Threads are Evil, Tasks are Good: Towards a Unified Resource Management Framework

Mats Brorsson
SICS & KTH Royal Institute of Technology, Sweden
“Though shalt not work with cores, work with chores…”

Freely from James Reinders, Intel
The current (architectural) situation

- Multicore (C/C++, Java, C# etc)
- Accelerators
  - GPGPU (OpenCL, CUDA, OpenACC)
  - Manycores (Tilera, Adapteva, Kalray etc)
  - Fixed function (Cryptography, FFT etc)
- Reconfigurable computing
  - CPU + FPGA
  - CGRA

- Bottom line: parallel, heterogeneous, difficult to program
The current (run-time) situation

- Rigid interfaces between layers
- No or little semantic information flowing across layers
- 40+ year old OS abstractions
- Multiprogramming in multicores not well supported (or even understood)
The current (compile-time) situation

- Explicit threads still rule
- Choice between parallel libraries or parallel programs
- Need to adapt to architecture(s)
- Difficult to compose different parallel components
- Difficulty to combine multiple languages
  - C/C++
  - Java/C#
  - Functional languages
  - DSL:s etc
Main challenges

• Difficult to write composable parallel software
• The parallel models of different languages do not work well together
• Poor resource management
• Difficult to write portable parallel software
Make *Tasks* a First Class Citizen

- **Separation of concerns**
  - Concentrate on exposing parallelism
  - Not how it is mapped onto hardware
An example of task-parallelism

- The (naïve) sequential Fibonacci calculation

```c
int fib(int n){
    if( n<2 ) return n;
    else {
        int a,b;
        a = fib(n-1);
        b = fib(n-2);
        return b+a;
    }
}
```

Parallelism in fib:
- The two calls to fib are independent and can be computed in any order and in parallel.
- It helps that fib is side-effect free but disjoint side-effects are OK.

The need for synchronization:
- The return statement must be executed after both recursive calls have been completed because of data-dependence on a and b.
A task-parallel fib in OpenMP 3+

Starting code:

```c
int fib(int n){
    if( n<2 ) return n;
    else {
        #pragma omp task shared(a)
        a = fib(n-1);
        #pragma omp task shared(b)
        b = fib(n-2);
        #pragma omp taskwait
        return b+a;
    }
}
```
Task-centric parallel models

• Gaining momentum

• C/C++:
  - OpenMP 3/4, Cilk Plus, TBB, GCD...

• C#:
  - Microsoft TPL

• Java:
  - fork/join

• Erlang:
  - processes

• X10:
  - activities

• Etc...
Task model benefits

- Automatic load-balancing through work-stealing
- Serial semantics => debug in serial mode
- Composable parallelism
- Parallel libraries can be called from parallel code
- Can be mixed with data-parallelism
  - SIMD/Vector instructions
  - Data-parallel workers can also be tasks
- Adapts naturally to
  - Different number of cores, even in run-time
  - Different speeds of cores (ARM big.LITTLE)
Greatest thing since sliced bread?

- Overheads are still too big to not care about when creating tasks
  - Tasks need to have high enough *arithmetic intensity* to amortize the cost of creation and scheduling
- Different models do not use same run-time
  - You can’t have a task in TBB calling a function creating tasks written in OpenMP
- The operating system does not know about tasks
  - Current OS:s only schedules threads
- Still no accepted way to target different ISA:s
  - Research is going on
OS scheduling abstractions: with threads

- The OS schedules threads on cores
- Threads constitute mandatory parallelism
- Threads of a single application are not distinguished from other threads
- Resource mapping done by programmer
OS scheduling abstractions: with tasks

- Tasks are scheduled onto user-level worker threads
- Tasks constitute potential parallelism
- The OS schedules threads on cores
- Threads of a single application are not distinguished from other threads
- Resource allocation done by programmer
  - Mapping by the RTS
OS scheduling abstractions: cooperative scheduling of tasks

- Abstract each physical core as a shared, persistent, worker thread
- Workers can accept tasks from any application
- The OS scheduler allots workers to applications
  - Optimistically
  - On demand
  - Asynchronously
- Cooperates with any task-centric programming model that can provide the *appropriate information* on resource requirement
Typically random victim-selection work-stealing is used
- Some nice theoretical properties are possible

We show that with a specific deterministic victim selection, we can:
- Increase likelihood of finding work when load balancing
- Estimate number of usable cores
Palirria: Adaptive resource allocation

- A mathematically proven method to estimate the number of resources an application can effectively use
- Can be used to save resources (power)
- Can be used to re-assign resource between applications dynamically
1) Make all inter-core communication explicit.
2) Make OS structure hardware-neutral.
3) View state as replicated instead of shared.
Greatest thing since sliced bread?

- Overheads are still too big to not care about when creating tasks
  - Tasks need to have high enough arithmetic intensity to amortize the cost of creation and scheduling
- Different models do not use same run-time
  - You can’t have a task in TBB calling a function creating tasks written in OpenMP
- The operating system does not know about tasks
  - Current OS:s only schedules threads
- Still no accepted way to target different ISA:s
  - Research is going on
Heterogeneous processing

- Same ISA, performance heterogeneous processing is transparent
- Real heterogeneity is challenging
OpenMP 4.0: extending OpenMP with dependence annotations

```c
#pragma omp task depend(inout:A)  
    sort(A);
#pragma omp task depend(inout:B)  
    sort(B);
#pragma omp task depend(inout:C)  
    sort(C);
#pragma omp task depend(inout:D)  
    sort(D);  
    // taskwait not needed  
#pragma omp task depend(in:A,B, out:E)  
    merge(A,B,E);
#pragma omp task depend(in:C,D, out:F)  
    merge(C,D,F);
```

A

B

C

D

A, B

C, D

A, B

C, D
Benefits of tasks in OpenMP 4.0

- More parallelism can be exposed
- Complex synchronization patterns can be avoided/automated
- Knowing a task's memory usage/footprint, offloading to accelerators can now be made almost transparent to the user
  - In terms of memory handling and execution!
Going heterogeneous

• **Goal**: Explore and improve execution performance when scheduling for heterogeneous architectures
  - Unconditionally given all system resources, how well can the run-time scheduler perform

• **Our platform**:
  - A 4-processor AMD x86-64 platform
  - 1 GPU (nVidia Quadro FX 570)
  - 1 TilePRO64 embedded processor
Going heterogeneous

- GPU – Traditional graphics processor enhanced with the ability to perform data-parallel work
- A “GPU” task is written in CUDA, and performs the same operation as a HOST(x86) task
- Under the OmpSs programming model, the run-time system will:
  - Manage memories, incl. Transfers and consistency
  - Schedule tasks to execute on the GPU
Writing heterogeneous code

### GPU

```c
#pragma omp task device(gpu)
implements(inc_arr)
void cuda_inc_arr(int *A, int *B) {
    cuda_inc_array_kernel <<<4,256>>>(A,B);
}
```

### TilePRO64

```c
#pragma omp task device(tilera) implements(inc_arr)
void tilera_inc_arr(int *A, int *B) {
    #pragma omp parallel
    {
        int i = omp_get_thread_num();
        B[i] += A[i];
    }
}
```

### Task Spawn

```c
#pragma omp task input(A) output(B) target(tilera,gpu,host)
inc_arr(&A[0], &B[0]);
```
Going heterogeneous

• TilePRO64
  - 64 MIPS-like cores
  - VLIW
  - Mesh NoC
  - No FPU

• The TilePRO64 was not intended to be used as a co-processor
  - Unlike the GPU whose solely purpose is just this
  - We created a micro-kernel capable of handling requests as well as distribute work amongst cores
Scheduling for heterogeneity

- We created a run-time system capable of handling OmpSs. We call it UnMP...
- Our task was to evaluate some standard scheduling policies
  - Random work-deal scheduling
  - Work-Steal scheduler
  - Weighted Random
  - ...and possible improving upon them!
Scheduling for heterogeneity

- Random Work-Dealer
Scheduling for heterogeneity

- Random Work-Stealer
Scheduling for heterogeneity

- Weighted-Random
Scheduling for heterogeneity

- We extended the weighted-random with history-based prediction
  - We call it the FCRM-scheduler
- Use execution history-based information together with regression to make a prediction concerning new tasks
- However, execution history is not always enough, neither is the amount of data required by a task:
  - A task need not to correlate with the amount of data it requires
Example Experimental Result

- **N-Body simulation**

  N-Body simulation for various input sizes

[Graph showing speedup normalized to sequential vs. number of bodies for different methods: Random, Work-Steal, Weighted-Random, FCRM, FCRM-PRELOAD, and GPU (reference).]
Training the scheduler

• **N-Body**

![N-Body Timestep Simulation](image)
Main challenges

• Difficult to write composable parallel software
  - Possible with task-centric models
  - Also for heterogeneous systems
• The parallel models of different languages do not work well together
  - Can be overcome with a unified task model
• Poor resource management
  - Can be overcome with task schedulers talking to the OS

OpenMP 4.0
Acknowledgments

- Karl-Filip Faxén
- Ananya Muddukrishna
- Artur Podobas
- Konstantin Popov
- Georgios Varisteas
- Vladimir Vlassov

- The Encore project team, www.ence-re-project.eu

- The PaPP project team, www.papp-project.eu
Thank you for your attention!

• Mats Brorsson, matsbror@kth.se
• Tel. 08-790 4121

• “Threads are evil”: 97 500 hits on Google search

• “Tasks are good”: 200 000 hits on Google search