Enhance your code
Parallelization as easy as pie (?)

D. Renault

The Thursday's Geekeries
ENSEIRB-MATMECA

April 2016, v. 1.2

A somewhat basic problem

Consider a task T such that :

- T is constituted of a rather large number of subtasks $\{t_1, \ldots, t_N\}$;
- The subtasks are completely **independent** ;
- Each subtask t_i outputs a result u_i ;
- The result of the global task T is a **combination** of the u_i's.

Examples :

- News aggregator : aggregate a series of requests to different servers ;
- Test harness : execute all the tests for a program in a distributed manner.

How to execute T while harnessing the parallelism of a personal computer ?

Recurring example

Problem

Compute an approximation of π using the Bailey–Borwein–Plouffe formula :

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

- Each thread computes a (finite) number of parts of the sum.
- The computation time of each thread is non trivial.
- The final code must sum the results computed by each thread.

Highlights the following aspects of the problem :

- **Parallelism** : all the quantities are independent.
- **Sharing** : each computation plays a role in the final result.

  $\Rightarrow$ leads to concurrency problems.

Concurrency and Atomicity

Atomic operation

An atomic operation is a sequence of one or more machine instructions that are executed sequentially, without interruption from the operating system.

When a thread performs an atomic operation, the other threads see it as happening instantaneously.

Example : `x += 1;` is **not** an atomic operation.

It is composed of three atomic operations :

- read the value of `x` and store it into a register ;
- compute the sum of this register and the value 1 ;
- store the result of the addition into `x`.

The value of `x` could be modified between the first and the last operation.

Naive implementation with fork

```
static mpf_t *glob_var;                          // Allocate shared memory
glob_var = mmap(NULL,sizeof *glob_var,PROT_READ|PROT_WRITE,
                MAP_SHARED|MAP_ANONYMOUS,-1, 0);
mpf_init(*glob_var);
pid_t *childPids = NULL;
childPids = malloc(NUM_THREADS * sizeof(pid_t));

for (int k = 0; k < NUM_THREADS; ++k) { // Fork children
  if ((p = fork()) == 0) {                       // Child starts to work here
    mpf_t work; mpf_init(work);
    bbp_computation(k, size, work);
    pthread_mutex_lock(&mtx);                    // Start atomic operation
    mpf_add(*glob_var,*glob_var,work);
    pthread_mutex_unlock(&mtx);                  // End atomic operation
    exit(0);                                     // Child's work is finished
  } else { childPids[k] = p; } }                 // Parent process

int stillWaiting;                                // Wait for children to exit
do {
  stillWaiting = 0;
  for (int k = 0; k < NUM_THREADS; ++k) {
    if (childPids[k] > 0) {
      if (waitpid(childPids[k], NULL, WNOHANG) != 0) {
        childPids[k] = 0;                        // Child is done
      } else { stillWaiting = 1; }               // Chils is not finished
} } } while (stillWaiting);
```

Problem : this approach is cumbersome and prone to mistakes.

- Necessity to handle the shared memory,
- Necessity to handle the mutexes for concurrency,
- Necessity to spawn and wait for the processes.

$\Rightarrow$ Some expertise is required to write correct code.

Optimally, one can encapsulate the system bits into a library or a framework.

Less naive implementation with threads

C and C++ offer a `thread` library for multithreading :

```
mpf_t glob_var;

void bbp_worker(unsigned int k, unsigned int size, mpf_t glob_var) {
  mpf_t work; mpf_init(work);
  bbp_computation(k, size, work);
  mtx.lock();                          // Start atomic operation
  mpf_add(glob_var, glob_var, work);
  mtx.unlock();                        // End atomic operation
}

int main(void) {
  mpf_init(glob_var);
  std::thread threadArray[num_threads];
  for(int i=0;i<num_threads;i++)   // Start children
    threadArray[i] = std::thread(bbp_worker, i,
                                 num_loops*size/num_threads,
                                 glob_var);
  for(int i=0;i<num_threads;i++)   // Wait for children to exit
    threadArray[i].join();
}
```

- Considerably simpler code ;
- Still necessary to handle explicitly the threads and mutexes.

Easier parallelization

Some frameworks for simplifying parallelization of tasks :

- C++ : Threading Building Blocks, `https://www.threadingbuildingblocks.org/`
- C, C++ : OpenMP, `http://openmp.org/wp/`, `http://bisqwit.iki.fi/story/howto/openmp/`
- C, C++ : CilkPlus, `https://www.cilkplus.org/`

Main ideas :

- Frameworks consisting of very few functions, keywords, or macros.
- Sprinkle the code with some parallelization annotations.
- The runtime handles the allocations and memory accesses.

Example in Openmp

```
mpf_t glob_var;
mpf_init(glob_var);

#pragma omp parallel for shared(glob_var) num_threads(num_threads)
  for(int k=0; k<num_loops; ++k)  {    // OpenMP parallel loop
    mpf_t work;
    mpf_init(work);
    bbp_computation(k, size, work);
    mpf_add(glob_var,glob_var,work);
  }
```

- Programmer inserts `#pragma` directives to indicate parallelism.
- Safety of shared access, even if it is still necessary to define what is shared.
- Few modifications of the existing code, though the `#pragma`s look untidy.

Example in Threading Building Blocks (1/2)

```
mpf_t glob_var;
mpf_init(glob_var);
task_scheduler_init init(num_threads); // Initialize TBB

parallel_for(0, num_loops,
             [&glob_var,size] (const int k) {
               mpf_t work;
               mpf_init(work);
               bbp_child(k, size, work);
               mpf_add(glob_var,glob_var,work);
             });
```

- The example uses a C++ lambda operation for brevity ;
  The more classical example uses a `struct` for storing the function.
- Requires more modifications of the code compared to OpenMP,
- But using functions gives more flexibility than a separate compilation phase.

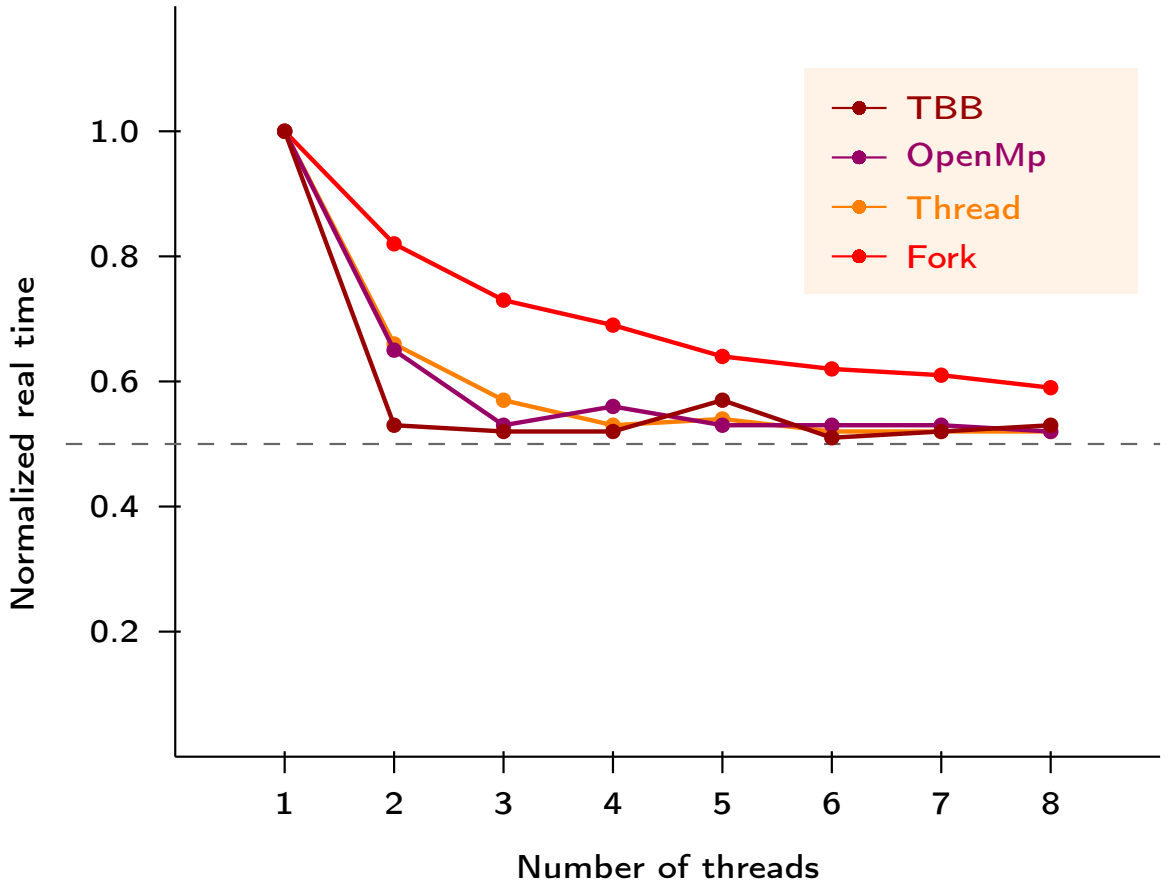
Example in Threading Building Blocks (2/2)

```
task_scheduler_init init(num_threads);              // Initialize TBB

return parallel_reduce(
    blocked_range<int>(0, num_loops*size), glob_var,
    // Range accumulation function
    [size] (const blocked_range<int>& r, mpf_t *init) -> mpf_t* {
          mpf_t* work;  mpf_p_init(work);
          mpf_set(*work, *init);
          for(int k=r.begin(); k<r.end(); ++k )  // work = Σ r_k
            mpf_add(*work, *work, *bbp_element(k));
          return work;
    },
    // Pair reduction function
    [] (mpf_t* work1, mpf_t* work2) -> mpf_t* {
          mpf_t* glob;  mpf_p_init(glob);
          mpf_add(*glob,*work1,*work2);             // glob = work1+work2
          return glob;
    });
```

- The same example using a `parallel_reduce` function.
- No global variable, the runtime passes the results from thread to thread. (this example does not handle freeing the memory)

Results



CPU : Intel Core 2 Duo (2 virtual processors)

In fact, these libraries are much more generic than simple loop parallelization :

```
class FibTask: public task {
public:
    const long n;      long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
    task* execute() { // Override virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y; // Allocate children tasks
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            set_ref_count(3);
            spawn( b );                    // Start b.
            spawn_and_wait_for_all(a);     // Start a and wait
            *sum = x+y;                    // Do the sum
        }
        return NULL; }};

long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;   }
```

. . . here, the computation of the Fibonacci numbers by spawning recursive tasks.

$$f_n = f_{n-1} + f_{n-2}$$

Aside : Example with GoRoutines

```
ch := make(chan Float)
glob_var := new(Float)
glob_var.SetPrec(prec)              // Set the precision

runtime.GOMAXPROCS(num_threads) // Set the number of threads
for i := 0; i < num_loops; i++ {
    go bbp_computation(size, i,  ch) }

for i := 0; i < num_loops; i++ {
    fl := ← ch
    glob_var.Add(glob_var, &fl)  }
```

- Uses **goroutines** and **channels** to represent independent computations and communications in between.
- Results are comparable with other C libraries.

Abstraction of a **generic parallel algorithm** that allows for :

- Efficient implementations :
  - Subscription : adapt the number of threads to the hardware capabilities.
  - Scheduling : the scheduler may adopt dedicated policy by balancing the loads of the threads, or delaying preemption times.
- Low code overhead.
- Portability of the code.

Is it possible to generalize these tactics ?

Skeletal programming

Skeletal programming

Compose high-level algorithms that are prone to parallelization.
Ultimately, the parallelism is handled by a framework or a compiler.

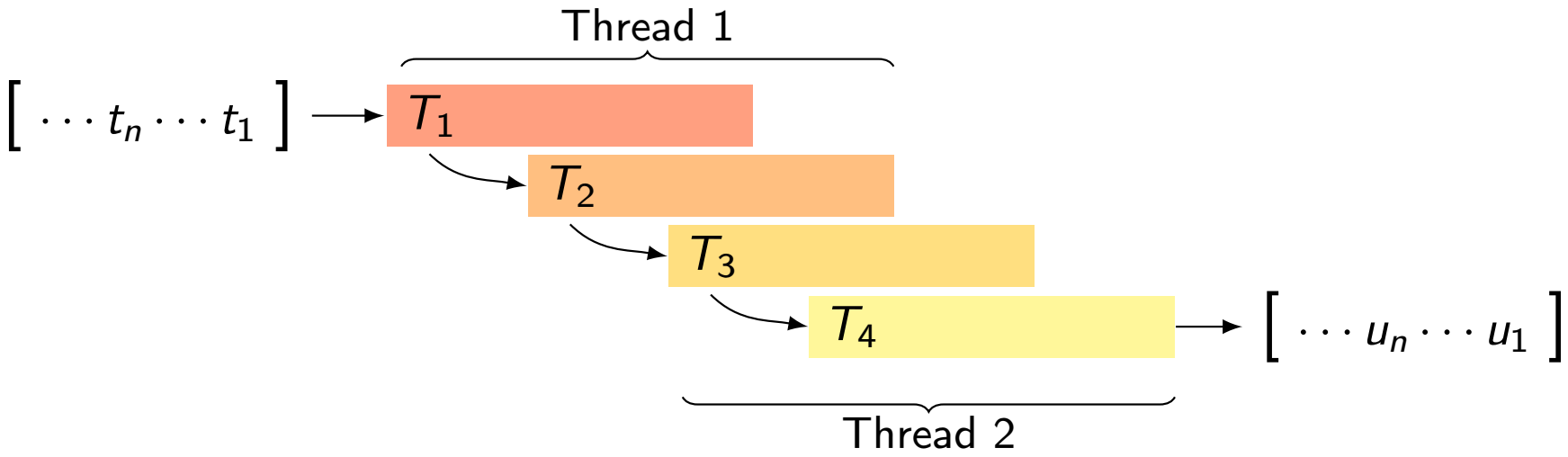
Cf. *Parallel Programming Using Skeleton Functions*, Darlington et al. in 1993,

• For, While	OpenMP, TBB
• Pipeline	Parallel LINQ, Java 8 Streams, MongoDB Aggregation
• Map-Reduce	Eden, Skandium, Hadoop, Scalding, Disco, Spark, Storm, HDInsight, Pig, Hive ...
• Divide & Conquer	Eden, Skandium

Some **algorithmic skeletons frameworks** implementations :

- Java : Skandium (discontinued), `http://didawiki.di.unipi.it/../skandium`
- Haskell : Eden, `http://www.mathematik.uni-marburg.de/~eden/`
- C++ : Fastflow, `http://calvados.di.unipi.it/fastflow`

Pipeline



Condition : each task must be independent from the others.

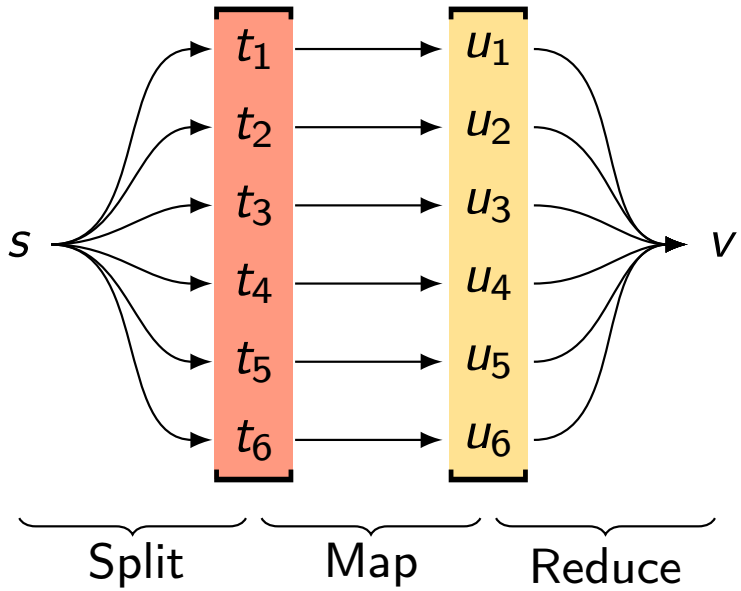
Examples

- composition of tasks on streams of data,
- chains of database queries,

```
from elem in array orderby elem descending where elem > 2 select elem;
```

- generalized in the producer/consumer pattern.

Map Reduce



Condition : each task of the map must be independent of the others.

Examples

- distributed grep,
- count of URL access frequency in a set of logs,
- reverse web-link graph on a set of URLs ...

Example using Java streams

Using the Java 8 Streams framework :

```
public static class BigDecimalSumCollector implements
       Collector<BigDecimal,BigMutableDecimal,BigDecimal> { ... }

List<Integer> array = iota(0,n);                          // [0,1,...,n-1]
BigDecimal res = array.parallelStream()
            .map(s → bbp_simple(PREC, s))                 // map
            .collect(new BigDecimalSumCollector());       // sum
System.out.println(res);
```

- May use lambda-expressions or classes inside higher-order functions.
- « When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. » *Java documentation*

Example with the Eden library in Haskell

Using the Eden library in Haskell :

```
workChild :: Int → Int
workChild x = x*x

main :: IO ()
main = do
  let res = foldr (+) ( parMap workChild [1..10] ) -- Map + reduce
  putStrLn ( "Result : " ++ show res )
```

- Single function to modify to switch between sequential and parallel.

Example with the Scalding library in Scala

Using the Scalding library in Scala :

```
val mx : Int = args("input").toInt
val mc = new java.math.MathContext(1000)
TypedPipe.from(new IterableSource(iota(0,n)))    // [0,1,...,n-1]
  .map { k : Int  ⇒ bbp_simple(mc, k) }
  .groupBy { _ ⇒ 0 } // single group
  .foldLeft(BigDecimal(0))((u : BigDecimal,v : BigDecimal) ⇒ (u+v))
  .write(TypedTsv(args("output")))
```

- Scalding is a frontend for the Hadoop framework and only handles Map-Reduce algorithms.
- More complex to deploy, but handles parallelism on clusters.

Example with the Eden library in Haskell

Using the Eden library in Haskell :

```
mergeSortBBP :: [Int] → BigFloat Prec
mergeSortBBP = parDC 1 trivial solve split combine where
  trivial    :: [Int] → Bool
  trivial xs = length xs <= 1
  solve      :: [Int] → BigFloat Prec
  solve [x]  = bbpPiTransform x
  split      :: [Int] → [[Int]]
  split      = splitIntoN 2
  combine    :: [Int] → [BigFloat Prec] → BigFloat Prec
  combine _  = foldl (+) 0

main :: IO ()
main = do
  let res = mergeSortBBP [0..(s-1)]
  putStrLn $ "Final result : " ++ show res
```

- Divide and Conquer algorithm here adapted to solve our problem.

Skeleton Patterns

MapReduce patterns (cf. *MapReduce Design Patterns*, Miner & Shook) :

- Sum & Group (counting, reverse index),
- Filtering, removing duplicates,
- Partitioning, clustering, **sorting**, shuffling.

Divide & Conquer algorithms :

- Merge/quick sort,
- FFT, matrix multiplication and diagonalization,
- Barnes-Hut algorithm for solving the N-body problem,
- Image processing algorithms (convexity, connexity).

In some cases, skeletons can be automatically **converted** into other skeletons.
Example : MapReduce may be encoded into Divide & Conquer.

Other types of parallelization

What kinds of problems are not fit for these techniques ?

- Problems involving a large number of blocking tasks (I/O, mutexes).
- Computations with a large number of communications (messages and data).

In some cases, message passing frameworks with a more precise grain for parallelism, such as PVM or MPI, may be more adapted.

Example : clustering algorithms (such as K-means)

Some good reading

- *Parallel Programming Using Skeleton Functions*,
  Darlington et al., PARLE Conference Proceedings, 1993,
  https://dl.acm.org/citation.cfm?id=691650
- *A Survey of Algorithmic Skeleton Frameworks : High-Level Structured Parallel Programming Enablers*,
  Horacio González-Vélez and Mario Leyton, Practice and Experience 2010,
  https://dl.acm.org/citation.cfm?id=1890757