# Parallel Optimizations

## Advanced Constructs and Compiler Optimizations for a Parallel, Object Oriented, Shared Memory Language running on a distributed System.

*Claudio Fleiner*

*cfl@zurich.ibm.com*

http://www.icsi.berkeley.edu/~fleiner/thesis

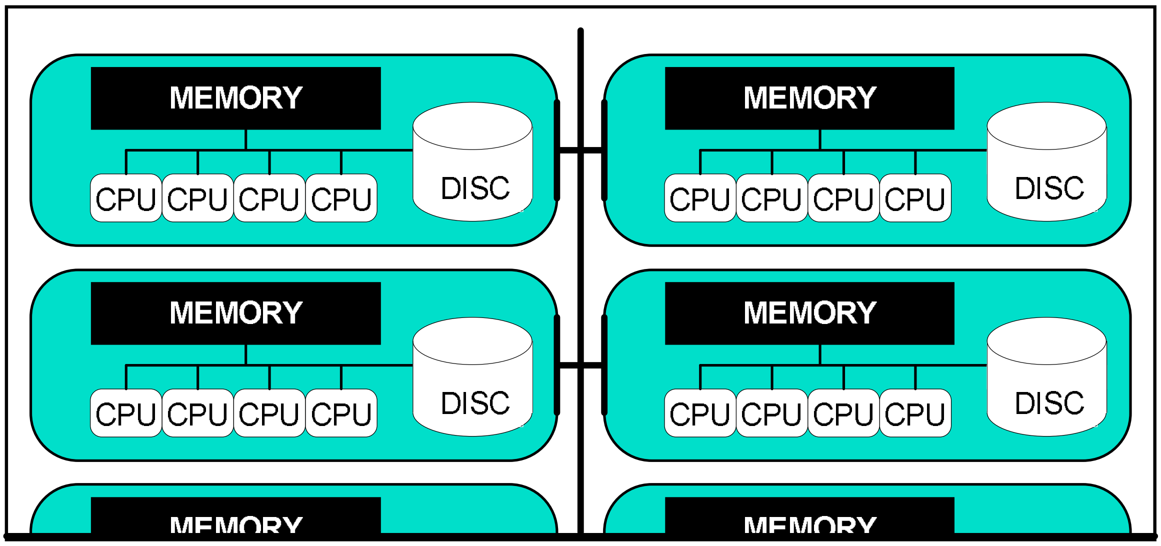
# Agenda

- Background
  - » Distributed Computer System
  - » pSather / Sather
- Language Design
  - » Advanced Lock Construct
- Compiler Implementation
  - » Memory Consistency
    - – Serial Consistency
    - – Release Consistency
  - » Optimizations
  - » Results
- Conclusions

# Distributed Computer System

The programs discussed in this thesis run on a distributed computer system with:

- One or more CPU's per node.
- A high speed network between nodes (round-trip < 30$\mu$sec).
- Optional Harddisc.



# Sather/pSather 1.1

***Sather** is similar to Java with some additional features.*

***pSather** adds support for parallel and distributed execution:*

- Data distribution:
  *Presents shared memory model to the programmer, even on distributed architectures.*
- Synchronization:
  *pSather offers a flexible disjunctive lock statement.*

# The Sather Group 95/96

- **Prof. Jerôme Feldman**
- Ben Gomes
  browser / TK interface / libraries
- David Stoutamire
  serial Compiler / parallel garbage collection
- Michael Philippsen
  pSather front end
- Boris Weissman
  serial Compiler / C & Fortran interface / libraries / new solaris thread library
- Claudio Fleiner
  pSather backend / pSather runtime (Meiko / Solaris Myrinet) / optimizations

# Disjunctive Lock
## *Example*

Get an element from `queue` until it is empty and all producers died:

```
loop
  lock
    when queue.not_empty then
      a:=queue.dequeue;
    when prod.no_threads,
         queue.empty then
           return;
  end;
  work_on(a);
end;
```

# Disjunctive Lock
## *Reason*

- Numerous times it would have been necessary to let a thread wait on either one of several locks, which was not possible in earlier versions of pSather.
- The implementation of migrating threads that can be interrupted is a nightmare.
- Killing or interrupting Threads is often not an option, but it must be possible to interrupt a thread waiting on a lock.
- The original pSather offered some conditional locks (for example `queue.not_empty()`), the new lock implementation should allow the user to define new lock types (i.e. Barrierlocks, Rendezvous locks, ...).

# Disjunctive Lock
## *Syntax*

```
lock lck1, lck2, .. then
    statements;
[else
    statement_else]
end;


lock
[guard cond] when lck1, .. then
    statement1;
[guard cond] when lck2, .. then
    statement2;
    .
    .
[else
    statement_else]
end;
```

# Disjunctive Lock
## *Properties*

- Deadlock free

  *All locks of a branch are locked atomically to avoid deadlocks of two threads trying to lock the same two locks at the same time.*

- Guaranteed Unlock

  *The system will unlock the locks as soon as the thread leaves the lock block, regardless if this happens because of a* `return`*, a* `raise` *or a* `quit`*.*

- Fairness

  *No starvation of threads or branches.*

# Memory Consistency Model

*defines the precise semantics of memory reads and writes:*

- the programmer needs it to reason about the correctness of programs,
- the compiler needs it to use the correct optimizations,
- the runtime needs it to implement the correct semantics.

# Sequential Memory Consistency Model

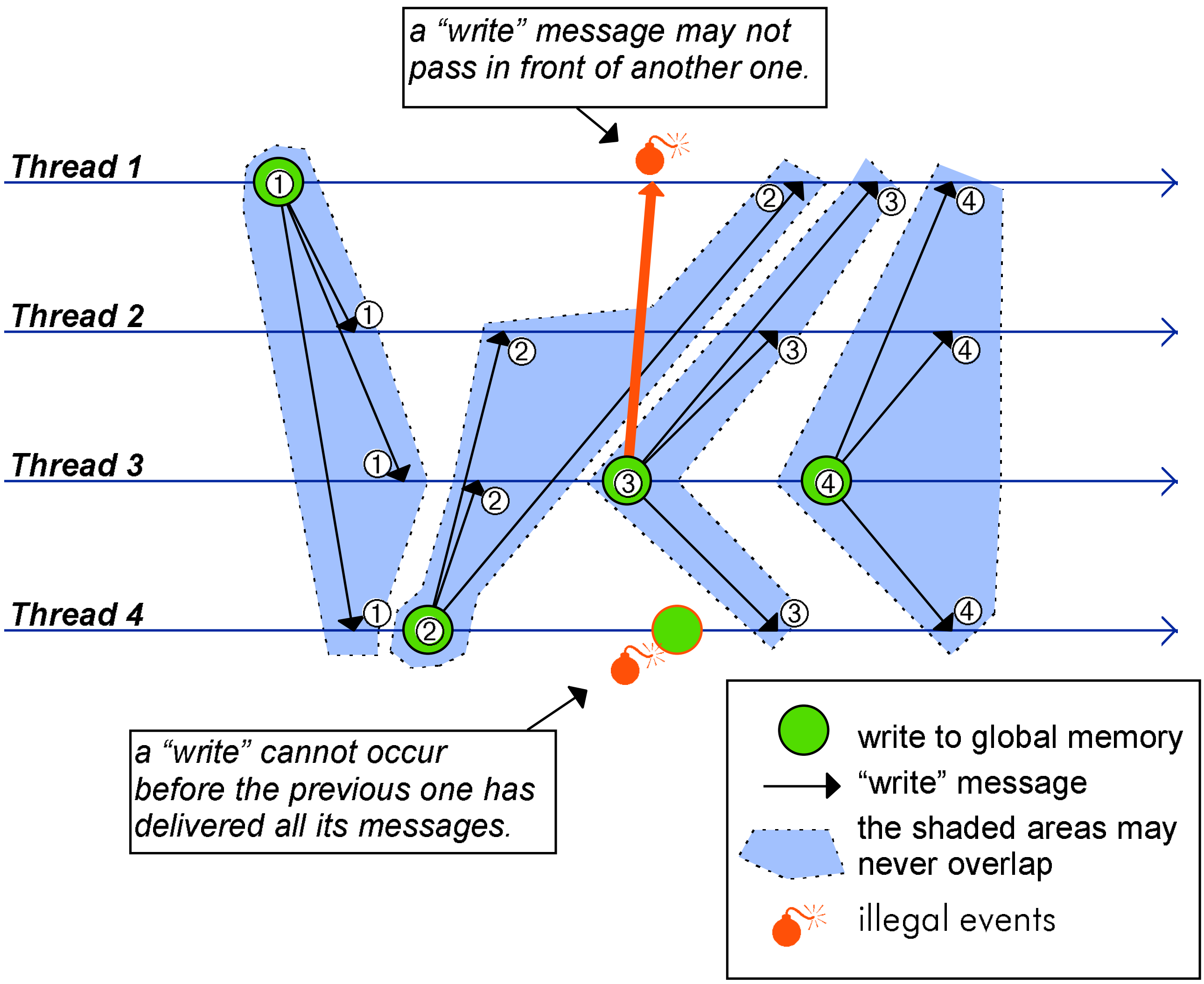
Lamport:

*A system is sequentially consistent if the result of execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

$\Leftrightarrow$

All threads observe all writes in the same order.

# Sequential Consistency
# *Example*



# Sequential Consistency
## *Problem*

- The following two expressions are NOT equivalent:
  `res1:=obj.a+obj.b+obj.a;`
  `res2:=2*obj.a+obj.b;`
- Suppose that `obj.a==2;obj.b==3;` and another thread executes
  `obj.a:=5; obj.b:=7;`
- Possible results:

```
res1:=obj.a+obj.b+obj.a; →  7
res1:=obj.a+obj.b+obj.a; → 10 ←
res1:=obj.a+obj.b+obj.a; → 14 ←
res1:=obj.a+obj.b+obj.a; → 17
res2:=2*obj.a+obj.b; →  7
res2:=2*obj.a+obj.b; → 11     ←
res2:=2*obj.a+obj.b; → 17
```

# Sequential Consistency
# *Conclusion*

## ***Advantages***:

- Intuitive to use, as it is the “expected” behavior and also true for sequential programs.
- Correctness proofs are relatively easy, as this model is more deterministic than weaker models.

## ***Disadvantages***:

- Many optimizations are no longer possible.
- Hard to implement without a significant efficiency loss, especially on a distributed system.

# Release Memory Consistency

We define two memory operations:

**export:**

*other threads may "see" writes to the global memory only after an export.*

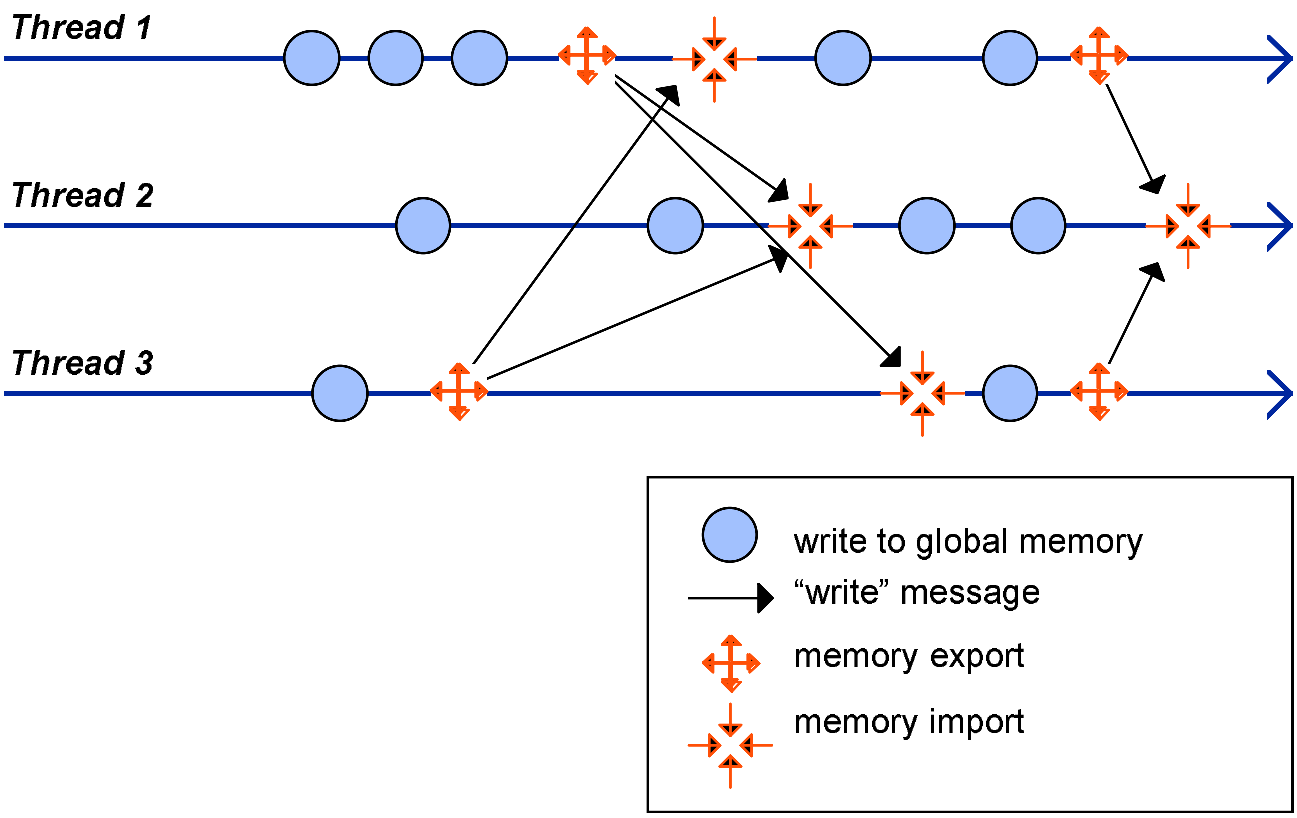
- Sparc: flush store buffers
- pSather: wait for write acknowledge

**import:**

*a thread may "see" writes exported by other threads only after an import.*

- pSather: flush import cash

# Release Consistency *Example*



# Release Consistency
# *implicit import/export*

Most, if not all, explicit imports and exports are unnecessary in pSather, as implicit imports and exports are done whenever the lock construct is used:

```
lock l1, .. ln then
        import; -- implicit
        ....
        export; -- implicit
end;
```

With the exception of some special library functions the import/export statements have never been used in practice.

# Release Consistency
# *Conclusion*

## *Advantages:*

- Same results as serial consistency, if all shared memory accesses are properly enclosed in lock statements.
- Higher efficiency: many standard serial optimizations are possible between *imports* and *exports*.

## *Disadvantages:*

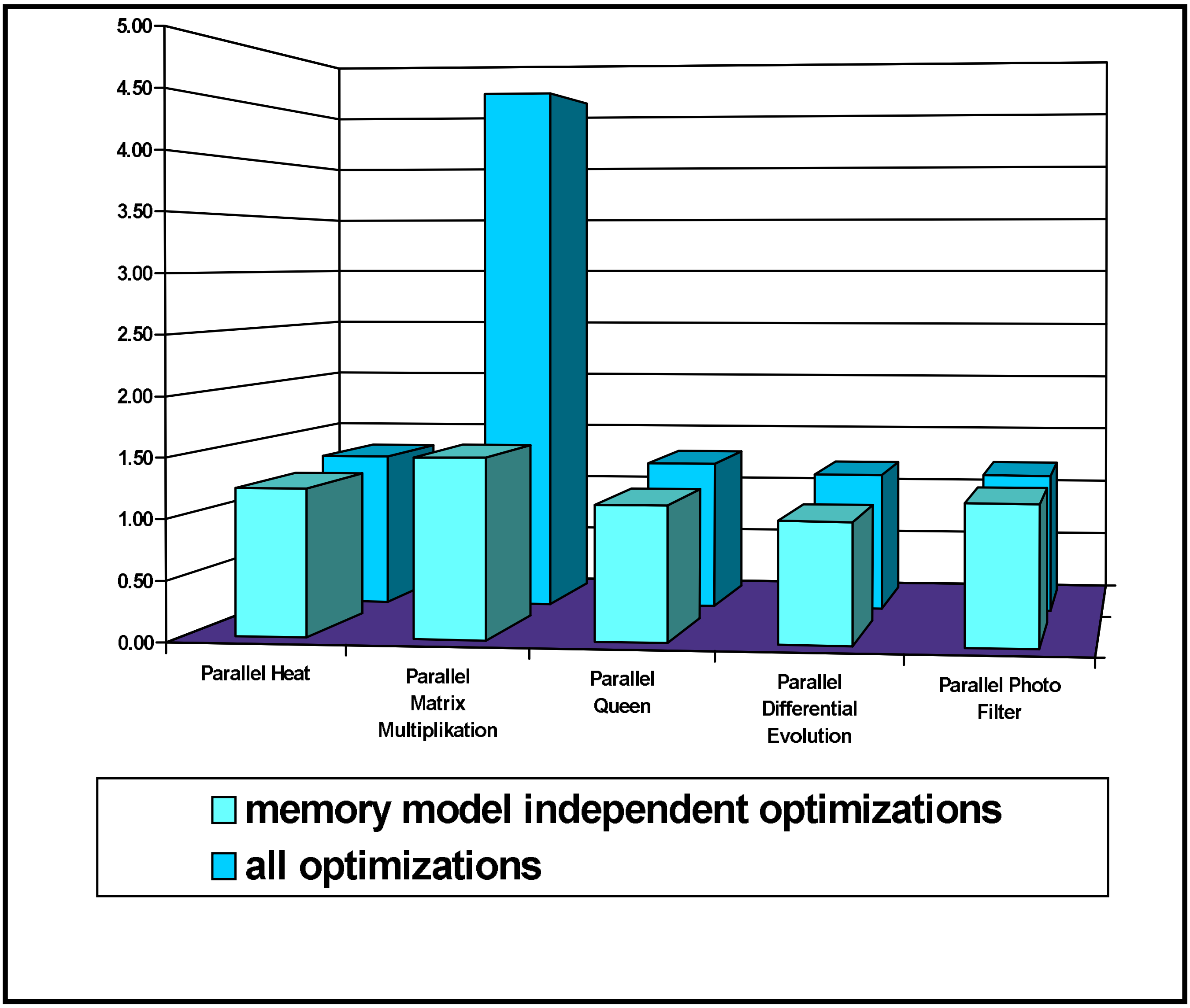
- Not as easy to use as the serial consistent memory model for programming and correctness proofs.
- Adds even more indeterministic behavior to parallel programs.

# Serial Optimizations

- **Independent of Memory Model:**
  Those are optimization that do not change the order of evaluation of expressions:
  - » Loop Unrolling,
  - » Inlining.

- **Only possible in some Memory Models:**
  Optimizations that change the order of evaluation of expressions depend on the memory model:
  - » Loop Invariant Hoisting,
  - » Iterator Initialization Hoisting,
  - » Common Subexpression Elimination.

# Serial Optimizations
# *Results*




Speedup for Serial Optimizations

# Parallel Optimizations

- ***Prefetching***

  reduce network latency by requesting values before they are needed.

  ***Problems:***

  - network / computers may be clogged with prefetch messages.
  - should speculative prefetch be used?

- ***Postwriting***

  reduce network latency by sending store messages as early as possible, but wait for acknowledge only when executing an **`export`**.

  ***Problem:***

  - when the same location is changed twice, two messages are sent out which must be received in the same order by the other node.

# Parallel Optimizations

- ***Loop Fusion***

  reduce thread creation / switching time by using only as many threads as available processors to execute a parallel loop.

  ***Problems***

  - works only if there is no possibility for a thread to block during the execution of the loop.

```
parloop i::=1.upto!(10);
do
    b[i]:=a[i]+a[i+1];
    sync;
    a[i]:=b[i];
end;
```

# Parallel Optimizations

- ***Specialization for Locality***
  - Avoid the test the check if some object is located in the local memory if this is already known.
  - Create different versions of the same function. Depending on the locality of the arguments the correct one is executed.

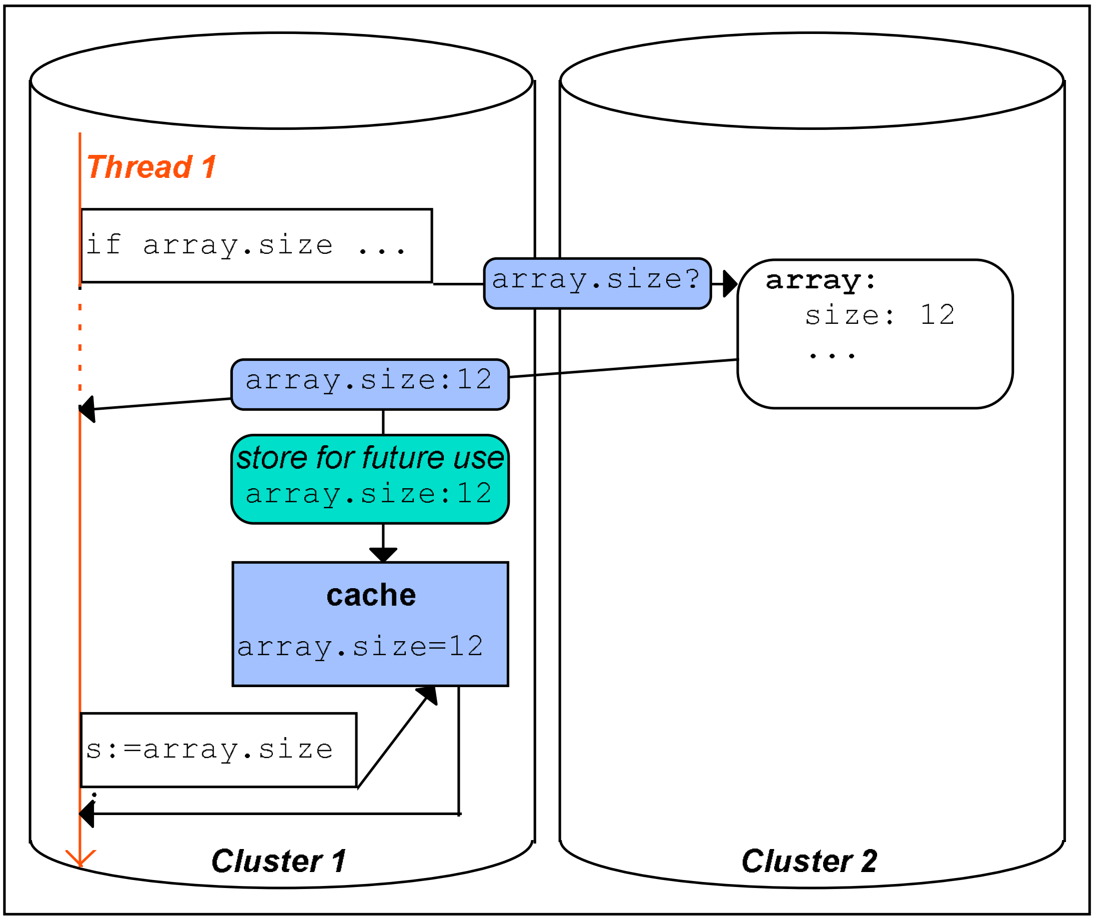
- ***Remote Execution***

  If the compiler can show that a function should be executed where the data is stored, it will be executed remotely.

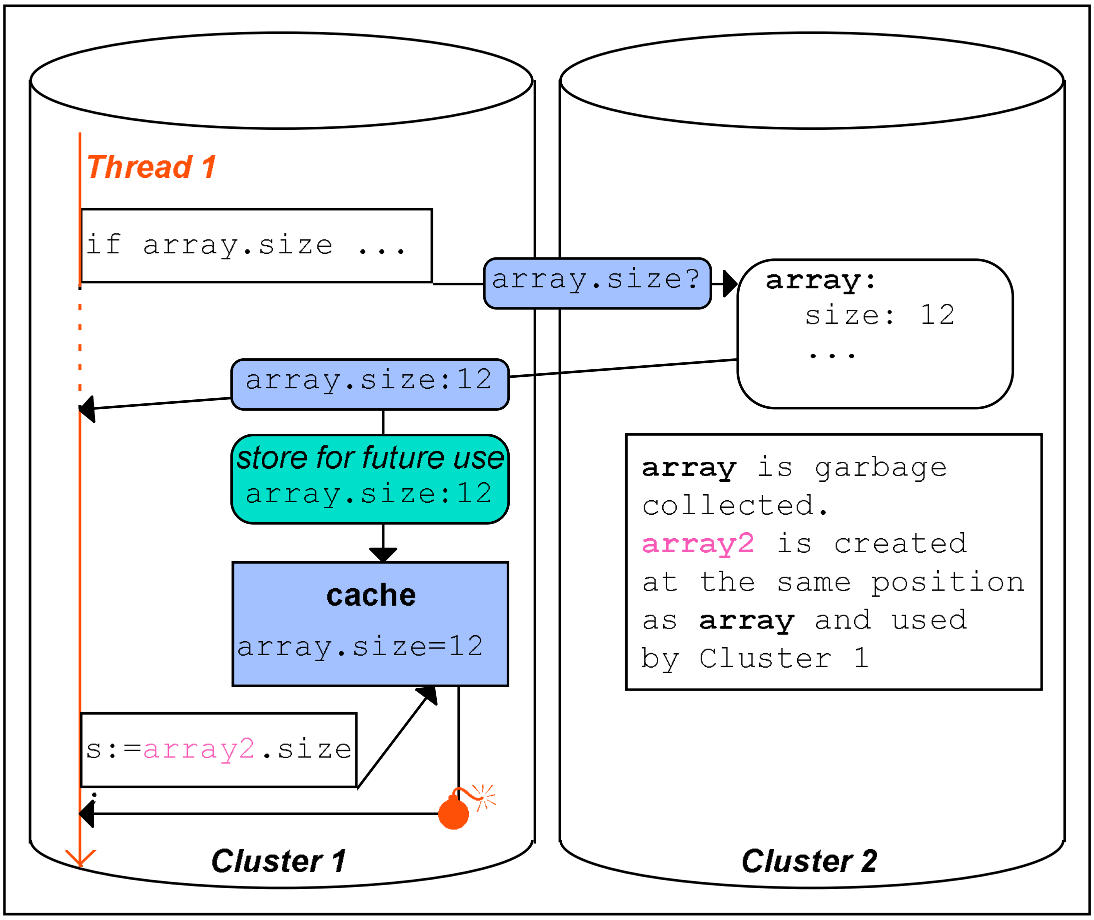
  ***Problems:***
  - caching may make a remote execution unnecessary.
  - the fact that some code is executed remotely must be undetectable.

# Caching

Whenever a node reads some value from a remote cluster, it can reuse this value until one of its threads executes an import.

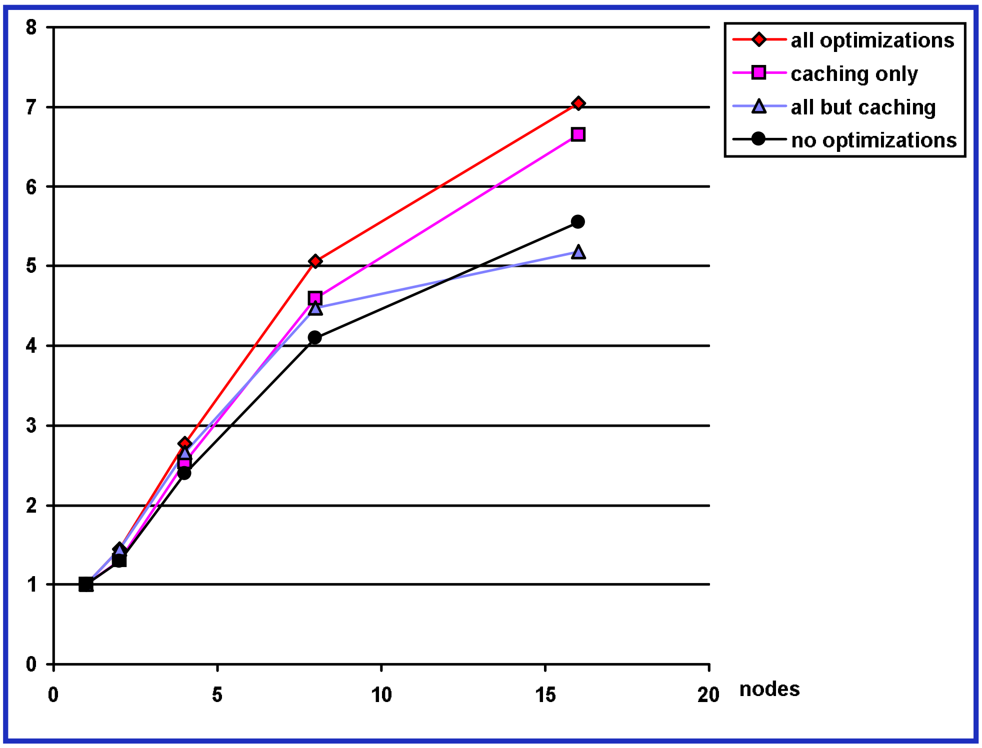


# Cache
# *Problems*




*Solution:*
the cache must be cleared during Garbage Collection!

# Caching
## *Result*



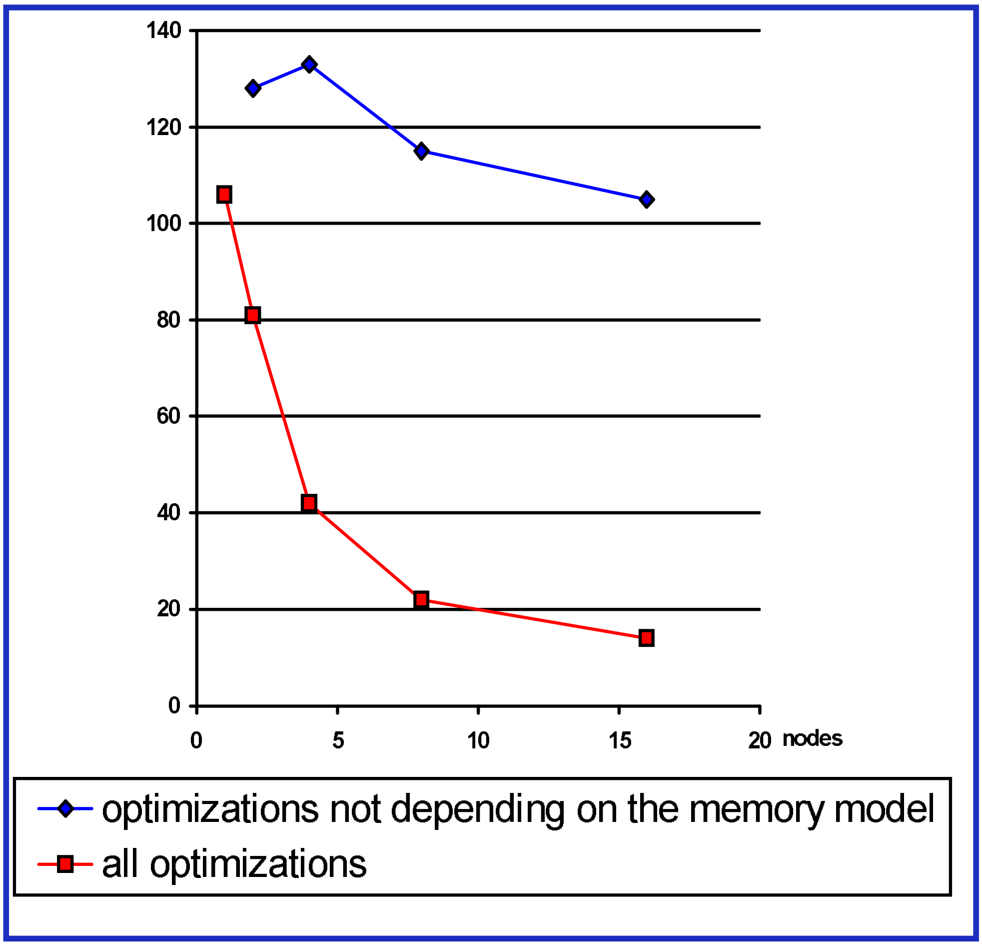

$$p_{x,y} = \frac{1}{d} \sum_{i=0}^{M_w} \sum_{j=0}^{M_h} M_{i,j} \, p_{x+i-\frac{M_w-1}{2},\, y+j-\frac{M_h-1}{2}}$$

$$M = \begin{pmatrix} -2 & -2 & -2 \\ -2 & 32 & -2 \\ -2 & -2 & -2 \end{pmatrix} \qquad d = 16$$

This charts shows how many times faster a photo filter program runs on a Meiko using 1 / 2 / 4 / 8 and 16 nodes.

# Overall Results

By using a weak memory model, the compiler is able to produce code that runs much faster than when using the serial consistent memory model:




*The Photo Retouching Program*

# Conclusion

- The disjunctive lock statement using synchronization objects that can be defined by the user leads to simple code and helps in reusing and managing parallel code.
- The strong serial consistency memory model simplifies programming, yet it prohibits many necessary compiler optimizations and cannot be used for a high speed parallel programming language.
- The release consistency model, even though it is a very weak consistency model, enables many crucial optimizations which are absolutely needed for high performance computing.

# Pointers / ongoing work

- **Sather/pSather on the WWW**
  http://www.icsi.berkeley.edu/~sather
  - » pSather: works on Meiko / Myrinet
- **Sather news groups**
  comp.lang.sather
- **ongoing work**
  - » Solaris Threads will be replaced (horribly slow switching times)
  - » parallel garbage collector
  - » better parallel libraries
  - » port to Myrinet on PC running Solaris