Implementing Dense LU Factorizations in Parallel

Isaac Dooley

8th Annual Workshop on Charm++ and its Applications
Friday April 30th 2010

Overview

- Introduction to LU Matrix Factorization
- How LU is done in parallel
- Important issues
- Description of a new Charm++ LU implementation

Why LU?

- Useful in some Science Applications
- Well known benchmark for evaluating parallel systems (Linpack Benchmark)
  - http://www.top500.org

What is LU?

- LU Factorization
  - Take a matrix M and convert it into two matrices L and U such that M = LU
  - L is lower triangular
  - M is upper triangular
- LU is one way to solve system of linear equations

What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal

$$\begin{matrix} 2 & 2 & 3 & 1 \\ 6 & 5 & 5 & 0 \\ 5 & 2 & 5 & 3 \\ 10 & 4 & 9 & 1 \end{matrix}$$

What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal

2	2	3	1
6	5	5	0
5	2	5	3
10	4	9	1

Subtract 3*row

Zero These Entries

What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal

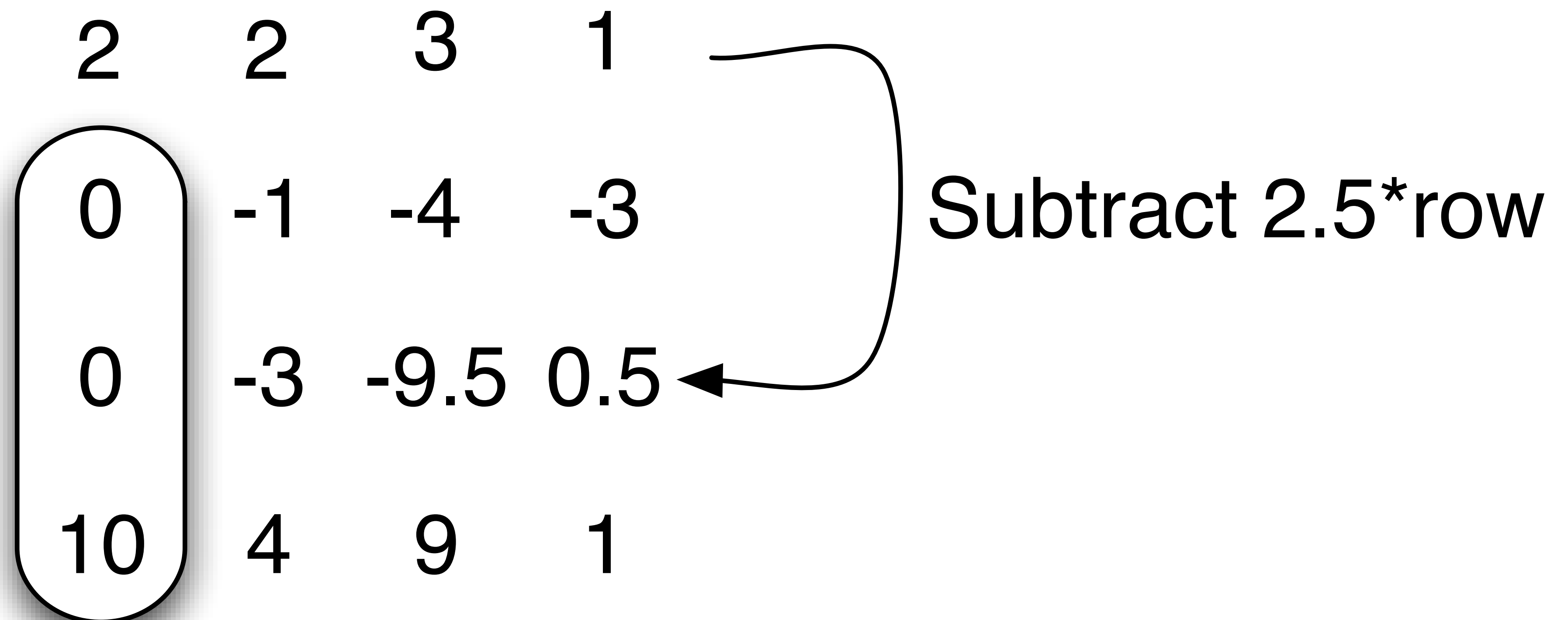
2	2	3	1
0	-1	-4	-3
5	2	5	3
10	4	9	1

Subtract 3*row

Zero These Entries

What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal



What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal

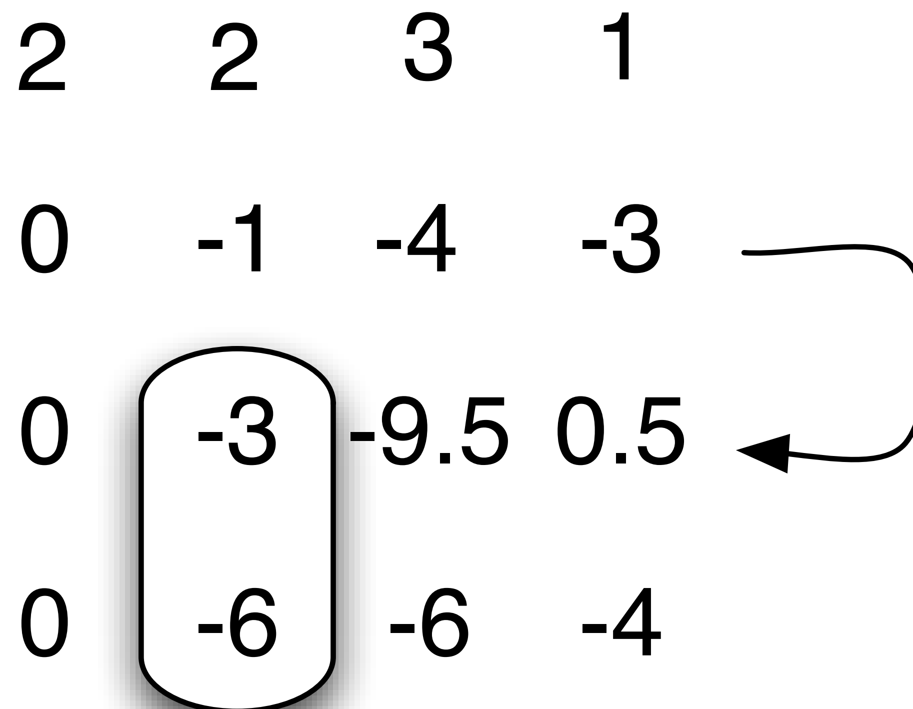
2	2	3	1
0	-1	-4	-3
0	-3	-9.5	0.5
0	-6	-6	-4

Subtract 5*row

Zero These Entries

What is LU?

- Remember Gaussian Elimination?
- Row operations to create zeros below diagonal



What is LU?

- Gaussian elimination produces an upper triangular matrix U.
- If we were to keep track of the multipliers for each row operation, we would have the lower triangular matrix L.
- $2/3(n^3)$ floating point operations for nxn matrix

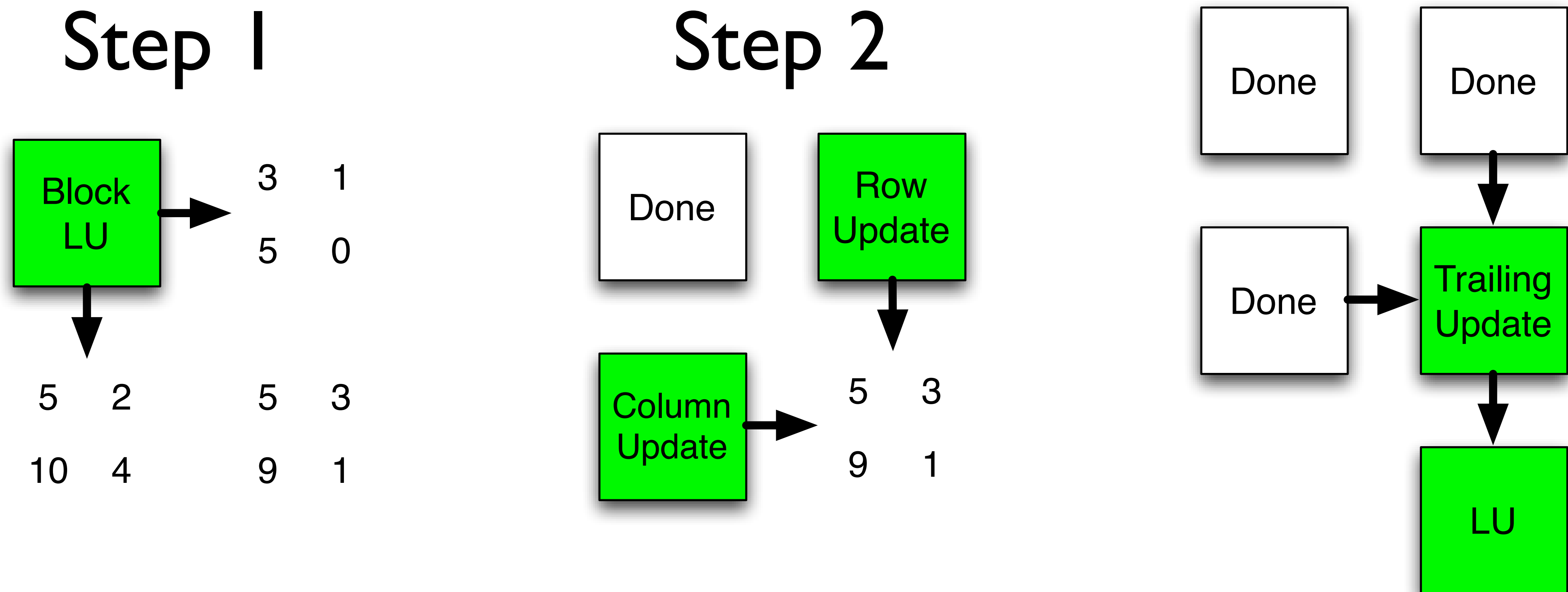
Blocked LU Example

- If we subdivide the matrix into square blocks, gaussian elimination can be expressed in four operations:

Operation	BLAS Routine
Block LU	dgetrf
U update	dtrsm
L update	dtrsm
Trailing update	dgemm

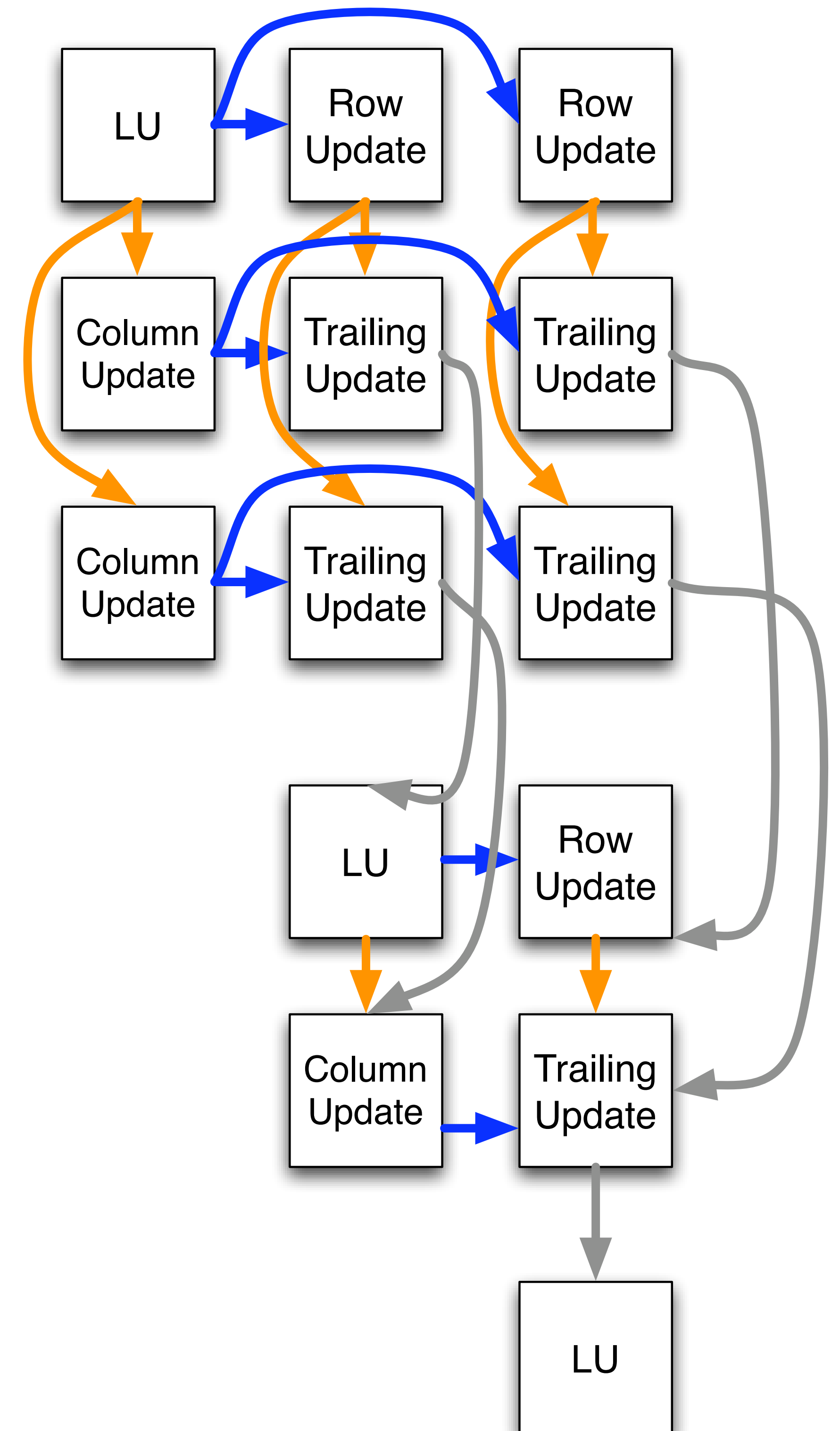
Blocked LU Example

- Partition into 4 blocks



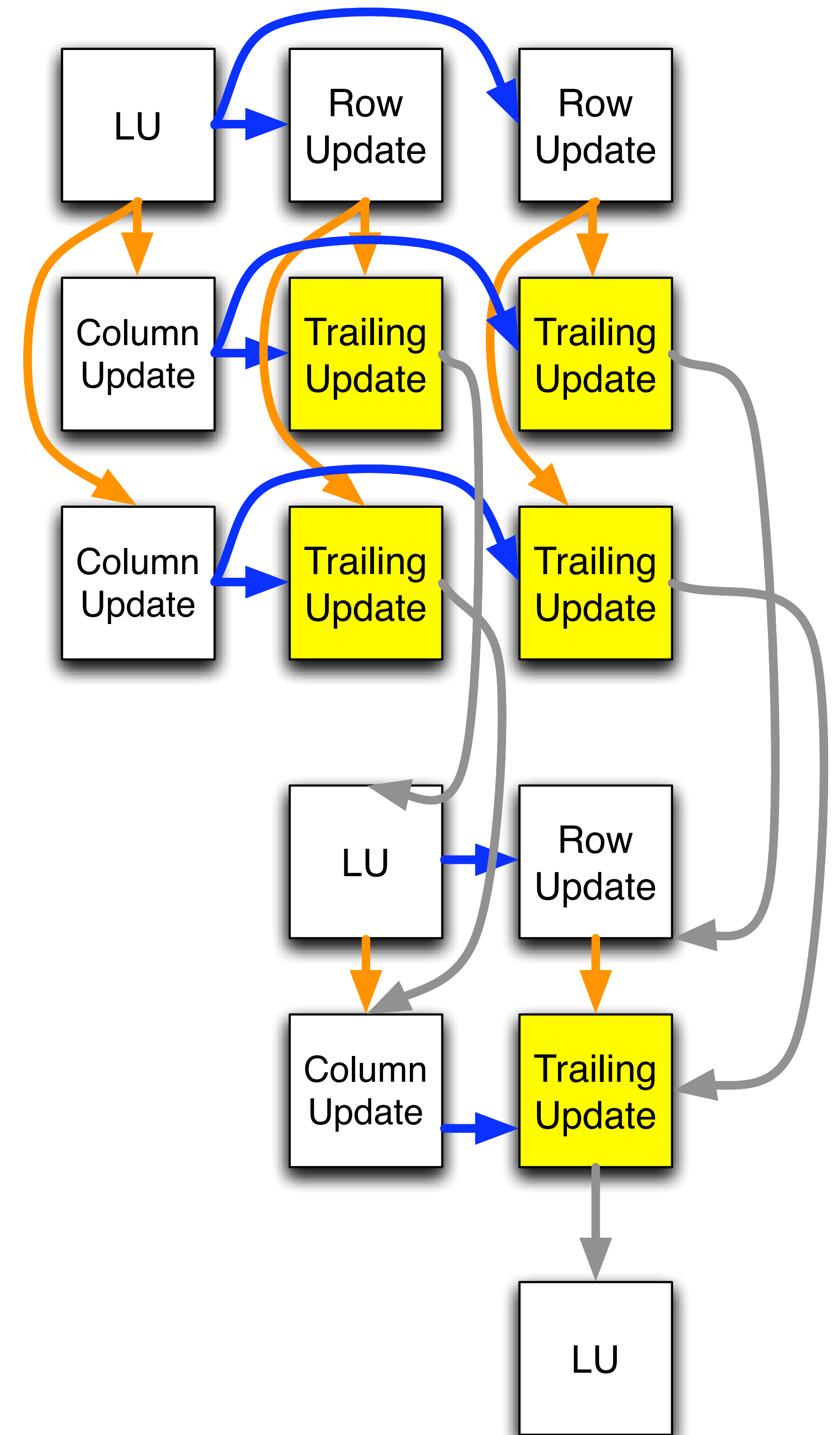
Blocked LU

- Now have multicasts along part of each row and column
- Why Blocks?
  - Serial BLAS 3 routines are fast



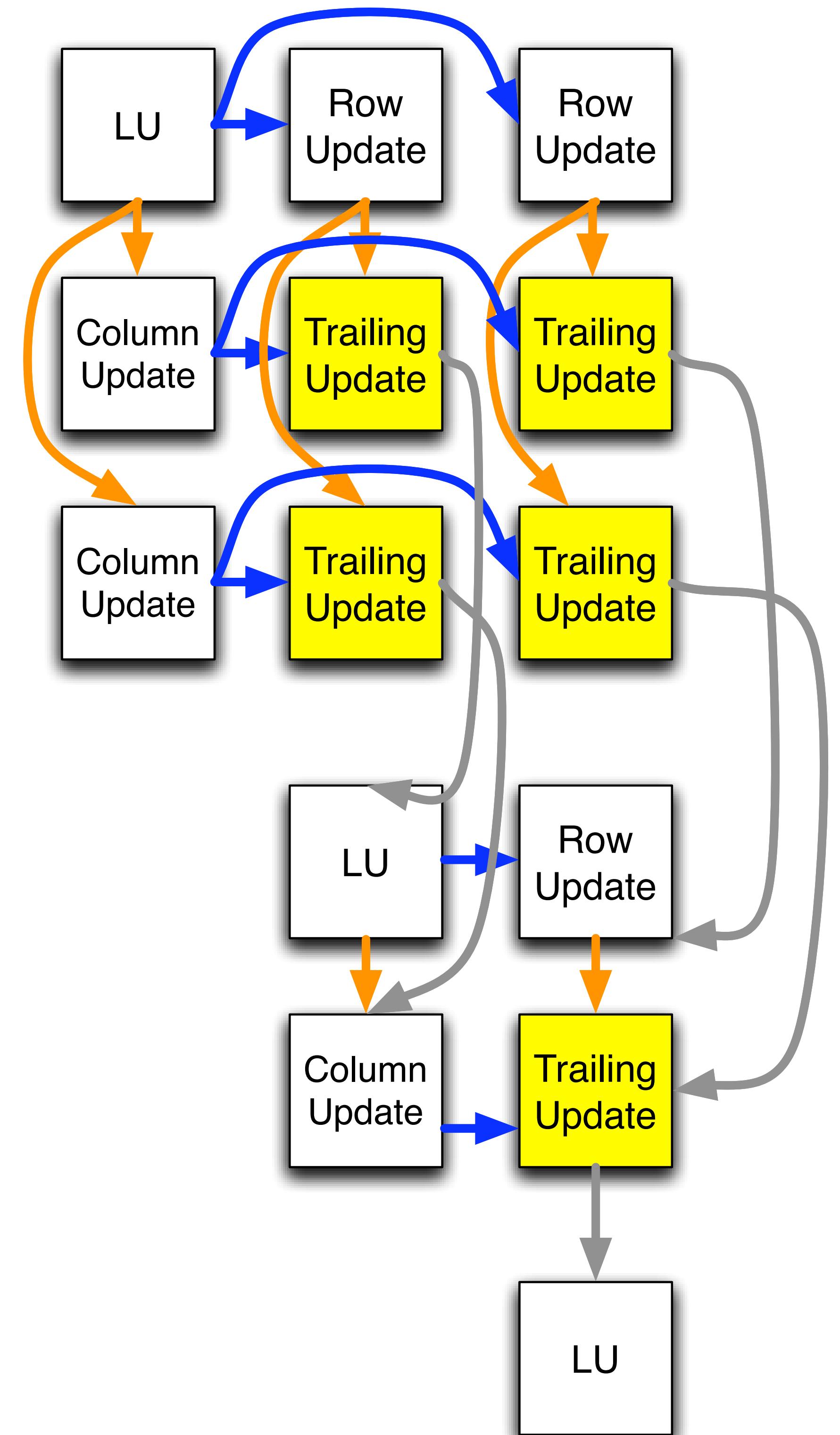
Blocked LU

- Block i,j performs:
  - One LU or U or L update
  - min(i,j) trailing updates



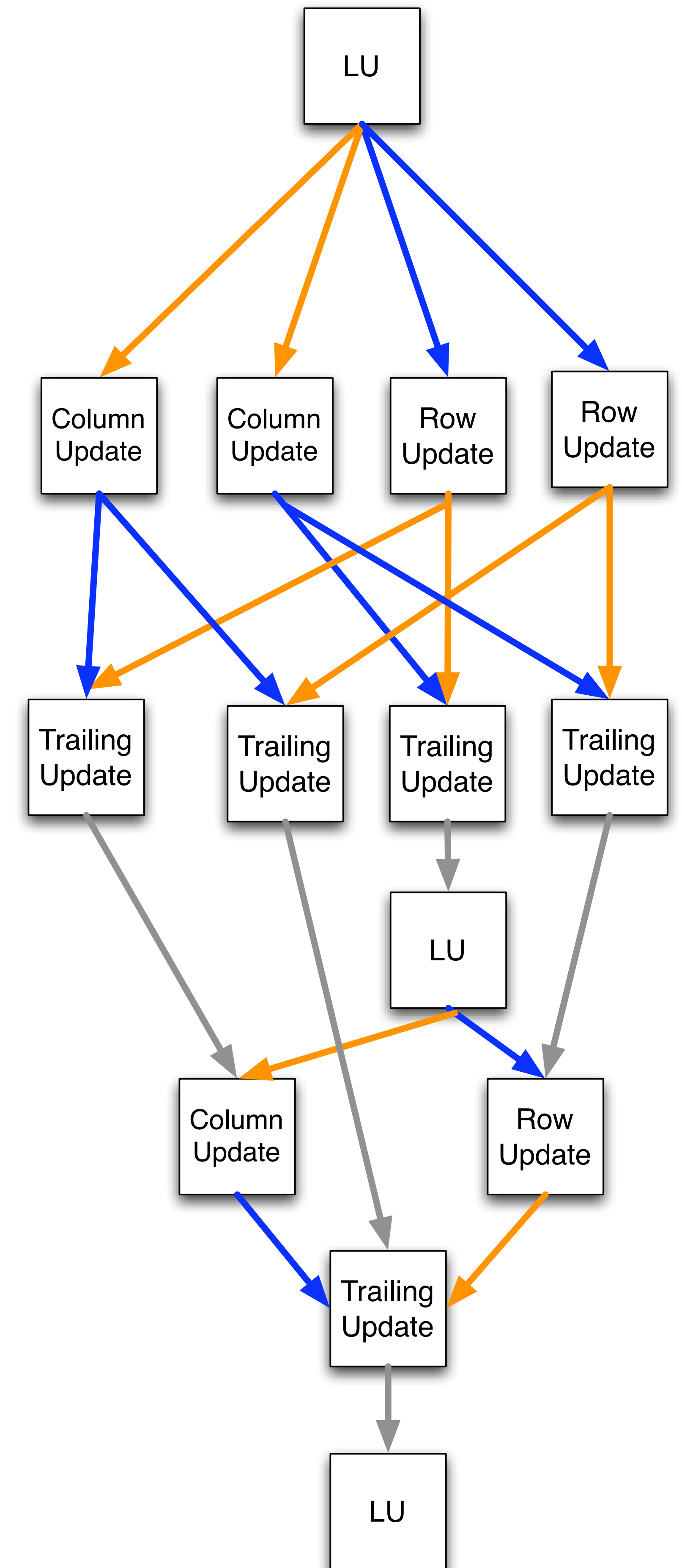
Blocked LU

- For large problem sizes most block operations are trailing updates, which are matrix-matrix-multiplies
- LU performance will be constrained by performance of dgemm



Blocked LU

- Parallel implementations will need to execute the tasks in the DAG

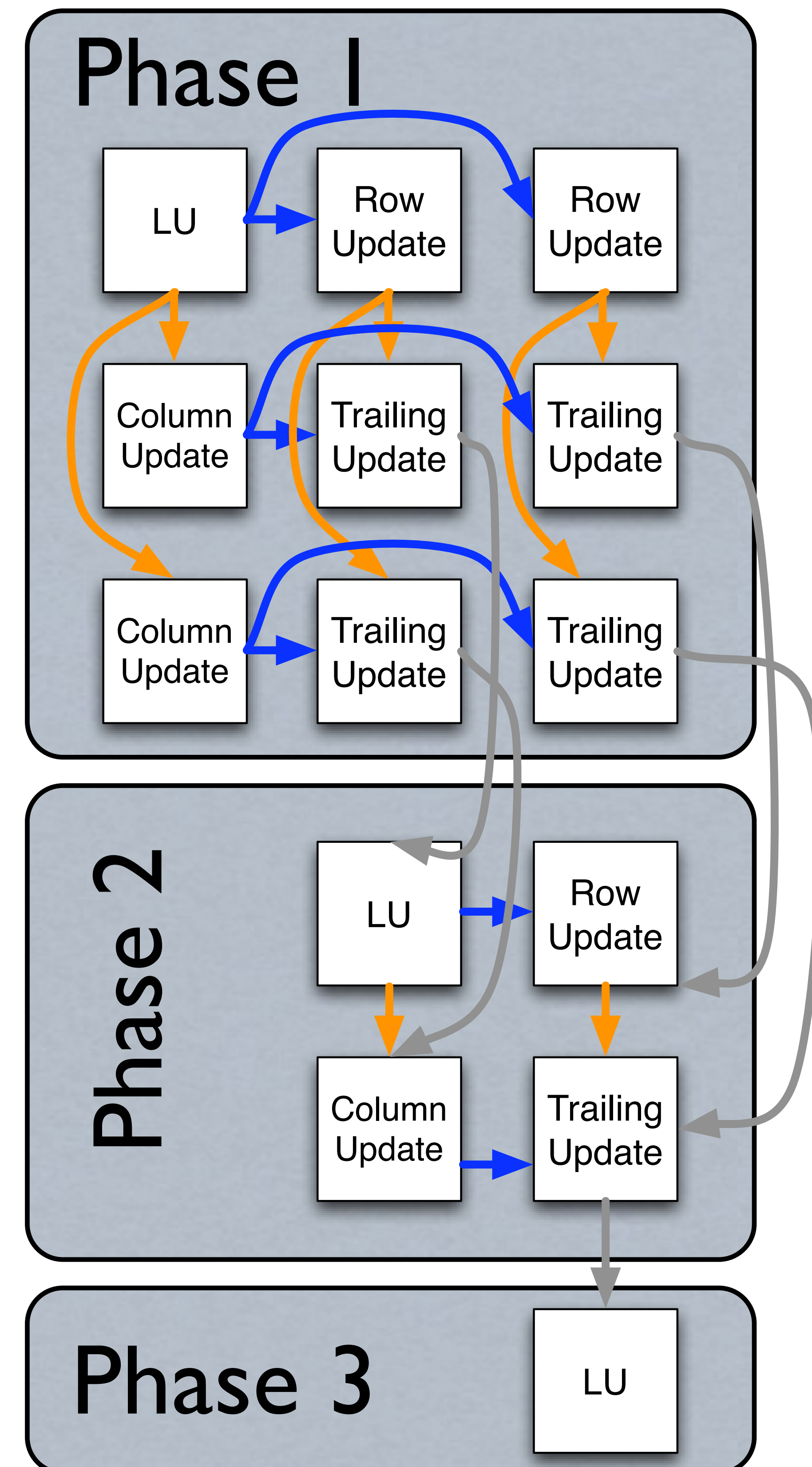


Parallel Implementations

- Necessary Parallel Operations:
  - Schedule block operations respecting dependencies
  - Multicast results across portions of a row or column. (Each multicast has unique set of destination blocks)

Parallel Implementations

- Scheduling Options
  - Synchronously execute one or more phases
  - Statically decompose whole DAG
  - Dynamically schedule blocks as dependencies happen to be satisfied (with / without priorities)



Parallel Implementations

Scheduling Method	Good	Bad
Synchronously execute phases	Simple	Limited Parallelism
Static whole DAG Decomposition	Optimal choices can be made?	Need to analyze the large graph
Dynamically schedule blocks as dependencies are satisfied	Fully Exploits parallelism	Memory usage is not necessarily bound Complex Code?

Parallel Implementations

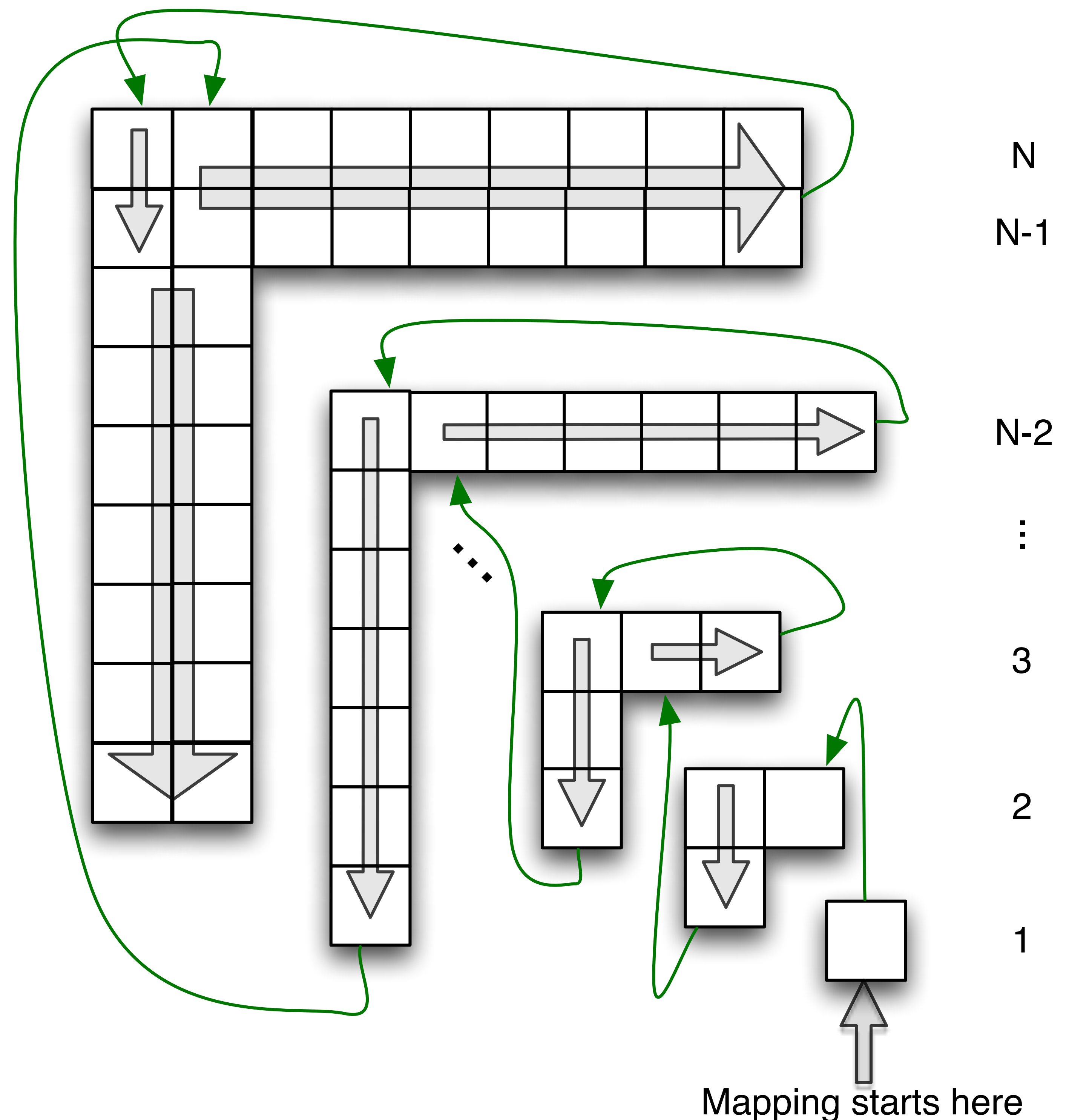
- Scalability
  - As problem size increases:
    - communication / computation decreases
    - Better load balance can be achieved
    - Program takes longer to run
      (bad for exascale)

Our Charm++ Implementation

- Each block is in a 2-D chare array element
- Prioritized messages: Prioritize work for upper left blocks higher than bottom right blocks
- Simple naive code
- No pivoting
- Custom mapping scheme
- Uses new optional Charm++ scheduler to reduce memory consumption

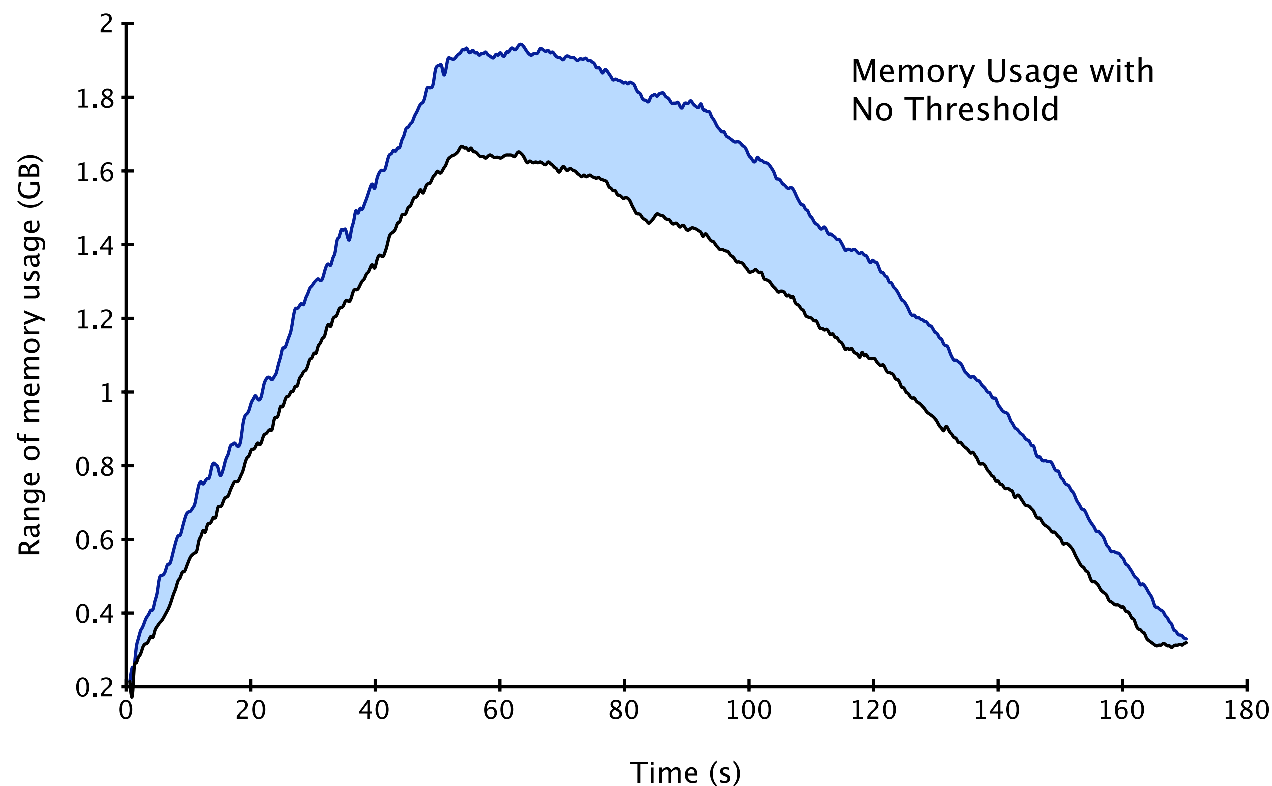
Our Charm++ Implementation

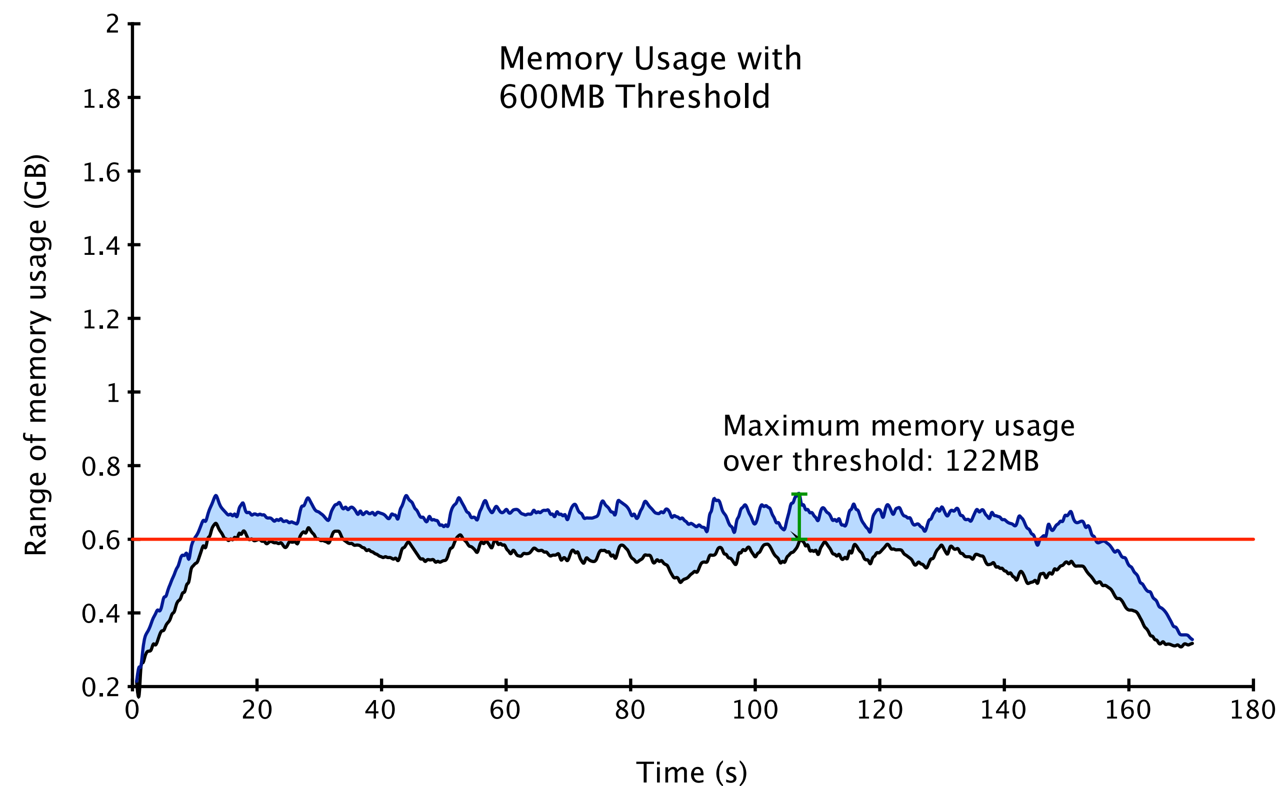
- Everyone else today uses block-cyclic mapping
- Custom mapping scheme is based on fact that more work is performed for bottom rightmost blocks.



Our Charm++ Implementation

- Uses new optional Charm++ scheduler to reduce memory consumption
- Annotate trailing updates because they reduce memory consumption by "consuming" two incoming messages and producing none.





Our Charm++ Implementation

- Performance exceeds that of HPL *(although we skip the pivoting)*
- Code is very simple:
  - No need for code that restricts progress
  - No application specific multicast code
  - Simple prioritization scheme
  - No application specific scheduler (as is done in UPC)
  - Easy to experiment with novel block to processor mappings

Future Work

- Add pivoting to our implementation
- Examine more mapping schemes
- Fix known performance issues (it currently sends larger messages than required).
- New types of automatic tuning

- *If anyone is interested in joining this project, email me!*

References

- *Matrix Computations*. Gene Golub and Charles Van Loan
- *Multi-threading and one-sided communication in parallel LU factorization*. Parry Husbands and Katherine Yelick. 2007 ACM/IEEE conference on Supercomputing
- *A Study of Memory-Aware Scheduling in Message Driven Parallel Programs*. Isaac Dooley, Chao Mei, Jonathan Lifflander, and Laxmikant V. Kale. PPL Technical Report 2010

Questions

Implementing Dense LU Factorizations in Parallel

Isaac Dooley

8th Annual Workshop on Charm++ and its Applications
Friday April 30th 2010