On the cost of managing data flow dependencies

- program scheduled by work stealing -

Thierry Gautier, INRIA,
EPI MOAIS, Grenoble France

Workshop INRIA/UIUC/NCSA

Outline

- Context
  - introduction of work stealing in Cilk+ / TBB
- How to add data flow dependencies between tasks?
  - experience with Kaapi software
- Preliminary experimental results
  - comparizon with Cilk/TBB
- Conclusions

Context: multicore

- Multicore is the basic building component of super computer

- Dynamic load balancing
  - correct ”unbalanced” work load
    - ✓ variation due to the application
    - ✓ variation due to the environment / OS / ...

- Work stealing scheduling is a good candidate
  - Cilk (Cilk-Mit, Cilk++, Cilk+), TBB

Cilk/TBB task model

- Cilk and TBB have
  - ”task parallelism” == cilk_spawn, tbb::task
    - ✓ task ≡ function call
    - ✓ theoretical foundation (only for Cilk scheduler), $T_P = O(T_1/p + T_\infty)$
  - ”data parallelism” == cilk_for, tbb::parallel_for
    - ✓ at runtime ”tasks” are created
    - ✓ TBB has support for ”affinity” (tbb::affinity_partitioner)

➡ **Independent** tasks

- no dependencies between tasks => synchronization of the control flow (cilk_sync, tbb::task::spawn_and_wait, ...)
- ~ Fork/Join model

Cilk/TBB work stealing

- Each thread owns a *work queue* (WQ)
  - 3 methods on a work queue
    - ✓ push/pop : only called by the owner
    - ✓ steal : only called by a thief
- Work stealing algorithm
  - push/pop to execute work
  - idle thread (with empty work queue) invokes 'steal'
    - ✓ randomly selected victim work queue

Some properties

1. Cilk and TBB have a "C++" elision
   - ✓ "sequential execution" is a valid execution order
2. Number of steal requests per core is $O(T_\infty)$
   - ✓ Low if small critical path T_∞ (highly parallel algorithm)
3. Work first principle
   - ✓ "Minimize scheduling overhead borne by work at the expense of increasing the critical path"
     - Extra operations during steal requests are reported into the critical path

Work queue protocols

- Management of the concurrency on the victim's work queue
  - ✓ several thieves, one victim thread
  - T.H.E. : Cilk, TBB
    - ✓ serialization of thieves using a lock + Dijkstra-like protocol between one thieves and the victim with lock in rare case
  - ABP (Arora, Blumofe, Plaxton), Chase & Lev, ...
    - ✓ concurrency is managed by read-modify-write atomic instruction (*i.e.* compare & swap)
- Need of memory barriers to ensure sequential consistency

Our goal

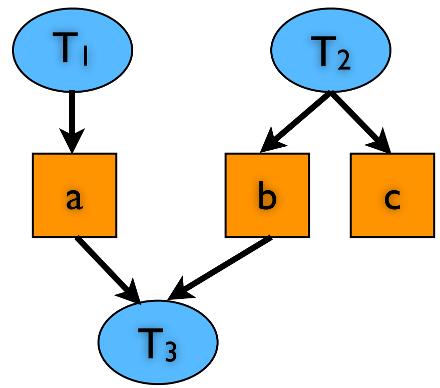
- Extend the ”task model” of TBB/Cilk+
  - tasks with data flow dependencies
    - ✓ fine model of application for scheduling
    - ✓ automatic management of communications between tasks
      - partitioning of the data flow graph / iterative application
      - data transfer between GPUs and CPUs into a multicore
      - add useful semantics to specialize coherency protocol in presence of copies
    - ✓ improvement of checkpoint/rollback protocol [X. Besseron]
      - 1 month at UIUC in march 2010

➡ How to manage efficiently such dependencies ?

- comparison with TBB/Cilk+

Data flow dependencies

- A task is ready iff all its inputs are produced
  - data flow machine
  - some runtime



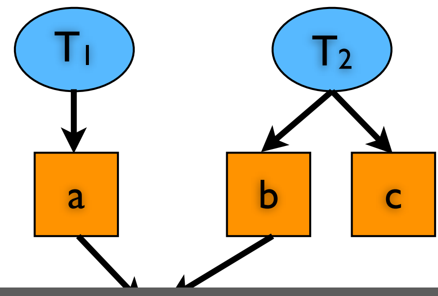
- Two main costs

1. at task creation: storing the data flow dependencies
2. at task execution: **computation** of "ready" property

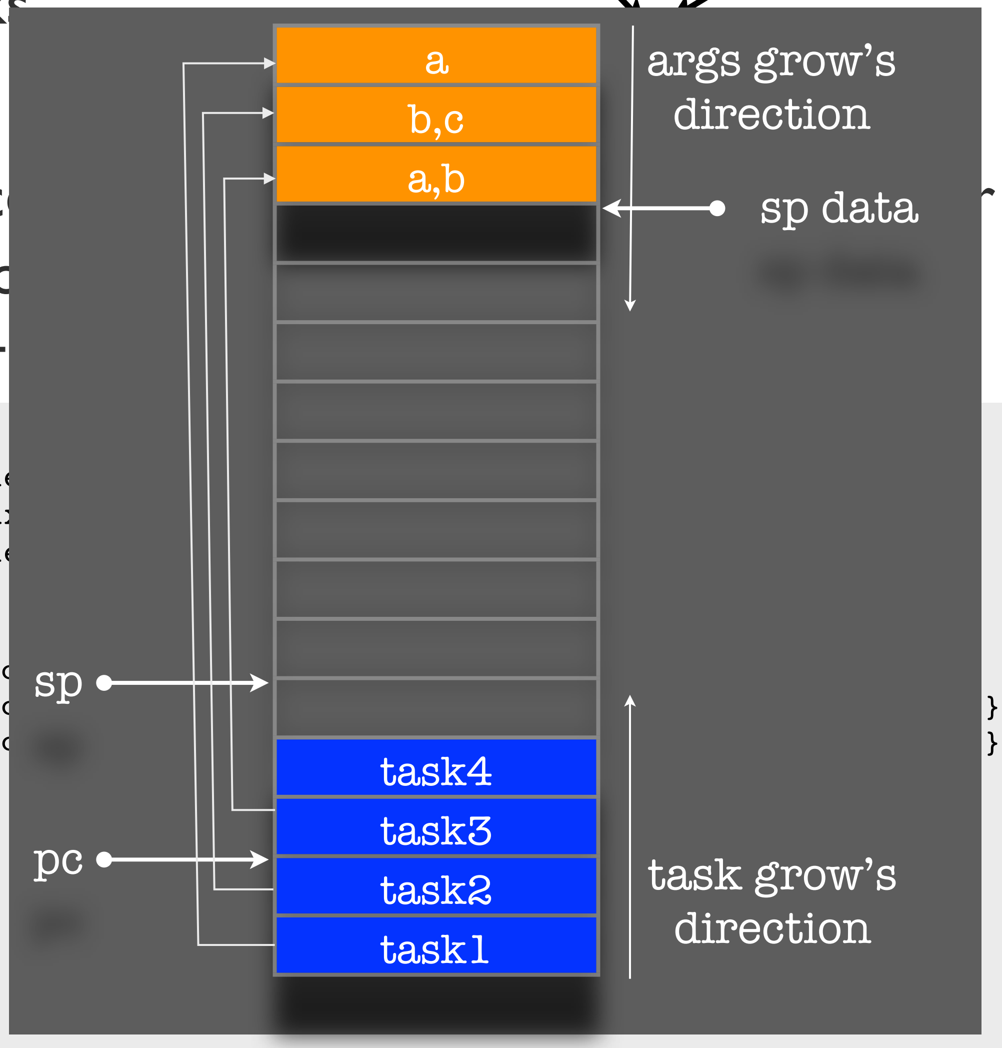
Overview of impl. in Kaapi

- Kaapi
  - C / C++ library
  - high level API: macro data flow programming
  - low level API: for fine grain adaptive algorithm
- Optimization of 3 aspects
  - Task representation
    - ✓ very light = function call + pointer to effective parameters
  - Task execution
    - ✓ take into account specificity of work stealing based execution
  - Data flow representation
    - ✓ lazy approach: compute data flow constraints only when required

Kaapi task model



- Global address space
  - ✓ data shared between tasks
- Task ~ function call
  - ✓ task has a signature = acc
  - ✓ several implementations c
  - ✓ sizeof task = 2 pointers +




```
/* Signature for T1, T2, T3 */
struct Task1: public Task<1>::Signature<W<double
struct Task2: public Task<2>::Signature<W<Matri
struct Task3: public Task<2>::Signature<R<double

/* Task body for T1, T2, T3 */
template<> struct TaskBodyCPU<Task1> { void operat
template<> struct TaskBodyCPU<Task2> { void operat                            };
template<> struct TaskBodyCPU<Task3> { void operat                            };

/* Previous graph: */
double* a = ..;
Matrix* b = ..;
int* c = ..;
Spawn<Task1>()( &a );
Spawn<Task2>()( &b, &c );
Spawn<Task3>()( &a, &b );
```

Task creation / execution

- Work first principle applied to Kaapi:
  - optimize the sequential execution...
    - ✓ tasks are executed following the sequential creation order, **without data flow dependencies computation**
  - ...at the expense of increasing the critical path during work stealing requests
    - ✓ **compute ready tasks**

Stealing into a work queue

- Iterate on all the tasks into the work queue
  - ✓ pseudo code:
    - for task *t* in the work queue
      if (compute_ready(*t*)) return *t*;
  - ✓ order of iteration = sequential order of creation

- Non constant complexity
  - ✓ bounded by the depth of the computation
  - ✓ constant if independent tasks
  - ✓ use hash map to retrieve a same data accessed by several tasks

- Main costs are reported during steal request but:
  - increasing the critical path reduce the scalability
  - average parallelism = T_1/T_∞

Cost to create task

- Task creation (average of 1000 spawns of task)
  - opteron 875, 2.2Ghz, gcc 4.4.2, -O3
  - Time(spawn) = 12 cycles (~5.6 ns) per task
  - + ~ 3cyles / pointer arguments

Execution overhead

- Objective: comparizon of Data Flow [Kaapi] vs Independent Task [Cilk/TBB] on multicore machine
- 1 program (fibonacci), 4 implementations
  - sequential
  - Cilk+
  - Kaapi
  - TBB
- Timing
  - T_1 for Cilk, Kaapi, TBB = time using 1 core
  - T_p using p cores
  - T_{seq} sequential implementation
- Environment
  - Intel X7460, 2.67Ghz, 4 x 24 cores = 96 cores, NUMA
  - gcc-4.4 -O3
  - Intel icpc 12.0 -O3

Cilk+

```
long fib(long n
{
  if (n < 2)
    return (n);
  else {
    long x, y;
    x = cilk_sp
    y = fib(n
    cilk_sync;
    return (x +
  }
}
```

TBB

```
struct FibContinuation: public tbb::task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {}
    tbb::task* execute() {
        *sum = x+y;
        return NULL;
    }
};

struct FibTask: public tbb::task {
    long n;
    long * sum;
    FibTask( const long n_, long * const sum_ ) :
        n(n_), sum(sum_)
    {}
    tbb::task* execute() {
        if (n<2) {
            *sum = n;
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            // Set ref_count to "two children".
            c.set_ref_count(2);
            c.spawn( b );
            return this;
        }
    }
};
```

```
s, const long n )
```

Sequential execution overhead

Sequential	Cilk+	TBB	Kaapi
1.67s (slowdown:1)	11.8s (x 7.07)	17.86s (x 10.69)	7.99s (x 4.78)

- No extra data flow constraints to solve in the Kaapi execution (1 core => no steal!)
- Grain size selection to amortize overhead

Multicore

- 4x24 cores machine from EPI RUNTIME [Namyst]
  - ✓ Intel X7460, 2.67Ghz, NUMA
  - ✓ Time in second, fibonacci(40)

#Cores	Kaapi	Cilk+	TBB
1	7.99	11.81	17.86
16	0.50	0.78	1.13
24	0.33	0.58	0.75
48	0.18	0.32	0.39
64	0.17	0.22	0.30
96	0.21	0.14	0.20

- Kaapi has good runtime up to #cores in]64,96]
  - Increase of the critical path T_∞ (less average parallelism)
- Small speeds (Tseq=1.67s) due to too fine grain

Conclusions

- For fork/join program & work stealing scheduling
  - data flow does not cost vs independent tasks
    - ✓ fine grain implementation + work first principle
  - drawback: reduction of the scalability

- On going optimizations
  - Use T.H.E work queue
    - ✓ currently based on costly atomic operation
  - Work stealing requests aggregation
    - ✓ critical path optimization + better load balance
  - Taking into account shared cache
    - ✓ lock, biased work stealing

Status of Kaapi software

- Beta version rc2 under testing
  - http://kaapi.gforge.inria.fr
- Next month: official release
  - work stealing + distributed memory architecture + initial pre decomposition (graph partitioning) + GPUs/CPUs
- Applications
  - SOFA (http://www.sofa-framework.org/)
    - ✓ multi CPUs, multi GPUs
  - numerical iterative application
  - parallelization of VTK