (in order to get rid off i, j) leading to the relations on the right

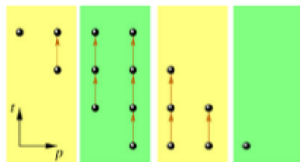
$$\begin{cases} o < n \\ 0 \le i \le n \\ 0 \le j \le n \\ t = n - j \\ p = i + j \\ 0 \le b \\ o \le u < B \\ p = bB + u, \end{cases} \qquad \begin{cases} B > 0 \\ n > 0 \\ 0 \le b \le 2n/B \\ 0 \le u < B \\ 0 \le u \le 2n - Bb \\ p = bB + u, \end{cases} \tag{1}$$

from where we derive the following program:

```
for (p=0; p<=2*n; p++) c[p]=0;
meta_for (b=0; b<= 2 n / B; b++) {
    for (u=0; u<=min(B-1, 2*n - B * b); u++) {
       p = b * B + u;
       for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
           c[p] = c[p] + a[t+p-n] * b[n-t];
   }
}
```

Generating parametric code: using tiles and scheduling them (4/4)

```
DO PAR r=0,NP-1
  DO t=0,n
    DO s=ceiling(-t/(NP*B) -r/NP +(n-B+1)/(NP*B)),
      floor(-t/(NP*B) -r/NP +(2*n)/(NP*B))
        DO u=max(0,-t-r*B-s*NP*B+n),min(B-1,-t-r*B-s*NP*B+2*n)
          DO p=max(n-t,r*B+s*NP*B+u),min(2*n-t,r*B+s*NP*B+u)
             C(p) = C(p) + A(t+p-n) * B(n-t)
```



Generation of parametric parallel programs

Summary

- Given a theoretically good algorithm (e.g. divide-and-conquer matrix multiplication) and
- a given a type of hardware that depends on various parameters (e.g. a GPGPU with amount S of the shared memory per streaming multiprocessor, maximum number P of threads supported by each streaming multiprocessor, etc.)
- our goal is to automatically generate code that depends on the hardware parameters (S, P, etc.)
- which, then, do not need to be known at compile-time.

Plan

1. Motivation
2. MetaFork: fork-join constructs and semantics
3. MetaFork: code translation examples
4. MetaFork: interoperability between CilkPlus and OpenMP
5. The MetaFork framework: generation of parametric code
6. Concluding remarks

Concluding remarks

Summary

- We presented a platform for translating programs between multithreaded languages based on the fork-join parallelism model.
- Translations are performed via MetaFork. a language which borrows from CilkPlus and OpenMP.
- Translation process does not add overheads on the tested examples.

Work in progress

- The MetaFork language is extending to pipeline parallelism
- The MetaFork framework is being enhanced with automatic generation of parametric parallel programs

Acknowledgments

We are grateful to the Compiler Development Team at the IBM Toronto Labs, in particular Priya Unnikrishnan and Abdoul-Kader Keita for their support and advice.