DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

Efficiently combining MPI and GPU-enhanced tasks within a large scale industrial application

Cédric AUGONNET

CEA/DAM, France

GTC Munich | October 2017

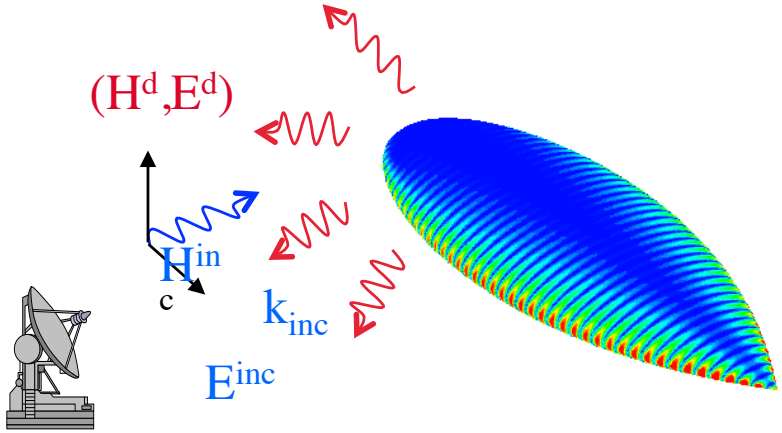
DE LA RECHERCHE À L'INDUSTRIE

cea

Solving Maxwell Equations on large problems

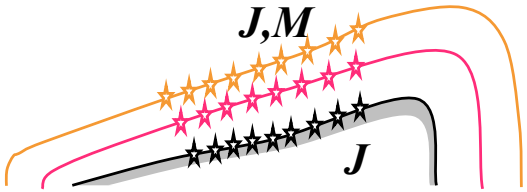
Context

Physical problem



- Radar Cross Section (RCS)
  - Ratio between incident and reflected energy
  - In a specific direction and polarisation
- Maxwell Equations
  - 2D axi-symetrical, 3D

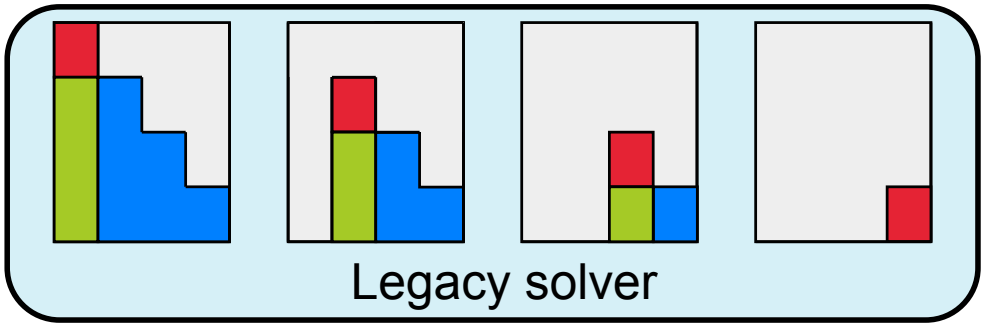
Boundary Elements Method

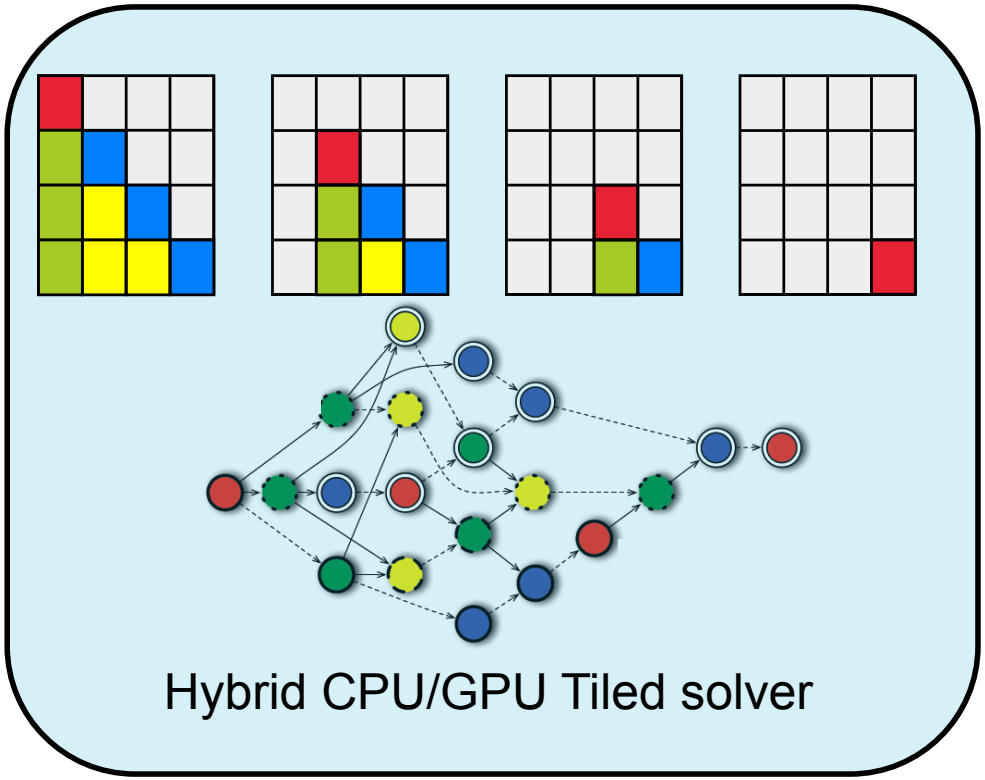


Compute currents at surfaces
=> Solve A X = B

- A : complex **dense** symmetric
- Direct LU or LL^T solver
- Millions of unknowns
- B : Thousands of RHS

Modernizing our legacy solver





- Legacy solver (25+ years)
  - *Similar* to Scalapack's LU / LL^T
  - No pivoting
  - Full MPI approach
- Tiled algorithms (UTK)
  - Better locality, more parallelism
  - Well suited for Accelerators
- Task parallelism
  - Parallel Application ⇔ Task graph
    - Machine independant !
  - Updating one block = one task (vertices)
  - Data dependencies (edges)
    - Intra-node => cudaMemcpy
    - Inter-node => MPI transferts
- Programmability & portability
  - GPU Kernels : NVIDIA CUBLAS
  - CPU Kernels : Intel MKL

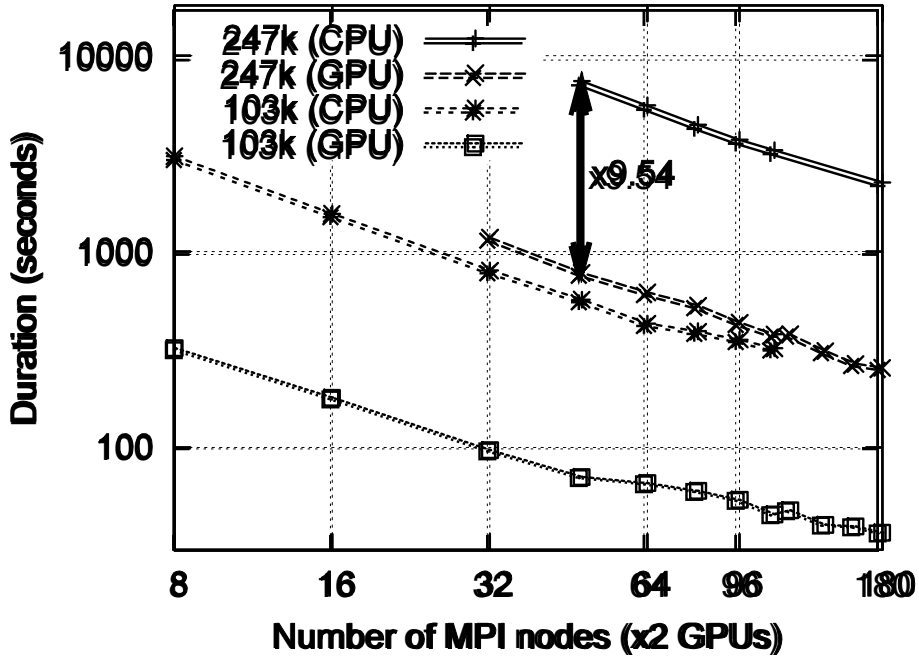
DE LA RECHERCHE À L'INDUSTRIE

cea

Performance evaluation

Performance on the TERA supercomputers

Speedups (8 CPU cores vs. 1 K20 GPU)



Measured performance ~ 83 TFlop/s

Sustained peak (*) ~ 120 TFlop/s

Energy consumption (520 000 d.o.f.)

520 000 unknowns
- Matrix size ~ 2 TB

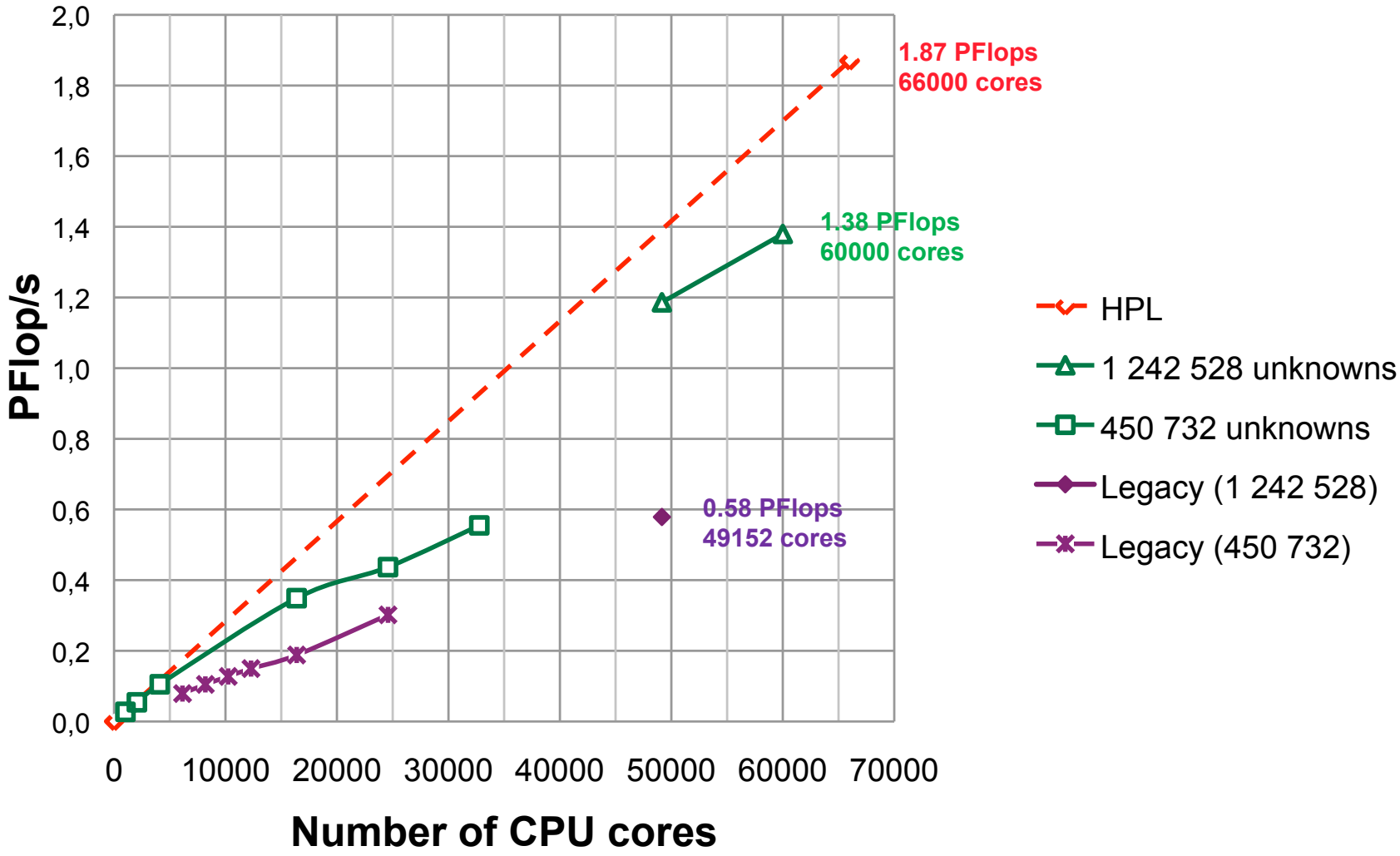
CPU only cluster:
- 5760 cores x 6.02H
- 34675 CPU.H
- 26 TFlop/s, **999 kWh**

Hybrid CPU/GPU cluster:
- (1440 cores + 360 GPUs) x 2.75 H
- 3960 CPU.H
- 83 TFlop/s, **223 kWh**

After upgrade : (TERA100 => TERA1000) **160 K80 GPUs ⇔ 207 Tflop/s**

(*) Sustained peak = performance of the $ZGEMM_{NT}$ kernel x #GPUs

Performance on the TERA-1000 supercomputer (no GPUs)



How did we design GPU-friendly tasks ?

Task programming interface

- programmability and maintainability concerns
  - Many kernels
  - Implicit task dependencies whenever possible
  - « insert_task » paradigm similar to PLASMA, MAGMA or StarPU

- **wait_for_task_group** primitive
  - Thread actually performs useful work while waiting (tasks, MPI progression, etc.)
  - Callable within tasks => recursion

```
void func(task_t *t)
{
    int a,b;
    task_unpack(t->args, &a, &b);
    printf(«a =%d b=%d », a, b);
}
```

```
int a = 42, b = 12;

insert_task(func,
            ARG, &a, sizeof(a),
            ARG, &b, sizeof(b),
            GROUP, &g,
            0);

wait_fot_task_group(&g);
```

Efficient CUDA synchronization mechanisms

- Minimizing synchronization overhead
  - Amdhal's law : e.g. 1% sequential code => maximum speedup = 100x
  - Keep host way ahead of *GPU work* to hide latency
  - Task submission should never block the calling thread

- Avoid **explicit** host-device synchronization
  - Avoid cudaStreamSynchronize
    - Rely on **cudaStreamWaitForEvent** instead !
    - Non-blocking stream synchronization primitive

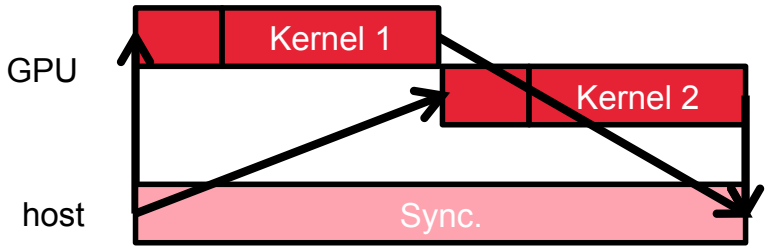
- Avoid **implicit** synchronization either
  - Never use the *null stream*
  - Avoid calling cudaMalloc/cudaFree/cudaHostRegister in the critical path
    - Use a preallocated memory pool instead
    - Pin host memory in advance
  - cudaEventCreate**WithFlags**(&event, **cudaEventDisableTiming**);
  - Some HW resources can get saturated
    - Clear improvement between each HW generation

Breaking kernel concurrency is really easy

BAD

```
cudaHostRegister(hA, …)
memcpyAsync(dA, hA, stream1);
kernel1<<<stream1>>>(dA);

cudaHostRegister(hB, …);
memcpyAsync(dB, hB, stream2);
kernel2<<<stream2>>>(dB);

cudaDeviceSynchronize();
```

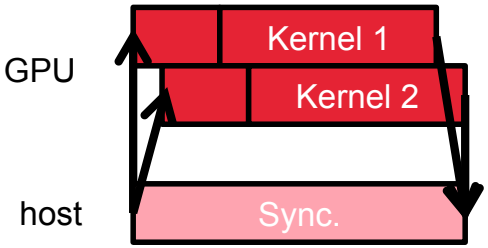
Observed behaviour :



BETTER

```
cudaHostRegister(hA, …)
cudaHostRegister(hB, …);

memcpyAsync(dA, hA, stream1);
kernel1<<<stream1>>>(dA);

memcpyAsync(dB, hB, stream2);
kernel2<<<stream2>>>(dB);

cudaDeviceSynchronize();
```

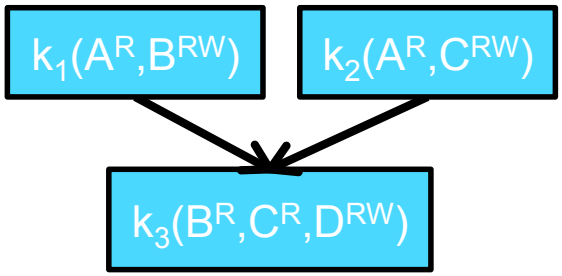
Expected :



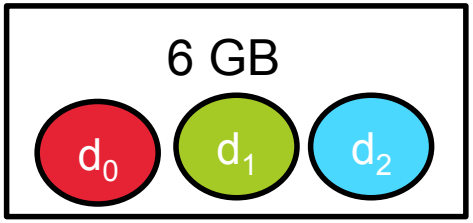
do use NVPROF !

What if host memory >> GPU memory ?

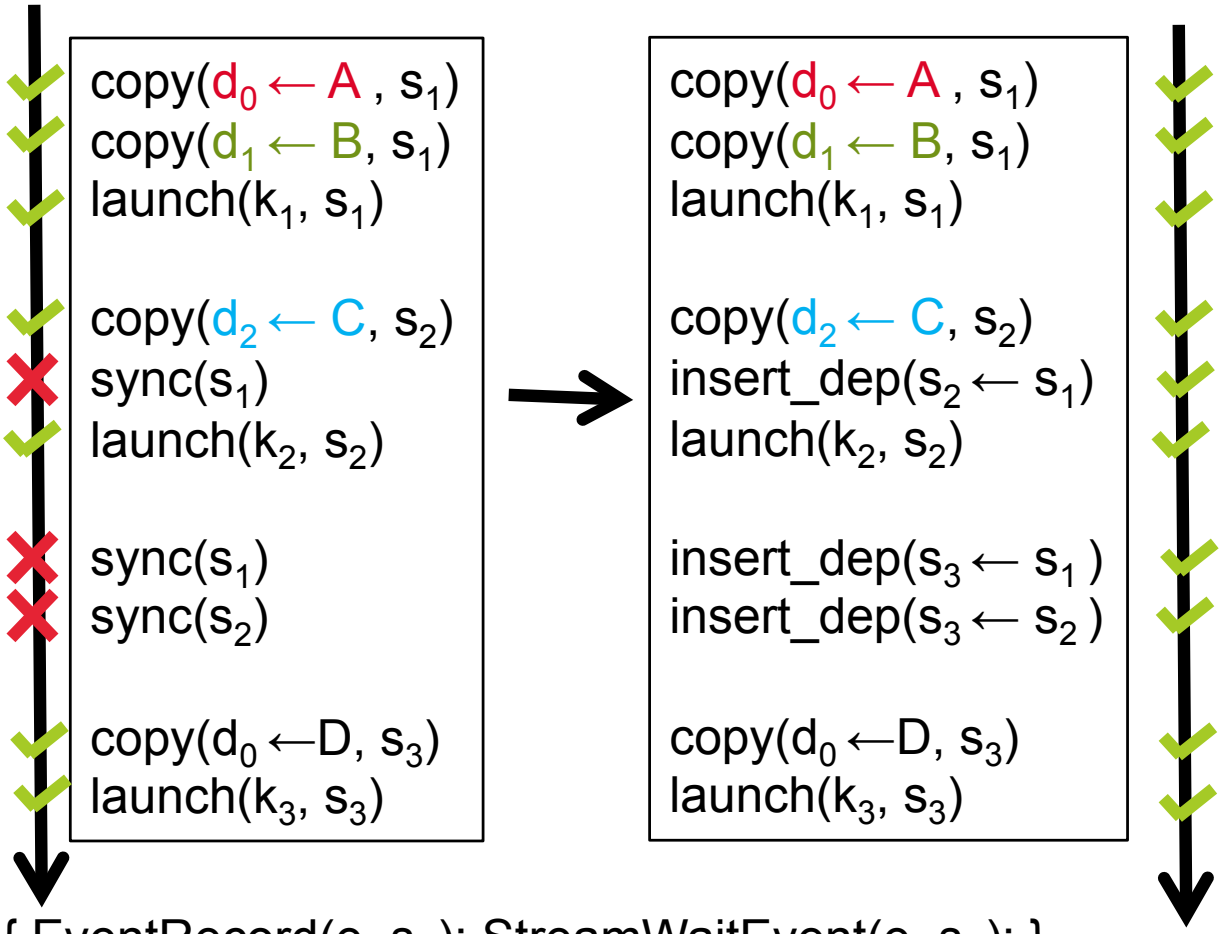
- Typically 256 GB on the host vs. devices with 6 GB
- Example with 3 GPU buffers



Graph of tasks



Pool =3 device buffers of 2GB



insert_dep(s_A,s_B) ⇔ { EventRecord(e, s_B); StreamWaitEvent(e, s_A); }

- => *Transparently* solve problems larger than GPU memory !

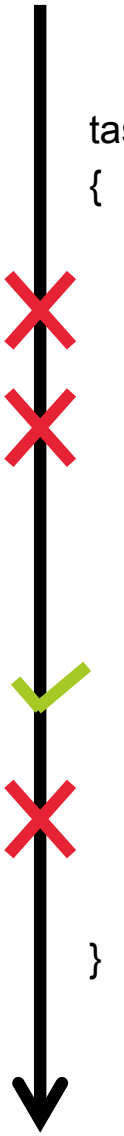
Translating this into a GPU-friendly tasking model

FROM USER PERSPECTIVE

```
task_foo(data[in] A, data[inout] B)
{

    dA = cudaMalloc(…);
    cudaMemcpy(dA ← A);
    dB = cudaMalloc(…);
    cudaMemcpy(dB ← B);


    /* typically a CUBLAS  call  */
    kernel_foo<<<…>>>(dA, dB);

    cudaMemcpy(B ← dB);
    cudaFree(dA);
    cudaFree(dB);
}
```

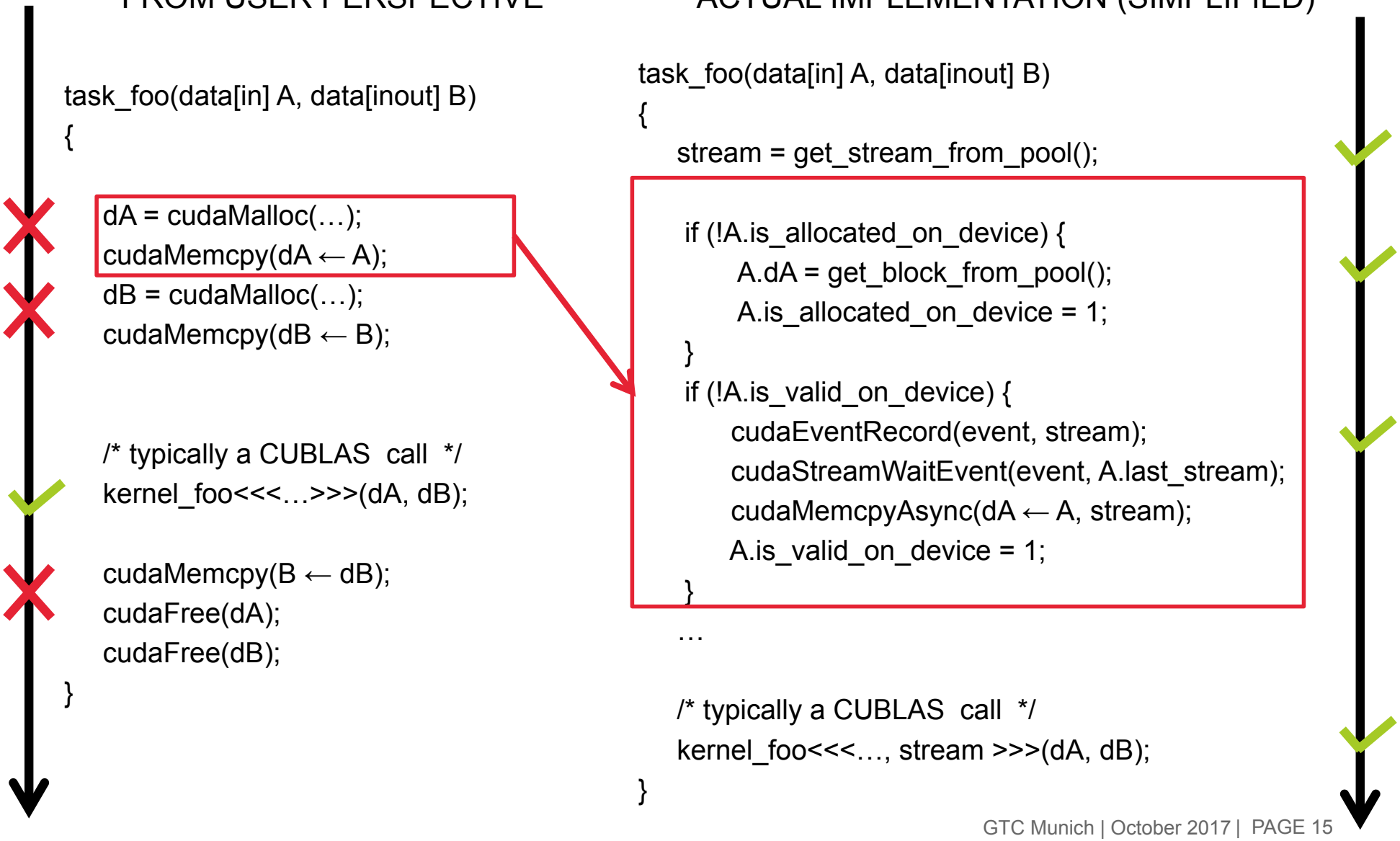
Translating this into a GPU-friendly tasking model

FROM USER PERSPECTIVE



```
task_foo(data[in] A, data[inout] B)
{

    dA = cudaMalloc(…);
    cudaMemcpy(dA ← A);
    dB = cudaMalloc(…);
    cudaMemcpy(dB ← B);


    /* typically a CUBLAS  call  */
    kernel_foo<<<…>>>(dA, dB);

    cudaMemcpy(B ← dB);
    cudaFree(dA);
    cudaFree(dB);
}
```

Translating this into a GPU-friendly tasking model

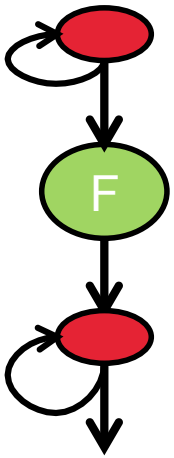


DE LA RECHERCHE À L'INDUSTRIE

cea

Going distributed with MPI

Properly mixing MPI and tasks (1/2)

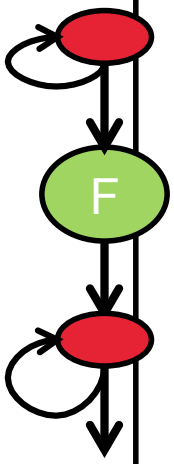



```
MPI_Irecv(&data, ....)
MPI_Wait(...)

insert_task(F, ARG, &data,
                GROUP, &g,
                ...);
wait_for_task_group(&g);

MPI_Isend(&data, ....)
MPI_Wait(...)
```

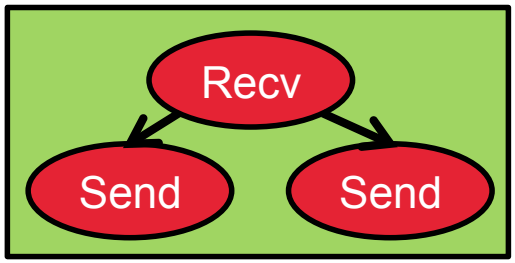
- Typical approach
  - Fully dissociate MPI from tasks
  - Two level programming model :
    - Tasks within a node
    - MPI between nodes
  - Too many synchronization points
    - Hardly scalable
  - Little overlap between tasks and communication
    - Possible, but tedious !
  - How to implement multiple concurrent Bcast ?

Properly mixing MPI and tasks (2/2)

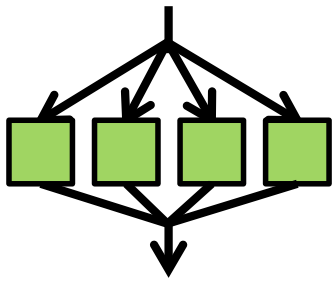
```
void G (task_t *t)
{
   t->restart = 1;
   switch (t->step) {
      case 0 :
         ... initialize_task...
         MPI_Irecv(&data, ... , &mpi_req);
         t->step++;
      case 1:
         if (MPI_Test(&mpi_req)) return;
         t->step++;
      case 2:
         F(data)
         MPI_Isend(&data, ...);
         t->step++;
       case 3:
         if (MPI_Test(&mpi_req)) return;
         t->restart = 0;
    }
}

Insert_task(G, ...)
```



- Adding *steps* in tasks
  - Task has a new step field
  - Task can specify that they are restartable
  - Task can return before completion
    - Restarted later on …
    - … At the current step
- Really simple to use
  - Appears sequential to programmer
- Powerful way to express complicated MPI transfer schemes
  - Possibly interleaved with tasks

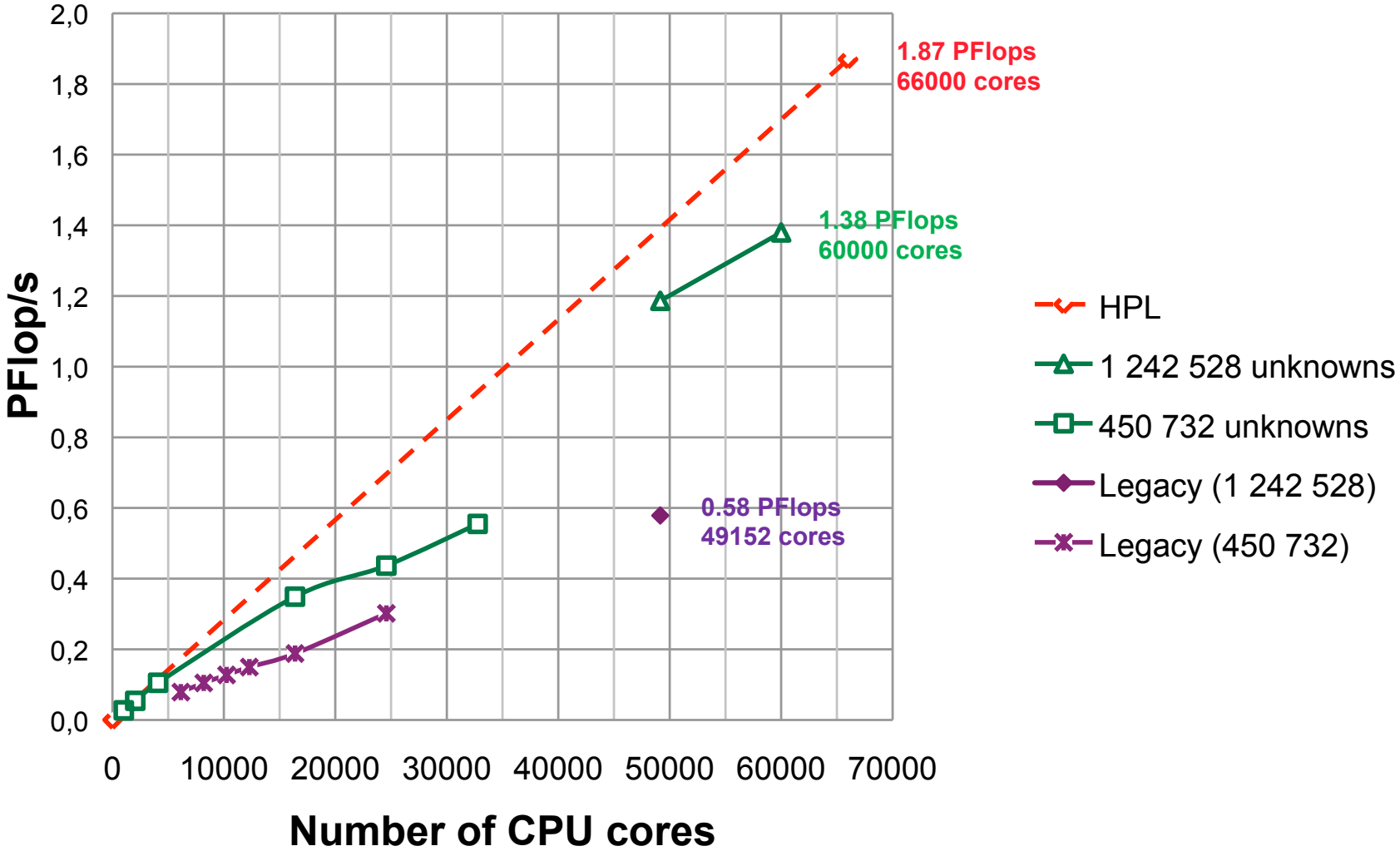


« Bcast » task



Concurrent IBcasts

Performance on the TERA-1000 supercomputer



160 K80 GPUs = 207 Tflop/s⇔ 10000 cores

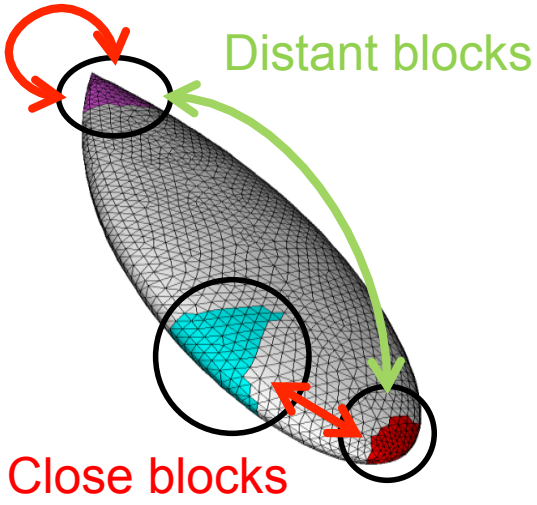
DE LA RECHERCHE À L'INDUSTRIE

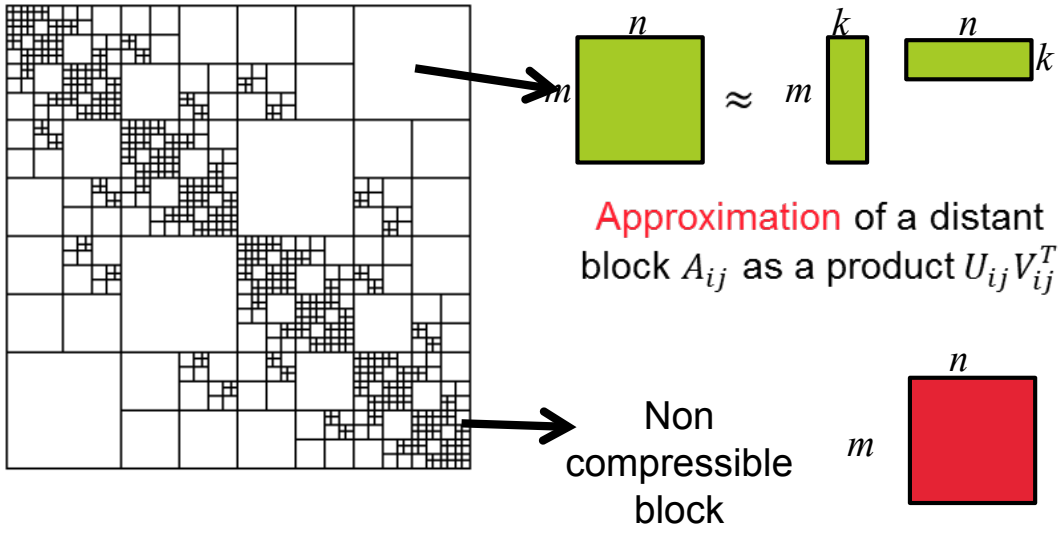
cea

Low-rank compression techniques

Low-rank compression techniques for direct solvers

- Take advantage of the inner properties of the matrices resulting from Electromagnetism problems
- Geometrical partitionning of mesh vertices
- Distant interaction → Block with low numerical rank *(k)*

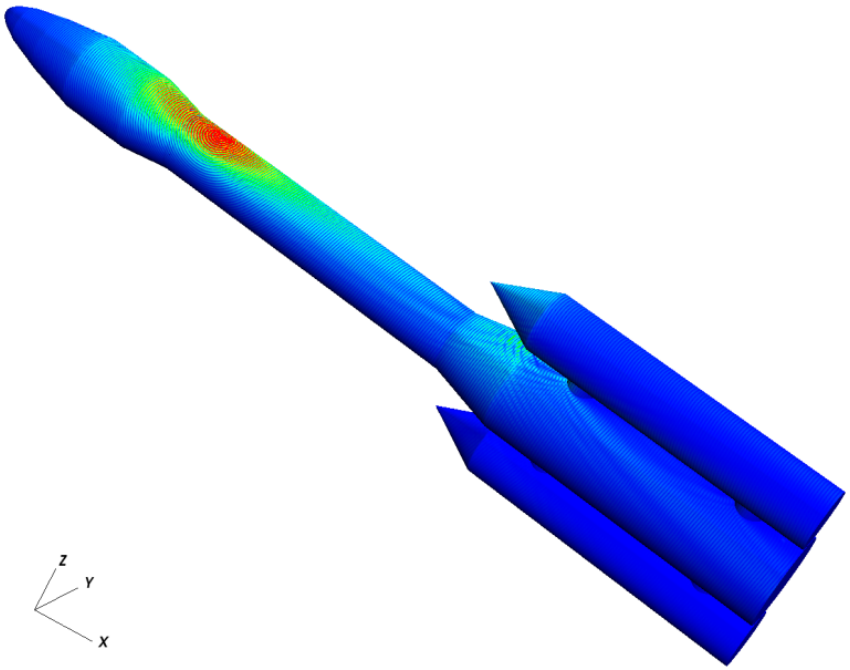




Without compression:
Memory $O(n^2)$, Ops. $O(n^3)$

Hierarchical compression:
Memory & Ops : $O(n \log n)$

Performance on a test case with 1 650 875 unknowns



« CNES Booster» test-case
(antenna-type computation)

ISAE'14 Workshop

Solver	Factorization time	Speedup
No compression (predicted, 15000 cores)	**15000 cores x 20.4H = 305870 CPU.H**	-
H-matrix (200 cores)	200 cores x 5.2H = 1043 CPU.H	**293x**

Challenges to get H-matrices on GPUs

- Memory management
  - *Memory reallocated very often => hard to design a memory pool*
  - *Complicated on both host and devices*

- Cannot *directly* rely on CUBLAS to obtain good performance
  - GEMM => QR / SVD / UNMQR kernels
  - Much less efficient on GPUs than GEMM

- Granularity issue
  - e.g. 512x512 matrices vs. 4096 x 17 or 128 x 5 matrices
  - Matrices with different sizes and different numerical ranks
    - 128 x 12, 4096 x 28
    - 4096 x 16, 4096 x 27, …
  - Much more kernel calls
    - Batching *should* help hiding overhead

- **What are we missing ?**
  - **Efficient QR / UNMQR kernels (possibly SVD)**
  - **(Truly) asynchronous QR/SVD/UNMQR kernels**
  - **(Truly) Asynchronous kernel batching**
  - **Efficient batched kernels with different input sizes**

Conclusion and Perspectives

- Solving Maxwell equations
  - Huge dense linear algebra problems
  - Solved on the TERA supercomputer at CEA/DAM

- Efficient management of concurrency
  - Event-based synchronization
  - NVPROF is extremely helpful
  - Transparently solve problems larger than GPU memory

- Flexible programming model
  - Task-based
  - Tightly integrate MPI, CUDA, and tasks
  - **The same code is running efficiently on GPUs, CPUs *(also on Intel KNLs…)***
    - 1.38 Pflops on 60000 CPU cores
    - 0.2 Pflops on 160 K80 GPUs

- Implemented a distributed H-Matrix solver
  - Allows to solve problems with millions of unknowns
  - Gained several orders of magnitude on CPUs
  - Still many challenges to efficiently port it on GPUs

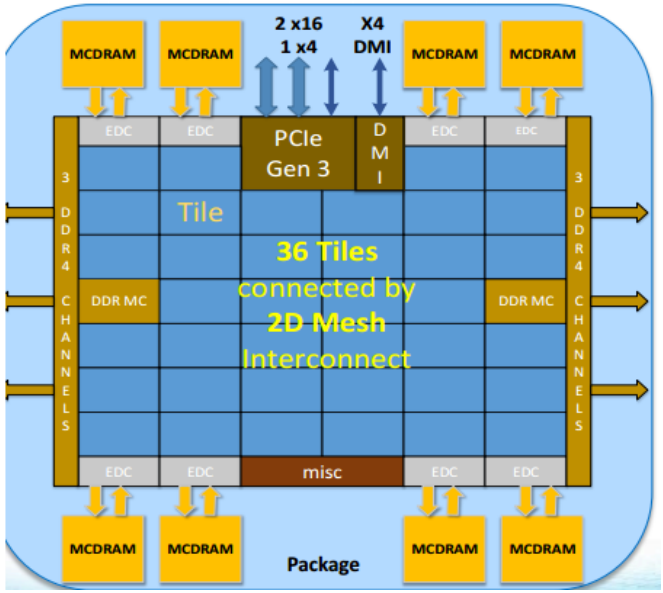
cedric.augonnet@cea.fr

DE LA RECHERCHE À L'INDUSTRIE

cea

Backup slides

Toward manycore architectures : supporting Intel KNL



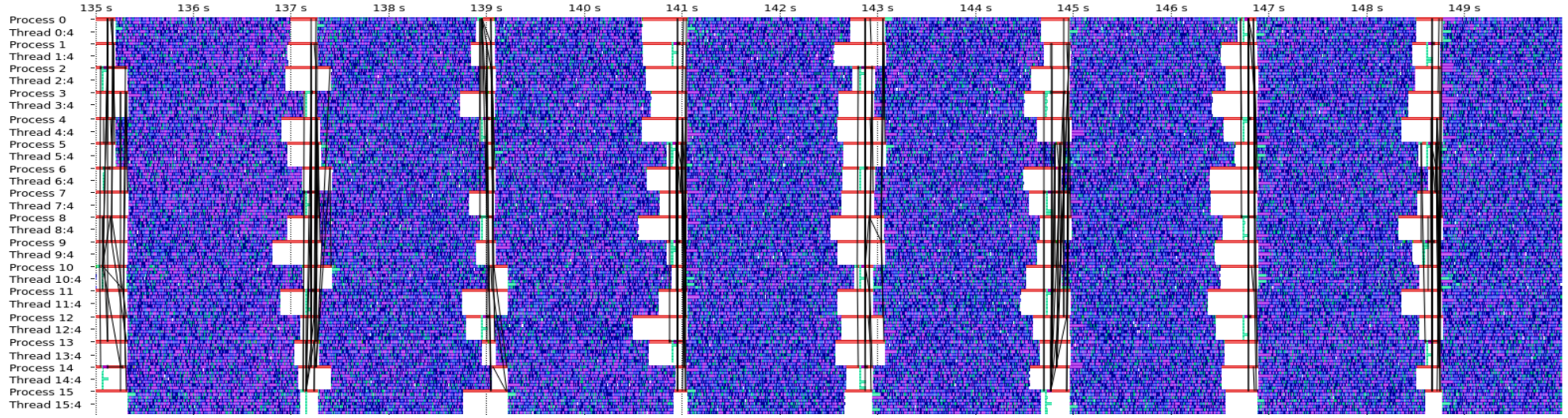
Intel KNL

- 68 cores with 4 hyperthreads
- Fast memory (=> NUMA issues)
- Supports OpenMP, pthreads, etc.

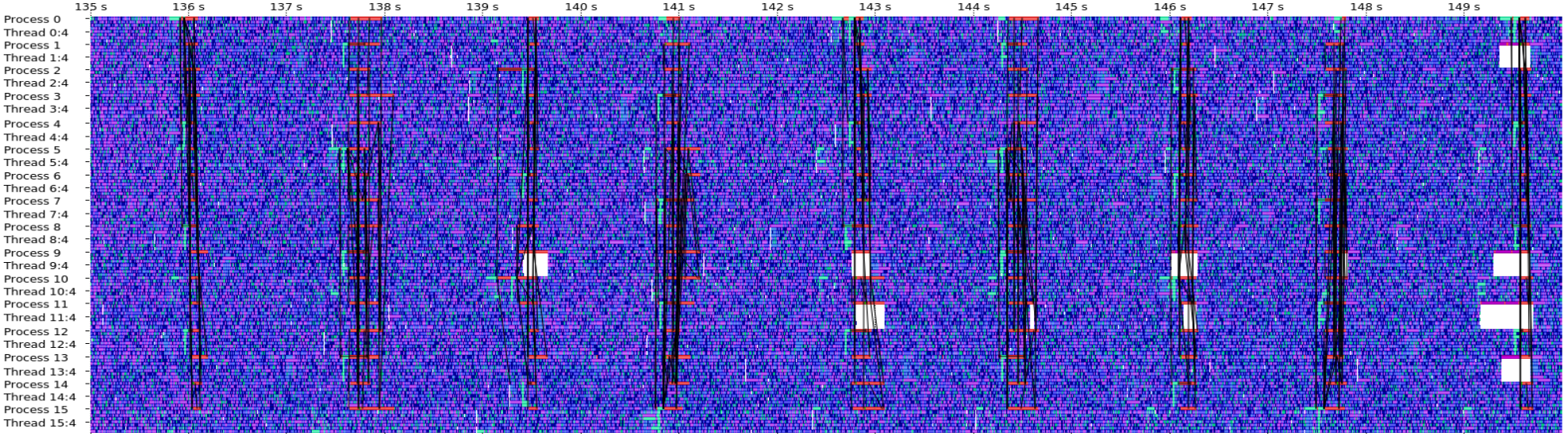
- Number of cores within each nodes keeps increasing
  - Task parallelism well suited !
- Straightforward approach
  - 4 processes per KNL
  - 17 threads per process
  - => LLt solver << 1 Tflops (double precision) per card
- Granularity issue
  - Large block sizes required to get performance
  - Increasing block size reduces the amount of parallelism
- Adding a new level of parallelism
  - 4 processes per KNL
  - 4 or 5 threads per process
  - Up to 4 dedicated cores per thread
    - Nested BLAS kernels

  => LLt solver ~1.5 Tflops (double precision) per card

Impact of Look-ahead on the BLR solver



Without Look-ahead



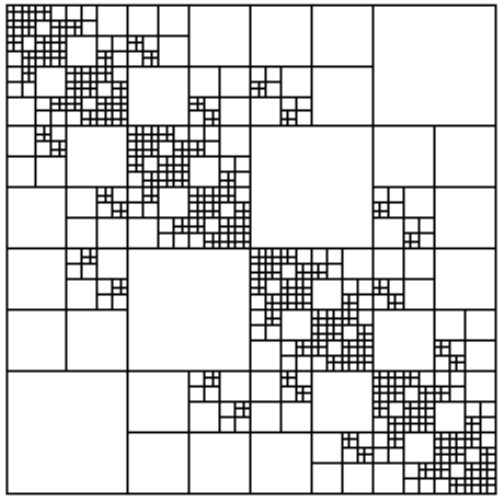
With Look-ahead

Challenges to design and implement H-matrix solvers

- Need to consider programmability and maintenability !
  - Many many new kernels (> 50)
  - Implicit task dependencies whenever possible
  - « insert_task » paradigm similar to PLASMA, MAGMA or StarPU

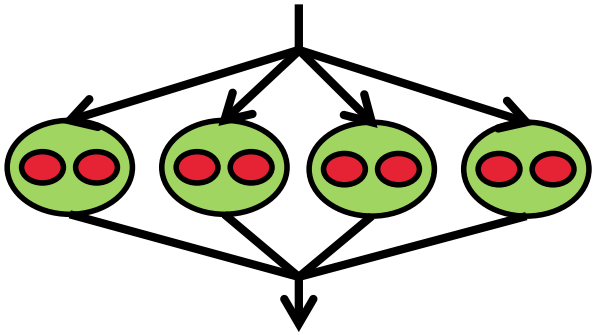
```
void func(task_t *t)
{
    int a,b;
    task_unpack(t->args, &a, &b);
    printf(«a =%d b=%d », a, b);
}
```

```
int a = 42, b = 12;
insert_task(func,
            ARG, &a, sizeof(a),
            ARG, &b, sizeof(b),
            0);
```



- Irregular and recursive workload
  - **How to implement recursive tasks ?**
- Severe load imbalance between MPI processes
  - Must be as asynchronous as possible (look-ahead)
  - Turn almost everything to tasks including MPI transfers
  - **How to combine MPI nicely with tasks ?**
- Should be suitable for tomorrows machines too !
  - **Manycore, Accelerators, …**

Supporting recursive tasks with task groups



```
void F1(task_t *t)
{
   task_group_t g2 = {0};
   insert_task(F2, GROUP, &g2, ...);
   insert_task(F2, GROUP, &g2, ...);
   wait_for_task_group(&g2);
}

task_group_t g1 = {0};
for (i = 0; i < 2; i++)
for (j = 0; j < 2; j++)
{
   insert_task(F1, GROUP, &g1, ...);
}
wait_for_task_group(&g1);
```

- Task Groups
  - ~ Reference counter
  - Synchronization with « wait_for_task_group»
  - Can be used within tasks too
    - => Recursion made possible
- Similar to sequential programming
  - Much easier to use than continuations (callbacks) !
- Execution *stalled* during synchronization
  - Thread actually performs useful work while waiting (tasks, MPI progression, etc.)