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

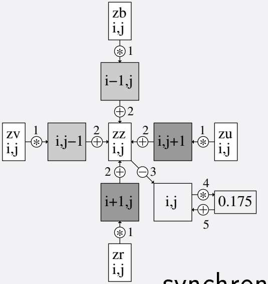
Outline

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

Iterative application

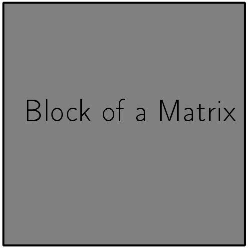
Properties — Design

- Discrete data space — blocks of data
- Local computation — local addressing
- Iterative computation — common loop
- Out-of-core computation — synchronize comm and comp

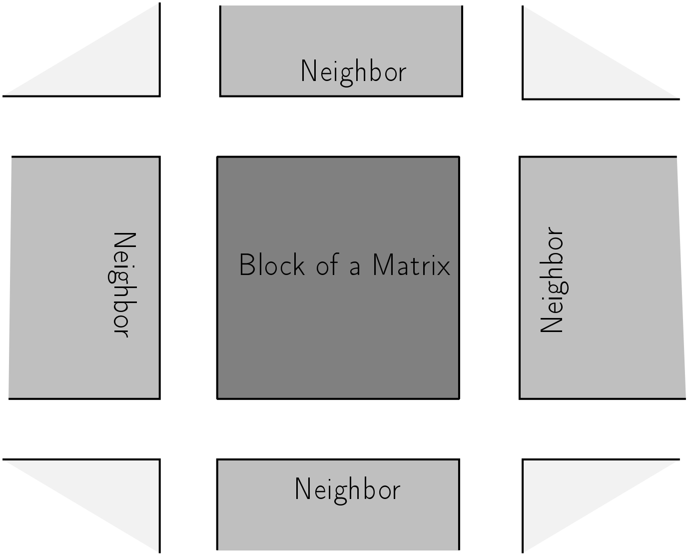


LINPACK example: Livermore Kernel 23

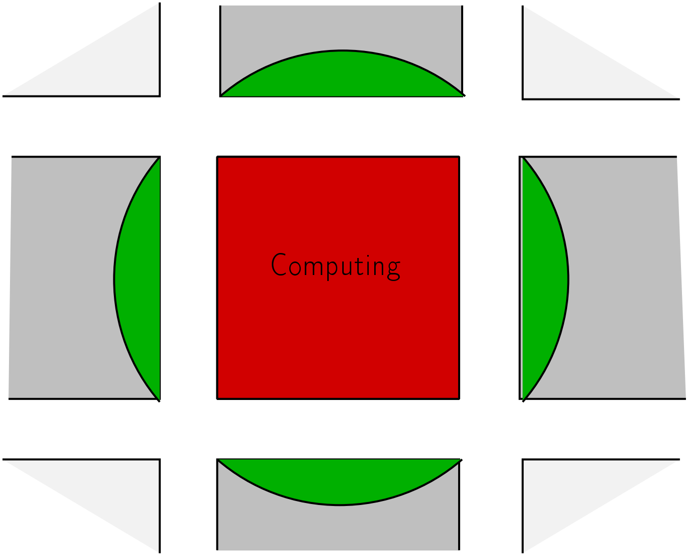
Local view of an iterative task



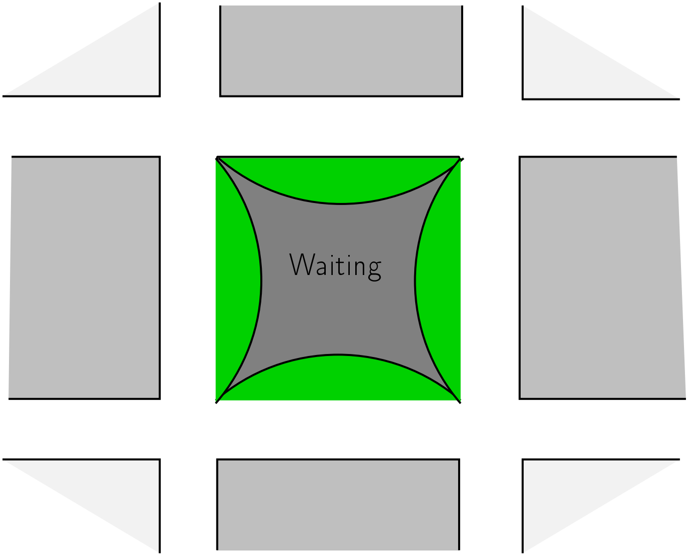
Local view of an iterative task



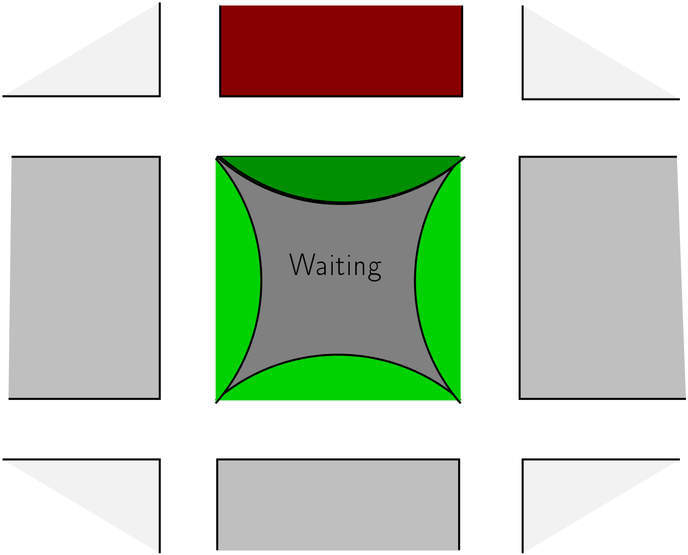
Local view of an iterative task



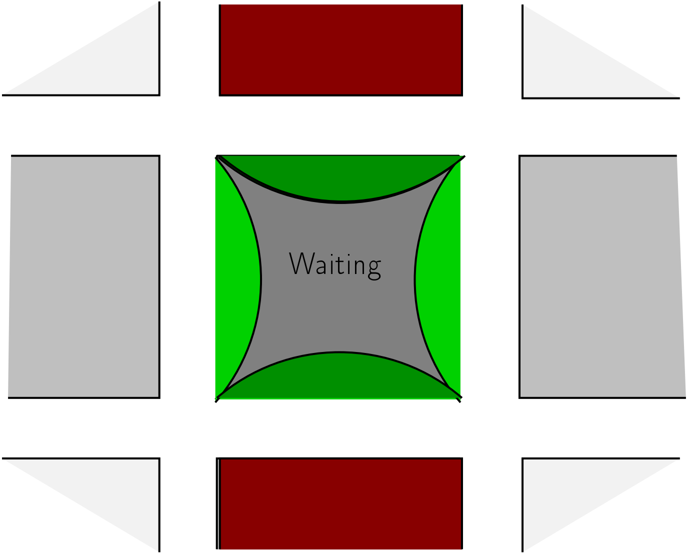
Local view of an iterative task



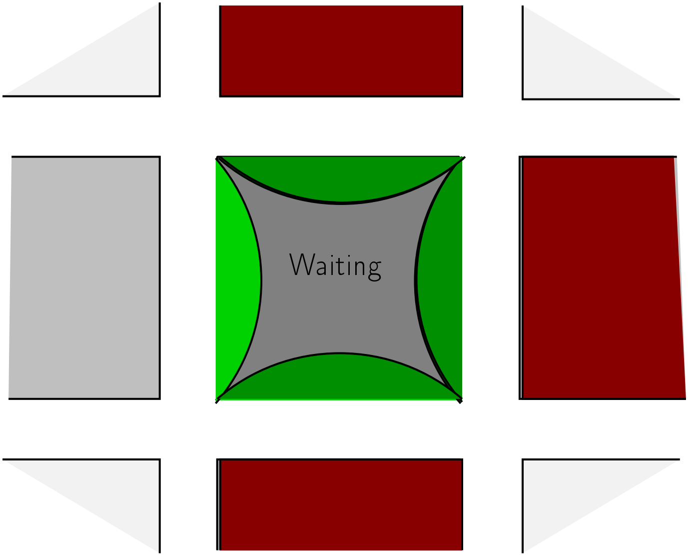
Local view of an iterative task



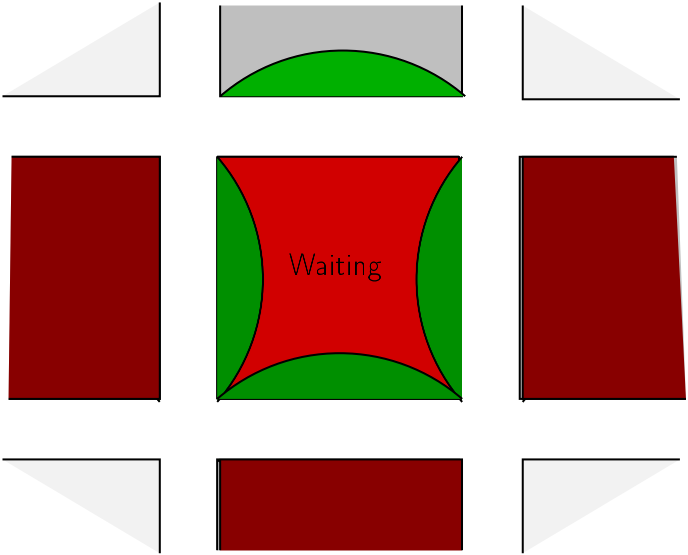
Local view of an iterative task



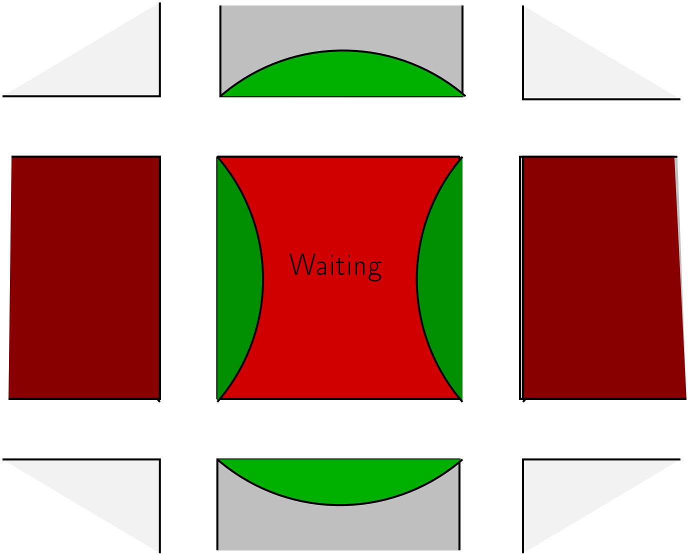
Local view of an iterative task



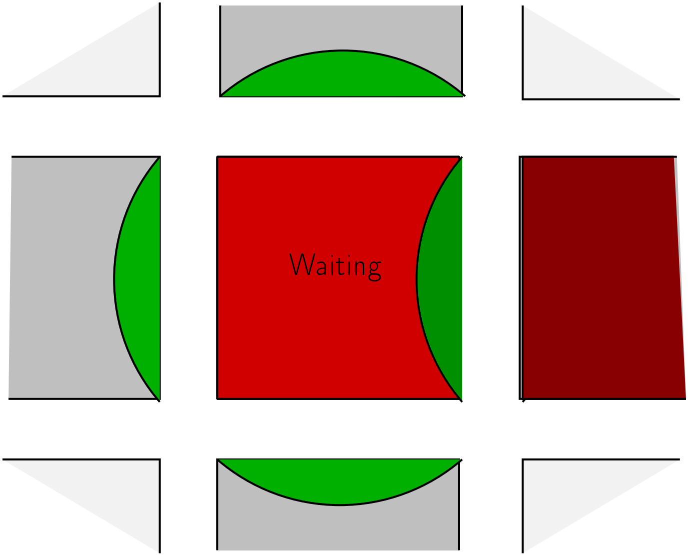
Local view of an iterative task



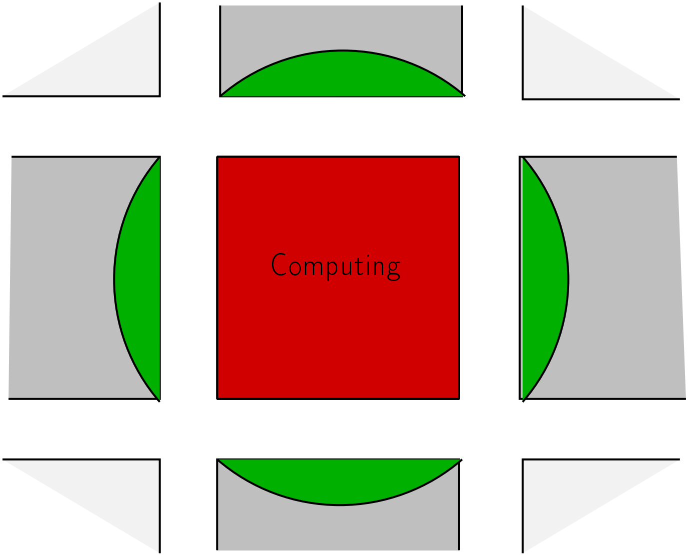
Local view of an iterative task



Local view of an iterative task



Local view of an iterative task



Synchronization requirements

Design a control and communication tool

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

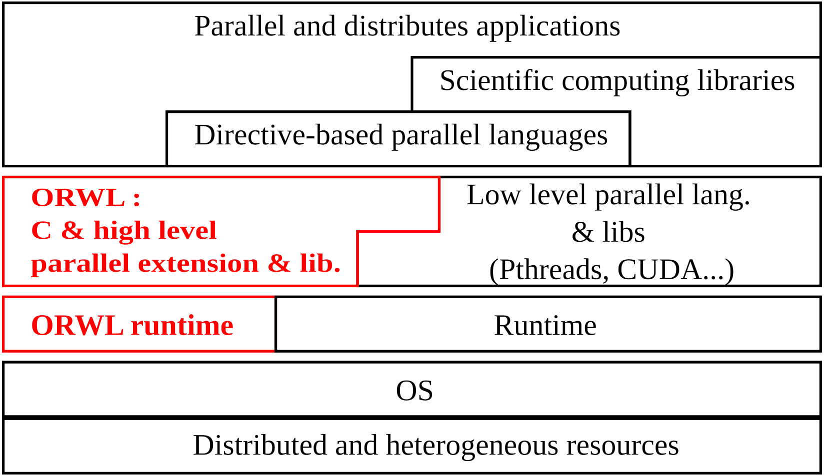
Access to resources is regulated through a FIFO

- 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



Outline

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

ORWL, Ordered Read-Write Locks

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*

ORWL: the task model

decomposition of an execution

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:

ORWL_LOCATION(taskID, locationID)

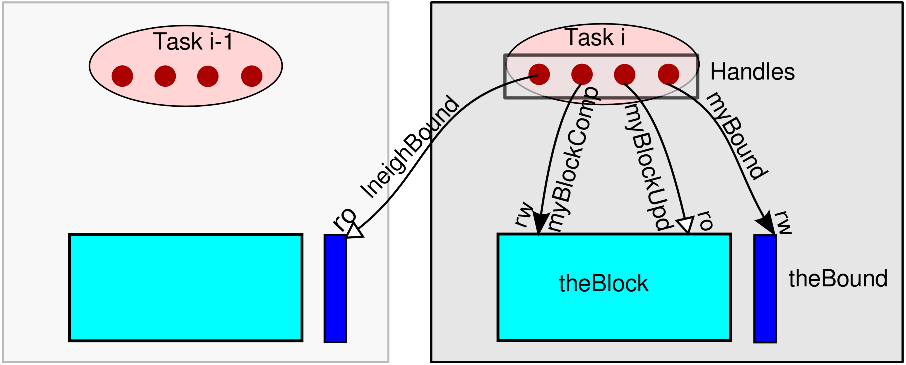
associated data
: a binary (untyped) *data* with a given *size*:
The application controls size (orwl_scale) and contents.

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



ORWL: inner computation loop

```
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);
    }
  }
}
```

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

Experimental Setting

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

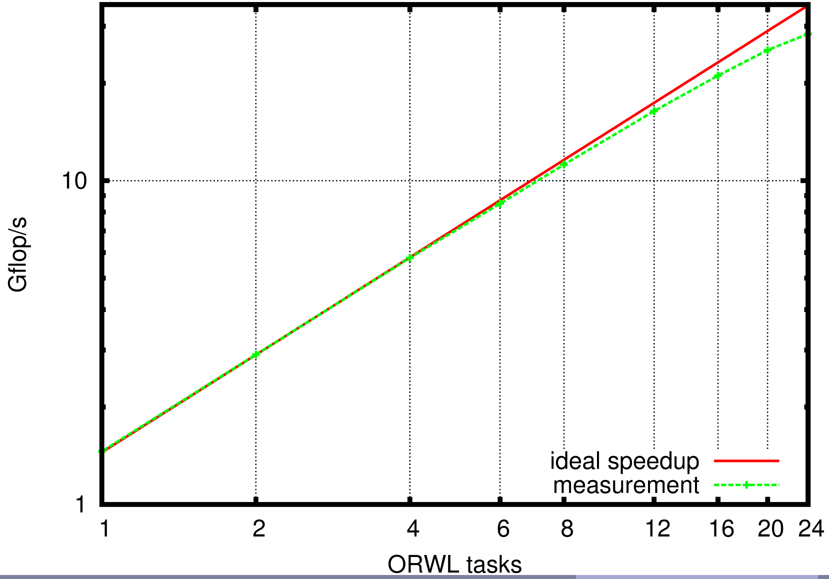
lemans multicore ICube lab

- 24 cores at 800 MHz

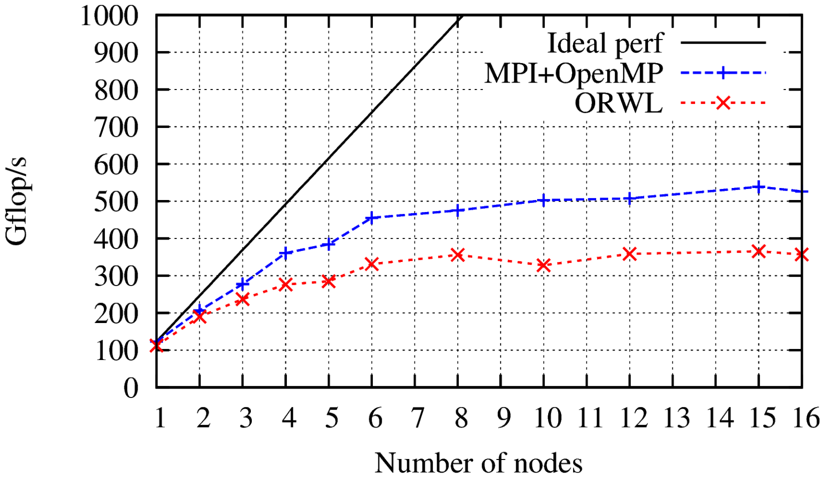
cameron cluster SUPÉLEC Metz

- 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

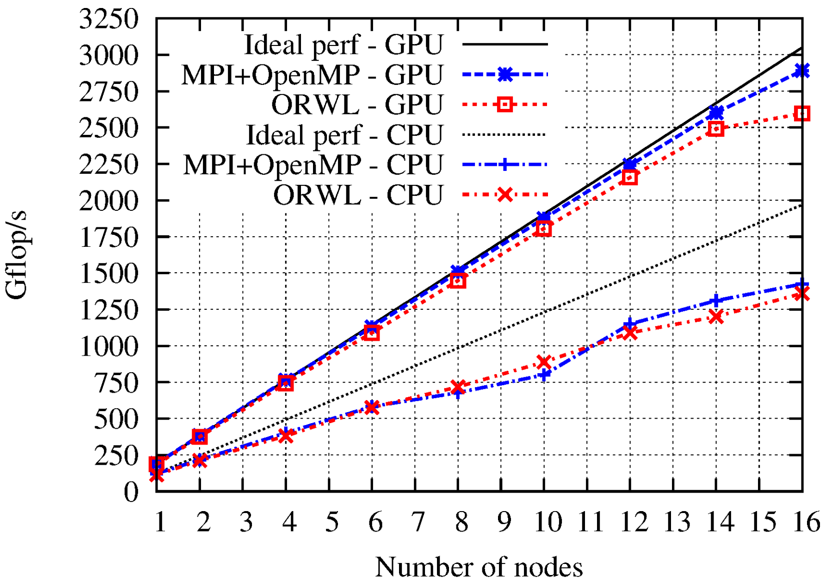
lemans, 24 core



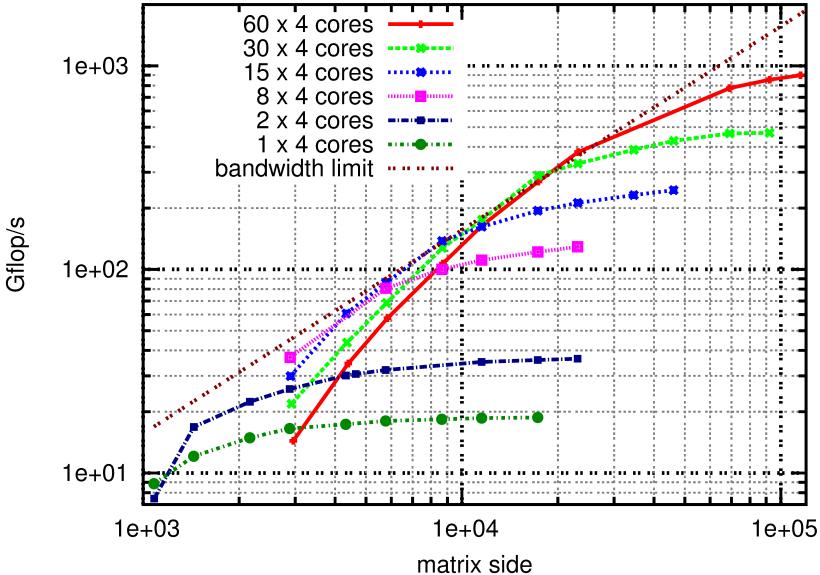
cameron, constant sized problem



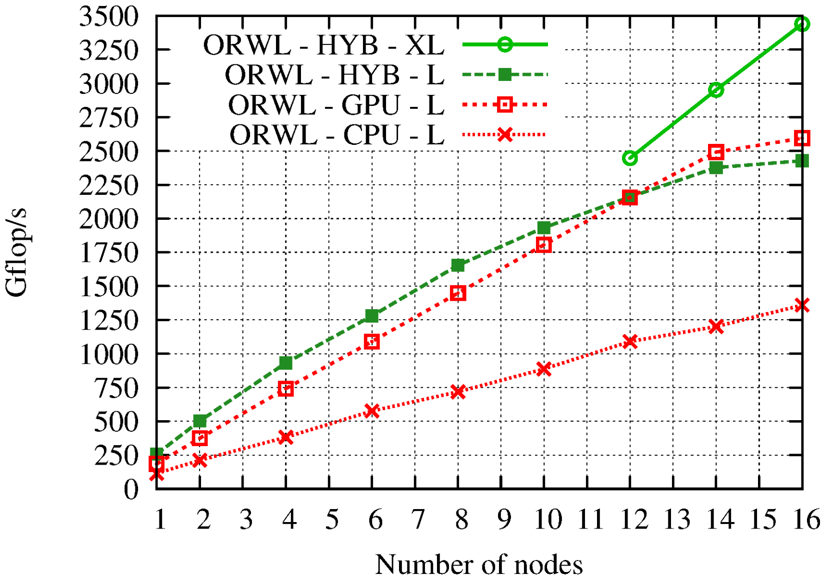
cameron, maximum sized problem



pastel, up to 60 processors 180 cores



cameron, maximum size problem, hybrid, including GPU



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

A new Synchronization Tool

Ordered Read-Write Locks

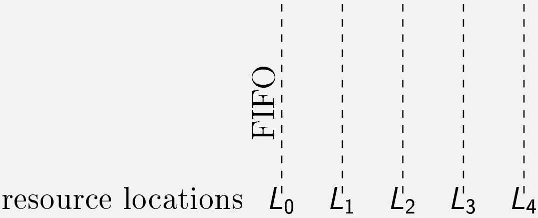
- simple usage for critical sections
- proactive announcement of requirements
- alternating resource allocation in iterative computations
- provably deadlock free
- offline copy between remote hosts
- zero copy between threads
- almost perfect computation/communication overlap
- weak scaling

Questions? jens.gustedt@inria.fr

Supplement: Execution of iterative tasks

Evolution rule

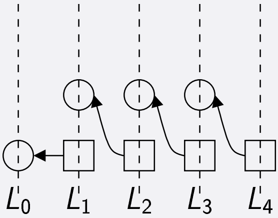
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

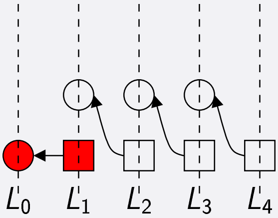
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

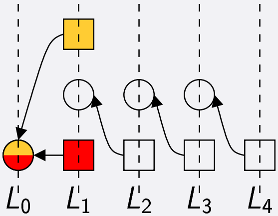
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

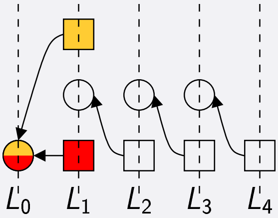
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

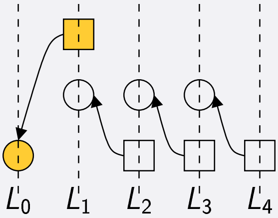
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

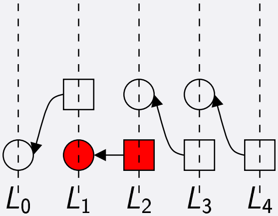
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

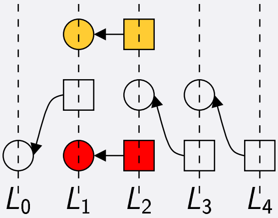
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

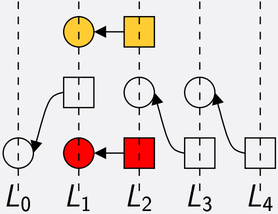
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

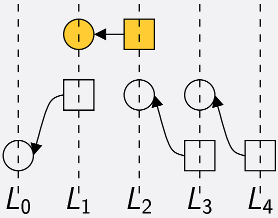
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

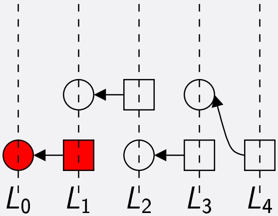
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

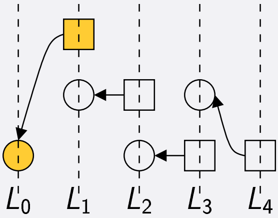
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

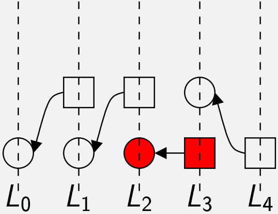
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

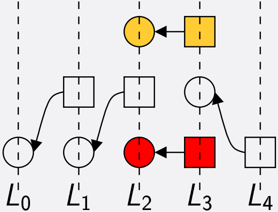
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

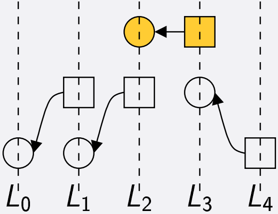
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

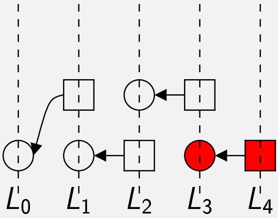
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

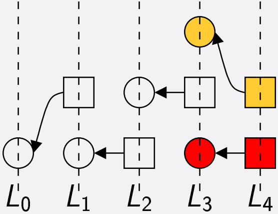
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



Supplement: Execution of iterative tasks

Evolution rule

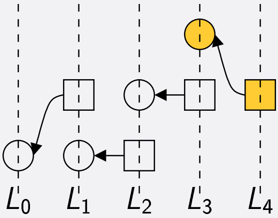
1. Acquire all requests (current iteration)
2. Post new requests (next iteration)
3. Compute
4. Release requests (current iteration)



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)