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 powerful 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 this 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 another core

Provable Performance

**Diagram:**
- Time axis
- Cores: core1, core2, core k
- Work request and work result arrows
- Blue: active
- Orange: inactive
KAAP\textbf{I}

\begin{itemize}
  \item A low hoverhead implementation of work stealing
  \item \textbf{Base API:}
    \begin{itemize}
      \item Shared memory model
      \item Spawn: delimit a task boundary
      \item Data dependencies: implicitly defined by type of task arguments (read, write, read/write)
      \item \texttt{sync}: wait for previously spawned task completion
      \item Similar to Cilk but with data flow dependencies
    \end{itemize}
\end{itemize}
Comparison with Cilk/TBB

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

![Graph showing comparison between STL Transform, TBB, Cilk, and X-Kaapi](image-url)

2009 Tests
Comparison with Cilk/TBB

- 8 processors NUMA machine
- STL Merge, Ratio $T_{stl} / T_{library}$ on 8 cores
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;
    }
};
struct Fibonacci {
    void operator()( int n, int * result )
    {
        if ( n < 2 ) *result = n ;
        else {
            int subresult1;
            int subresult2;
            Spawn<Fibonacci>()(n-1, &subresult1);
            Spawn<Fibonacci>()(n-2, &subresult2);
            Spawn<Sum>()(result, &subresult1, &subresult2);
            Sync();
        }
    }
};

struct Sum {
    void operator()(
        int * result,
        int * srl,
        int * sr2
    )
    {
        *result = *srl + *sr2;
    }
};
Data Flow Graph

- Fibonacci
  - subres2
  - Sum
  - result

- Fibonacci
  - subres1
  - Sum
  - result

- Fibonacci
  - subres1.1
  - subres1.2
  - Sum
  - result

Time
2-Level Scheduling

K-Thread

K-Processor

KAAPI scheduler

OS scheduler

other process

OS CPU CPU

2010/10/6 - MOAIS team - INRIA
**Data dependencies: pointers**

**Task signature**

```c++
/* 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)**

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

**Data dependency:**

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

R-W dependencies

```c++
Spawn<TaskPrint>()( &sum );
```
Spawn/Sync Interface

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

```c
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/KAAPL 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

• Early experiments:
  - 8 GPUs: 7x (or 60x compared to single CPU core execution)
  - Cooperative speedup on 4 CPUs + 4 GPUs:

    \[ \text{Speedup}(4\text{CPUs} + 4\text{GPUs}) > \text{Speedup}(4\text{CPUs}) + \text{Speedup}(4\text{GPUs}) \]