Lessons learned from the development of a parallel sparse direct solver

Kyungjoo Kim

Aerospace Engineering and Engineering Mechanics
The University of Texas at Austin

Sep 6, 2013

Outline

1. Source of sparse matrix: *hp*-FEM
2. Parallelization strategy
3. Heterogeneous architectures: multi-level matrix blocking
4. Discussion
5. Summary

Source of sparse matrix: *hp*-Finite Element Method (FEM)

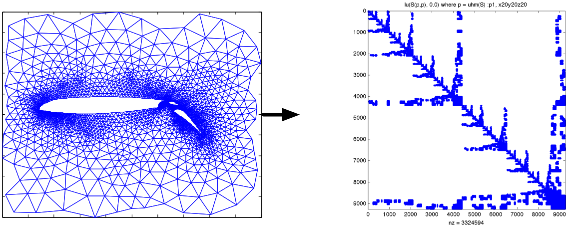


Figure : *A sparse system of equations is generated based on a FE-mesh* [1].

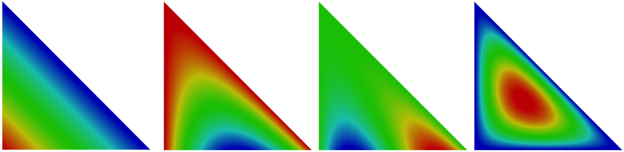
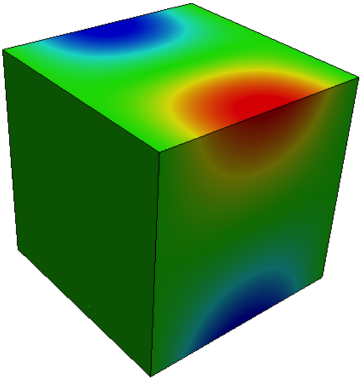


Figure : *Example of high order basis functions.*

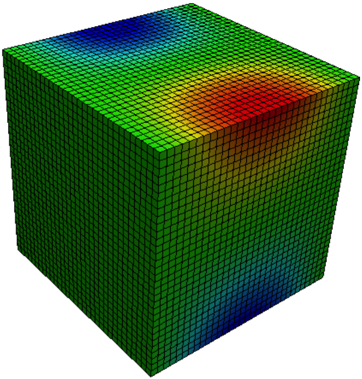
[1] The airfoil mesh is obtained from Matlab.

hp-FEM delivers fast convergence rate

Example: projection of the manufactured solution: $\psi = \sin(x)\cos(y)z$



p=4, # of FEs=1, err=1.27%



p=1, # of FEs=32,768, err=1.19%

Figure : *For a smooth solution, the use of high p delivers a fast convergence rate.*

Application: wave propagation problems

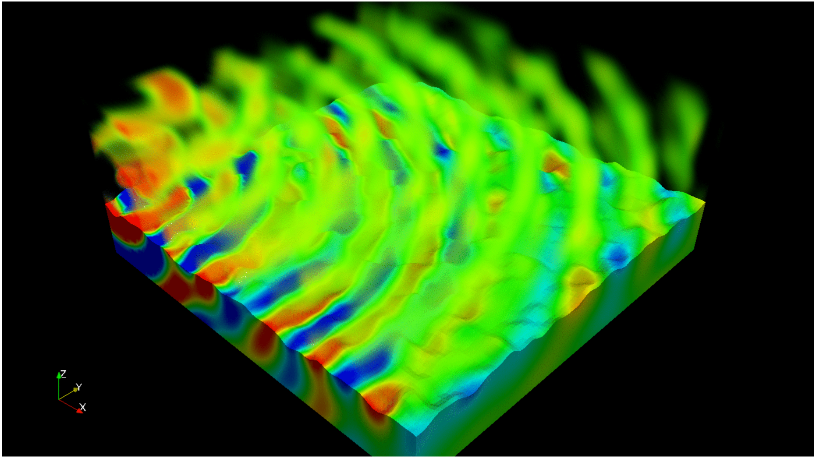


Figure : *Underwater acoustics with a rouch seabed* [2], *approximated by p=6, # of elements= 1,130 (368 in the domain of interest), # of DOFs= 200k.*

[2] The image is produced by Jeffrey Zitelli.

Application: wave propagation problems

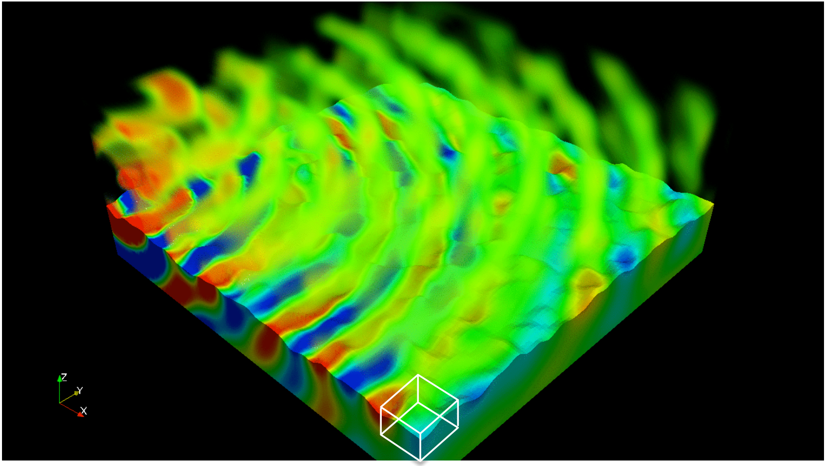


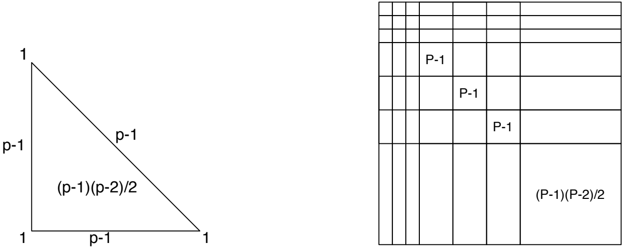
Figure : *Underwater acoustics with a rouch seabed* [2]*, approximated by p=6, # of elements= 1,130 (368 in the domain of interest), # of DOFs= 200k.*

[2] The image is produced by Jeffrey Zitelli.

Sparse system of equations generated by *hp*-FEM

- All operations are essentially dense.

Node Type	Edge	Face	Interior
# of DOFs	$O(p)$	$O(p^2)$	$O(p^3)$



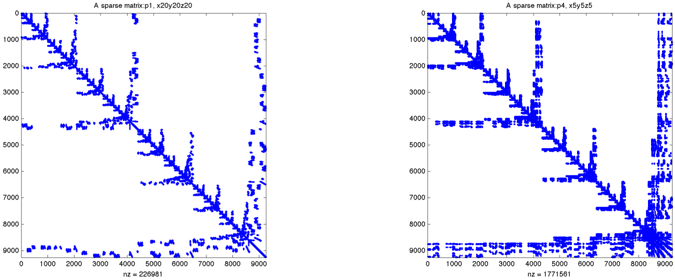
(a) *High order FE* **(b)** *Element matrix*

Figure : *The shape of an unassembled element matrix.*

Sparse system of equations generated by *hp*-FEM

- All operations are essentially dense.

Node Type	Edge	Face	Interior
# of DOFs	$O(p)$	$O(p^2)$	$O(p^3)$



(a) $p = 1$, *nz = 226,981* **(b)** $p = 4$, *nz = 1,771,561*

Figure : *Nonzero patterns keeping the same system DOFs.*

Multifrontal factorization in FEM

- Characterized by **recursive** procedure on the assembly tree.
- Performs **supernodal** elimination and assembly for each frontal matrix.
- Converts the sparse matrix factorization into **multiple dense subproblems**.

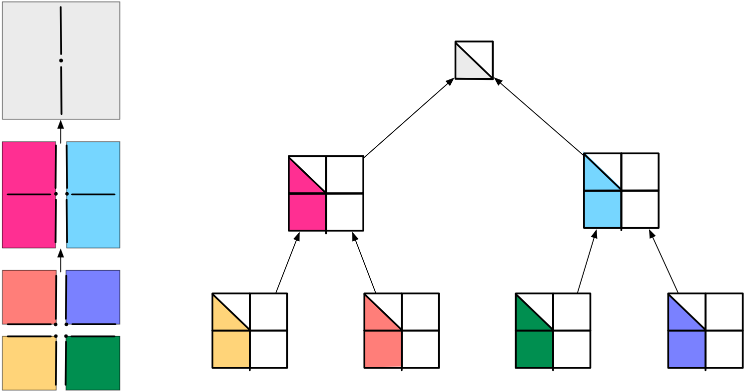
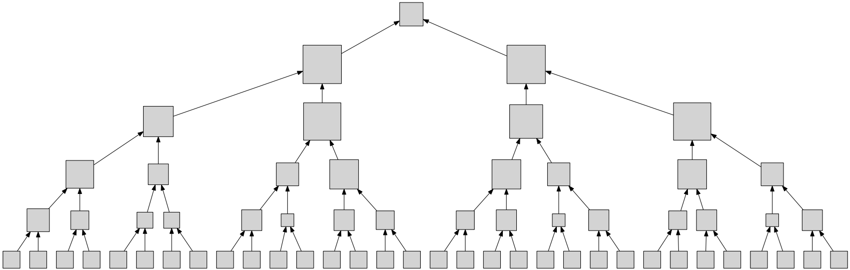


Figure : *The factorization is completed ascending the assembly tree.*

Two-level parallelism

- High degree **tree-level** parallelism on leaves.
- Increasing opportunity in **matrix-level** parallelism.

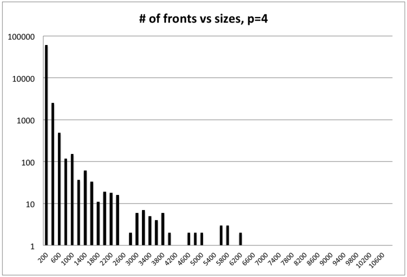


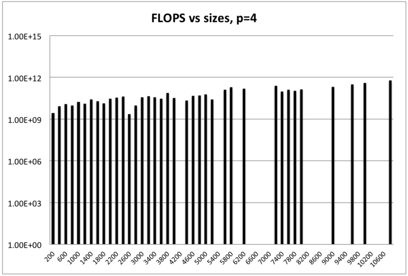
Asynchronous task execution in harmony with two-level parallelism

- Load imbalance due to irregular task sizes.
- Bandwidth-bounded tasks on leaves **vs** compute-bounded tasks nearby the root.

Two-level parallelism

- High degree **tree-level** parallelism on leaves.
- Increasing opportunity in **matrix-level** parallelism.

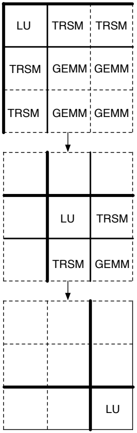




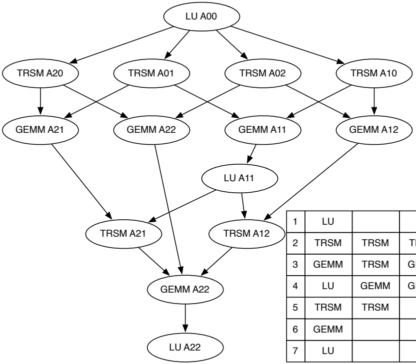
Asynchronous task execution in harmony with two-level parallelism

- Load imbalance due to irregular task sizes.
- Bandwidth-bounded tasks on leaves **vs** compute-bounded tasks nearby the root.

Fine-grained task generation: algorithms-by-blocks



LU



1	LU		
2	TRSM	TRSM	TRSM
3	GEMM	TRSM	GEMM
4	LU	GEMM	GEMM
5	TRSM	TRSM	
6	GEMM		
7	LU		

DAG of tasks

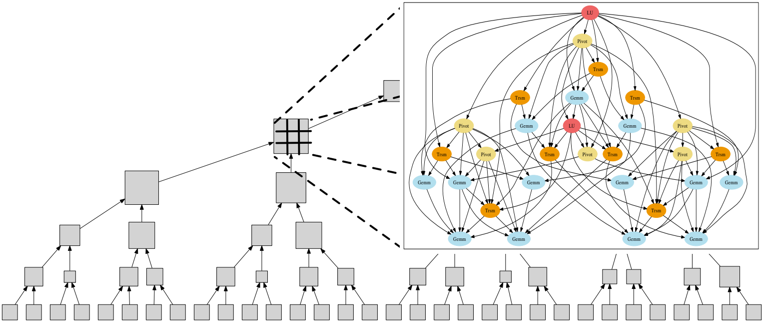
E.Chan *et al.*, 2007., Satisfying your dependencies with Supermatrix.

G.Quintana-Ortí *et al.*, 2009., Programming matrix algorithms-by-blocks for thread-level Parallelism.

Hierarchical DAG scheduling

Tasks are locally analyzed and globally ordered.

- Tree-level tasks (priori known structure) are generated via parallel post-order tree traversal.
- Fine-grained tasks are generated by using algorithms-by-blocks.
- Tasks are hierarchically ordered together with multiple Directed Acyclic Graph (DAG) schedulers.



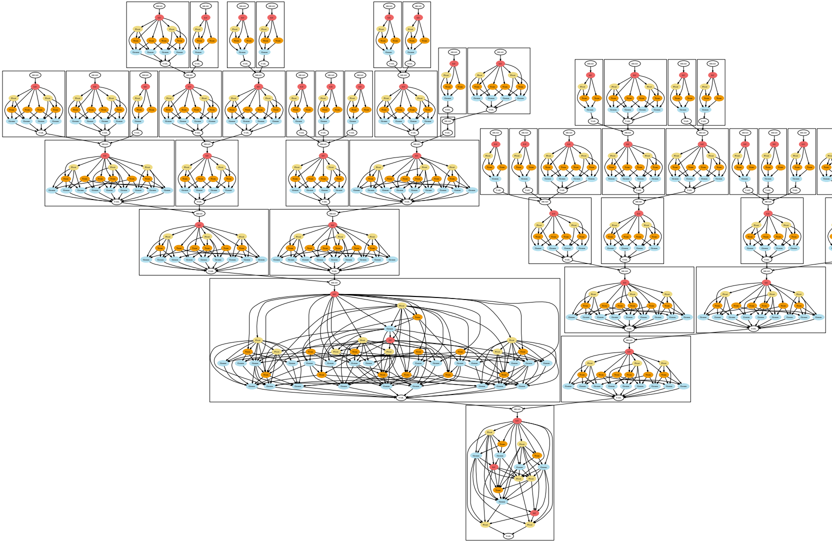


Figure : *An example of hierarchical DAGs.*

Strong scale

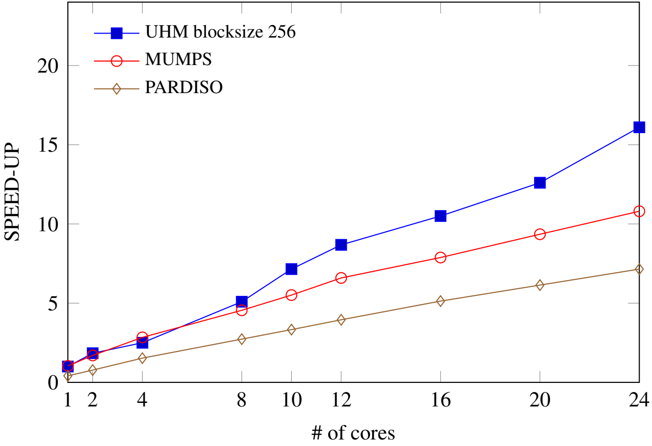


Figure : *Factorization phase for fixed* $p = 4$ *with a reference time of the sequential UHM solver.*

Scheduling tasks to multiple GPUs

We want to achieve portable performance with manageable programming complexity.

Challenges:

- A large front may not fit into a **small device memory (6 GB)**.

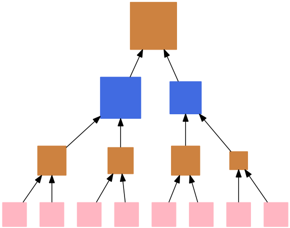
→ A large matrix is decomposed of blocks; only computing blocks are transferred to devices.

- **Different programming models** can be used.

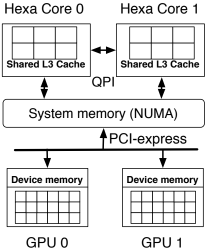
→ Blocks are computed via vendor provided libraries (*e.g.*, MKL, CUBLAS).

- Efficient workload balancing among **asymmetric computing units**.

→ Workloads are dynamically partitioned based on device performance.



Assembly tree



Lonestar @TACC

Scheduling tasks to multiple GPUs

We want to achieve portable performance with manageable programming complexity.

Challenges:

- A large front may not fit into a **small device memory (6 GB)**.

→ A large matrix is decomposed of blocks; only computing blocks are transferred to devices.

- **Different programming models** can be used.

→ Blocks are computed via vendor provided libraries (*e.g.*, MKL, CUBLAS).

- Efficient workload balancing among **asymmetric computing units**.

→ Workloads are dynamically partitioned based on device performance.

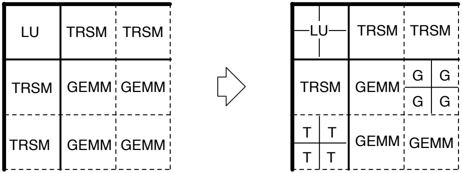
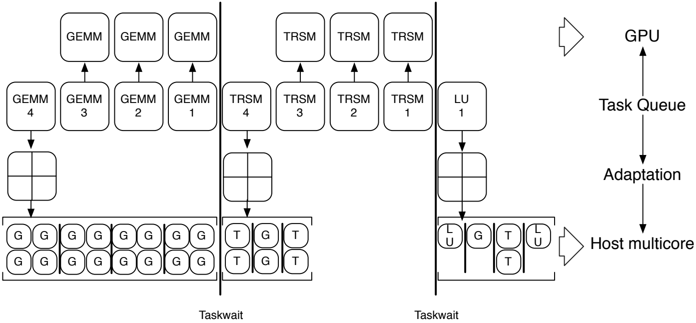


Figure : *Multi-level matrix blocking improves unit performance and efficiency.*

Task handling: bulk-synchronous approach

Suppose that the target architecture has four computing units and one GPU whereas their performance ratio is 1:3.



- Dense subproblems are computed within a sequence of supersteps.
- Each superstep consists of tasks that can be executed concurrently.
- Tasks are dispatched to heterogeneous computing units in a round-robin fashion (easy to exploit multiple GPUs).

Dense problems: two Fermi GPUs

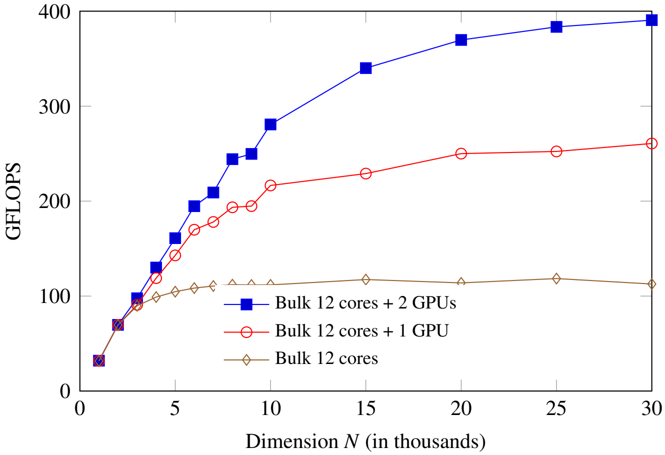


Figure : *Dense LU factorization without pivoting accelerated by multiple GPUs.*

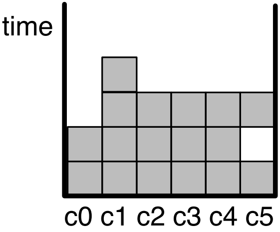
Sparse factorization: two Fermi GPUs

		Performance		Speed-up	
Cores	GPUs	Time [sec]	GFLOP/sec	vs 1 core	vs 12 cores
1	0	291	11.13	1.00	-
2	0	213	15.21	1.36	-
4	0	85	38.11	3.42	-
8	0	45	71.98	6.46	-
12	0	33	98.16	8.81	1.00
12	1	23	140.84	12.65	1.43
12	2	19	170.50	15.31	1.69

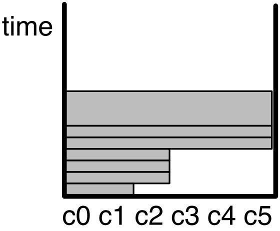
Table : *Sparse LU with partial pivoting accelerated by multiple GPUs.*

Discussion: runtime parallelism vs structured parallelism

How can we put runtime parallelism in harmony with structured parallelism ?



Runtime task parallelism

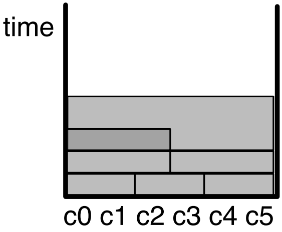


Structured parallelism

	Runtime	Structured
Granularity	Fine	Finer
Concurrency	Out-of-order scheduling	Dependent on algorithms
Locality	Data affinity scheduling	Predefined data partitions
Parallel overhead	High	Low

Discussion: runtime parallelism vs structured parallelism

How can we put runtime parallelism in harmony with structured parallelism ?



Requirements in DLA interface:

- DLA algorithms are designed with abstract communicators.
- Tasks are generated from DLA algorithms with light-weight communicators.
- Runtime resource manager dynamically controls resource allocation for given tasks.

Lessons learned

Increased reliance on DLA libraries.

- Application problem is characterized by **dense** block sparse matrix.
- Supernodal sparse factorization forms a tree of **dense** problems.

High performance computing in the application context.

- Multi-level tasking effectively combines multifrontal factorization with runtime task parallelism.
  ✓ high performance of DLA libraries → high performance sparse direct solver.
- Dynamic task subdivision approach provide reduce the number of data transfer to devices and provide a suitalbe granularity to devices.

Can BLIS provide building blocks for users to build their own parallelism ?

Thank you.