MetaFork: A Compilation Framework for Concurrency Platforms Targeting Multicores

Xiaohui Chen, Marc Moreno Maza & Sushek Shekar

University of Western Ontario, Canada

IBM Toronto Lab
February 11, 2015

Plan

Plan

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.

Motivation: optimization of parallel programs

Challenge:

A parallel program written and optimized for one architecture may loose performance when ported, say via translation, to another architecture. Possible causes:

- change of memory access policies (say from multi-cores to GPUs)
- change in the number of cores,
- change in the cache sizes.

Proposed solution:

Given a parallel algorithm and formal machine parameters (number of physical cores, cache sizes) generate a parametric parallel code

- valid for any values of those parameters in prescribed ranges,
- specializable at *installation time* on a particular machine.

Plan

The fork-join concurrency model

Principles

- 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.

Plan

OpenMP

OpenMP uses the fork-join model:

- All OpenMP programs begin as a single thread: the master thread.
- The master thread then creates a team of parallel threads when parallel region construct is encountered.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

OpenMP uses the shared-memory model :

- All threads share a common address space (shared memory)
- Threads can have private data

OpenMP

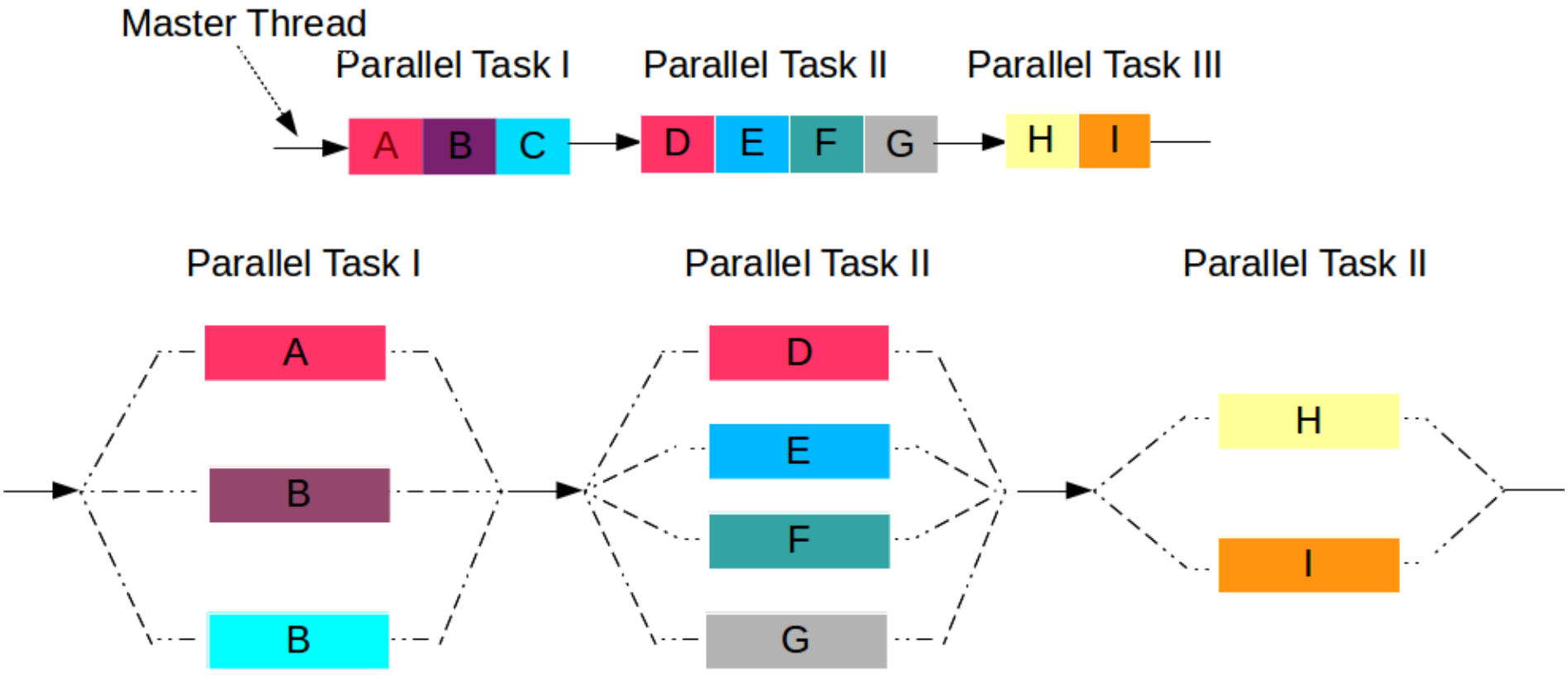


Figure: OpenMP fork-join model

OpenMP

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

The syntax of this construct is as follows:

```
#pragma omp parallel [ private (list), shared (list) ... ]
structured_block
```

When a thread reaches a `parallel` directive:

- It creates a team of threads and becomes the master of the team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel region.
- Only the master thread continues execution past this point.

OpenMP work-sharing construct

Work-sharing construct

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads.
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work-sharing construct.

There are three different work-sharing constructs.

- parallel for-loop construct
- parallel sections construct
- single construct

OpenMP work-sharing construct

OpenMP for-loop shares iterations of a loop across the team.

```
#pragma omp for [schedule(type [,chunk]), private(list) ...]
for_loop
```

Example: Saxpy operation:

$$y = ax + y \tag{1}$$

```
void saxpy() {
    const int n = 10000;
    float x [ n ], y [ n ], a;
    int i;

    #pragma omp parallel
    #pragma omp for
    for (i=0; i<n; i++) {
      y [ i ] = a * x [ i ] + y [ i ];
  }
 }
```

OpenMP work-sharing construct

OpenMP sections:

- Sections breaks work into separate, discrete sections.
- Each section is executed by a thread.

```
#pragma omp sections [shared(list), private(list) ...]
structured_block
```

Example:

```
#define N 1000
int main () {
   int i;
   double a [ N ], b [ N ], c [ N ], d [ N ];
   for (i=0; i < N; i++) {
     a [ i ] = i * 1.5;
     b [ i ] = i + 22.35;
   }
   #pragma omp parallel shared(a,b,c,d) private(i)
    {
     #pragma omp sections
       {
         #pragma omp section
          {
              for (i=0; i < N; i++)
                c [ i ] = a [ i ] + b [ i ];
          }
         #pragma omp section
          {
              for (i=0; i < N; i++)
                d [ i ] = a [ i ] * b [ i ];
          }
       } /* end of sections */
     } /* end of parallel section */
}
```

OpenMP task directives

Parallel sections are established upon compilation and number of threads is fixed. Sometimes more flexibility is needed, such as parallelism within `if` or `while` block.

In OpenMP, an explicit task is specified using the task directive.

- whenever a thread encounters a task construct, a new task is generated.
- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time.
- If task execution is deferred, then the task is placed in a pool of tasks.
- A thread that executes a task may be different from the thread that originally encountered it
- The taskwait directive specifies a wait on the completion of children tasks generated since the beginning of the current task.

Example:

```
/*pseudo code*/
int main () {
    my_pointer = listhead;
    #pragma omp parallel
    {
      #pragma omp single
      {
         while(my_pointer) {
          #pragma omp task
          {
             do_independent_work(my_pointer);
          }
          my_pointer = my_pointer->next ;
        }
      } // End of single
    } // End of parallel region - implied barrier here
}
```

OpenMP synchronization directives

There are various synchronization constructs available to coordinate the work by multiple threads.

- #pragma omp master: species a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.
- #pragma omp critical: species a region of code that must be executed by only one thread at a time.
- #pragma omp barrier: synchronizes all threads in the team. When a barrier directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.
- #pragma omp atomic: species that a specic memory location must be updated atomically.

Example: Computing the sum:

```
#define N 1000
int main () {
   int sum = 0, sum_local = 0, a [ N ];
   #pragma omp parallel shared(a,sum) private(sum_local)
   {
      #pragma omp for
      for (i=0; i<N; i++)
         sum_local += a [ i ]; // form per-thread local sum

       #pragma omp critical
       {
          sum += sum_local; // form global sum
       }
    }
}
```

Plan

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 constructs, the MetaFork language is currently a super-set of CilkPlus 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`.
- 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.

Motivations

- MetaFork principles encourage a programming style limiting thread communication to a minimum so as to
  - prevent from data-races while preserving satisfactory expressiveness,
  - minimize parallelism overheads.
- The original purpose of MetaFork is to facilitate automatic translations of programs between the above concurrency platforms.

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.

MetaFork constructs for parallelism

MetaFork 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:

```
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 ];
      }
}
```

- **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**
- **pragma omp sections**
- **pragma omp for**
- **pragma omp taskwait**

MetaFork data attribute rules (1/2)

MetaFork

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 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* 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 wraping this code block into a function which is, then, called.

Variable attributes of MetaFork: example

```
extern int var;          //shared
void test(int *array)
{              // array is shared
   int basecase = 100; //shared
   meta_for(int j=0;j<10;j++)
   {   // j is sprivate
       static int var1; //shared
       int i = array[j]; //private
       if( i < basecase )
           array[j]++;
   }
}
```

```
int a;          //shared
long par_region(long n)
{               // n is private
  int b;           //private
  int *c=(int*)malloc();//shared
  int d[10];        //shared
  const int f;  //shared
  static int g; //shared
  meta_fork{
     int e = b;   //private
     foo(c,d);
     meta_fork
     { ... }
  }
}
```

Plan

Original CilkPlus code and translated MetaFork code

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

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 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 MetaFork code

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

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

Original MetaFork code and translated CilkPlus code

```
1 void task() {
  2 int a100;
  3 int b100;
  4 int k = 100;
  5 meta_fork shared(a) {
  6    for(int i=0;i<k; i++){
  7            ai = i;
  8            bi = i;
  9     }
 10   }
 11 }
```

==========

Outlined function:

```
1 static void * _taskFunc0_(void *);
  2 void task() {
  3   int a 100;
  4   int b 100;
  5   int k = 100;
  6   {
  7     struct __taskenv__ {
  8         int b 100;
  9         int (* a) 100;
 10         int k;
 11       } * _tenv;
 12
```

```
13     _tenv = (struct __taskenv__ *) malloc(
14               sizeof(struct __taskenv__));
15     _tenv->a = &a;
16     _tenv->k = k;
17     memcpy((void *) _tenv->b, (void *) b, sizeof(b));
18     cilk_spawn _taskFunc0_(_tenv);
19   }
20 }
21
22 static void * _taskFunc0_(void * __tdata) {
23   struct __taskenv__ {
24       int b 100;
25       int (* a) 100;
26       int k;
27     };
28   struct __taskenv__ * _tenv = (struct __taskenv__ *) __tdata;
29   int (* a) 100 = _tenv->a;
30   int k = _tenv->k;
31   int (* b) 100 = &(_tenv->b);
32   {
33     for (int i = 0; i < k; i++) {
34         (*a)i = i;
35         (*b)i = i;
36       }
37   }
38   free(_tenv);
39   return (void *) 0;
40 }
```

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 (1/4)

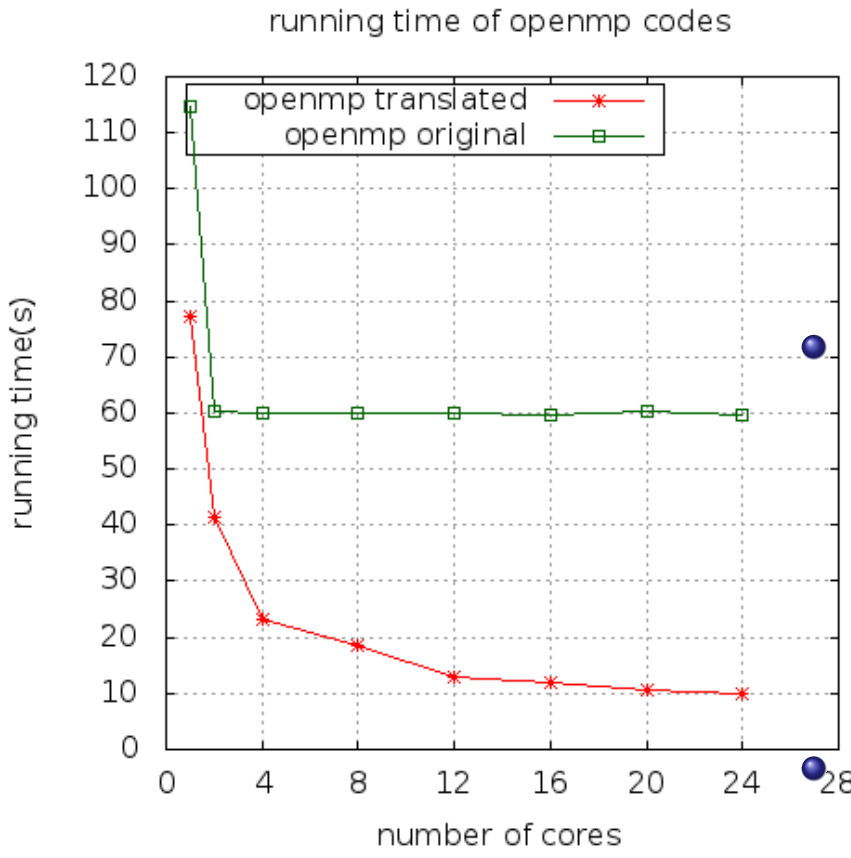


Figure: Mergesort: $n = 5 \cdot 10^8$

- 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 (2/4)

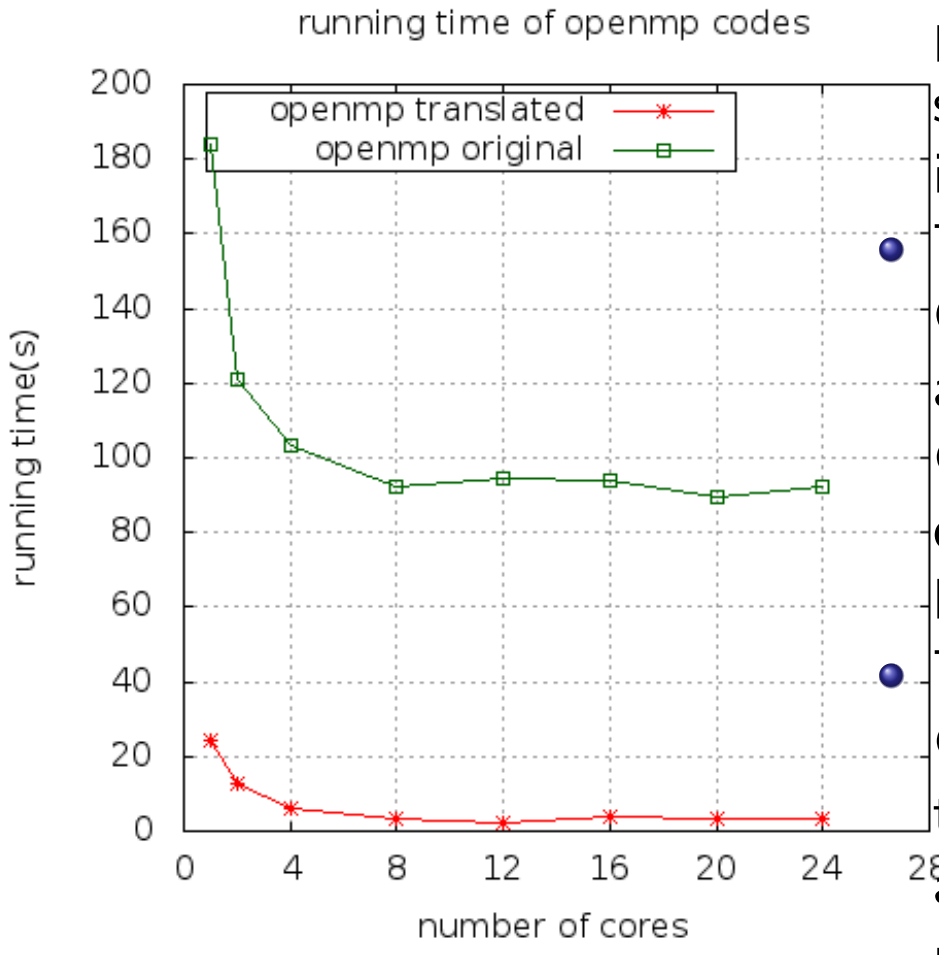
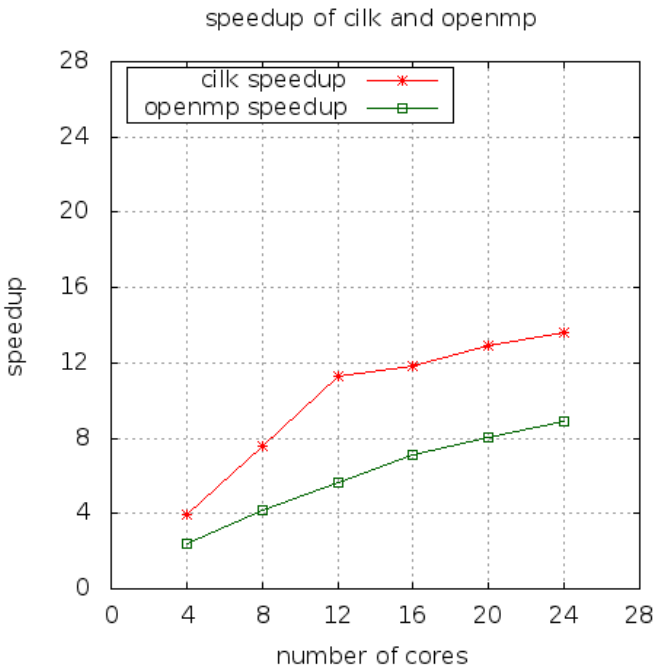


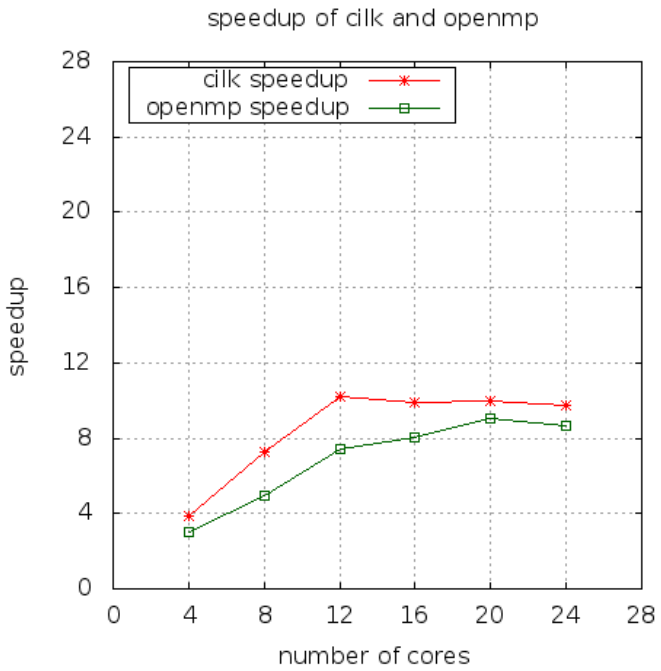
Figure: Matrix inversion: $n = 4096$

- 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 (2/9)



: DnC MM: 4096



: 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 (7/9)

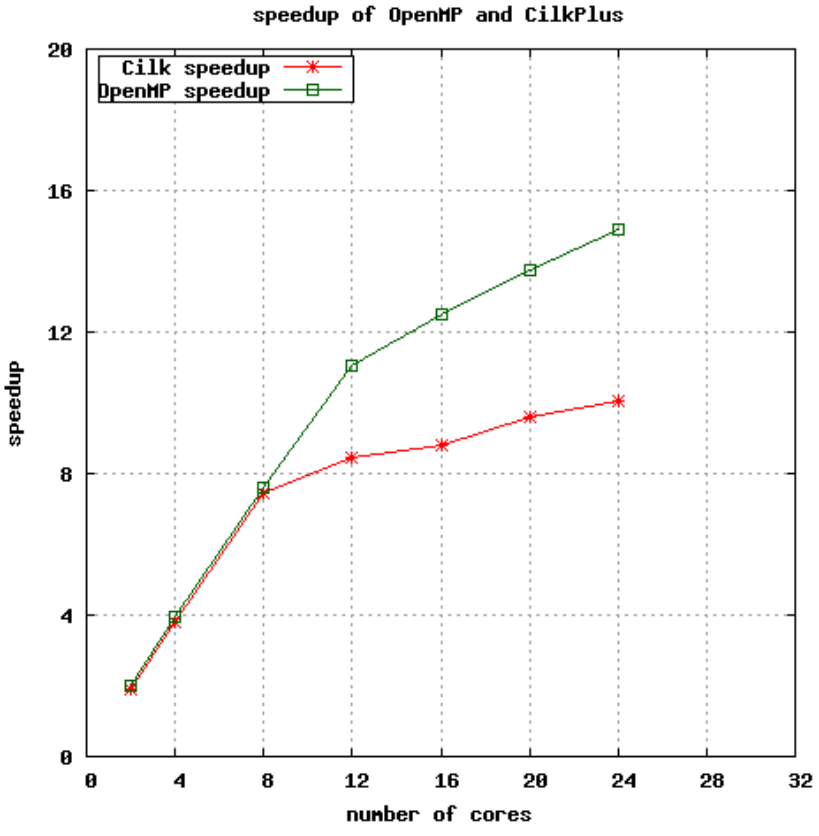


Figure: Protein alignment sequence: speedup curves.

- 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.

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

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 extending to accelerator model(GPU)
- The MetaFork framework is being enhanced with automatic generation of parametric parallel programs