Assembly Operations for Multicore Architectures using Task-Based Runtime Systems

Genet Damien, INRIA, Bordeaux, France

Bosilca George, University of Tennessee, Knoxville, USA

Guermouche Abdou, INRIA, LaBRI, Univ. Bordeaux, Bordeaux, France

ROME14 - Porto - August 26, 2014

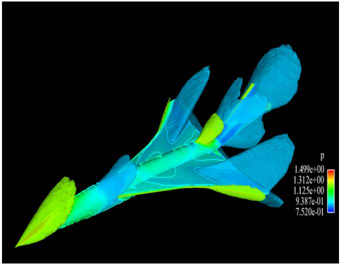
Introduction

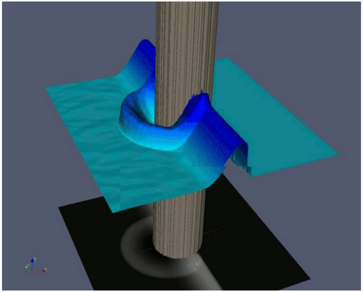
In numerical simulations :

→ we want to study a physical phenomenon;

→ we will use numerical methods to model it (finite differences, finite volumes, finite elements);

→ we will develop an application to simulate it.

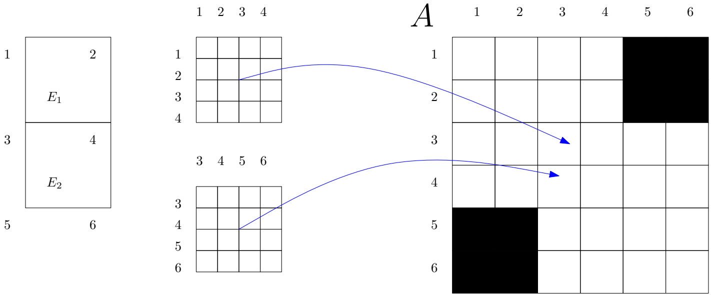




In numerical simulation :

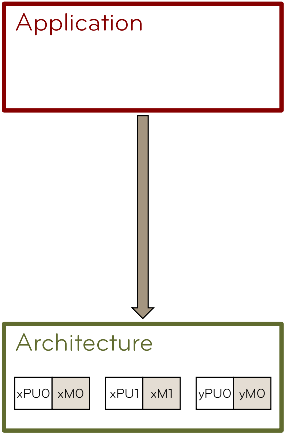
→ Finite element methods work on meshes of elements.

→ Each element will contribute to many entries in the global matrix.



The assembly operation can be found in sparse direct solver based on multifrontal methods.

Application



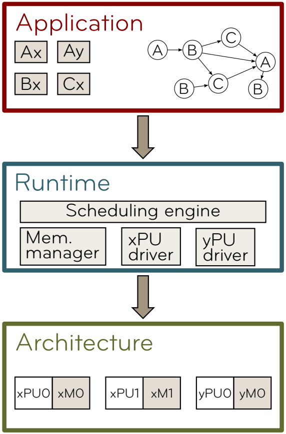
The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which

→ requires a big programming effort

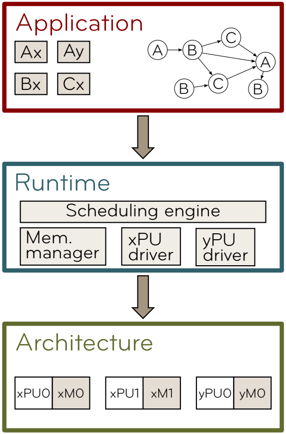
→ is difficult to maintain and update

→ is prone to (performance) portability issues

Application

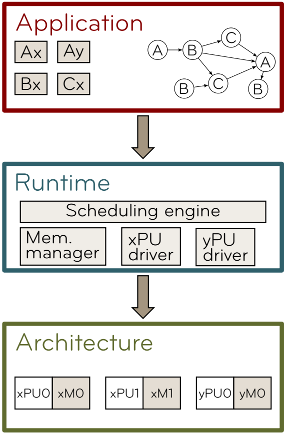


Application



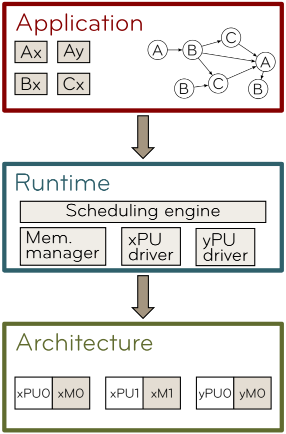
→ runtimes provide an abstraction layer that hides the architecture details

Application



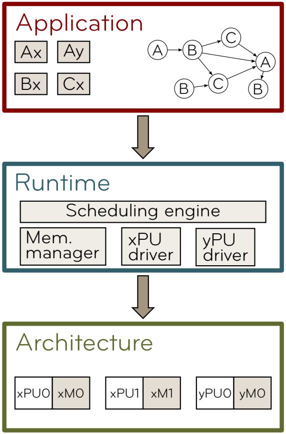
- → runtimes provide an abstraction layer that hides the architecture details
- → the workload is expressed as a DAG of tasks where the dependencies are
  - → defined explicitly
  - → defined through rules
  - → automatically inferred

Application



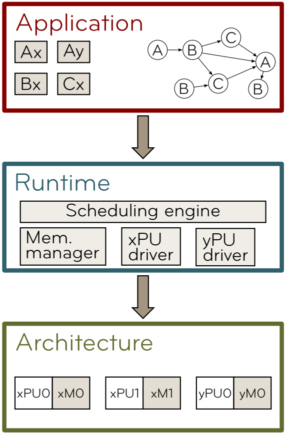
- → runtimes provide an abstraction layer that hides the architecture details
- → the workload is expressed as a DAG of tasks where the dependencies are
  - → defined explicitly
  - → defined through rules
  - → automatically inferred
- → the scheduler decides when/where to execute a task

Application



- → runtimes provide an abstraction layer that hides the architecture details
- → the workload is expressed as a DAG of tasks where the dependencies are
  - → defined explicitly
  - → defined through rules
  - → automatically inferred
- → the scheduler decides when/where to execute a task
- → the drivers deploy the code on the devices

Application

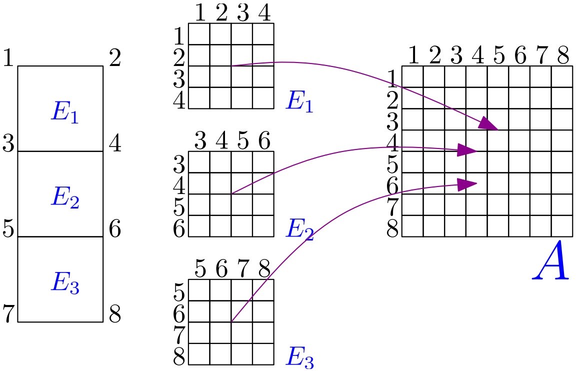


- → runtimes provide an abstraction layer that hides the architecture details
- → the workload is expressed as a DAG of tasks where the dependencies are
  - → defined explicitly
  - → defined through rules
  - → automatically inferred
- → the scheduler decides when/where to execute a task
- → the drivers deploy the code on the devices
- → the data manager does the data transfers and guarantees the consistency

1 State of the art

Assembly operations

```
do i=1, Nelement
    call assemble( A, block[i] )
end do
```

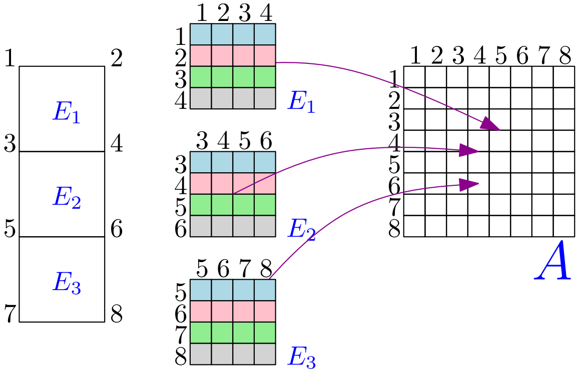


→ Parallelise inside a block -or- parallelise the blocks loop.

Assembly operations

```
do i=1, Nelement
    call assemble( A, block[i] )
end do
```

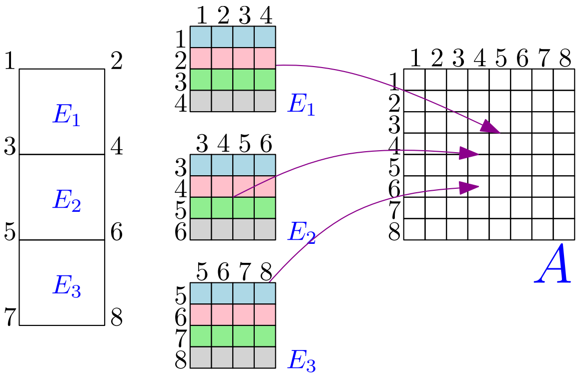
First idea : Use `OpenMP` to parallelise the assemble function.



Assembly operations

```
do i=1, Nelement
     call assemble( A, block[i] )
end do
```

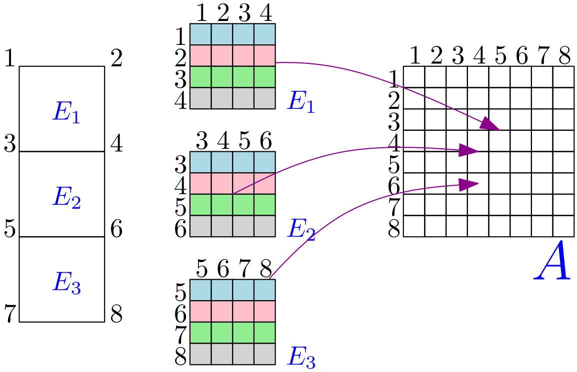
⇒ **Bigger the blocks, the more parallelism!**



Assembly operations

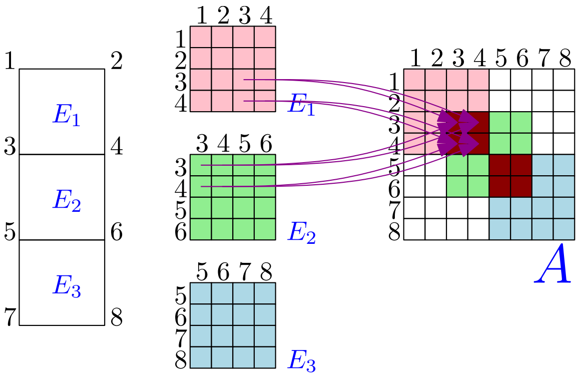
```
do i=1, Nelement
     call assemble( A, block[i] )
end do
```

⇒ **With small blocks, not enough parallelism!**



Parallelising the outer loop

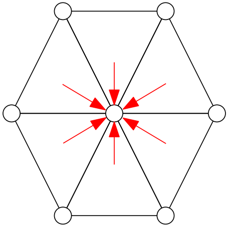
If you try to parallelise the outer loop with `OpenMP` , it goes bad :



Graph Coloring

Introduced by Cecka *et al.*[1]

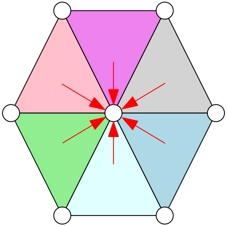
A set of elements will update the same entries in the matrix.



[1] Assembly of finite element methods on graphics processors. Int. J. for Numerical Methods in Engineering

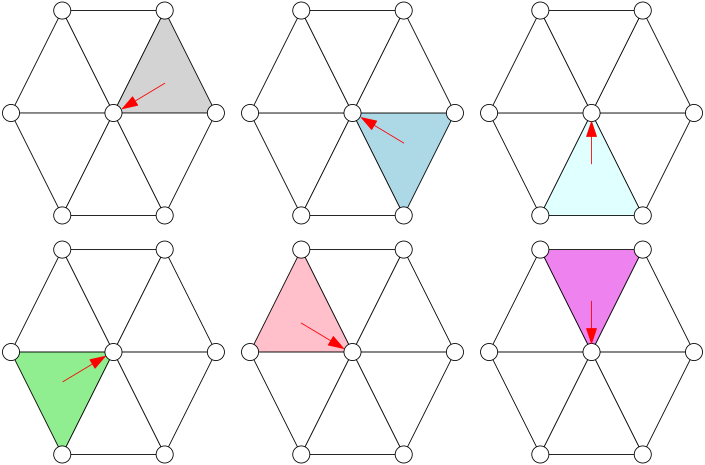
Graph Coloring

Give a different color to each of these elements. And gather them in list of elements of the same color



Graph Coloring

Treat each element in a color set in parallel.



2
Runtime systems

Applications using runtimes

Runtime systems are becoming widely adopted in scientific computing especially for dense linear algebra libraries :

→ PLASMA (QUARK)

→ DPLASMA (PaRSEC)

→ MAGMA-MORSE (StarPU)

→ FLAME (SuperMatrix)

In sparse linear algebra :

→ `PasTiX` (PaRSEC, StarPU)

→ `qr_mumps` (PaRSEC, StarPU)

StarPU

The `StarPU` runtime system :

$\rightarrow$ dynamically schedules tasks on all processing units

$\Rightarrow$ see a pool of heterogeneous processing units;

$\rightarrow$ avoids unnecessary data transfers between accelerators

$\Rightarrow$ software VSM for heterogeneous machines;

$\rightarrow$ open scheduling platform

$\Rightarrow$ different schedulers to meet different needs.

$\rightarrow$ `StarPU` will infer the dependencies between the tasks.

StarPU

An example :

```
do i=1, N
    call toto( x, y )
end do
```

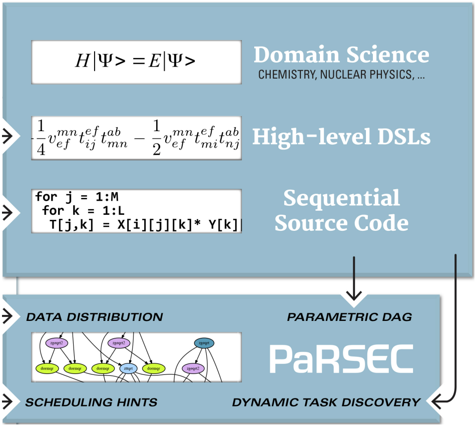
The example with StarPU :

```
do i=1, N
    call starpu_submit( toto, x, y )
end do
call starpu_wait_all()
```

PaRSEC: focus on the algorithm

Concepts

- Clear separation of concerns: compiler optimize each tasks, developer describe dependencies between tasks, the runtime orchestrate the dynamic execution
- Interface with the application developers through specialized domain specific languages (PTG, insert_task, fork/join, …)
- Separate algorithms from data (dataflow)
- Make control flow executions a relic



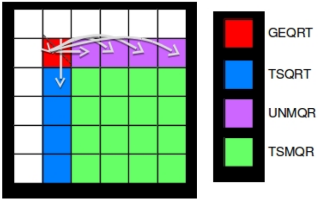
Runtime

- Permeable portability layer for heterogeneous architectures
- Scheduling policies adapt every execution to the hardware & ongoing system status
- Data movements between consumers are inferred from dependencies. Communications/ computations overlap naturally unfold
- Coherency protocols minimize data movements
- Memory hierarchies (including NVRAM and disk) integral part of the scheduling decisions

PaRSEC: Symbolic representation (Parameterized DAG)

```
GEQRT(k)
 /* Execution space */
 k = 0..( MT < NT ) ? MT-1 : NT-1 )
 /* Locality */
 : A(k, k)
 RW     A <- (k == 0)     ? A(k, k)
                          : A1 TSMQR(k-1, k, k)
         -> (k < NT-1)   ? A UNMQR(k, k+1 .. NT-1)  [type = LOWER]
         -> (k < MT-1)   ? A1 TSQRT(k, k+1)         [type = UPPER]
         -> (k == MT-1) ? A(k, k)                   [type = UPPER]
 WRITE T <- T(k, k)
         -> T(k, k)
         -> (k <  NT-1) ? T UNMQR(k, k+1 .. NT-1)
 /* Priority */
 ;(NT-k)*(NT-k)*(NT-k)

BODY
   zgeqrt( A, T )
END
```



Advantages

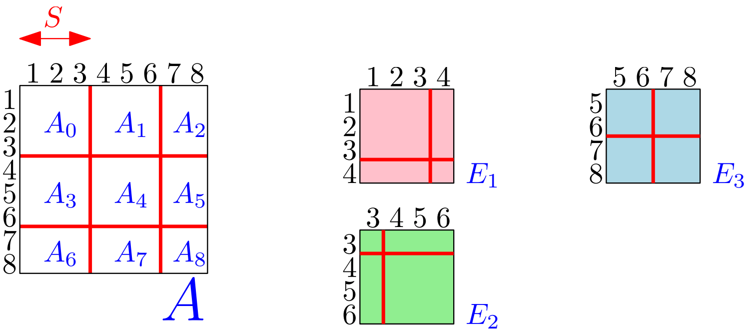
- Local view of the DAG
- No need to unroll the full graph
- Full knowledge of Input/Output for each task
- Separation of concerns (Data/task distribution, algorithm)

3
Taskified Assembly

Flat strategy

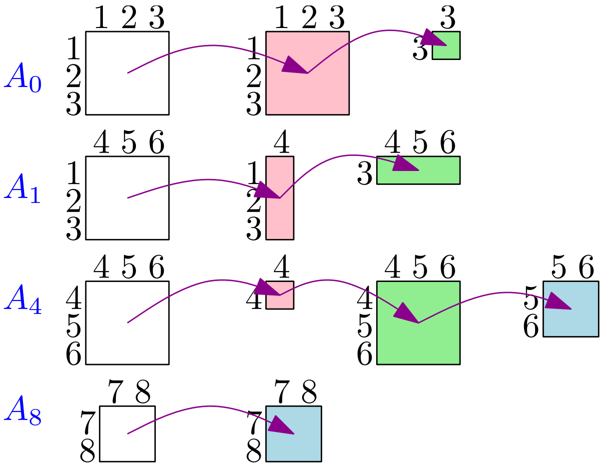
→ The matrix is divided (2D) in blocks using a fix size.

→ Elements blocks are divided according to the matrix.



Flat strategy

→ Every blocks overlapping the same block of the matrix are gathered in a chain.

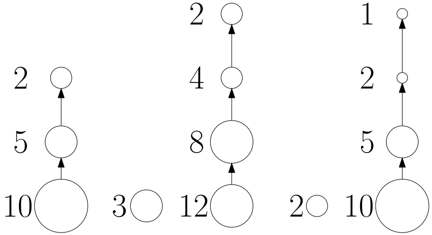


Adaptive strategy

→ Same chain as the flat strategy.

→ Every chain is weighted according to the amount of computation required to assemble it.

→ The weight of each chain serves as a priority.



4
Results

2 applications

Experiments

Runs done on the `riri` platform :

→ 4 INTEL E7-4870 processors of 10 cores clocked at 2,40 GHz ⇒ **40 cores**;

→ 30MB of L3 cache;

→ uniform memory access (**UMA**) to its 1TB of RAM.

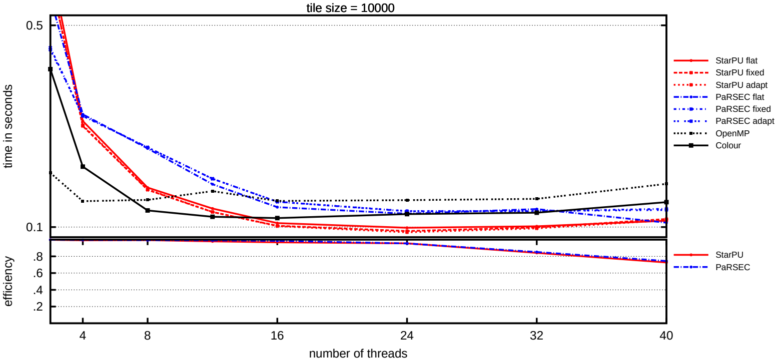
1 finite element experiment :

→ 3D grid with big blocks, a high number of elements overlapping the same rows of the matrix;

1 sparse direct solver experiment :

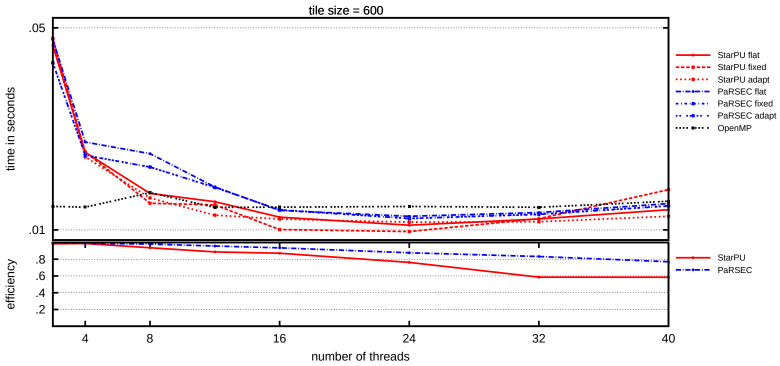
→ sub-problem from `MUMPS` ⇒ Blocks with a size of 1% to 100% of the matrix size, and very irregular overlapping between blocks.

Results in 3D



Assembly of 512 blocks of size 512^2 in a matrix with 185k entries.

Results for MUMPS case



Assembly of 42 blocks of size varying from 1 to 4279 in a matrix with 4279 entries.

5 Conclusion

Perspectives

Conclusions

→ The assembly operation is a core kernel of the numerical application

→ We proposed a new taskified assembly operation scheme.

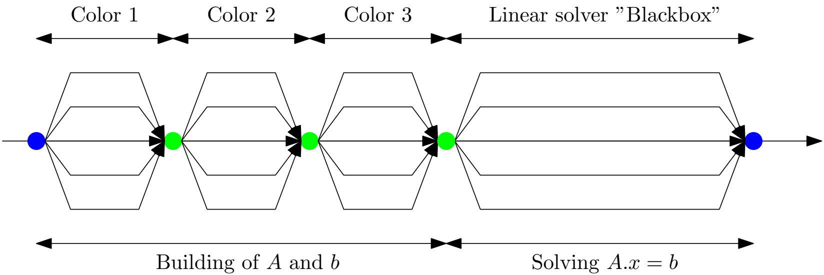
→ The behaviour of our approach is closed to the state-of-the-art approach while fully relying on a runtime system.

→ Exploit the intra-block parallelism.

→ This work is the first step to a fully taskified simulation application.

Perspectives

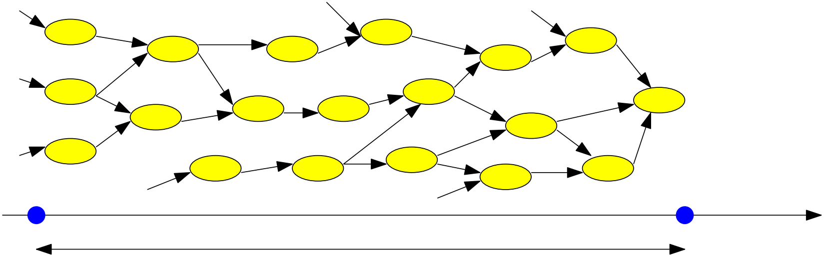
- → We have too many synchronization points using the fork-join paradigm.
- → We want to express the whole chain in a dataflow to suppress those points.



Perspectives

→ We have too many synchronization points using the fork-join paradigm.

→ We want to express the whole chain in a dataflow to suppress those points.



Continuous dataflow for the assembly of the system and its resolution

THANKS!