RUNNING SLATE USING THE PARSEC RUNTIME SYSTEM

Florent Lopez
Lunch talk, October 2020

Innovative computing laboratory, University of Tennessee

Slate:

- Parallel implementation of LAPACK, ScaLAPACK and BLAS routines
- C++ API
- Capable of exploiting multicore CPUs and GPU devices
- MPI, OpenMP and CUDA

Slate and Parsec

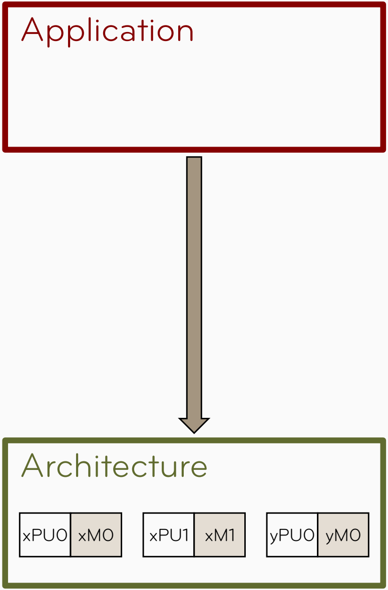
Slate:

- Parallel implementation of LAPACK, ScaLAPACK and BLAS routines
- C++ API
- Capable of exploiting multicore CPUs and GPU devices
- MPI, OpenMP and CUDA

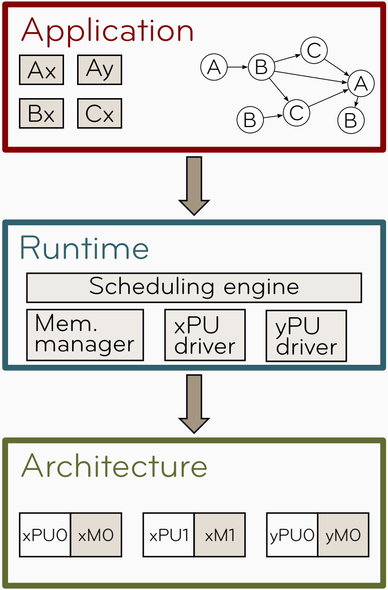
Parsec:

- General purpose runtime system
- Supports various domain specific language such as DTD, JDF and TTG
- Asynchronous distributed-memory communication engine
- GPU driver currently supporting NVIDIA GPUs
- Capable of managing data consistency across various memory nodes

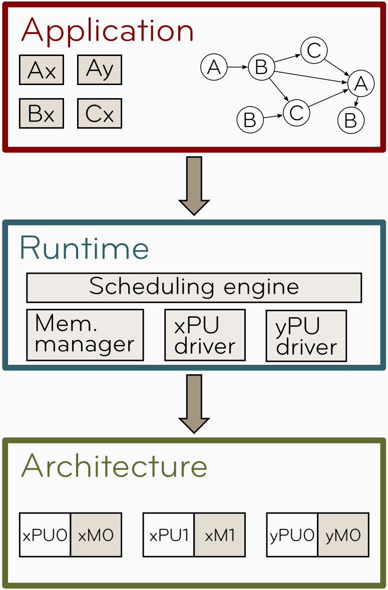
Objective: Integrate the PaRSEC runtime system into the Slate library as an alternative to the standard MPI+OpenMP+CUDA model.



- 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.



- 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.
- Runtime systems provide an abstraction layer that hides the architecture details:
  - Portable programming interface
  - Data management
  - Support for multiple architectures
  - Customizable scheduling strategies



- 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.
- Runtime systems provide an abstraction layer that hides the architecture details:
  - Portable programming interface
  - Data management
  - Support for multiple architectures
  - Customizable scheduling strategies
- The workload is expressed as a DAG (Directed Acyclic Graph) of tasks.

Objectives

Objective: Integrate the PaRSEC runtime system into the Slate library as an alternative to the standard MPI+OpenMP+CUDA model.

Objective: Integrate the PaRSEC runtime system into the Slate library as an alternative to the standard MPI+OpenMP+CUDA model.

- Ability to exploit of fine-grain parallelism.
- Let the runtime system handle memory transfers and data coherency.
- Maximize computation-communication overlapping while benefiting from collective communications.
- Enable task pipelining between sequence of routine calls.

Objectives

Objective: Integrate the PaRSEC runtime system into the Slate library as an alternative to the standard MPI+OpenMP+CUDA model.

- Ability to exploit of fine-grain parallelism.
- Let the runtime system handle memory transfers and data coherency.
- Maximize computation-communication overlapping while benefiting from collective communications.
- Enable task pipelining between sequence of routine calls.

Make this integration as transparent as possible for the Slate developers:

- High-level algorithm description.
  - Issue with hard-coded OpenMP directive at the higher level.
- Data-distribution: 2D block-cyclic.
- GPU task mapping.

This approach has been assessed in the domain of dense linear algebra and sparse algorithms.

- Programming models: Sequential Task Flow (STF), Parametrized Tasks Graph (PTG)
- Runtime systems: PaRSEC, StarPU, QUARK, IntelTBB
- Scientific libraries: (D)PLASMA, Chameleon, SyLVER, `qr_mumps`, PasTiX

Slate potrf

Slate algorithms relies on control flow dependencies and nested parallelism

```
for (p = 0; p < nc; ++p) {
   Factor(p);

   for (u = p+1; u < nc; ++u) {
      Update(p, u)                         ;
   }
}
```

- At the higher level Slate expresses dependencies using a 1D block partitioning

Slate algorithms relies on control flow dependencies and nested parallelism

```
for (p = 0; p < nc; ++p) {
   Factor(p);

   for (u = p+1; u < nc; ++u) {
      Update(p, u)                    ;
   }
}
```

- At the higher level Slate expresses dependencies using a 1D block partitioning
- `Factor(p)` and `Update(p, u)` are executed asynchronously

Slate potrf

Slate algorithms relies on control flow dependencies and nested parallelism

```
for (p = 0; p < nc; ++p) {
   Factor(p);

   for (u = p+1; u < nc; ++u) {
      Update(p, u).depends_on(Facor (p));
   }
}
```

- At the higher level Slate expresses dependencies using a 1D block partitioning
- `Factor(p)` and `Update(p, u)` are executed asynchronously
- Dependencies between panel factorization and trailing submatrix update are expressed explicitly to the runtime system

Slate algorithms relies on control flow dependencies and nested parallelism

```
for (p = 0; p < nc; ++p) {
   Factor(p);
   send(data(p), processes 1 to q);
   for (u = p+1; u < nc; ++u) {
      Update(p, u).depends_on(Facor (p));
   }
}
```

- At the higher level Slate expresses dependencies using a 1D block partitioning
- `Factor(p)` and `Update(p, u)` are executed asynchronously
- Dependencies between panel factorization and trailing submatrix update are expressed explicitly to the runtime system
- Tasks are not associated with real data (factors) but symbolic ones: data transfer must be done explicitly i.e send if the tile is local, receive otherwise

Nested parallel within Factor and Update

```
Factor(p) {

   Potrf(tile(p,p));



   for (r = p+1; r < nr; ++r) {

      Trsm(tile(p,p), tile(r,p));
   }

}
```

- Numerical operations execute on underlying 2D data partitioning

Nested parallel within Factor and Update

```
Factor(p) {
#omp task
   Potrf(tile(p,p));



   for (r = p+1; r < nr; ++r) {
#omp task
      Trsm(tile(p,p), tile(r,p));
   }

}
```

- Numerical operations execute on underlying 2D data partitioning
- Execution of independent tasks is done asynchronously using OpenMP

Nested parallel within Factor and Update

```
Factor(p) {
#omp task
    Potrf(tile(p,p));
    wait_for_all();


    for (r = p+1; r < nr; ++r) {
#omp task
        Trsm(tile(p,p), tile(r,p));
    }
    wait_for_all();
}
```

- Numerical operations execute on underlying 2D data partitioning
- Execution of independent tasks is done asynchronously using OpenMP
- Dependency are expressed through synchronization barriers

Nested parallel within Factor and Update

```
Factor(p) {
#omp task
    Potrf(tile(p,p));
    wait_for_all();
    send(tile(p,p)); or recieve(tile(p,p));

    for (r = p+1; r < nr; ++r) {
#omp task
        Trsm(tile(p,p), tile(r,p));
    }
    wait_for_all();
}
```

- Numerical operations execute on underlying 2D data partitioning
- Execution of independent tasks is done asynchronously using OpenMP
- Dependency are expressed through synchronization barriers
- Data transfer must be done explicitly

Nested parallel within Factor and Update

```
Factor(p) {
#omp task
    Potrf(tile(p,p));
    wait_for_all();
    send(tile(p,p)); or recieve(tile(p,p));
    wait_for_all();
    for (r = p+1; r < nr; ++r) {
#omp task
        Trsm(tile(p,p), tile(r,p));
    }
    wait_for_all();
}
```

- Numerical operations execute on underlying 2D data partitioning
- Execution of independent tasks is done asynchronously using OpenMP
- Dependency are expressed through synchronization barriers
- Data transfer must be done explicitly

Sequential Task Flow (STF) model:

Sequential Task Flow (STF) model:

1) Task submission follows the sequential algorithm

Sequential Task Flow (STF) model:

1) Task submission follows the sequential algorithm
2) Tasks are associated with data (manipulated within computational kernels) along with access mode (Read, Write, Read-Write) and shape (dimensions, stride, type, etc) information

Sequential Task Flow (STF) model:

1) Task submission follows the sequential algorithm
2) Tasks are associated with data (manipulated within computational kernels) along with access mode (Read, Write, Read-Write) and shape (dimensions, stride, type, etc) information

$\Rightarrow$ Task dependencies are inferred to ensure the sequential consistency of the parallel code

Sequential Task Flow (STF) model:

1) Task submission follows the sequential algorithm
2) Tasks are associated with data (manipulated within computational kernels) along with access mode (Read, Write, Read-Write) and shape (dimensions, stride, type, etc) information

⇒ Task dependencies are inferred to ensure the sequential consistency of the parallel code

⇒ Data transfers can be automatically issued by the runtime system

Using STF model within Slate

```
Factor(p) {
#omp task
   Potrf(tile(p,p));
   wait_for_all();
   send(tile(p,p)); or recieve(tile(p,p));
   wait_for_all();
   for (r = p+1; r < nr; ++r) {
#omp task
      Trsm(tile(p,p), tile(r,p));
   }
   wait_for_all();
}
```

Using STF model within Slate

```
Factor(p) {

   insert(Potrf, tile(p,p)   );
   wait_for_all();
   send(tile(p,p)); or recieve(tile(p,p));
   wait_for_all();
   for (r = p+1; r < nr; ++r) {

      insert(Trsm, tile(p,p)  , tile(r,p)   );
   }
   wait_for_all();
}
```

- Use Parsec to asynchronously execute the task on tiles

Using STF model within Slate

```
Factor(p) {

   insert(Potrf, tile(p,p):RW);
   wait_for_all();
   send(tile(p,p)); or recieve(tile(p,p));
   wait_for_all();
   for (r = p+1; r < nr; ++r) {

      insert(Trsm, tile(p,p):R, tile(r,p):RW);
   }
   wait_for_all();
}
```

- Use Parsec to asynchronously execute the task on tiles
- Add data access and shape information to the runtime system

Using STF model within Slate

```
Factor(p) {

   insert(Potrf, tile(p,p):RW);

   send(tile(p,p)); or recieve(tile(p,p));
   wait_for_all();
   for (r = p+1; r < nr; ++r) {

      insert(Trsm, tile(p,p):R, tile(r,p):RW);
   }

}
```

- Use Parsec to asynchronously execute the task on tiles
- Add data access and shape information to the runtime system
- Remove explicit synchronization barrier

Using STF model within Slate

```
Factor(p) {

   insert(Potrf, tile(p,p):RW);



   for (r = p+1; r < nr; ++r) {

      insert(Trsm, tile(p,p):R, tile(r,p):RW);
   }

}
```

- Use Parsec to asynchronously execute the task on tiles
- Add data access and shape information to the runtime system
- Remove explicit synchronization barrier
- Remove explicit data transfer routines

The DPlasma library uses a PTG model.

The DPlasma library uses a PTG model.

Parametrized Task Graph (PTG) programming model:

- The DAG is represented using a compact format which is problem size independent.

The DPlasma library uses a PTG model.

Parametrized Task Graph (PTG) programming model:

- The DAG is represented using a compact format which is problem size independent.
- The dataflow is explicitly encoded i.e. task dependencies are explicitly given to the runtime system.

The Parametrized Task Graph Model

The DPlasma library uses a PTG model.

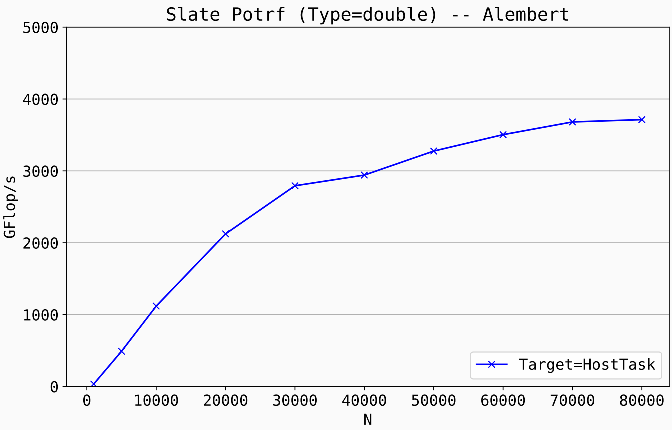
Parametrized Task Graph (PTG) programming model:

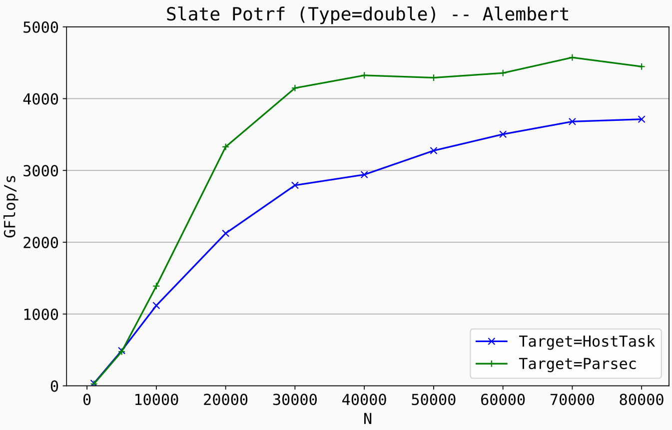
- The DAG is represented using a compact format which is problem size independent.
- The dataflow is explicitly encoded i.e. task dependencies are explicitly given to the runtime system.
- The runtime handles the communications implicitly using the dataflow representation.

System Alembert (Saturn):

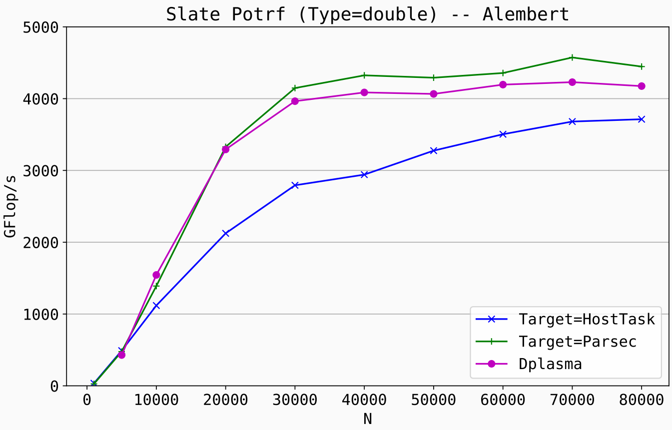
- Haswell E5-2650 v3 @ 2.30GHz, 2×10 cores
- 64GB RAM (NUMA)
- 9 nodes
- Infiniband EDR 100G

- Compiler GCC 7.3
- BLAS and LAPACK libraries from Intel MKL 19.3
- Open MPI 3.0


Slate Potrf (Type=double) -- Alembert
GFlop/s
0
1000
2000
3000
4000
5000
0
10000
20000
30000
40000
50000
60000
70000
80000
N
Target=HostTask



- The Parsec target compares favourably to Slate implementation

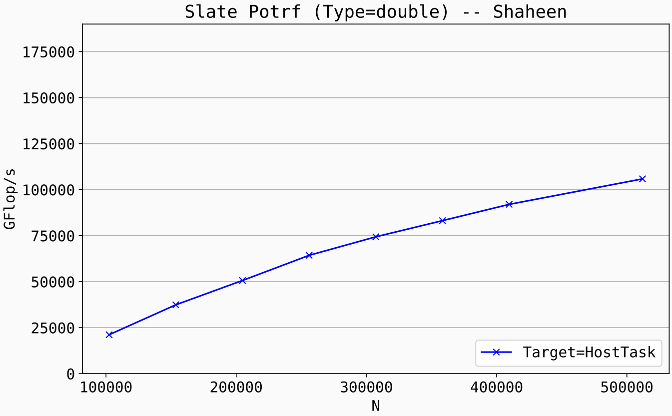


- The Parsec target compares favourably to Slate implementation
- It outperforms DPlasma

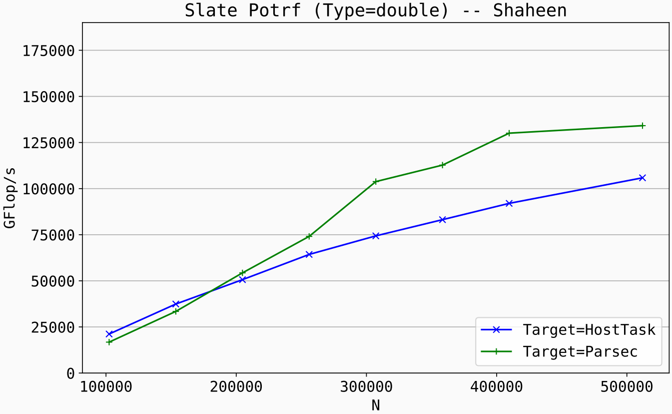
System Shaheen (Kaust):

- Intel Haswell @ 2.30GHz, 2×16 cores
- 128GB RAM (NUMA)
- 256 nodes (8192 cores)
- Cray Aries interconnect with Dragonfly topology

- Intel C++ compiler
- BLAS and LAPACK libraries from Intel MKL 19.3
- Cray MPI (MPICH3)

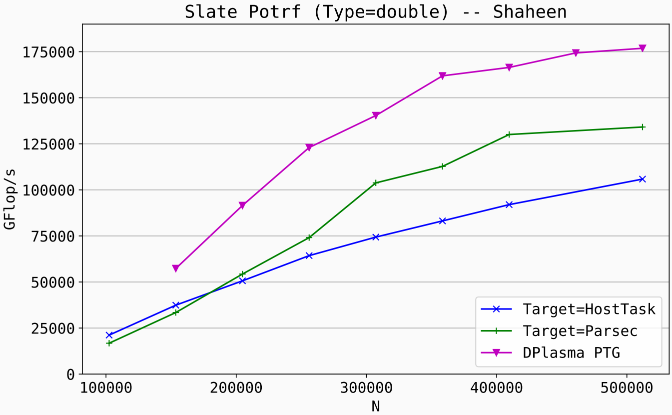

Slate Potrf (Type=double) -- Shaheen
GFlop/s
0
25000
50000
75000
100000
125000
150000
175000
100000
200000
300000
400000
500000
N
Target=HostTask

Numerical experiments: Shaheen



- The Parsec target compares favourably to Slate implementation

Numerical experiments: Shaheen

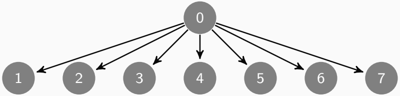


- The Parsec target compares favourably to Slate implementation
- Performs poorly compared to DPlasma

Issue: The DTD interface lacks collectives communication

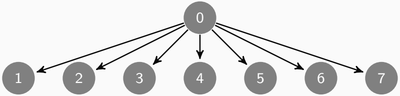
Issue: The DTD interface lacks collectives communication

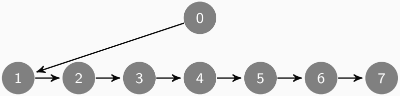
Use of collective communication patterns that better fits the network topology e.g. Broadcast



Issue: The DTD interface lacks collectives communication

Use of collective communication patterns that better fits the network topology e.g. Broadcast

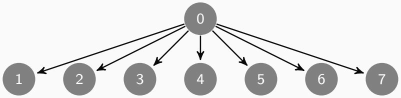


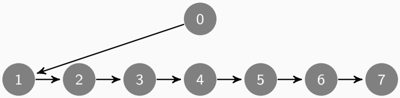


Chain

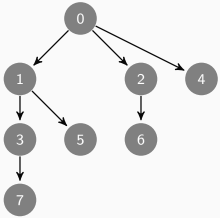
Issue: The DTD interface lacks collectives communication

Use of collective communication patterns that better fits the network topology e.g. Broadcast





Chain



Binomial tree

Using STF model within Slate and integrating collective communications

```
Factor(p) {
   Potrf(tile(p,p):RW);




   for (r = p+1; r < nr; ++r) {
      Trsm(tile(p,p):R, tile(r,p):RW);
   }

}
```

Using STF model within Slate and integrating collective communications

```
Factor(p) {
    Potrf(tile(p,p):RW);

    insert(send, tile(p,p)  ); or insert(receive, tile(p,p)   );

    for (r = p+1; r < nr; ++r) {
        Trsm(tile(p,p):R, tile(r,p):RW);
    }

}
```

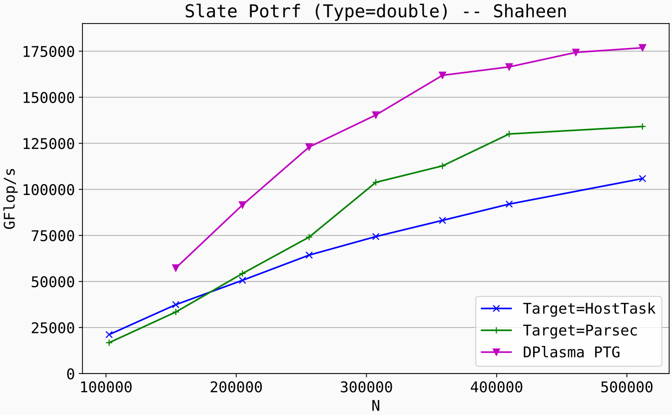
- Integrate the communication information in the task flow by inserting a task in the DAG

Slate-Parsec potrf

Using STF model within Slate and integrating collective communications

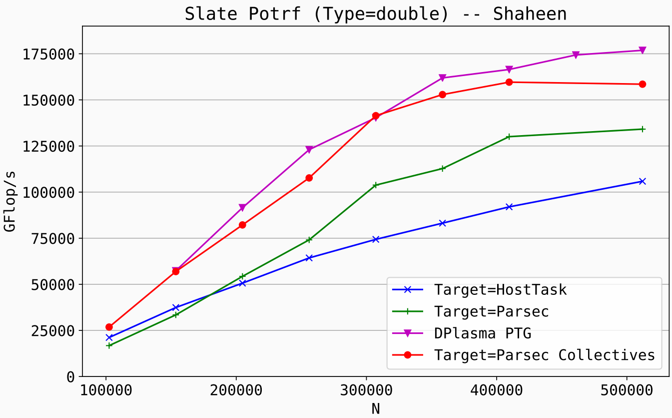
```
Factor(p) {
   Potrf(tile(p,p):RW);

   insert(send, tile(p,p):R); or insert(receive, tile(p,p):RW);

   for (r = p+1; r < nr; ++r) {
      Trsm(tile(p,p):R, tile(r,p):RW);
   }

}
```

- Integrate the communication information in the task flow by inserting a task in the DAG
- Communication in non-blocking but data access information guarantee sequential consistency



- The Parsec target compares favourably to Slate (main branch) implementation

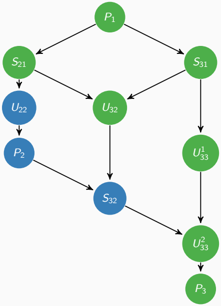
Numerical experiments: Shaheen



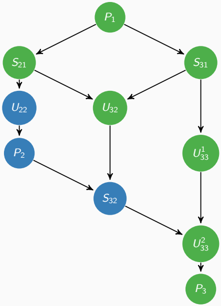
- The Parsec target compares favourably to Slate (main branch) implementation
- The Parsec target performance is dramatically improved by using collective communication patterns

In a distributed memory context, one limitation of the DTD interface lies in the fact that the whole DAG is unrolled on every nodes.

In a distributed memory context, one limitation of the DTD interface lies in the fact that the whole DAG is unrolled on every nodes.

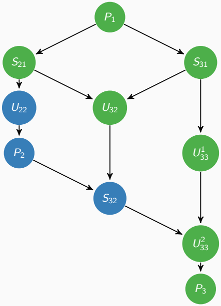


Node 0

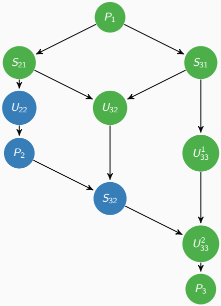


Node 1

In a distributed memory context, one limitation of the DTD interface lies in the fact that the whole DAG is unrolled on every nodes.



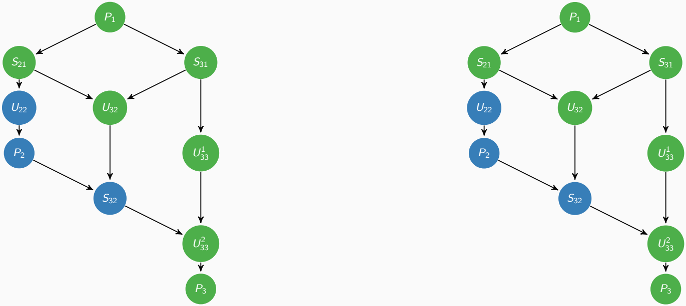
Node 0



Node 1

- The number of local tasks in the factorization only represent a fraction ($\frac{1}{np}$) of the total tasks $O((\frac{n}{nb})^3)$

In a distributed memory context, one limitation of the DTD interface lies in the fact that the whole DAG is unrolled on every nodes.



Node 0

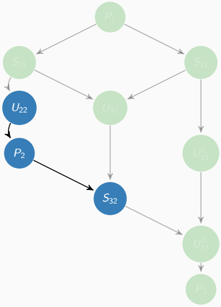
Node 1

- The number of local tasks in the factorization only represent a fraction $(\frac{1}{np})$ of the total tasks $O((\frac{n}{nb})^3)$
- The cost for the task creation and submission is not negligible for large matrix size and when increasing the number of processes

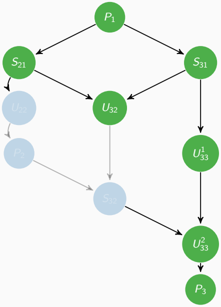
DAG trimming

DAG trimming: only create and insert tasks that are executed locally

DAG trimming: only create and insert tasks that are executed locally



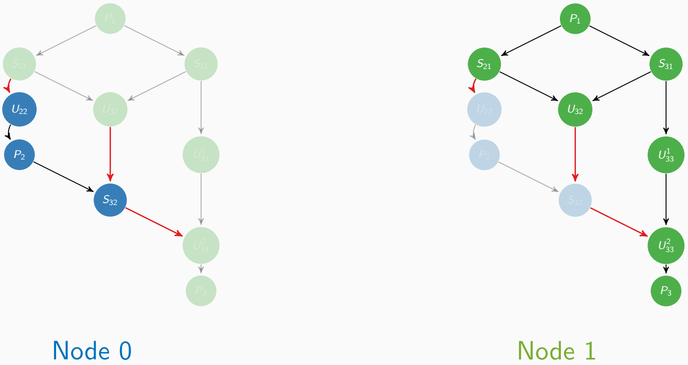
Node 0



Node 1

DAG trimming

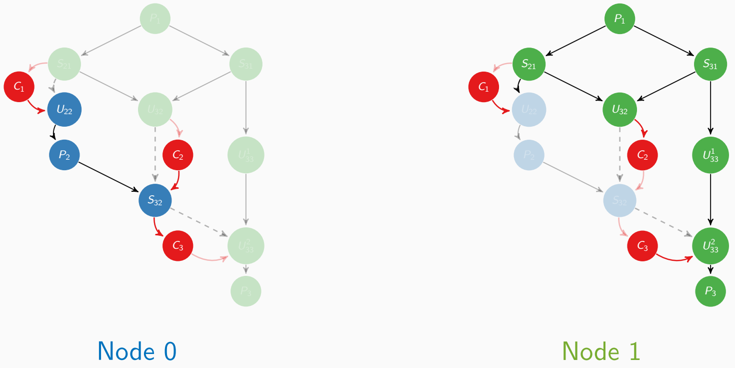
DAG trimming: only create and insert tasks that are executed locally



- Problem: Only partial information available locally, matching message tags becomes difficult

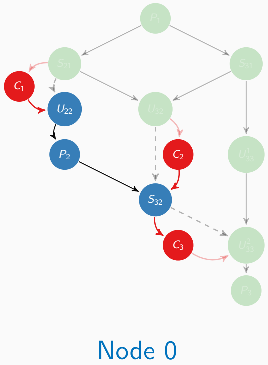
DAG trimming

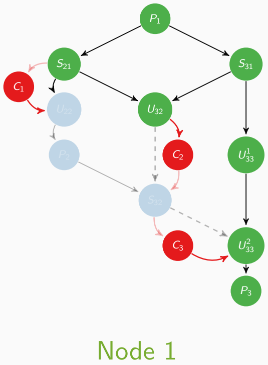
DAG trimming: only create and insert tasks that are executed locally



- Problem: Only partial information available locally, matching message tags becomes difficult
- Locally insert remote tasks responsible for communications on every nodes: globally consistent message identification

DAG trimming: only create and insert tasks that are executed locally





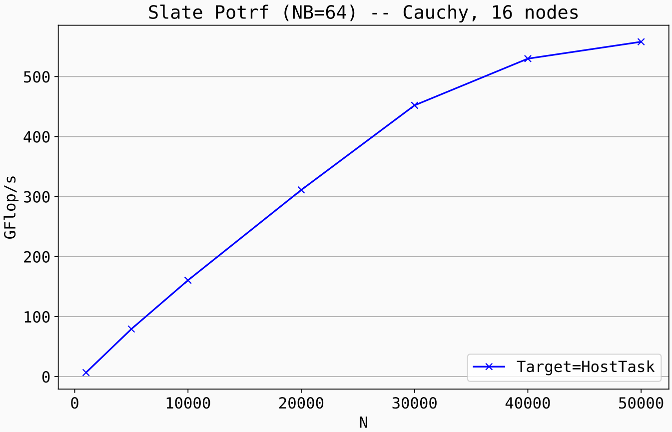
- Problem: Only partial information available locally, matching message tags becomes difficult
- Locally insert remote tasks responsible for communications on every nodes: globally consistent message identification
- Slate provide this information to the runtime system

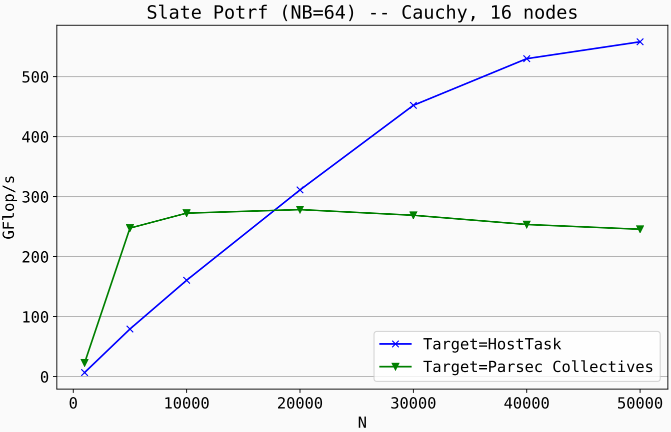
System Cauchy (Saturn):

- Westmere-EP E5606 @2.13GHz, 2×4 cores
- 24GB RAM (NUMA)
- 16 nodes
- Infiniband SDR 10G

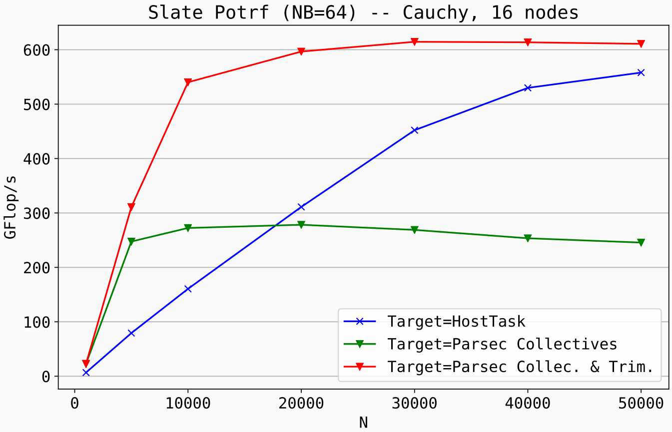
- Compiler GCC 7.3
- BLAS and LAPACK libraries from Intel MKL 19.3
- Open MPI 3.0

Numerical experiments: Cauchy





- Using a relatively small tile size, the time spent for the task creation and submission becomes greater than the execution of local tasks



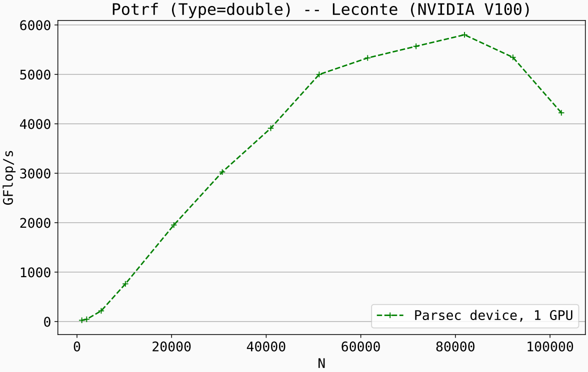
- Using a relatively small tile size, the time spent for the task creation and submission becomes greater than the execution of local tasks
- Trimming the DAG allows us to alleviate this issue thus improving the scalability of the factorization

System Leconte (Saturn):

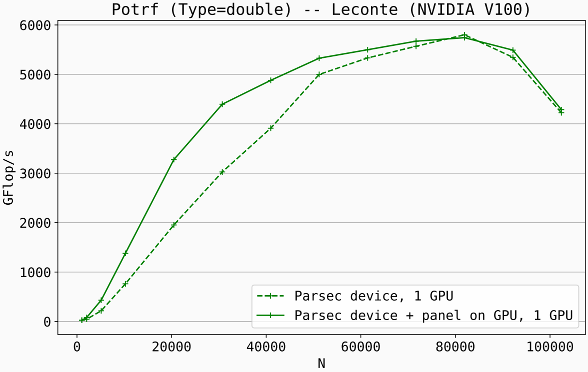
- Broadwell Intel CPU E5-2698 v4 @ 2.20GHz, 2×20 cores
- 500GB RAM (NUMA)
- 8 NVIDIA V100 GPUs

- Compiler GCC 8.4
- BLAS and LAPACK libraries from Intel MKL 20.0
- NVIDIA CUDA 10.2

- Panel factorization and update on GPU
- Using one GPU

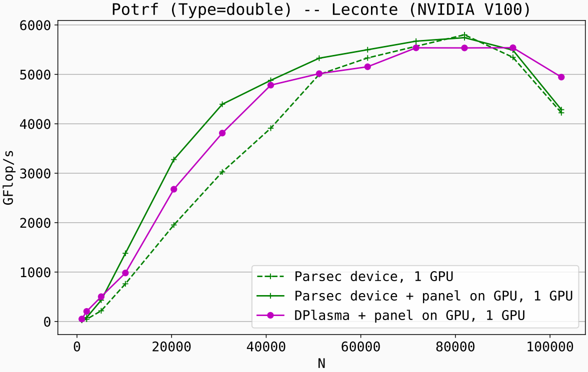


- Panel factorization and update on GPU
- Using one GPU



- Scalability improved by executing panel on GPU

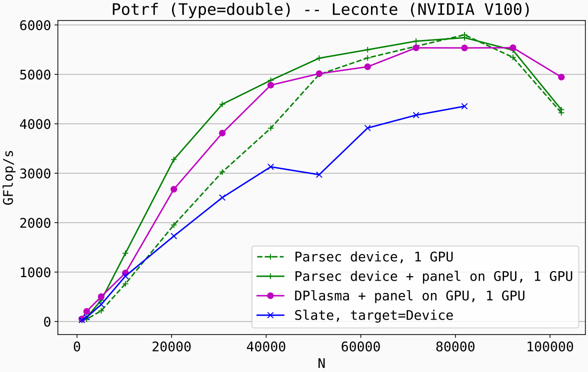
- Panel factorization and update on GPU
- Using one GPU



- Scalability improved by executing panel on GPU
- The Parsec target Compares favourably to current DPlasma code

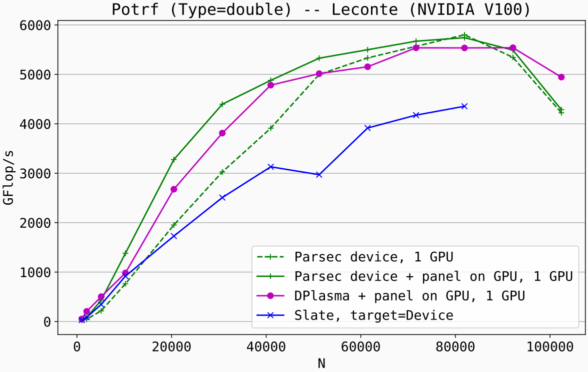
Numerical experiments: Leconte

- Panel factorization and update on GPU
- Using one GPU



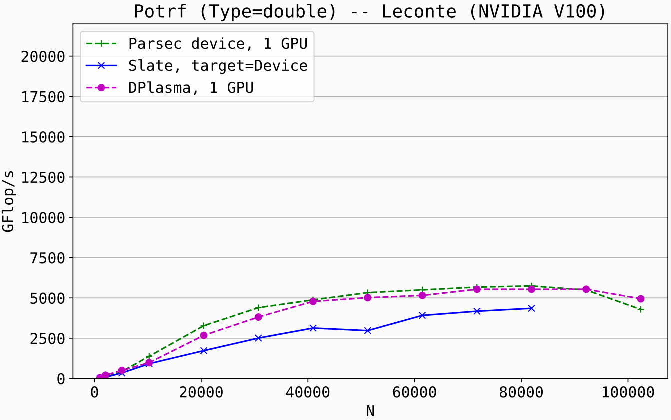
- Scalability improved by executing panel on GPU
- The Parsec target Compares favourably to current DPlasma code
- Slate-Parsec is about 20% faster than the original Slate

- Panel factorization and update on GPU
- Using one GPU

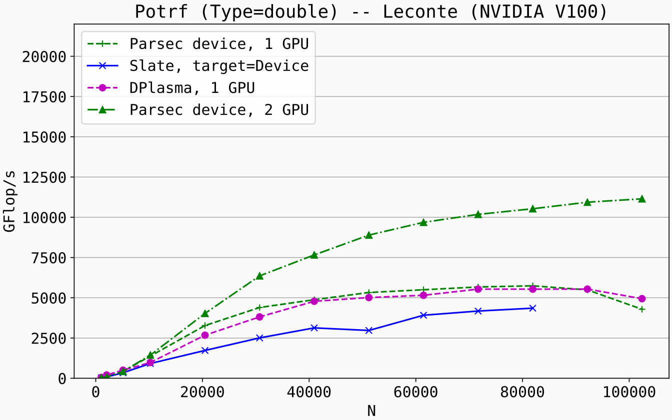


- Scalability improved by executing panel on GPU
- The Parsec target Compares favourably to current DPlasma code
- Slate-Parsec is about 20% faster than the original Slate
- Slate can no longer perform the factorization if the matrix does not fit in the GPU memory

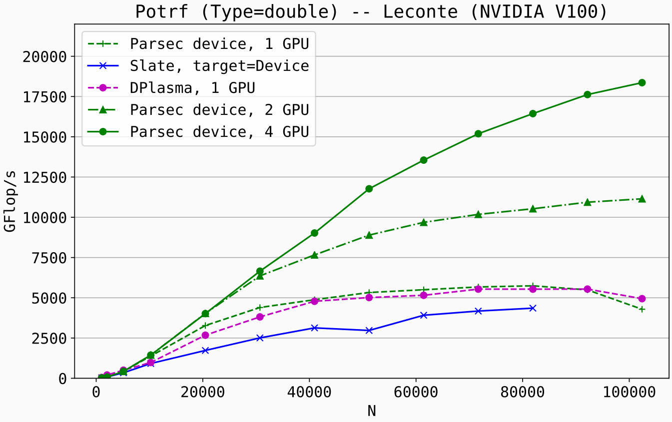
- Panel and update on GPU
- Using multiple GPUs



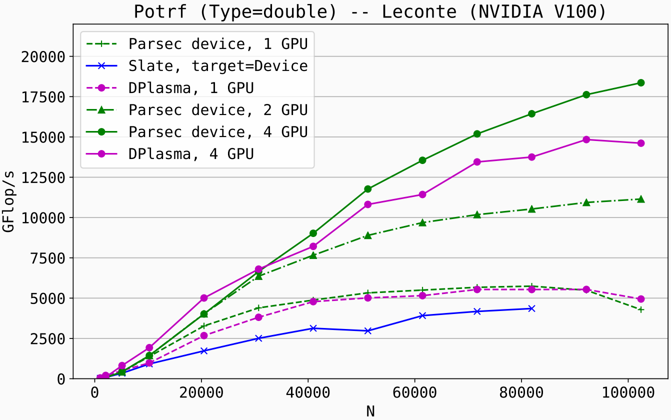
- Panel and update on GPU
- Using multiple GPUs



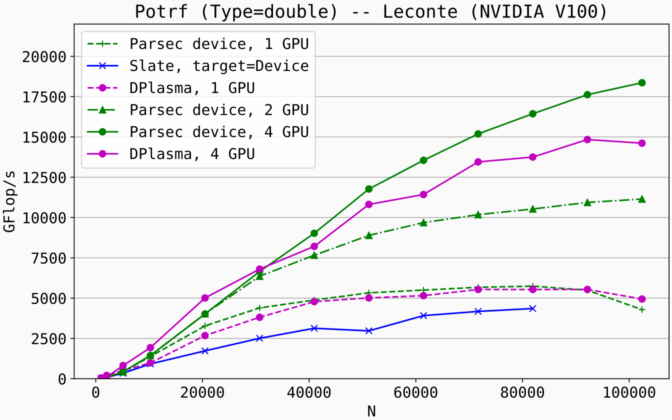
- Panel and update on GPU
- Using multiple GPUs



- Panel and update on GPU
- Using multiple GPUs



- Panel and update on GPU
- Using multiple GPUs



Conclusions:

- Parsec helps to efficiently run Slate algorithms by exploiting fine-grained parallelism and maximizing computation-communication overlap
- Parsec ease data management on heterogeneous architectures

Future work:

- Experiment on GPU-based distributed memory systems (e.g. Summit)
- Add support for AMD GPU architectures in Parsec