

METAFORK: A Compilation Framework for Concurrency Platforms Targeting Multicores

Presented by

Xiaohui Chen

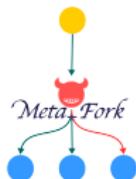
Joint work with

Marc Moreno Maza, Sushek Shekar & Priya Unnikrishnan

University of Western Ontario, Canada

IBM Toronto Lab

Sept 29, 2014



Plan

- 1 Motivation
- 2 METAFORK: fork-join constructs and semantics
- 3 METAFORK: interoperability between CILKPLUS and OPENMP
- 4 METAFORK: experimentation
- 5 Conclusion

Plan

- 1 Motivation
- 2 METAFORK: fork-join constructs and semantics
- 3 METAFORK: interoperability between CILKPLUS and OPENMP
- 4 METAFORK: experimentation
- 5 Conclusion

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.

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: interoperability between CILKPLUS and OPENMP
- 4 METAfork: experimentation
- 5 Conclusion

METAfork

The language

- The METAfork language 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.
- The semantics of the parallel constructs of METAfork are formally stated by defining the serial C elision of a METAfork program.
- By its parallel constructs, the METAfork language is currently a super-set of CILKPLUS and OPENMP and offers counterparts for 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`.

METAfork

The compilation framework

- Today, 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).
- The METAfork language is rich enough to capture the semantics of large bodies of OPENMP code, such as the *Barcelona OPENMP Tasks Suite* and *translate faithfully to CILKPLUS* each of the BOTS test cases.

METAFOK constructs for parallelism

METAFOK has four parallel constructs:

- **meta_fork** ⟨function – call⟩
 - 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,
 - on the contrary of CILKPLUS, no implicit barrier is assumed at the end of a function spawn.
- **meta_for** (start, end, stride) ⟨loop – body⟩
 - 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:

```
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);
}
```

Example:

```
void test() {
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:

```
void 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 ];
    }
    foo();
}
```

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

Counterpart directives in CILKPLUS & OPENMP

CILKPLUS

- **cilk_spawn**
- no construct for parallel regions
- **cilk_for**
- **cilk_sync**

OPENMP

- **pragma omp task/sections/single/master**
- **pragma omp for**
- **pragma omp taskwait/barrier**

METAFOREK data attribute rules (1/2)

METAFOREK terminology:

Local and non-local 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 defined in Y is said *local* to Y otherwise we call it 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.)

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* by 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* to Y .

METAfork data attribute rules (2/2)

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 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 by Y whenever it is declared `static`.
 - otherwise it is private to Y .
- In particular, loop control variables are private to 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.
- In the dissertation, 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: interoperability between CILKPLUS and OPENMP**
- 4 METAFORK: experimentation
- 5 Conclusion

Original CILKPLUS code and translated METAFORK code

```

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);
    }
}

```

```

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);
    }
}

```

```

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);
    }
}

```

Original OPENMP code and translated CILKPLUS code

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

```
int main()
{
  int a[ N ];

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

```
int main()
{
  int a[ N ];

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

Original OPENMP 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};
  #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;
}

```

Plan

- 1 Motivation
- 2 METAfork: fork-join constructs and semantics
- 3 METAfork: interoperability between CILKPLUS and OPENMP
- 4 METAfork: experimentation**
- 5 Conclusion

Experimentation: set up

Source of code

- John Burkardt's Home Page (Florida State University)
http://people.sc.fsu.edu/~jburkardt/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 $p = 1, 2, 4, 6, 8, \dots$ 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 procedure.

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: three 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.
- Parallelism overheads

Comparing hand-written codes (1/2)

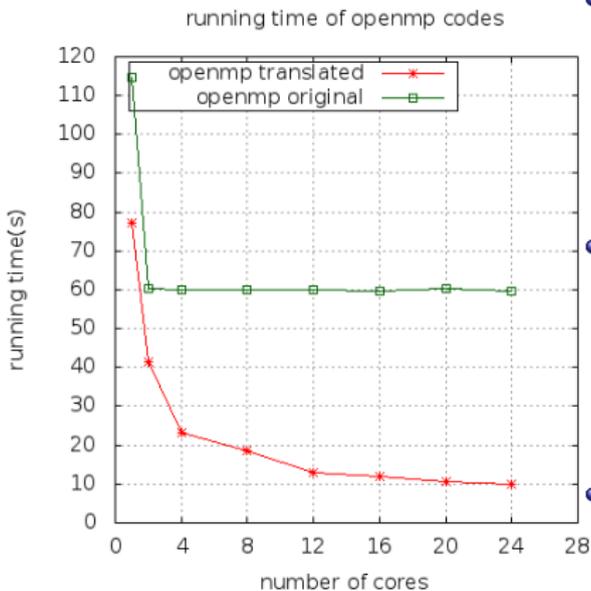
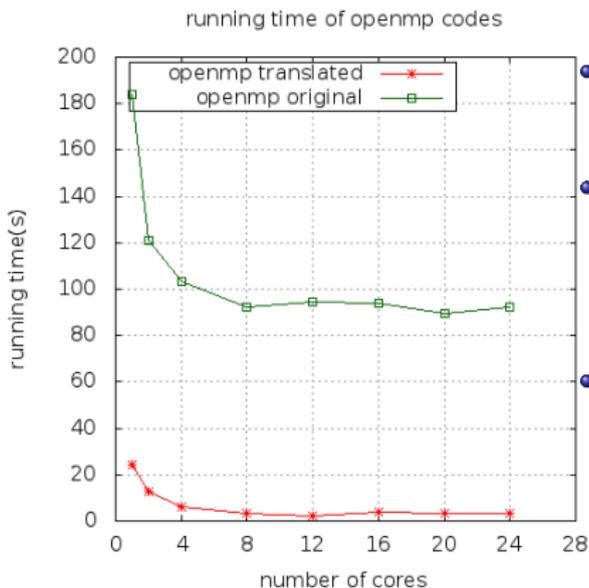


Figure : Mergesort: $n = 5 \cdot 10^8$

- 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 in the original hand-written OPENMP code.

Comparing two hand-written codes (2/2)



- 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.

Figure : Matrix inversion: $n = 4096$

Interoperability: automatic translation of highly optimized code

Test	Input size	CILKPLUS		OPENMP	
		T_1	T_{16}	T_1	T_{16}
8-way Toom-Cook	2048	0.423	0.231	0.421	0.213
	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 Plain Polynomial Multiplication	2048	0.874	0.259	0.867	0.299
	4096	3.95	1.264	3.925	1.123
	8192	18.196	3.335	18.154	4.428
	16384	77.867	12.778	75.885	12.674
	32768	331.351	55.841	332.126	55.925

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

Parallelism overhead measurements

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>

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

Experiment conclusion

- Our experimental results suggest that our translators can be used to narrow performance bottlenecks.
- The speed-up curves of the original and translated codes either match or have similar shape. Nevertheless, in some cases, either the original or the translated program outperforms its counterpart.

Plan

- 1 Motivation
- 2 METAFORK: fork-join constructs and semantics
- 3 METAFORK: interoperability between CILKPLUS and OPENMP
- 4 METAFORK: experimentation
- 5 Conclusion

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.
- Project website: www.metafork.org.

Work in progress

- The METAFORK language is being migrated to clang platform
- The METAFORK language is extending to pipeline parallelism and accelerator model
- 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 Abdoul-Kader Keita for the support and advice.