MetaFork: A Metalanguage for Concurrency Platforms Targeting Multicores

Xiaohui Chen, Marc Moreno Maza & Sushek Shekar

University of Western Ontario, Canada

WG14 C Standards Committee Meeting
October 1st, 2013

Our team and projects

Our team

- Postdoctoral Fellow: Changbo Chen
- PhD Students: Xiaohui Chen, Ning Xie, Parisa Alvandi, Sardar Anisul Haque, Paul Vrbik
- Master Students: Farnam Mansouri, Sushek Shekar

Our research subjects

- Computer algebra: polynomial system solving, Maple's `solve` command
- Models of computation for HPC, optimization of multithreaded code
- Parallel code generation (automatic parallelization, etc.)

Our projects

- The `RegularChains` library, integrated into Maple
- The `cumodp` (CUDA) and `modpn` (C code) libraries, part of Maple
- The `BPAS` (`CilkPlus`) library, to be integrated into Maple
- The MetaFork project, part of our IBM CAS project

Plan

1. Introduction
2. The MetaFork Language
3. Experimentation
4. Conclusion

Plan

1. Introduction
2. The MetaFork Language
3. Experimentation
4. Conclusion

Motivation

Hypothesis and objective

- The same multithreaded algorithm (say based on the fork-join parallelism model) implemented with two different concurrency platforms could result in very different performance, often very hard to analyze and improve.
- We propose to identify performance bottlenecks by comparative implementation.

Possible performance bottlenecks:

- 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.

Comparative Implementation (1/2)

Code Translation:

- Writing comparative multithreaded code based on different concurrency platforms, say P_1, P_2, is clearly more difficult than writing code in P_1 only.
- Thus, we propose automatic code translation between P_1 and P_2.

Solution 1:

Translate code from P_1 to P_2 directly:

- unfortunately preserves wrong scheduling decisions
- requires $\frac{n(n-1)}{2}$ mappings if n concurrency platforms $P_1, \ldots, P_n$ are involved.

Comparative Implementation (2/2)

Solution 1 (recall):

Translate code from P_1 to P_2 directly:

- unfortunately preserves wrong scheduling decisions
- requires $\frac{n(n-1)}{2}$ mappings if n concurrency platforms $P_1, \ldots, P_n$ are involved.

Solution 2:

Use an intermediate high-level language (meta-language) abstracting from scheduling decisions and focusing on algorithms:

- requires only $2n$ mappings to support $P_1, \ldots, P_n$.
- offers a well-defined language to express algorithms and,
- an easier path between languages providing different forms of concurrency (fork-join, SIMD, pipelining, etc.)

Plan

1 Introduction

2 The METAFORK Language

3 Experimentation

4 Conclusion

MetaFork

Definition

- MetaFork is an extension of C/C++ and a multithreaded language based on the fork-join parallelism model.
- By its parallel programming constructs, this language is currently a common denominator of CilkPlus and OpenMP.
- However, this language does not compromise itself in any scheduling strategies (work-stealing, work-sharing)
- Thus, it 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 basic Principles

Borrowed from CilkPlus

- Similarly to CilkPlus, the semantics of a MetaFork program is identical to those of its serial elision.
  - in particular, a MetaFork function spawn is a C function, and
  - in a MetaFork parallel for-loop, every variable is shared unless it is declared within the for-loop.

Borrowed from OpenMP

- Similarly to OpenMP, MetaFork has parallel regions (these are neither spawned function calls, nor parallel for loops):
  - in this case, and in this case only, explicitly qualifying a variable *shared* is possible;
  - in fact, in MetaFork parallel regions, variables are private by default as in OpenMP tasks.

MetaFork constructs for parallelism

- **meta_fork** ⟨function − call⟩
  - we call this construct a function spawn,
  - 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,
  - there is an implicit barrier at the end of the parallel area;
  - every variable is shared unless it has been declared within the meta_for loop
- **meta_fork** [**shared**(variable)] ⟨body⟩
  - every variable is private unless it has been declared shared,
  - we call this construct a parallel region,
  - no equivalent in `CilkPlus`.
- **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**

Translation Strategy (1/2)

CilkPlus *to* MetaFork

- Easy in principle, since CilkPlus is a subset of MetaFork.
- However, implicit CilkPlus barriers need to be explicitly inserted in the generated MetaFork code.

MetaFork *to* CilkPlus

- Since CilkPlus has no constructs for parallel regions, we use the *outlining technique* (widely used in OpenMP)
  - to wrap the parallel region as a function, and then call that function concurrently,
  - this requires to record the variable information and the function name.
- All other constructs are easy to translate.

Translation Strategy (2/2)

OpenMP *to* MetaFork

- To translate task manipulation:
  - if function spawn, use the corresponding feature of MetaFork,
  - otherwise, we use the parallel region feature of MetaFork.
- A parallel section of OpenMP is translated using the corresponding feature of MetaFork:
  - but extra synchronization points are inserted;
- Every variable specified *private* in an OpenMP parallel for loop is declared in the parallel for loop of the MetaFork translation;
  - we just ignore any *shared* attribute in an OpenMP parallel for loop.

MetaFork *to* OpenMP

- Easy in principle, since MetaFork is a subset of OpenMP.
- We decided to translate the function spawns (constructed with `meta_fork`) using OpenMP `task`.

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 Meta 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 Meta code

```
void parallel_qsort(int* begin,
                    int* end,
                    int base)
{
    int length;
    length = end - begin;
    if (length < base)
        serial_qsort(begin, end);
    else if (begin != end)
    {
        --end;
        int* middle = std::partition(begin, end,
          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);

        #pragma omp task
        parallel_qsort(begin,
                       middle, base);
        parallel_qsort(++middle, ++end, base);
        #pragma omp taskwait
    }
}
```

```
void parallel_qsort(int* begin,
                    int* end,
                    int base)
{
    int length;
    length = end - begin;
    if (length < base)
        serial_qsort(begin, end);
    else if (begin != end)
    {
        --end;
        int* middle = std::partition(begin, end,
          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);

        meta_fork parallel_qsort(begin,
                                 middle, base);
        parallel_qsort(++middle, ++end, base);
        meta_join;
    }
}
```

Original Meta code and translated CilkPlus code

```
void parallel_qsort(int* begin,
                    int* end,
                    int base)
{
    int length;
    length = end - begin;
    if (length < base)
        serial_qsort(begin, end);
    else if (begin != end)
    {
        --end;
        int* middle = std::partition(begin, end,
          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);

        meta_fork parallel_qsort(begin,
                                 middle, base);
        parallel_qsort(++middle, ++end, base);
        meta_join;
    }
}
```

```
void parallel_qsort(int* begin,
                    int* end,
                    int base)
{
    int length;
    length = end - begin;
    if (length < base)
        serial_qsort(begin, end);
    else if (begin != end)
    {
        --end;
        int* middle = std::partition(begin, end,
          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);

        cilk_spawn parallel_qsort(begin,
                                  middle, base);
        parallel_qsort(++middle, ++end, base);
        cilk_sync;
    }
}
```

Original OpenMP code and translated Meta 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 Meta code and translated CilkPlus code

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

Plan

1 Introduction

2 The MetaFork Language

3 Experimentation

4 Conclusion

Experimentation: set up

Source of code

- OpenMP Microbenchmarks
- `http://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html`
- Cilk++ distribution examples
- Students' code

Architecture and compiler options

- Intel Xeon 2.66GHz/6.4GT with 12 physical cores and hyperthreading, sharing 48GB RAM
- CilkPlus code compiled with GCC 4.8 using -O2 -g -lcilkrts -fcilkplus
- OpenMP code compiled with GCC 4.8 using -O2 -g -fopenmp

Measures

Running time on $p = 1, 2, 4, 6, 8, \ldots$ processors. Benchmarks were repeated/verified on AMD Opteron 6168 48core nodes with 256GB RAM and 12MB L3.

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
- 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 the performance bottleneck remains or not, so as to narrow the performance bottleneck in the inefficient program.

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 (1/3)

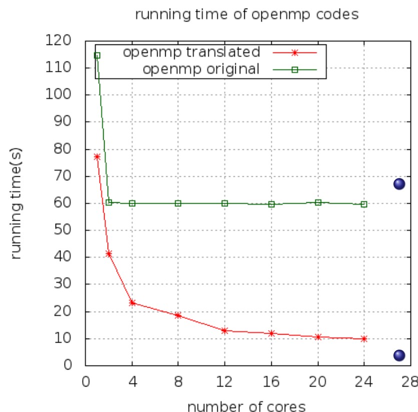


Figure: Mergesort: $n = 5 \cdot 10^8$

- 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`

Comparing two hand-written codes (2/3)

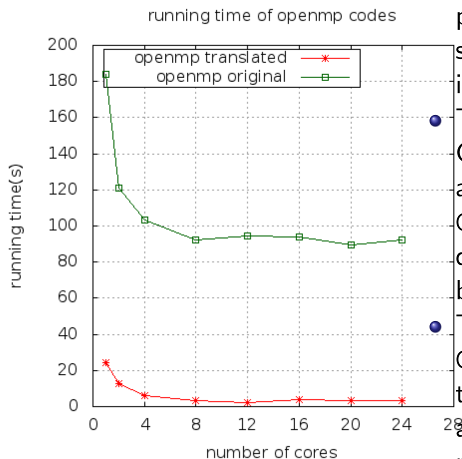


Figure: Matrix inversion: $n = 4096$

- 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.

Comparing two hand-written codes (3/3)

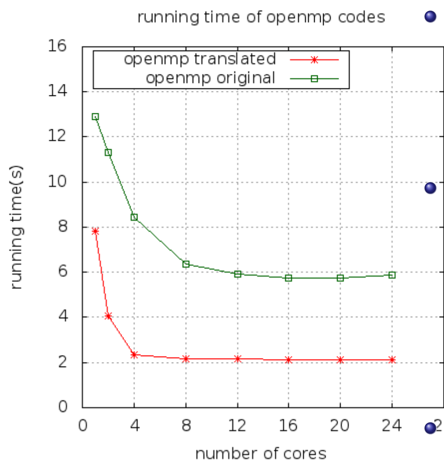
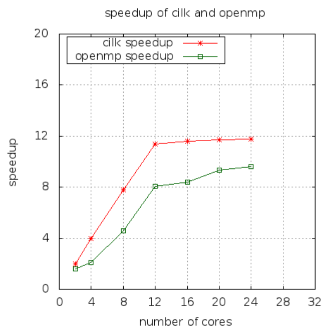


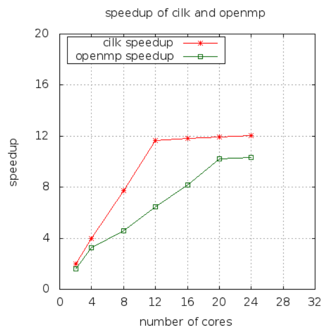
Figure: Matrix_transpose: $n = 32768$

- The original programs implement different algorithms for matrix transposition, which is a challenging operation on multicore architectures.
- Without doing complexity analysis, discovering that the `OpenMP` code (written by a student) runs in $O(n^2 \log(n))$ bit operations instead of $O(n^2)$ as the `CilkPlus` (written by Matteo Frigo) is very subtle.
- Here again, the translation narrows the algorithmic issue.

Automatic translation of highly optimized code (1/6)



(a) Fibonacci: 45

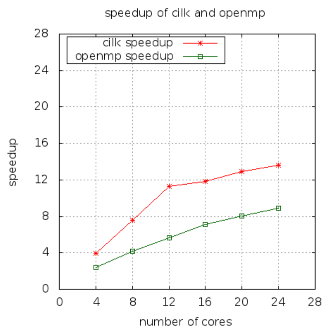


(b) Fibonacci: 50

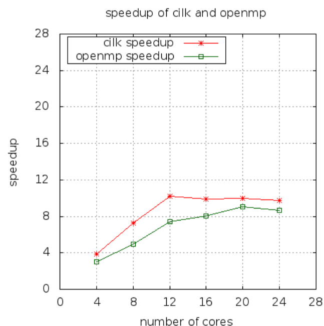
Figure: Speedup curve on Intel node

- About the algorithm (Fibonacci computation): high parallelism, no data traversal
- `CilkPlus` (original) and `OpenMP` (translated) codes scale well

Automatic translation of highly optimized code (2/6)



(a) DnC MM: 4096

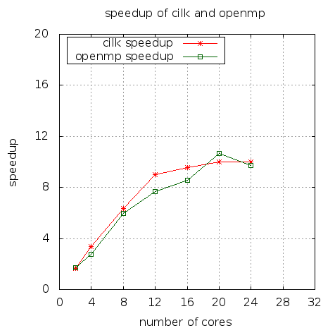


(b) DnC MM: 8192

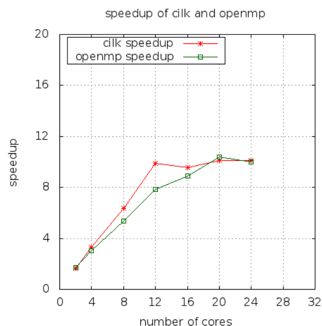
Figure: Speedup curve on intel node

- 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 (3/6)



(a) Prefix_sum: $n = 5 \cdot 10^8$



(b) Prefix_sum: $n = 10^9$

Figure: Speedup curve on Intel node

- About the algorithm (prefix sum): high parallelism, low work-to-memory-access ratio ($O(log(n))$ traversals for a $O(n)$ work.
- `CilkPlus` (original) and `OpenMP` (translated) codes scale well and at almost the same rate.

Automatic translation of highly optimized code (4/6)

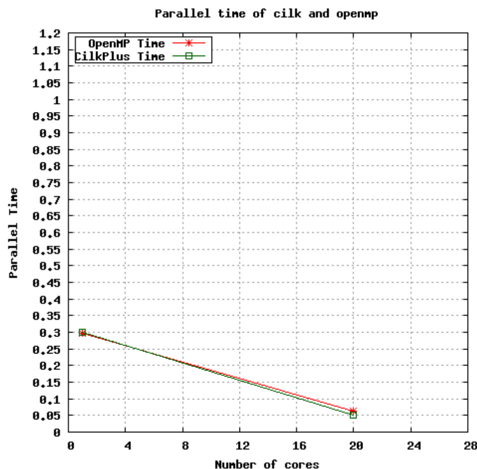


Figure: Mandelbrot set for a 500×500 grid and 2000 iterations.

- Nearly embarrassingly parallel. This application is compute-intensive and does not traverse large data.
- `OpenMP` (original) and `CilkPlus` (translated) codes scale well and at the same rate.

Automatic translation of optimized code (5/6)

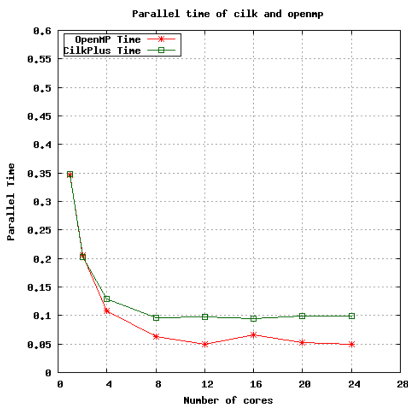


Figure: Linear system solving (dense method).

- Lots of parallelism. However, minimizing parallelism overheads and memory traffic is a challenge for this operation.
- `OpenMP` (original) and `CilkPlus` (translated) codes scale well up to 12 cores; recall that we are experimenting on a Xeon node with 12 physical cores with hyperthreading turned on.

Automatic translation of highly optimized code (6/6)

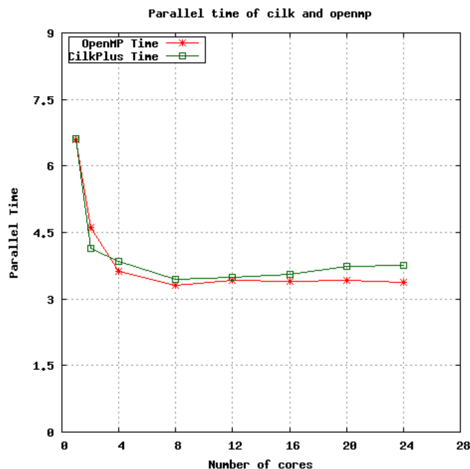


Figure: FFT over the complex in size 2^{25}.

- Low work-to-memory-access ratio: challenging to implement efficiently on multicores
- `OpenMP` (original) and `CilkPlus` (translated) codes scale well up to 8 cores.

Plan

1 Introduction

2 The MetaFork Language

3 Experimentation

4 Conclusion

Concluding remarks (1/2)

Summary

- We have realized a platform for translating programs between multithreaded languages based on the fork-join parallelism model.
- Currently the supported languages are `CilkPlus` and `OpenMP`.
- Translations are performed via a meta-language, called METAFORK. which is a common denominator of the parallel constructs of `CilkPlus` and `OpenMP`, without compromising in any scheduling decisions.
- Experimentation shows that this platform can be used to narrow performance bottlenecks. More precisely, for each of our test cases, the overall trend of the performance results can be used to narrow causes for low performance.
- Moreover, the translation process seems not to add any overheads on the tested examples.

Concluding remarks (2/2)

Future works for our team

- Define a model for analyzing the overheads of various scheduling strategies, as it was done for the randomized work-stealing scheduling strategy (Leiserson & Blumofe).
- Integrate Intel's TBB into the MetaFork family.

A proposal submitted to the WG14 Committee

- Extend the C language with parallel constructs similar to MetaFork's constructs `meta_fork`, `meta_join`, `meta_for`, with names like `fork`, `join`, `parallel_for`.
- This would provide a bridge between popular concurrency platforms based on fork-join parallelism such as OpenMP and CilkPlus, with benefits like comparative implementation, library collaboration.

Demo

```
./autotest.sh cilk compile cilk_examples

./autotest.sh openmp compile openmp_examples

./autotest.sh cilk verify cilk_examples

./autotest.sh openmp verify openmp_examples
```