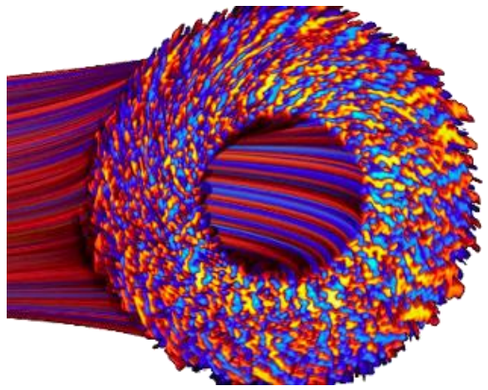
On the Impact of OpenMP Task Granularity

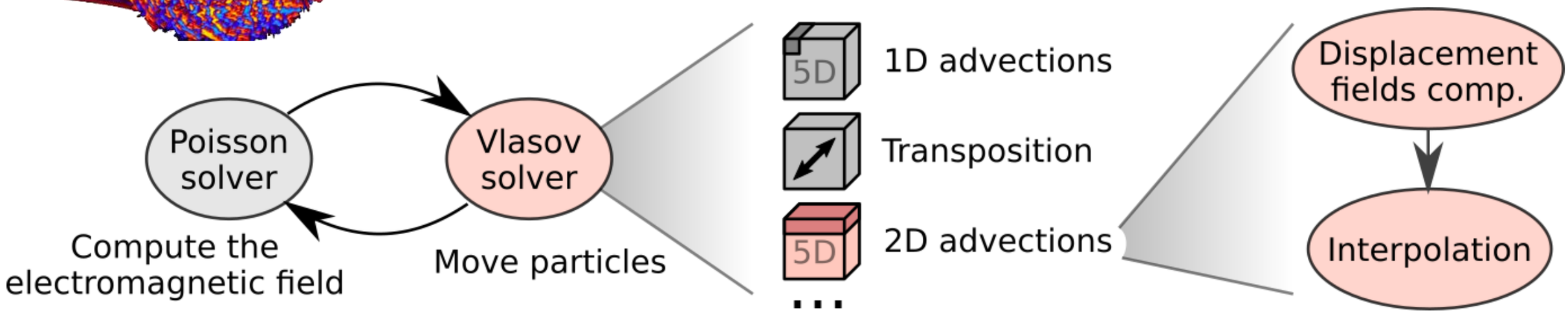
Thierry Gautier, Christian Perez, Jérôme Richard

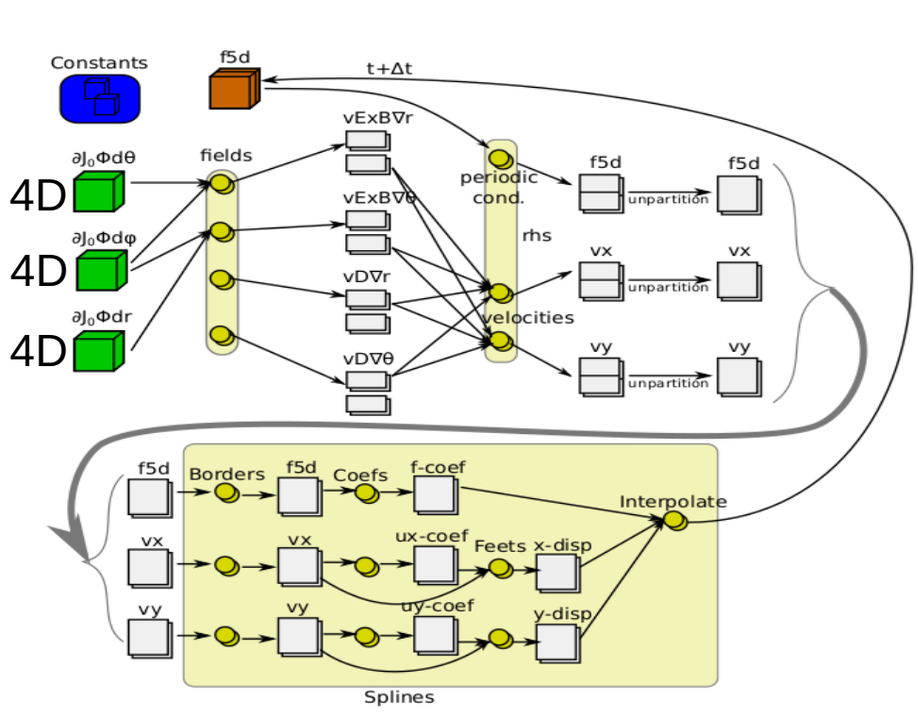
Most of this work has been published at IWOMP 2018, Barcelona, Spain.

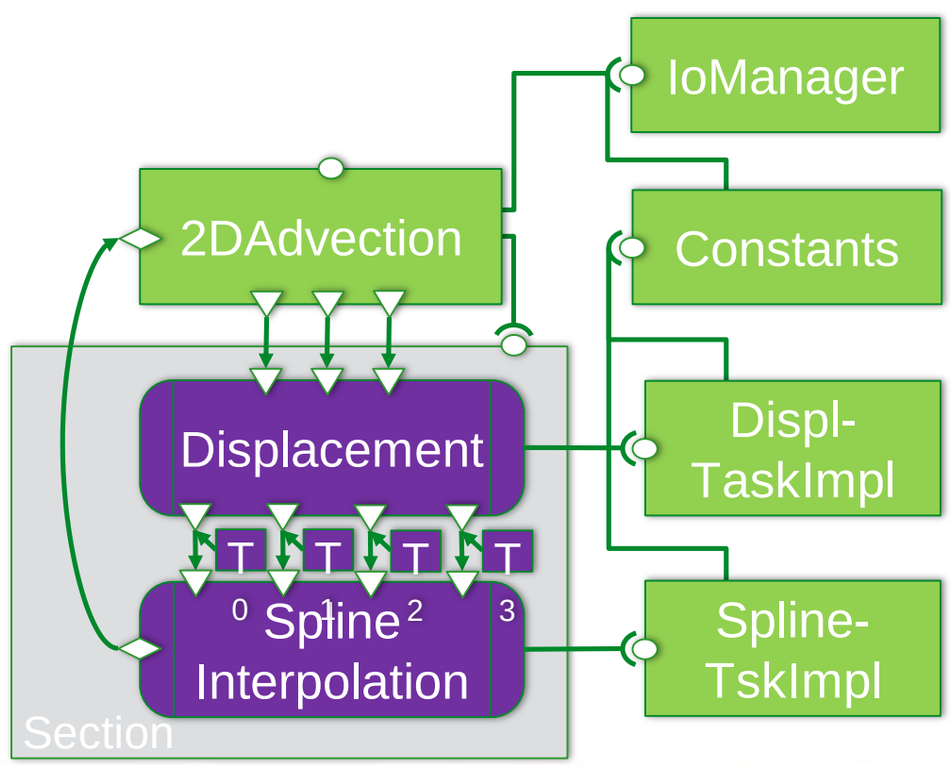
Gysela, 2D Advection, and COMET



Simulate the plasma of tokamaks
Developed by CEA, since 2000
~80,000 lines of FORTRAN





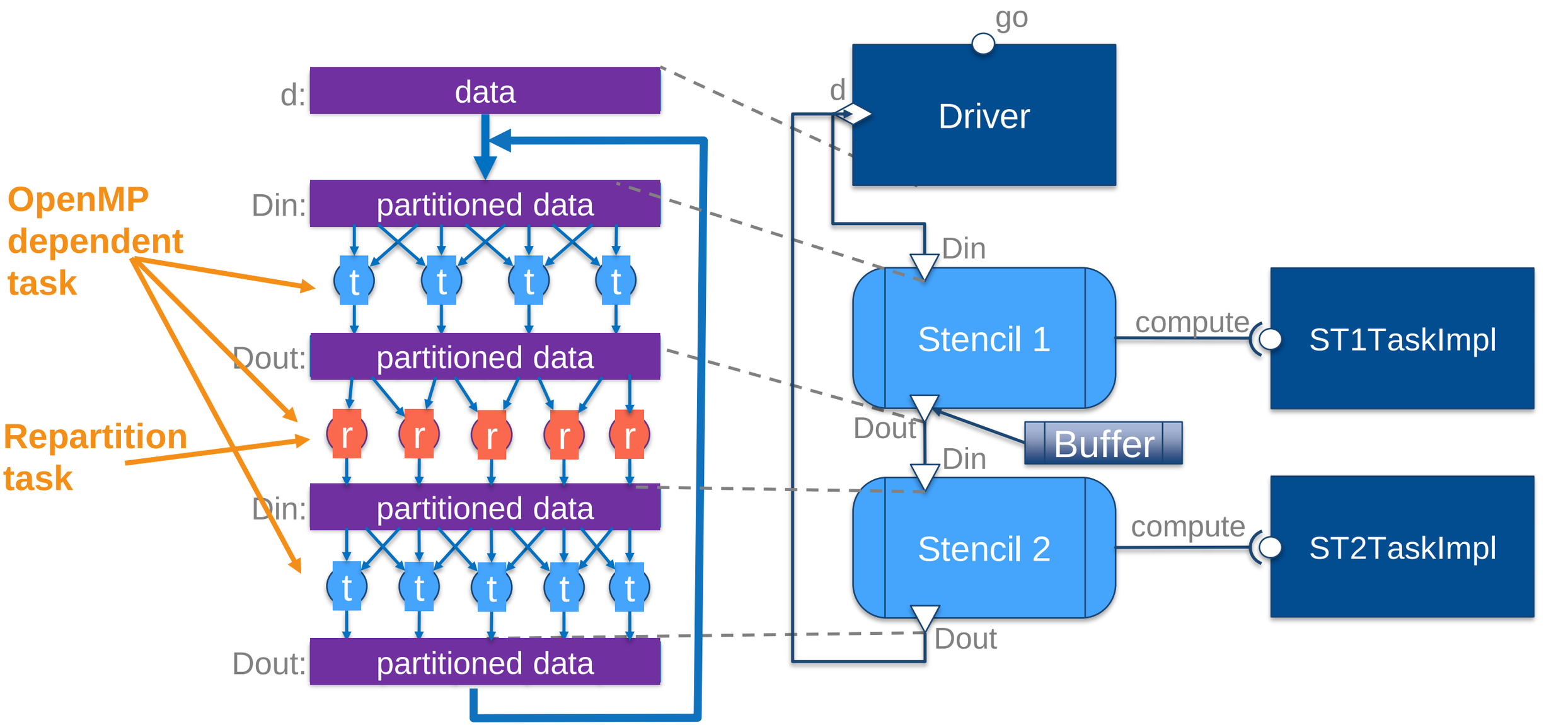


From COMET to OpenMP Code: Stencil-Stencil Use Case

Runtime execution
==
OpenMP program

Metatask

Implementation



Performance of ST-ST: Gant Chart

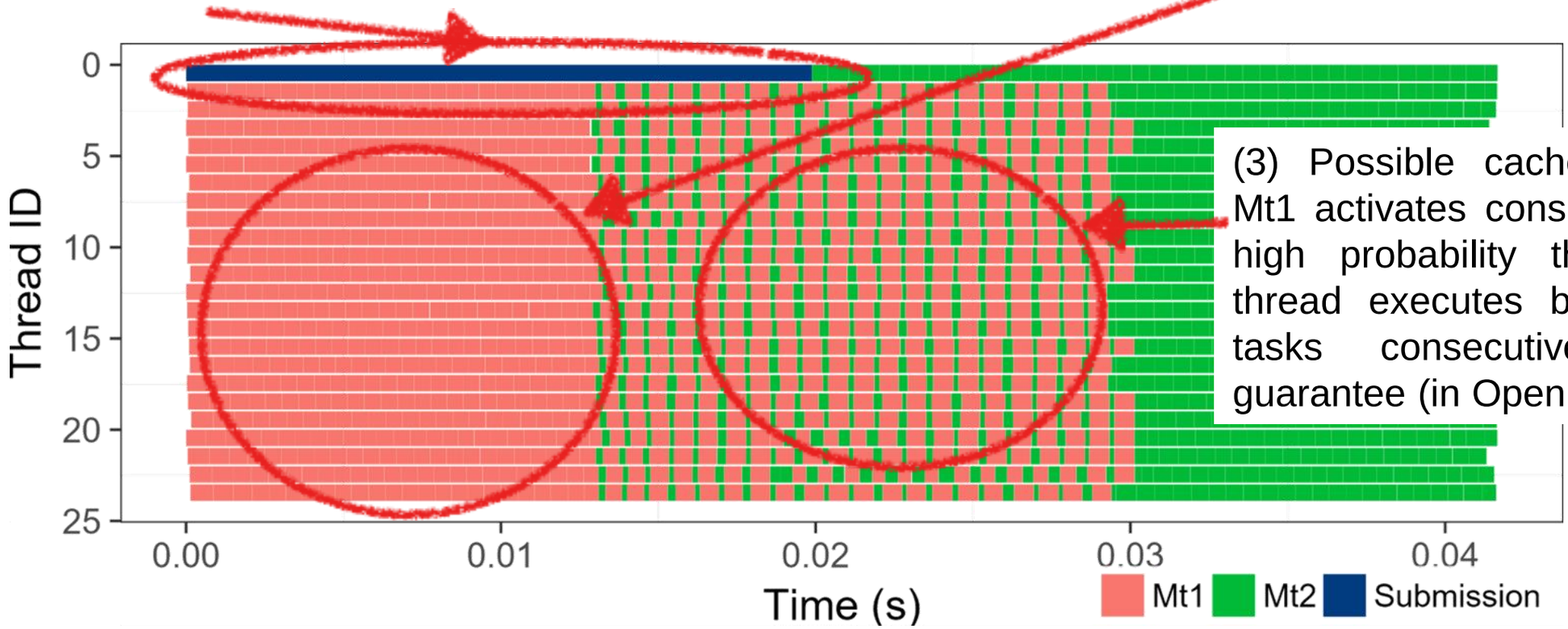
Completion time for 20 iterations

bloc size	1024x512	1024x256	1024x128	1024x64	1024x32	1024x16	1024x8
#tasks per it	128	256	512	1024	2048	4096	8192
time (s)	1.20	1.19	1.05	**1.01**	1.19	1.21	1.42

24 cores, bloc size 1024x64 ➡ 1024 tasks

(1) **Seems a very long submission time. Is it normal?**
- but short task duration (~ 1.1µs)
- about 42968 cycles of the machine / tasks

(2) **No cache reuse**:
producer task metatask Mt1 does not activate consumer Mt2 because it is not yet submitted !



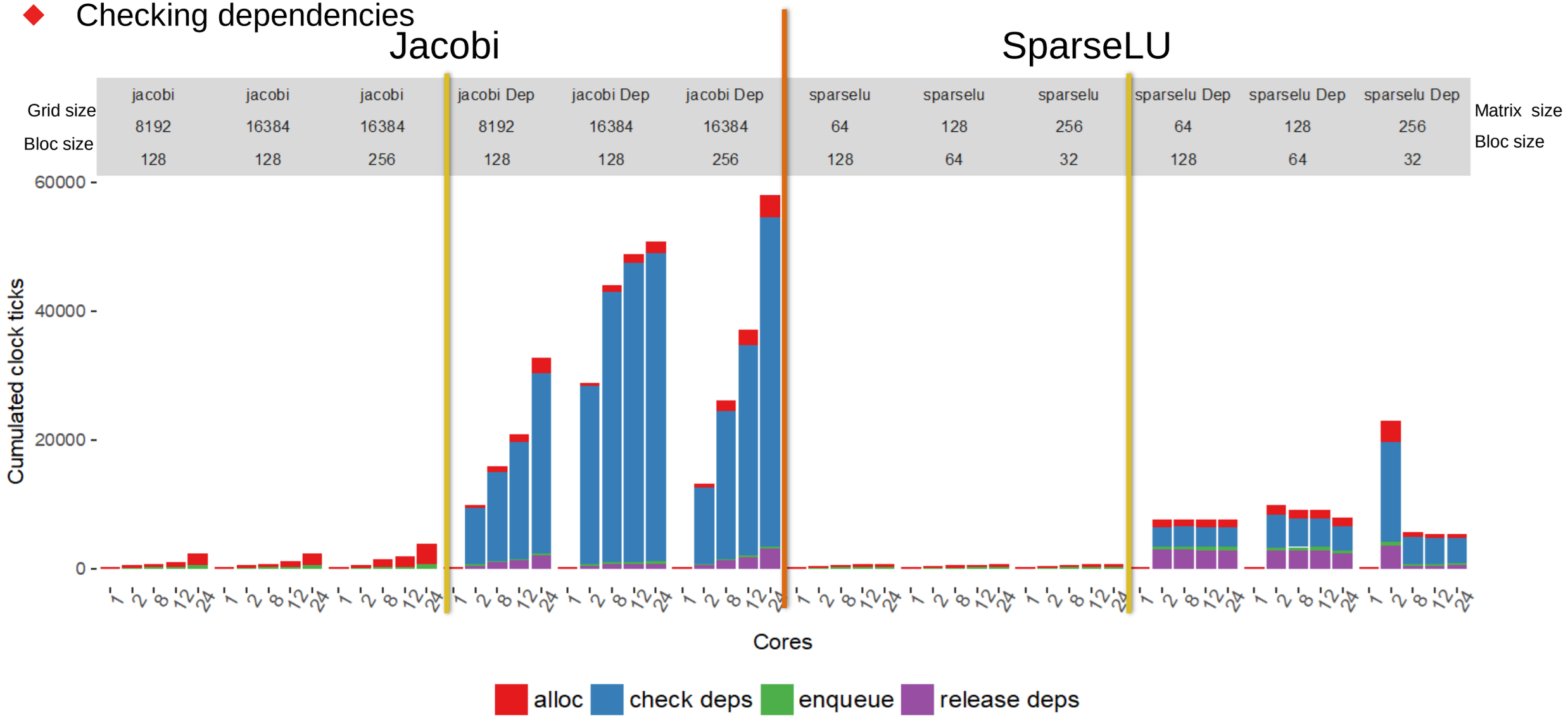
(3) Possible cache reuse when Mt1 activates consumer Mt2, with high probability that a running thread executes both dependent tasks consecutively, but no guarantee (in OpenMP)

➡ **Can performance be improved?**

Dependent Task Management Overheads (LLVM)

Dependent task management ~ 10 times cost of independent task management

- Dependent version (Jacobi Dep, sparselu Dep) versus independent version (Jacobi, sparselu)
- Checking dependencies



Costly operation ➜ task throttling to limit creation overhead

LLVM Overhead in Checking Deps

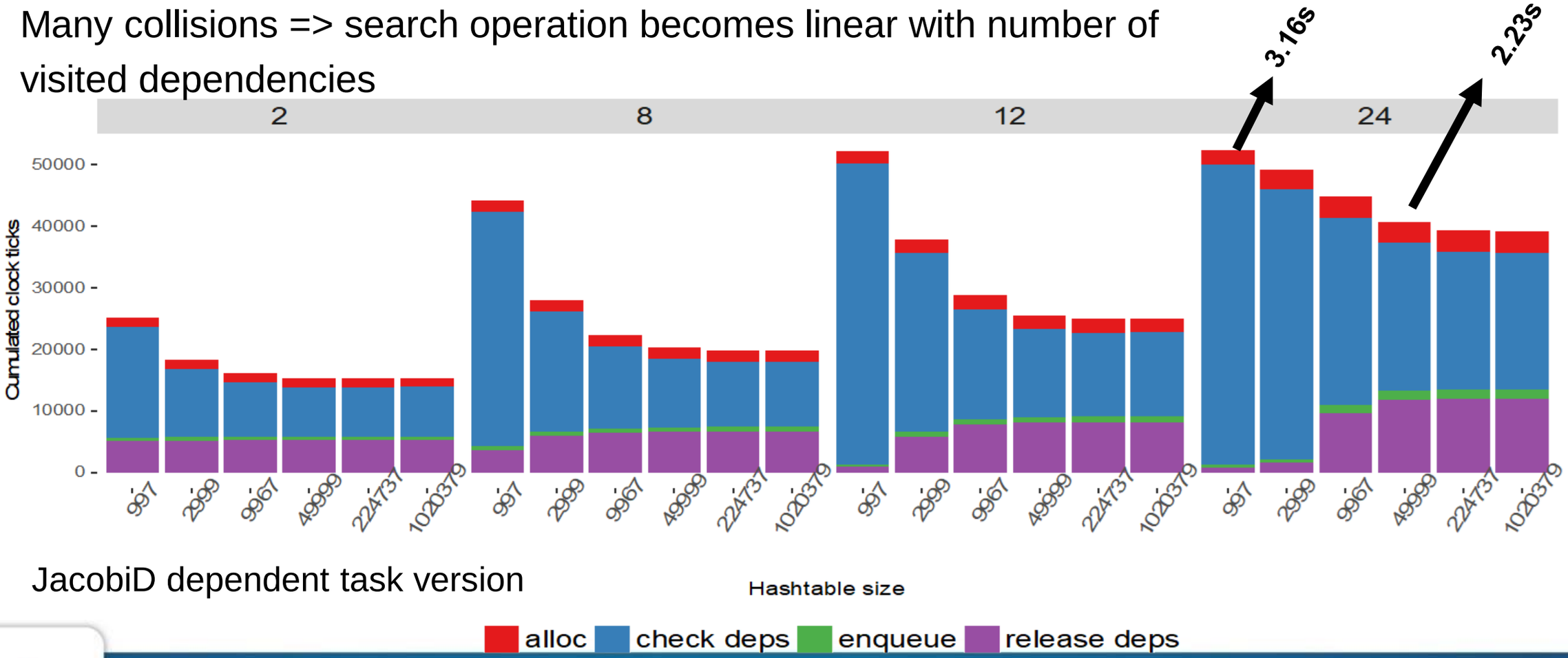
Checking dependencies in LLVM

- Retrieving the last created task that write data
- Hash table, key = memory address

We have been faced with a classical performance problem

- Too small static size for hash table (997)
  - Why this constant?
- Many collisions => search operation becomes linear with number of visited dependencies

30 % of gain on the completion time



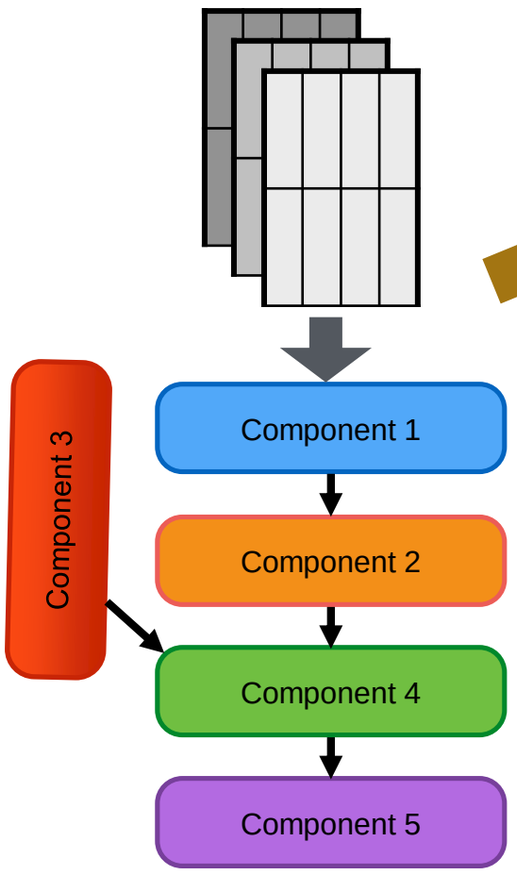
JacobiD dependent task version

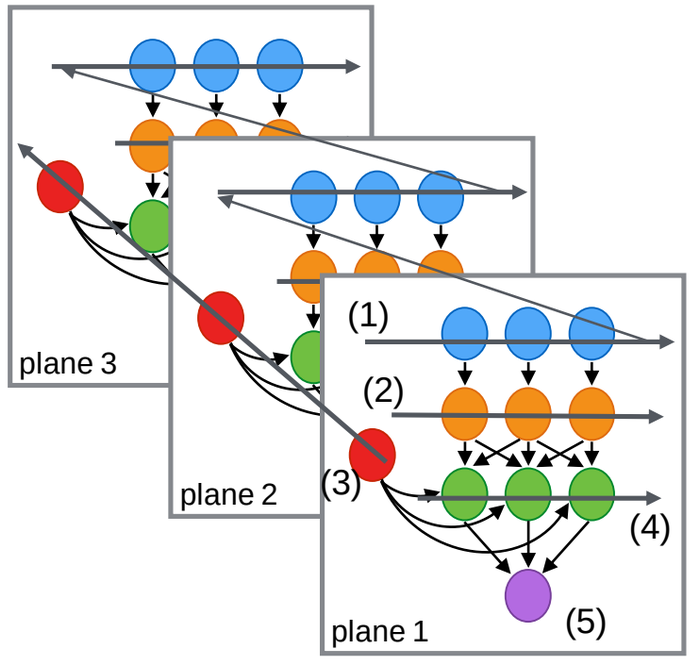
2D Gysela Advection

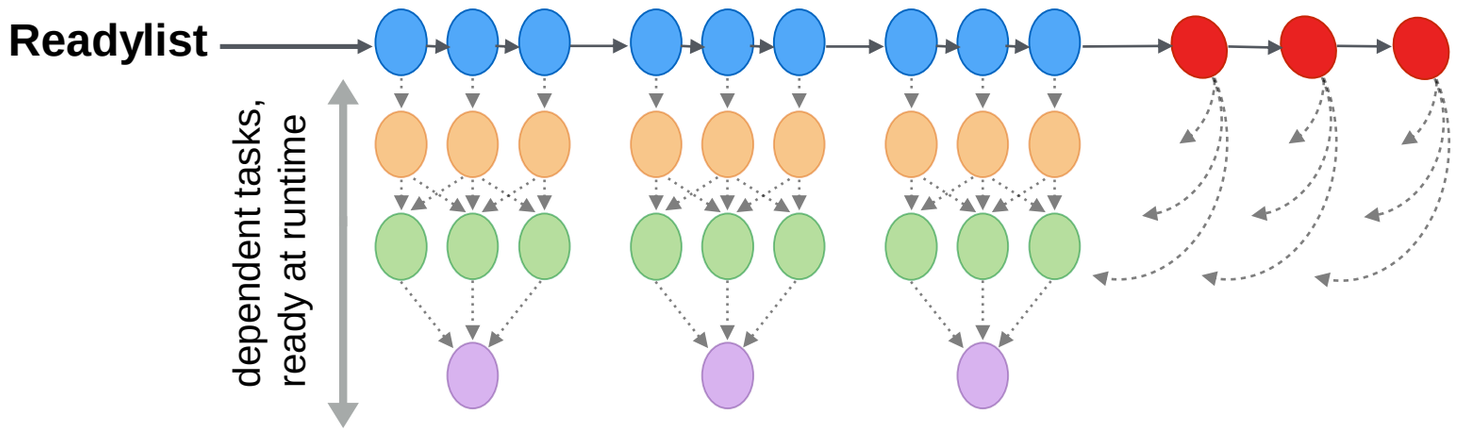
Generation of task follows

- for each component
  - for each plane

#pragma omp task…



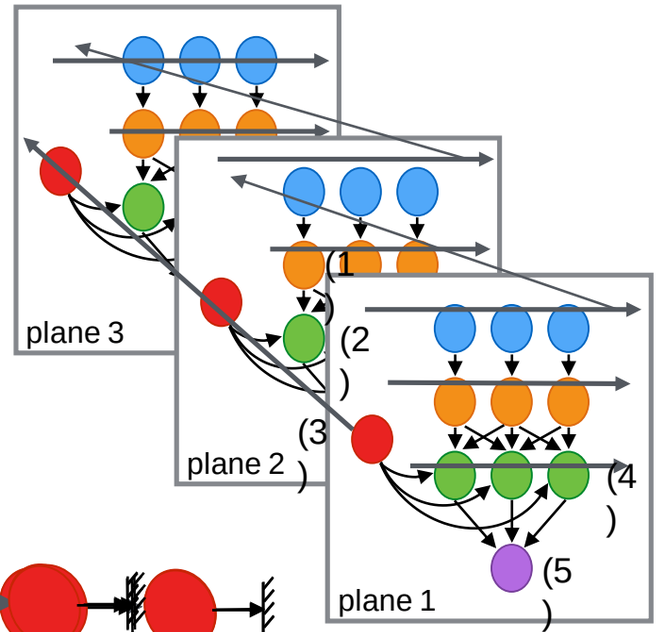




Problem: few cache reuse

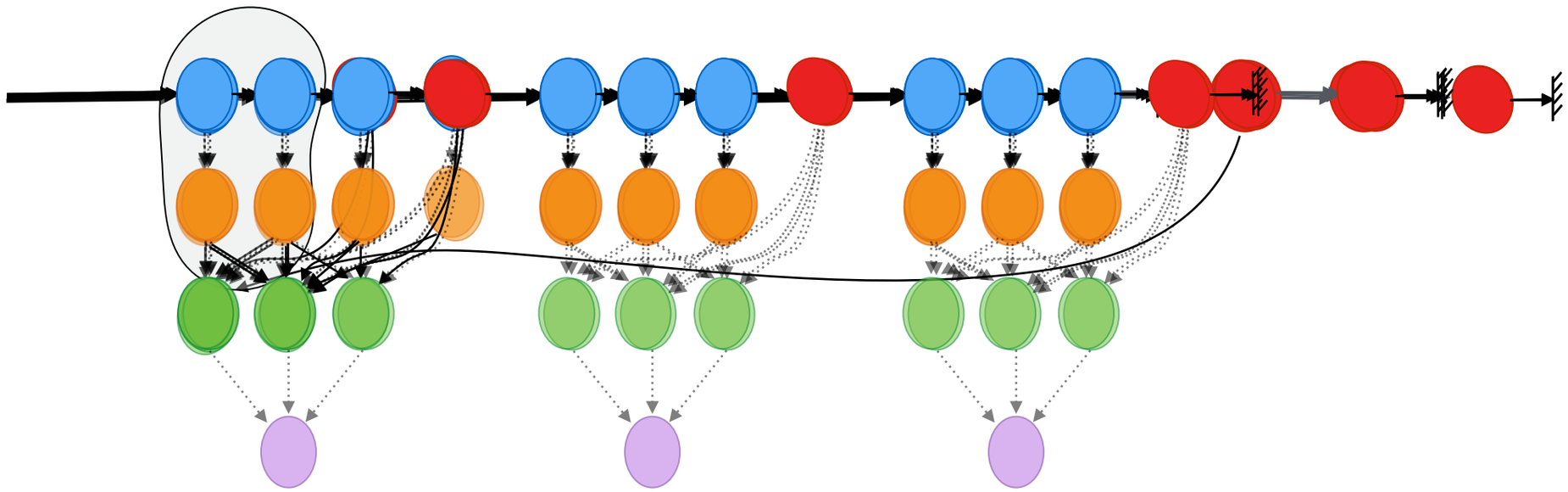
- breath first creation order
- work-stealing policy is non cache constructive

One Line Rescheduling



During task insertion

- Keep track of the range of tasks in the ready list on which it depends



Little overhead with respect to current code for checking dependencies

- Complexity O(1) with respect to default depend task implementation

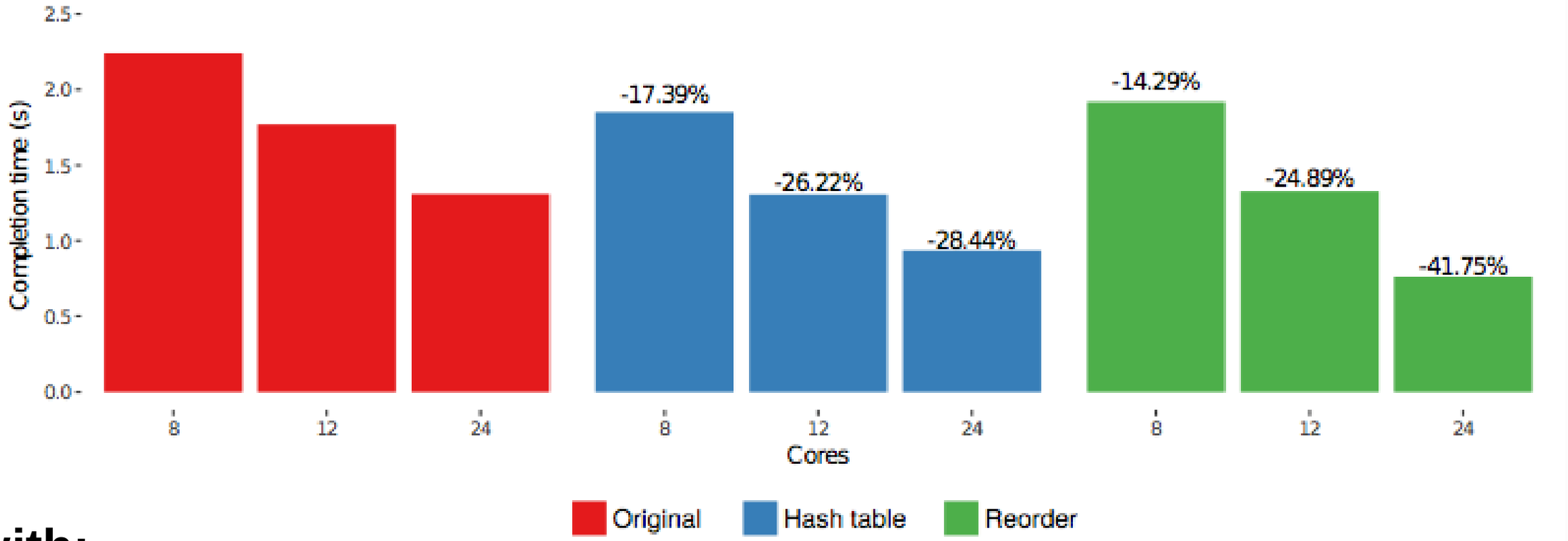
Change of work stealing implementation

- Constructive cache sharing version where both thief/victim access to same endpoint of the dequeue

Final 2D Gysela Advection Results

Up to 40% gain with respect to standard LLVM runtime

- Larger hash table: always interesting if many dependencies
- Reordering gain when the core count is high



with:

- hash table means hash table size=132069 in place of 997
- reorder= rescheduling with big hash table

- LLVM runtime, branch release_5.0
- Broadwell socket 24 cores@2.2Ghz

Bilan

libKOMP: research in productive runtime (LLVM based)

- Extensions: e.g. task/data affinity, scheduling heuristic, …
- OMPT module for tracing execution
- http://gitlab.inria.fr/openmp/libkomp

Limitation	Type	Solution	Solution in libKOMP
task creation overhead	size of internal descriptor	optimization	*reduce size of task descriptor*
		parallelization	*waiting for weak dependencies implementation in LLVM*
	checking deps	parallelization [20,14]	
		work first principle to checks deps on steal operation [15]	*possible integration of Kaapi algorithm [15] for nested dependent tasks*
		better algorithm [21]	dynamic resizing of hash table, Purpose of OpenMP-5.0 depobj ?
scheduling	task throttling	better heuristics [12] could be disable	**suppress heuristic**
	plugin algorithm	OpenMP extension + expert function	plug in infrastructure for new scheduling algorithm

Conclusion

Task implementation (in LLVM-) OpenMP runtime has limitations

- Overhead would be reduced in the future
- Several algorithmic decisions are hard coded
  - e.g. task throttling ≡ dequeue size in LLVM (for libGOMP: #task <= 64 #threads)
  - e.g. hash table size ≡ 997 in LLVM,
- Scheduling algorithms cannot be plug in the runtime

Perspectives

- Future nested tasks + weak dependencies may reduce time in task management
  - if implementation support high degree of concurrency with many fine grain tasks
- Interaction with the way component assembly generates task parallelism
  - impact in our compiler and may be into the component model is not well define

Open questions

- Should OpenMP specify basic task scheduling constraints?
  - a task creation is always a non blocking operation, except if explicit ‘if’ clause is false
- Universal scheduling algorithm does not exist. How to specialize it in OpenMP?