X-KAAPI :
Adaptive Runtime System for Parallel Computing

Thierry Gautier, thierry.gautier@inrialpes.fr
Bruno Raffin, bruno.raffin@inrialpes.fr

MOAIS team
INRIA Grenoble Rhône-Alpes

Challenge

- Multi-core processors make parallelism a central paradigm
- Automatic parallelization is not yet reay for prime time. Users need to parallelize their codes by hand
- But we can help them providing easy to handle APIs and powerfull run-time environments that take care of many nasty details.

Work Stealing

Two aspects:

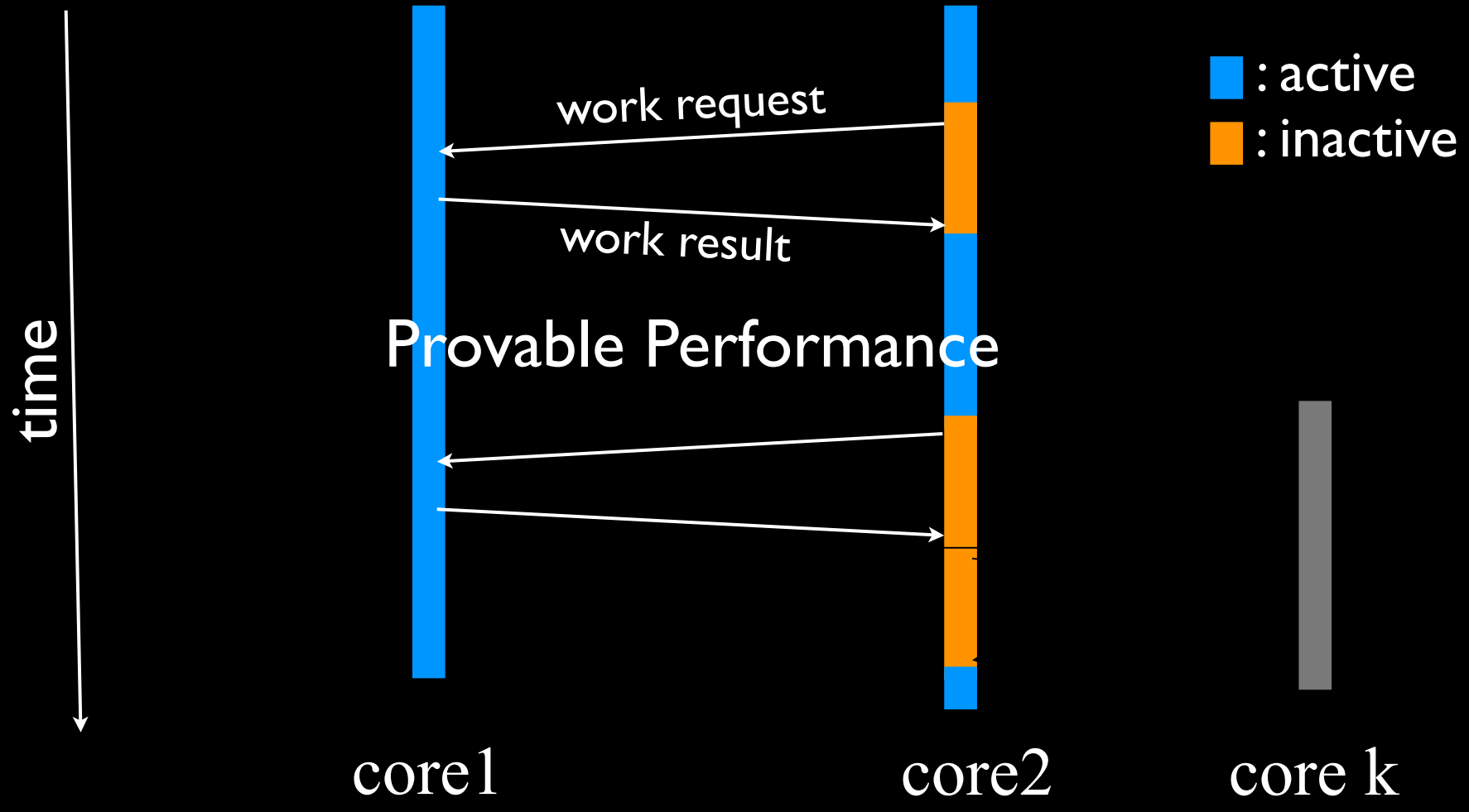
- Task scheduling and load balancing: run-time job (not exposed to programmer)
- Adapted APIs that keep the sequential semantics of the program (easier to understand)

Work stealing has gained a lot of attention:

- A promising approach for multi-core programming (overhead can be really small in htis context)
- Work stealing based tools: Intel TBB, Cilk (MIT -> ClikArt -> Intel), Kaapi (INRIA)

Work Stealing Principle

- Cores execute tasks locally first (newly created tasks pushed on local stack)
- If idle (no local task), steal work from an other core

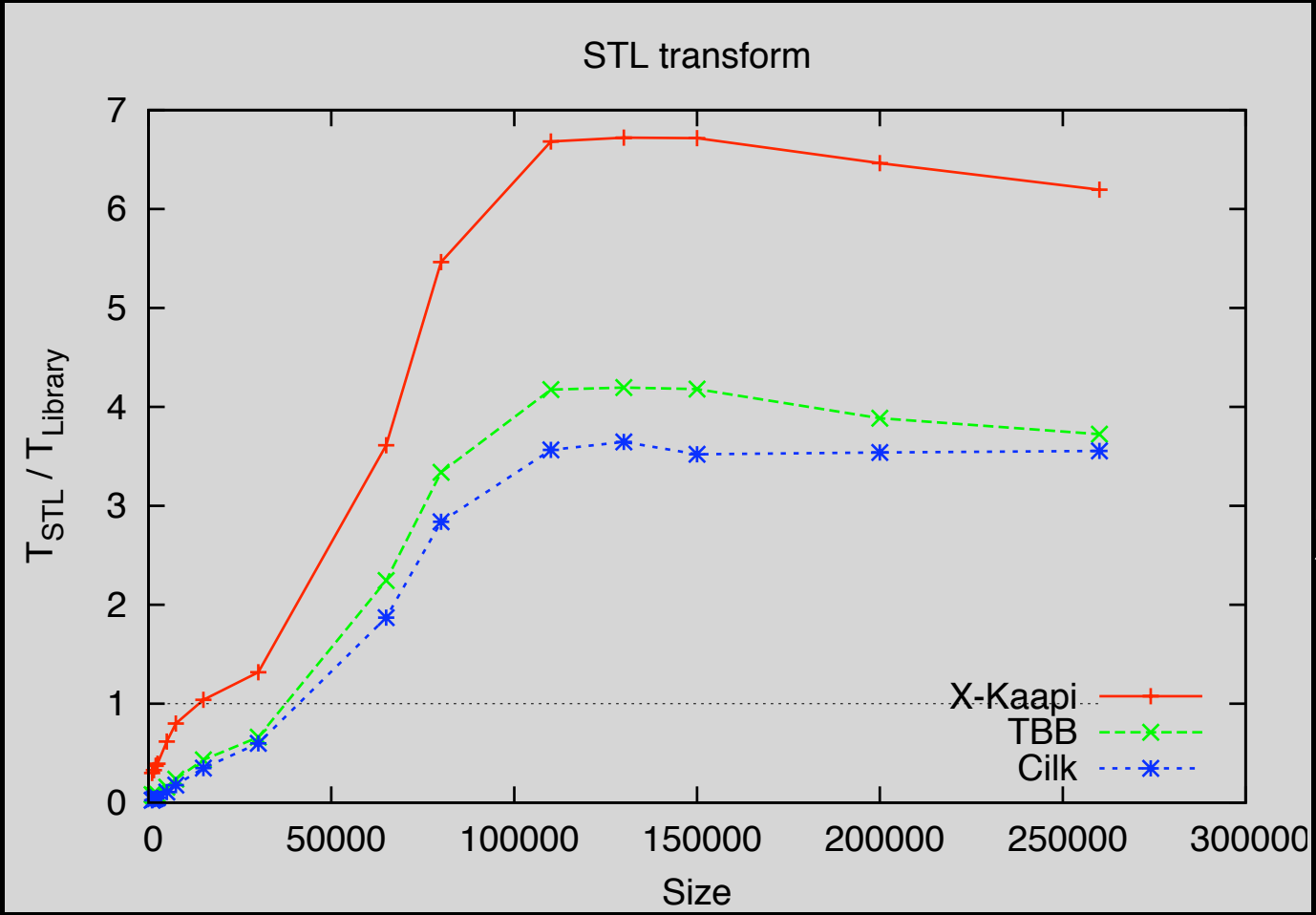


KAAPI

- A low hoverhead implementation of work stealing
- Base API:
  - Shared memory model
  - Spawn: delimit a task boundary
  - Data dependencies: implicitly defined by type of task arguments (read, write, read/write)
  - sync: wait for previously spawned task completion
  - Similar to Cilk but with data flow dependencies

Comparison with Cilk/TBB

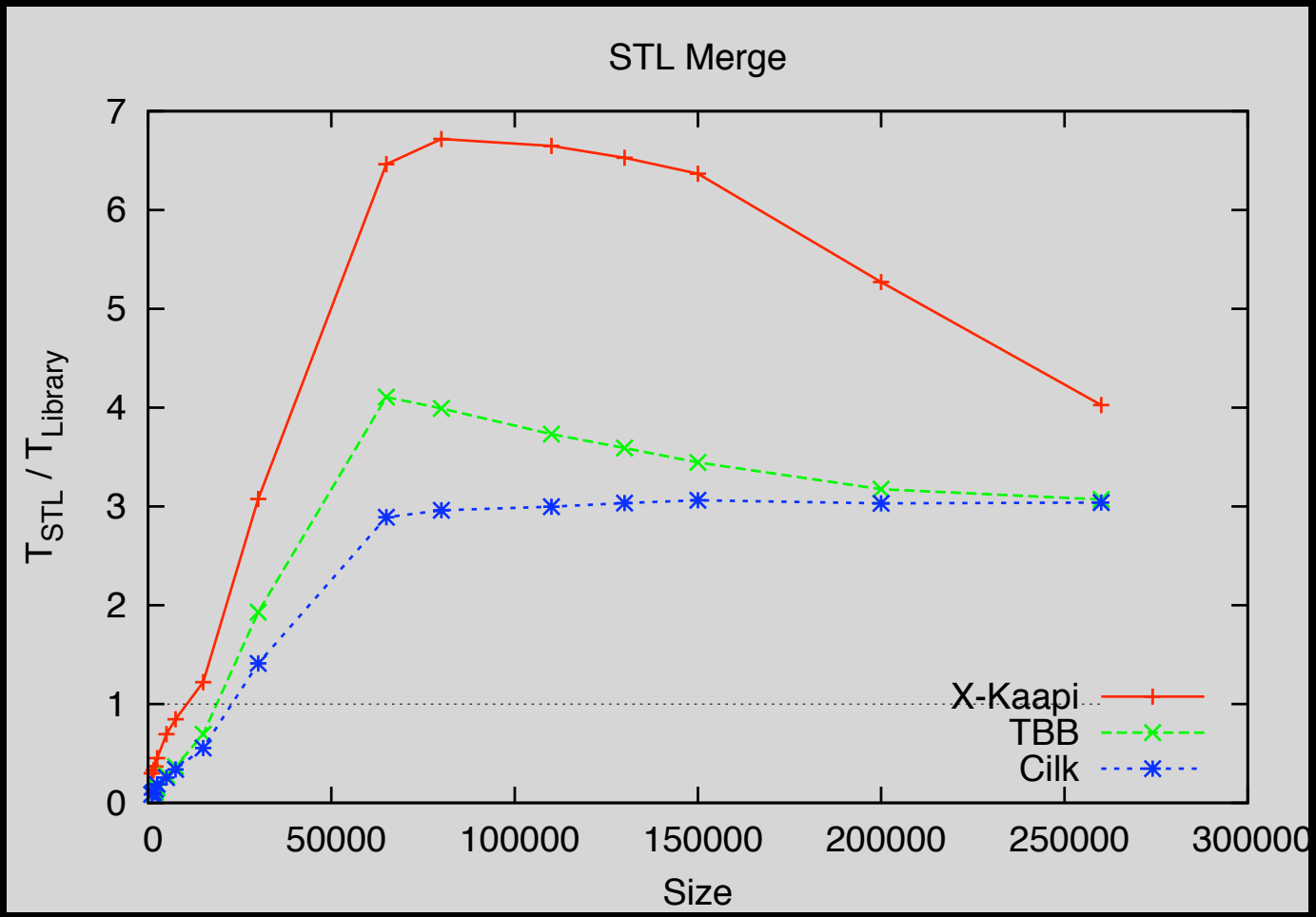
- 8 processors NUMA machine
  - STL Transform, Ratio T_{stl} / $T_{library}$ on 8 cores



2009 Tests

Comparison with Cilk/TBB

- 8 processors NUMA machine
  - STL Merge, Ratio T_{stl} / $T_{library}$ on 8 cores



2009 Tests

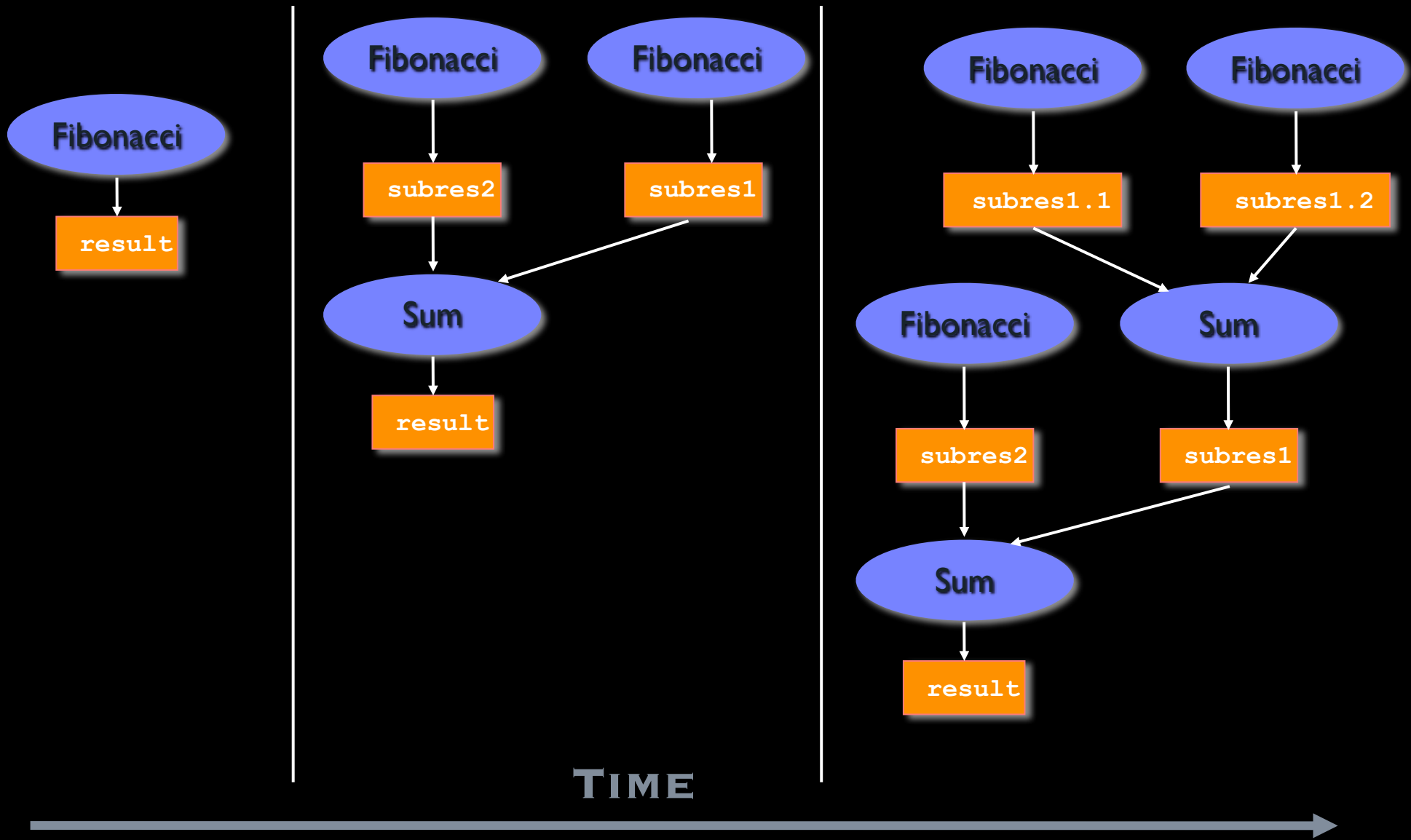
Fibonnacci (Sequential)

```
struct Fibonacci {
  void operator()( int n, int * result )
     {
         if (n < 2) result = n ;
         else {
             int subresult1;
             int subresult2;
             Fibonacci () (n-1, &subresult1);
             Fibonacci () (n-2, &subresult2);
             Sum()(result, &subresult1, &subresult2);

         }
     }
};

struct Sum {
  void operator()(   int * result,
                     int * sr1,
                     int * sr2 )
     { *result =  *sr1 + *sr2; }
}
```

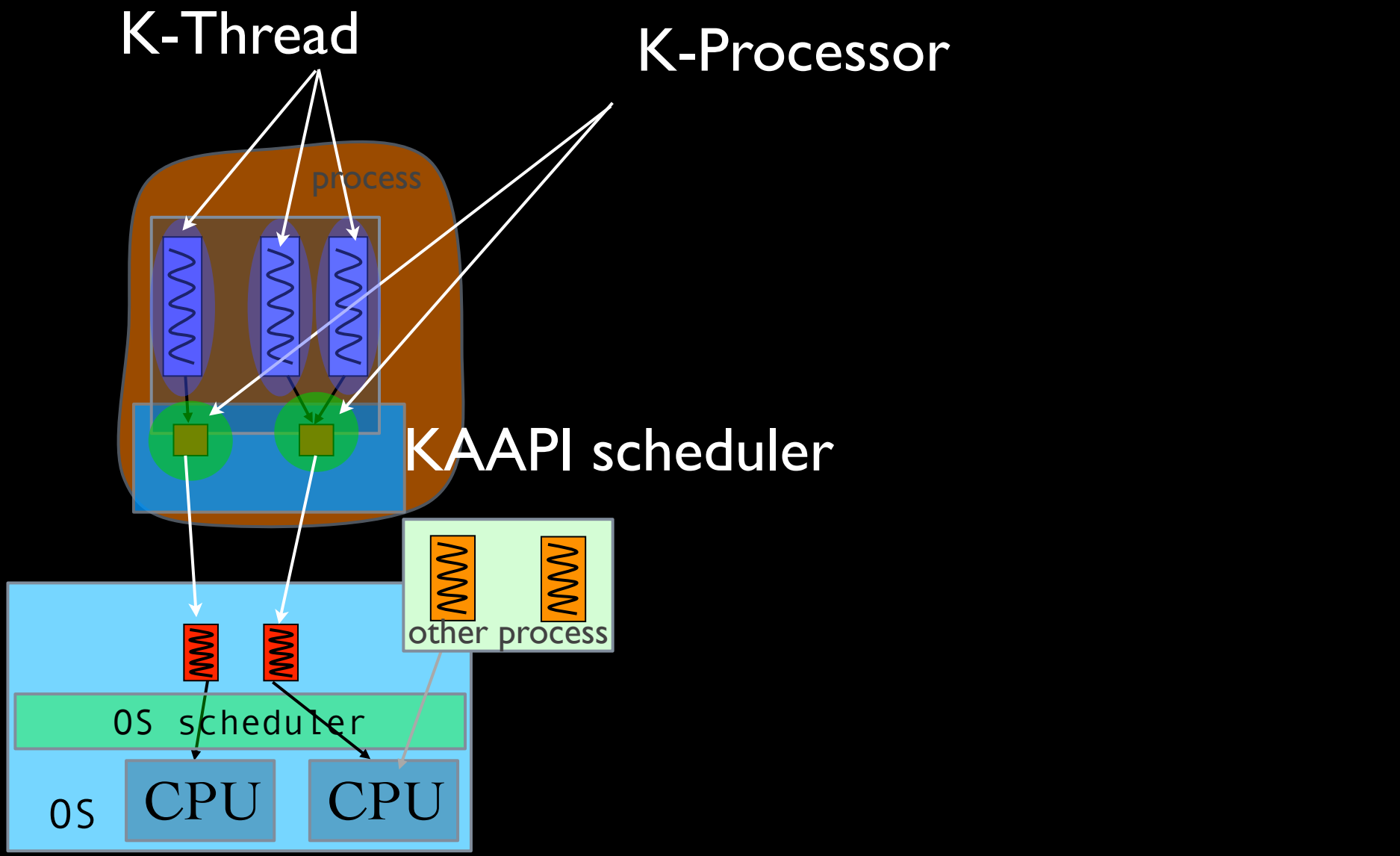
Fibonnacci (KAAPI)

```
struct Fibonacci {
  void operator()( int n, int * result )
     {
         if (n < 2) *result = n ;
         else {
             int subresult1;
             int subresult2;
             Spwan<Fibonacci>()(n-1, &subresult1);
             Spawn<Fibonacci>()(n-2, &subresult2);
             Spwan<Sum>()(result, &subresult1, &subresult2);
             Sync();
         }
     }
};

struct Sum {
  void operator()(   int * result,
                     int * sr1,
                     int * sr2 )
     { *result =  *sr1 + *sr2; }
}
```

Data Flow Graph



2-Level Scheduling



Data dependencies: pointers

Task signature

```
/* Kaapi Accum task: takes an array, its size, and compute the sum of the elements */
struct TaskAccum: public ka::Task<3>::Signature<
                                        ka::W<double>, // output (write) sum
                                        ka::R<double>, // input (read) array
                                        int>           // size of the array
{};
```

```
struct TaskPrint: public ka::Task<1>::Signature< ka::R<double> > {};
```

Task CPU implementation (GPU also supported)

```
template<> struct TaskBodyCPU<TaskAccum> {
  void operator()(double * sum, double * array, int size )
  {
    * sum = kastl::accumulate( array, array+size );
  }
};
```

Data dependency:

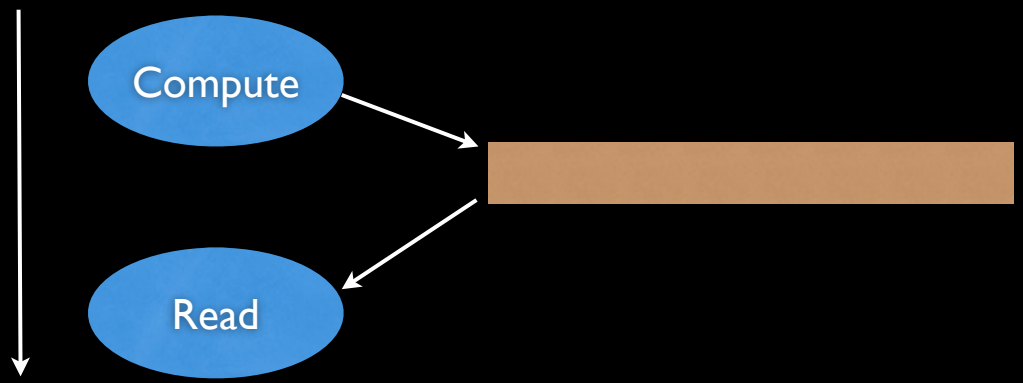
```
double* array = new double[size];
double sum;
Spawn<TaskAccum>()( &sum, array, size );

Spawn<TaskPrint>()( &sum );
```

R-W dependencies

Spawn/Sync Interface

- **Principle: detection of data flow dependencies between a sequence of function calls (Tasks)**

```
double* myarray = ...;

Spawn<Compute>()( myarray ); // W access

Spawn<Read>()( myarray ); // R access
```



Online computation of data flow dependencies

KaSTL

- Parallel STL algorithm on top of random access iterators
  - std::for_each(C.begin(), C.end(), Op);
  - ...and all others STL algorithms...
- Extension: Parallel algorithm for forward iterators

Adapative Application Interface

- Low level but very efficient interface
- Allows a direct interaction with the work stealing scheduler:
  - Adaptive behavior: dynamic task granularity
  - Possibility to aggregate steal requests and to preempt other cores.
  - Execution of the sequential (efficient) code and extraction of parallel work when other cores are inactive (work stealing)

VTK/KAAPI Project

- Develop a VTK specific API following a STL style but adapted to VTK data structures.
- Target intra algorithm parallelism first
- Need to rewrite algorithms (but expect parallel patterns to be reused in several algorithms)

GPU support

- Task body can have several implementations (CPU + GPU)
- Work stealing schedules tasks on both CPUs and GPUs
- Ealry experiments:
  - 8 GPUs: 7x (or 60x compared to single CPU core execution)
  - Cooperative speedup on 4 CPUs + 4 GPUs:

    Speedup(4CPUs + 4 GPUs) > Speedup(4CPUs)+ Speedup(4GPUs)