Resource Centered Computation with Ordered Read-Write Locks

Jens Gustedt
INRIA & ICube
Strasbourg, France

Stéphane Vialle
SUPELEC
Metz, France

collaboration with:
Pierre-Nicolas Clauss, Emmanuel Jeanvoine

INRIA project lab MULTICORE
Large scale multicore virtualization for performance scaling and portability
Outline

1. Iterative algorithms
2. Resource centered computing
3. An adaptative tool for resource control
4. Experiments
5. Conclusions
### Properties
- Discrete data space
- Local computation
- Iterative computation
- Out-of-core computation

### Design
- Blocks of data
- Local addressing
- Common loop
- Synchronize comm and comp

### LINPACK example: Livermore Kernel 23
Local view of an iterative task

Block of a Matrix
Local view of an iterative task
Local view of an iterative task
Local view of an iterative task

Waiting
Local view of an iterative task
Local view of an iterative task

Waiting
Local view of an iterative task
Local view of an iterative task
Local view of an iterative task

Waiting
Local view of an iterative task
Local view of an iterative task
Design a control and communication tool

1. Locks with read-write (inclusive-exclusive) semantics
2. A predictable scheduling semantic:
   - avoid deadlocks
   - progress uniformly
   - control operation order
3. Control overhead shouldn’t dominate resource utilization

- liveness
- equity
- reproducibility
- efficiency
Outline

1. Iterative algorithms
2. Resource centered computing
3. An adaptative tool for resource control
4. Experiments
5. Conclusions
# Resource centered computing

## Everything is resource

- **data**: input, output, temporaries
- **hardware**: CPU, memory (L1, L2, RAM), GPU, communication links
- **software**: specialized functions, data transformations

## Desired properties for each operation

- **feasibility**: resources are available
- **consistency**: resources are in defined states
- **performance**: computations don’t step on each other

## Access to resources is regulated through a FIFO

- **dead lock free**: check at compile time or startup
- **homogeneous**: all “operations” should get equal share
- **simple**: easy to use
Existing tools

- Designed for either parallel computing (threads, atomic ops) or distributed computing (MPI)
- Local copying between buffers (MPI)
- Separation of control and data (mutex)
- Modification order is scheduling dependent
- Lock order is either arbitrary or priority based (threads)
- Atomic operations are limited to word-sized data (or inefficient)
Software Stack

- Parallel and distributes applications
- Scientific computing libraries
- Directive-based parallel languages
- Low level parallel lang. & libs (Pthreads, CUDA...)
- ORWL: C & high level parallel extension & lib.
- ORWL runtime
- Runtime
- OS
- Distributed and heterogeneous resources
Outline

1. Iterative algorithms
2. Resource centered computing
3. An adaptative tool for resource control
4. Experiments
5. Conclusions
Properties

- FIFO-policy based waiting queue
- A distinction between request and acquire operations
- A distinction between locks (as opaque objects) and lock-handles (as user interfaces acting on locks).
- A distinction into exclusive or write locks and inclusive or read locks.

The typical sequence for an access is

request  acquire  release
**Task** is a *logical* unit of execution. It describes a set of computations that belong together from an application point of view, example:

*manipulation of a matrix block in one iteration step*

**Operation** is a specific computation that a task has to perform on a particular resource, examples:

- the computation that is to be performed on a block of the matrix
- the update to the boundary information that the task has to perform
- a collective operation to verify the quality of the result
ORWL: the resource model

Identifiable resources: unity of localization, data and control

Each resource in ORWL has

- **location**: a “primary” task and unique ID that identifies the resource:
  \[ \text{ORWL\_LOCATION}(\text{taskID}, \text{locationID}) \]

- **associated data**: a binary (untyped) data with a given size:
  The application controls size (\text{orwl\_scale}) and contents.

- **abstract**: There is no off-limits interface to control the data directly, only several competing “handles” to the same location.

Quantifiable resources: relaxed variant

We only need “one of many” for a computation: CPU, L1 cache, L2 cache, blocks of RAM
ORWL: schematic task view
An adaptative tool for resource control

ORWL: inner computation loop

```c
for (size_t orwl_phase = 0; orwl_phase < maxPhases; ++orwl_phase) {

    // computation operation
    ORWL_SECTION(&myBlockComp) {
        double* data = orwl_write_map(&myBlockComp);
        ORWL_SECTION(&lneighBound) {
            double const* lData = orwl_read_map(&lneighBound);
            // do the real computation here
            block_computation(n, data, m, lData);
        }
    }

    // update operation
    ORWL_SECTION(&myBlockUpd) {
        double const* data = orwl_read_map(&myBlockUpd);
        ORWL_SECTION(&myBound) {
            double* bData = orwl_write_map(&myBound);
            update_boundary(m, bdata, n, data);
        }
    }
}
```

Resource Centered Computation with Ordered Read-Write Locks

Jens Gustedt
An adaptative tool for resource control

ORWL: properties

Model for iterative computation
- deadlock-free
- homogeneous progression of tasks

Implementation
- transparent use on multi-core or cluster
- build on top of the C11 thread model
- type-generic interfaces
- OpenMP compatible
- CUDA compatible
Outline

1. Iterative algorithms
2. Resource centered computing
3. An adaptative tool for resource control
4. Experiments
5. Conclusions
Dense matrix multiplication

A common framework implemented with ORWL, block-cyclic MM.

Three compute kernels, work seamlessly together:

- Hand crafted legacy code
- BLAS/ATLAS `dgemm` optimized for the target architecture
- CUBLAS for GPU computations
Experimental Setting

**pastel cluster Grid5000 platform**
- up to 60 processors, 180 cores

**lemans multicore ICube lab**
- 24 cores at 800 MHz

**cameron cluster SUPÉLEC Metz**
- 16 nodes, each 6 cores at 3.2 GHz and 8 GiB of memory
- per node 12 cores hyperthreaded, but only 6 L2 caches
- per node 1 NVIDIA GeForce GTX580 with 512 CUDA cores and 1.5 GiB
- 10 Gigabit Ethernet interconnection network
lemans, 24 core

Experiments

Resource Centered Computation with Ordered Read-Write Locks

Jens Gustedt
Experiments

Resource Centered Computation with Ordered Read-Write Locks

Jens Gustedt

17/21

cameron, constant sized problem

![Graph showing performance comparison]

- **Ideal perf**
- **MPI+OpenMP**
- **ORWL**

- **Gflop/s** on the y-axis
- **Number of nodes** on the x-axis
Resource Centered Computation with Ordered Read-Write Locks

Cameron, maximum sized problem

Experiments

Gflop/s

Ideal perf - GPU
MPI+OpenMP - GPU
ORWL - GPU
Ideal perf - CPU
MPI+OpenMP - CPU
ORWL - CPU

Number of nodes

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Experiments

pastel, up to 60 processors 180 cores

Resource Centered Computation with Ordered Read-Write Locks

Jens Gustedt
Outline

1. Iterative algorithms
2. Resource centered computing
3. An adaptative tool for resource control
4. Experiments
5. Conclusions
A new Synchronization Tool

Ordered Read-Write Locks

- simple usage for critical sections
- proactive announcement of requirements
- alternating resource allocation in iterative computations
- provably deadlock free
- offline copy between remote hosts
- zero copy between threads
- almost perfect computation/communication overlap
- weak scaling

Questions?  jens.gustedt@inria.fr
Conclusions

A new Synchronization Tool

Ordered Read-Write Locks

- simple usage for critical sections
- proactive announcement of requirements
- alternating resource allocation in iterative computations
- provably deadlock free
- offline copy between remote hosts
- zero copy between threads
- almost perfect computation/communication overlap
- weak scaling

Questions?  jens.gustedt@inria.fr
Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)

FIFO

resource locations $L_0 \ L_1 \ L_2 \ L_3 \ L_4$
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
**Conclusions**

**Supplement: Execution of iterative tasks**

**Evolution rule**

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. **Acquire all requests (current iteration)**
2. **Post new requests (next iteration)**
3. **Compute**
4. **Release requests (current iteration)**
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Conclusions

Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Supplement: Execution of iterative tasks

Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)
Evolution rule

1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)