Using DryadLINQ for Large Matrix Operations

Thomas L. Rodeheffer
Frank McSherry
Microsoft Research, Silicon Valley

June 17, 2011

1 Overview

DryadLINQ [7] is a system that facilitates the construction of distributed execution plans for processing large amounts of data on clusters containing potentially thousands of computers. In this paper, we explore how to use DryadLINQ to perform basic matrix operations on large matrices.

DryadLINQ uses Dryad [4] as its execution engine. A Dryad execution plan is a directed acyclic graph in which computation occurs in the vertices and communication occurs on the edges. Dryad schedules vertices onto compute nodes, buffers communication in temporary files on disk, and masks failures by automatically restarting failed vertex computations on another compute node.

As compared against cluster computing using MPI [1], in Dryad the communication graph must be acyclic but the number of vertices can vastly exceed the number of compute nodes and failure recovery is automatic.

Our approach is to chop matrixes into an array of square tiles of uniform size and use DryadLINQ to construct and execute a distributed execution plan for passing tiles between compute nodes from stage to stage of the computation of a large matrix operation. We use HPC Dryad Beta [5] running on an 85-node cluster for distributed execution and BLAS routines from the Intel® Math Kernel Libary [3] (via the C# wrappers provided by Coconut [2]) for operations on individual tiles. We discuss various limitations of HPC Dryad Beta and provide preliminary performance numbers.

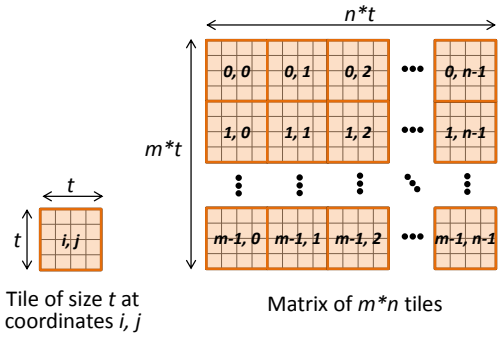


Figure 1: Matrix representation

2 Matrix representation

In order to coordinate many computers operating on matrices that are too large to fit into primary memory, we chop a matrix into an array of square *tiles* of uniform size, as shown in Figure 1. Each tile has a position in the matrix given by its *tile row* and *tile column* numbers, which are its *tile coordinates*. We number the rows and columns starting at zero.

A tile of size t consists of tile coordinates and a square $t * t$ array of doubles. In this paper, we consider only matrices of doubles and we ignore the complexity of handling matrices that are not evenly divisible into tiles.

A matrix consists of an *unordered* collection of tiles and metadata that describes the numbers of tile rows and columns that define the shape of the matrix. The value of a tile of the matrix at a given tile coordinate is determined by summing over all tiles in the collection with that coordinate.

Observe that the matrix representation admits of *sparse*, *dense*, *compact*, and *uncompact* variations. Tiles that contain only zeros can be omitted, giving a sparse representation. If tiles at all coordinates are present, the representation is dense. If each coordinate appears at most once, the representation is compact. Finally, if some coordinate appears multiple times, the representation is uncompact.

This matrix representation gives several advantages in organizing distributed computation with DryadLINQ. First, the fact that the order of tiles is irrelevant is especially important, since many of DryadLINQ's distributed operations run more efficiently when they are not required to preserve order. Second, since each tile carries its coordinates, collections of tiles can be partitioned as desired without having to worry about tracking which tile is which. Third, since tiles whose values are all zero can be omitted, sparse matrices or matrices with particular shapes (such as lower-triangular matrices) can be stored and operated upon more efficiently.[1] Finally, the tile size can be chosen to balance a trade off between memory consumption and the efficiency of tile operations.

3 Tile operations

Table 1 lists some basic operations on tiles. Since tiles carry coordinates in addition to their array of doubles, the operations also have to specify the coordinates of the result. For $a + b$, $a - b$, and $a * s$, the result takes the coordinates of a. (Presumably, tile b has the same coordinates as tile a in order for these operations to make sense.) For $a * b$, the result takes the row of a and the column of b. For $\mathrm{Lts}(a, b)$ the result takes the coordinates of b.

Example listings can be found in the Appendix. Appendix A shows an example C# implementation of tile coordinates and Appendix B an example C# implementation of tiles.

For best performance, we actually implemented tile operations using the BLAS routines from the Intel® Math Kernel Libary [3], via the C# wrappers provided by Coconut [2]. Table 2 shows the performance we measured for a tile size of 4096. Although we were unable to find an efficient use of the MKL for adding two matrices, the performance for multiply and LTsolve is over 1000 times faster than the naive example implementations shown in Appendix B.

[1] However, the possibility of omitted tiles creates complexity in some of the algorithms, as discussed in Section 6.2.

$a + b$	add two tiles
$a - b$	subtract two tiles
$a * s$	scale tile a by double s
$a * b$	multiply two tiles
$\mathrm{Lts}(a, b)$	find tile x such that $a * x = b$, where a is lower-triangular

Table 1: Basic tile operations

time	**operation**
1.24s	add two tiles
1.24s	subtract two tiles
3.23s	multiply two tiles (MKL)
2.53s	lower-triangular solve (MKL)
0.6s	read from local disk
5.7s	read from remote disk
0.8s	write to local disk

Table 2: Tile operation performance

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD .6+Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Disk operations using HPC Dryad Beta 3690 with custom serialization. No contention. Average operation time in seconds over a long sequence of repeating the same operation.

To compare computation against I/O, we also measured the time required to read or write a tile to disk using HPC Dryad Beta [5] running on our cluster. Dryad always writes to the local disk but it can read from either the local disk or across the network from a remote disk. Although we measured access to the local disk proceeding at a reasonable fraction of the disk bandwidth, access from a remote disk was almost ten times slower, even in the absence of contention. With contention, the remote access time degrades further.

4 Matrix operations

In the following sections we explore methods of performing basic matrix operations in DryadLINQ. Since we are interested in large matrices, it is important to discover which methods actually scale. As will become apparent in later discussion, some innocuous-looking expressions consume unexpected amounts of memory.

When a compute node runs out of physical memory, first it starts thrashing and, if demand increases enough, the computation fails. Although the Dryad vertex scheduler automatically tries the computation again on another compute node, since the failure is deterministic it repeats. Eventually Dryad gives up and the job fails.

To explore large matrix operations, we adopt the following configuration parameters. We take a tile size of 4096 by 4096 doubles, which results in a tile that occupies 134 MB. We take a matrix size of 16 by 16 tiles, which results in a matrix of 65536 by 65536 doubles, occupying 34 GB. We partition a matrix into 8 parts, so that each part occupies about 4 GB.

Since our compute nodes each have 16 GB of physical memory, these parameters mean that each node can hold three matrix parts simultaneously in physical memory, with about 3 GB left over for operating system, program, and other working storage. This design accomodates, for example, holding two input parts and one output part.

Table 3 summarizes our preliminary performance results for various matrix operations. These results are discussed in more detail in the following sections.

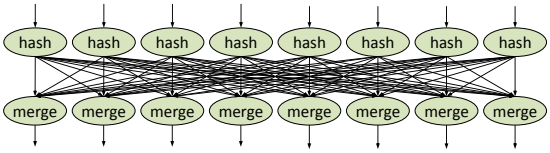
5 Repartition

Often when programming DryadLINQ it turns out that subsequent operations will be more efficient (or even possible) if only the input data is reorganized in a certain way. For example, it might help to repartition the matrix so that the tiles in any given column all fall in the same partition. This is easy to accomplish in DryadLINQ using HashPartition. Figure 2 shows an example listing and the two-stage execution plan created by DryadLINQ.

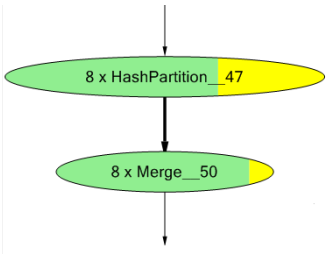
In the first stage of the execution plan, there is one vertex for each of the existing parts. Each vertex reads a part, hashes the tiles into bins according to the new partition-

```
public static IQueryable<Tile> ByCols (
  IQueryable<Tile> A)
{
  return A.HashPartition(a => a.col, numParts);
}
```

Example code listing.



Execution plan showing all vertices. Observe the all-to-all communication pattern between the hash stage and the merge stage.



Execution plan showing just the stages. Since diagrams showing all vertices rapidly get cluttered with detail, it is more useful to show just the stages. In the stage diagram, an all-to-all communication pattern is indicated by a heavy arrow.

Figure 2: Matrix partition by rows.

job runtime in minutes					
8 nodes	**16 nodes**	**24 nodes**	**32 nodes**	**64 nodes**	**job**
8					§5 repartition
					matrix addition
-	-	-	-	-	§6.1 AddByConcat
41	38	29	29	31	§6.2 AddByCoordUnion
33	29	30	29	35	§6.3 AddByCoordUnionV2
18	14	18	14	14	§6.4 AddByHashApply
24	16	15	14	14	§6.5 AddByHashmanyApply
					matrix multiplication
-	-	-	-	-	§7.1 MulByJoinAgg
119	97	110	118	105	§7.2 MulByAstreamBcols
109	71	66	45	34	§7.3 MulByAmwhBcolsApplySide
139	83	59	52	31	§7.4 MulByAmwhBcolsApplyLdoc
100	65	60	47	33	§7.5 MulByAmwhBcolsSelect
143	94	72	63	39	§7.6 MulByAmwhBcolsSelectJoin
249	140	97	81	48	§7.7 MulByArowsBcolsEnv
					matrix lower-triangular solve
77	70	73	71	78	§8.1 LtsByAorderBcols
77	71	79	75	78	§8.2 LtsByArangeBcols
78	75	75	76	80	§8.3 LtsByAmwhBcolsmwh
88	56	55	50	62	§8.4 LtsByAmwhmwhBcolsmwh

Table 3: Performance of matrix operations (preliminary).

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Number of nodes does not count the job manager. Average operation time in minutes over a small number of repetitions.

ing, and writes them out, creating a temporary file for each bin. These vertices are "hash" vertices.

In the second stage of the execution plan, there is one vertex for each of the new parts. Each vertex reads all of the temporary files destined for its parts and writes a merged output file. These vertices are "merge" vertices.

The performance of the execution plan depends on how the input is partitioned, where the files are, how compute nodes are assigned to vertices, how many files have to be read remotely, and how much contention there is. For our configuration parameters, repartitioning a dense compact matrix generally takes about 8 minutes if 8 compute nodes are available.

The performance can be analyzed as follows. First we consider the hash vertices.

Generally, each of the hash vertices has to read and write 32 tiles. Writing always happens to the local disk. If the reading is from the local disk, based on the tile operation performance listed in Table 2 this would take each hash vertex 44 seconds. Instead each hash vertex takes 230 seconds on average. So we looked at detailed performance logs.

We discovered that, although in almost all cases the hash vertex did read its input from the local disk, reading a tile took on average 2.4 seconds. Writing a tile took on average 4.6 seconds. This contrasts with 0.6 seconds and 0.8 seconds, respectively, listed for these operations in Table 2. One difference is that when tile operation performance was measured for Table 2, we performed a long sequence of the same operation and measured the average time. The hash vertex, in contrast, reads a single tile and then writes it. Presumably, the interleaved reading and writing of tiles on the local disk seriously affects the disk performance. Distributing the writing of tiles among several files may also affect performance.

Combining 32 reads at an average of 2.4 seconds each and 32 writes at an average of 4.6 seconds each gives each hash vertex an estimated run time of 224 seconds. This nicely accounts for almost all of the average 230 second measured hash vertex run time.

Next we consider the merge vertices.

Generally, each of the merge vertices also has to read and write 32 tiles, although in their case most of the reading will necessarily be from a remote disk. Looking at detailed performance logs, we discovered that the merge vertices took on average 5.0 seconds to read a tile (by far most of the time remotely) and 0.9 seconds to write a tile. This corresponds fairly well with the measurements listed in Table 2.

We note that although the merge vertex reads each tile and then writes it, almost all of its reading comes from a remote disk and there seems to be little impact on the performance of writing to the local disk. This strengthens the argument that the poor write performance observed in the hash vertices is due to their alternately reading and writing from the local disk.

Combining 32 reads at an average of 5.0 seconds each and 32 writes at an average of 0.9 seconds each gives each merge vertex and estmated run time of 189 seconds. This nicely accounts for almost all of the average 190 second measured merge vertex run time.

Combining the average 230 second measured hash vertex run time with the average 190 second measured merge vertex run time would yield an estimated run time of 420 seconds, or 7 minutes, if 8 compute nodes were available and everything performed at its average. Unfortunately, we actually find that it takes more like 8 minutes to run this job on 8 compute nodes. This can be explained by the fact that there is a considerable variance in individual compute node performance, and the job time reflects the worst case critical path. Note that none of the merge vertices can start until all of the hash vertices are complete.

6 Addition

Addition of $n*n$ matrices is theoretically an n^2 operation, since all that needs to happen is the addition of elements at corresponding coordinates.

For tile size 4096 by 4096 doubles, our measured tile addition time is 1.24 seconds. Given a matrix size 16 by 16 tiles it theoretically should take one compute node about 317 seconds to add two matrices. With 8 compute nodes it should theoretically take about 40 seconds.

Of course, these theoretical times are not achievable, because it takes time to read and write the matrices on disk and also time to send tiles around to the right place. In the case of using DryadLINQ to add large matrices, almost all of the elapsed time goes into such organizational overhead.

Next we present various methods for adding large matrices using DryadLINQ.

```
public static IQueryable<Tile> AddByConcat (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  return A.Concat(B)
    .GroupBy(c => c.coord)
    .Select(cc => cc.Aggregate((x,y) => x + y));
}
```

Figure 3: Matrix addition method AddByConcat.

6.1 Method AddByConcat

A straightforward method to add two matrices would be to concatenate their collections of tiles, group tiles by coordinate, and then add up the tiles in each group. We call this method "AddByConcat". Figure 3 shows an example listing. If an uncompact representation of the result were acceptable, we could even skip the group and aggregate steps.

Unfortunately, this method cannot be evaluated because HPC Dryad Beta does not implement Concat.

6.2 Method AddByCoordUnion

If all we had to deal with were dense matrices, in which tiles of all coordinates were present, we could make each matrix compact using a GroupBy and Aggregate step, then perform a Join between the two matrices based on coordinate. However, such an approach will not work if the matrices are sparse, because any coordinate that appears in one matrix but not in the other will be omitted from the Join. The fact that zero-valued tiles may be omitted from the matrix representation makes it tricky to perform coordinate-wise aggregation over two matrices.

One method that works is to compute the union of all coordinates in the two matrices, convert each coordinate to an explicit zero-valued tile, and then use GroupJoin to aggregate first matrix A and then matrix B by coordinate into the collection. We call this method "AddByCoordUnion". Figure 4 shows an example listing and the resulting execution plan.

Observe that while the execution plan usually sends tiles between vertices, in some cases it sends coordinates. This shows some of the expressive power of DryadLINQ. As long as a type is serializable, DryadLINQ can auto-

```
public static IQueryable<Tile> AddByCoordUnion (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  return A
    // get union of all coordinates in A or B
    .Select(a => a.coord)
    .Union(B.Select(b => b.coord))

    // convert to zero tile at each coordinate
    .Select(c => new Tile(tileSize, c))

    // accumulate all A tiles at each coordinate
    .GroupJoin(A, c => c.coord, a => a.coord,
      (c,aa) => aa.Aggregate(c, (x,y) => x + y))

    // accumulate all B tiles at each coordinate
    .GroupJoin(B, c => c.coord, b => b.coord,
      (c,bb) => bb.Aggregate(c, (x,y) => x + y));
}
```

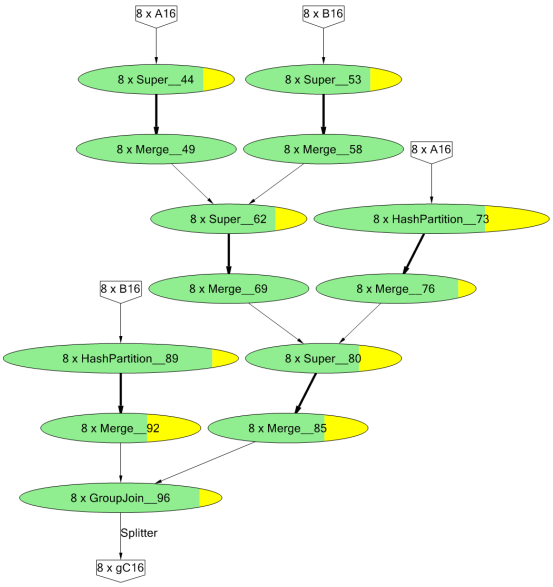


Figure 4: Matrix addition method AddByCoordUnion.

matically arrange to send records of that type betweeen vertices, without any special code needed by the programmer.

Next we describe how the execution plan performs when eight compute nodes are available.

Selecting the coordinates requires reading the input matrixes. Each vertex reads 32 tiles on average, but the DryadLINQ scheduler almost always arranges for this to be local, so only about half a minute is required. Less than a second is required to merge the coordinates.

Computing the union of the coordinates is fast, but then a zero tile is created for each coordinate, and these are rehashed by coordinate. The vertexes in this stage take about 3 minutes to run.

Following the rehash is a merge stage. The merge vertices read and write the zero tiles. These vertices each take about 4 minutes to run.

Each of the input matrixes has to be hash partitioned by coordinate. The hash partition vertices each take about 3 minutes and the following merge vertices about 2 minutes.

Then a group join stage adds the parts of matrix A into the zero tiles and rehashes by coordinate. The vertices in this stage each take about 6 minutes. Following is a merge stage whose vertices take about 2 minutes.

Then a group join stage adds the parts of matrix B into the partial sum. The vertices in this stage each take about 6 minutes.

Table 4 summarizes the breakdown of work in this matrix addition method. Recall that theoretically only about 5 vertex*minutes of work is needed to perform the actual multiplication. Everything else is overhead.

If every vertex performed at its average rate, eight compute nodes could complete the execution plan in about 32 minutes. We measure the job run time of about 41 minutes. The excess is due to variance in the execution times of vertices and the fact that often a subsequent stage (such as a merge stage) cannot start until all vertices in the previous stage are complete.

Inspecting the plan, we see that DryadLINQ spends a lot of work repartitioning the various data sets by coordinate. It is doing so in order to guarantee that records relating to the same coordinate appear in corresponding parts, so that the Union of coordinates and the GroupJoin according to coordinate can be computed correctly in independent computations distributed over the separate vertices of a stage.

n	m	$n * m$	**what**
16	0.5	8	select coord
16	0	0	merge coord
8	3	24	union coord, make zeros
8	4	32	merge zeros
16	3	48	hash matrixes
16	2	32	merge matrixes
8	6	48	group join A, add, rehash
8	2	16	merge group join A
8	6	48	group join B, add
		256	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 4: Breakdown of work in matrix addition method AddByCoordUnion.

6.3 Method AddByCoordUnionV2

Rather than require so many repartitioning steps, we can instead initially partition the input matrices by coordinate and then carry that partitioning through the computation. We call this method "AddByCoordUnionV2". Figure 5 shows an example listing and the resulting execution plan.

Although each Select transforms the data without changing the partitioning, DryadLINQ is unable to deduce it, so we use AssumeHashPartition to inform DryadLINQ of the fact. The same thing applies to GroupJoin. It is also important to know that Union produces its result partitioned according to AssumeHashPartition.

The "Tee" nodes in the execution plan do not correspond to any compute node. Rather, they are a bookkeeping notation used by Dryad to manage the fact that an intermediate result is consumed in more than one place.

Table 5 summarizes the breakdown of work in this matrix addition method. Recall that theoretically only about 5 vertex*minutes of work is needed to perform the actual multiplication. Everything else is overhead.

If every vertex performed at its average rate, eight com-

```
public static IQueryable<Tile> AddByCoordUnionV2 (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  A = A.HashPartition(a => a.coord, numParts);
  B = B.HashPartition(b => b.coord, numParts);

  var Ac = A.Select(a => a.coord)
    .AssumeHashPartition(x => x);
  var Bc = B.Select(b => b.coord)
    .AssumeHashPartition(x => x);

  return Ac.Union(Bc)
    // convert to zero tile at each coordinate
    .Select(c => new Tile(tileSize, c))

    // accumulate all A tiles at each coordinate
    .AssumeHashPartition(c => c.coord)
    .GroupJoin(A, c => c.coord, a => a.coord,
      (c,aa) => aa.Aggregate(c, (x,y) => x + y))

    // accumulate all B tiles at each coordinate
    .AssumeHashPartition(c => c.coord)
    .GroupJoin(B, c => c.coord, b => b.coord,
      (c,bb) => bb.Aggregate(c, (x,y) => x + y));
}
```

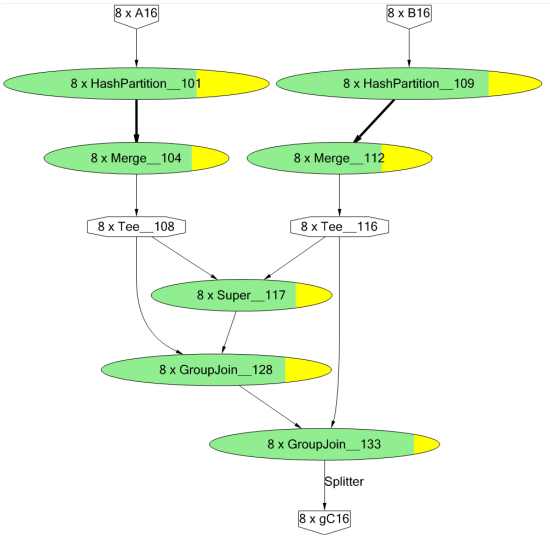


Figure 5: Matrix addition method AddByCoordUnionV2.

n	m	$n * m$	**what**
16	3	48	hash matrixes
16	2	32	merge matrixes
8	4	32	select coord, union, make zeros
8	6	48	group join A, add
8	6	48	group join B, add
		208	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 5: Breakdown of work in matrix addition method AddByCoordUnionV2.

pute nodes could complete the execution plan in about 26 minutes. We measure the job run time of about 34 minutes. The excess is due to variance in the execution times of vertices and the fact that often a subsequent stage (such as a merge stage) cannot start until all vertices in the previous stage are complete.

By getting rid of the coordinate repartitioning stages, the execution plan in this method is simpler than for method AddByCoordUnion, there is less total work to do, and it runs considerably faster.

One further improvement that could be made is to eliminate the zero tiles, which, for a dense matrix, amounts to reading and writing 34 GB worth of zeroes. However, since these reads and writes are almost always to the local disk, this only accounts for about one minute of waste in running the plan. Fixing it produces only a marginal improvement.

6.4 Method AddByHashApply

Another method for performing matrix addition is to partition the matrices by coordinate and then, within each pair of corresponding parts, concatenate the pair of parts, group by coordinate, and aggregate. We call this method "AddByHashApply". Figure 6 shows an example listing and the resulting execution plan.

By using Apply with a DistributiveOverConcat sub-

```
public static IQueryable<Tile> AddByHashApply (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  A = A.HashPartition(a => a.coord, numParts);
  B = B.HashPartition(b => b.coord, numParts);
  return A.Apply(B, (aa,bb) => LocalAdd(aa,bb));
}

[DistributiveOverConcat]
public static IEnumerable<Tile> LocalAdd (
  IEnumerable<Tile> A,
  IEnumerable<Tile> B)
{
  return A.Concat(B)
    .GroupBy(c => c.coord)
    .Select(cc => cc.Aggregate((x,y) => x + y));
}
```

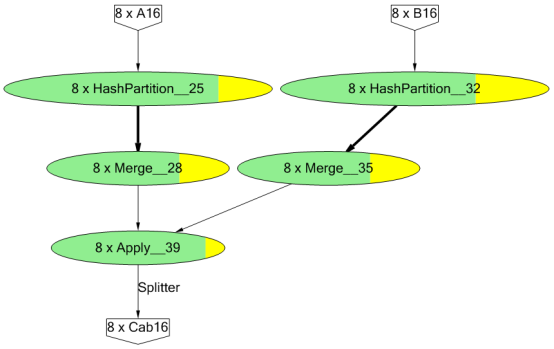


Figure 6: Matrix addition method AddByHashApply.

n	m	$n * m$	**what**
16	1	16	hash input by coord
16	4	64	merge
8	6	48	compute sum
		128	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 6: Breakdown of work in matrix addition method AddByHashApply.

routine we can control how each vertex in a stage reads through its input parts and writes its output part. The LocalAdd subroutine uses the IEnumerable methods to concatenate, group, and aggregate tiles by coordinate locally within the vertex.

A disadvantage of using Apply is that we loose any automatic data parallelism that could be provided by DryadLINQ. (Many DryadLINQ methods actually spawn multiple threads within each vertex and distribute their work among the threads.) On the other hand, we get the advantage of being able to coordinate how the two input parts are processed.

Table 6 summarizes the breakdown of work in this matrix addition method. Recall that theoretically only about 5 vertex*minutes of work is needed to perform the actual multiplication. Everything else is overhead.

If every vertex performed at its average rate, eight compute nodes could complete the execution plan in about 16 minutes. We measure the job run time of about 18 minutes, which is very close.

This method has a simpler execution plan, less work, and completes considerably sooner than method AddByCoordUnionV2. Both plans first partition their input matrices by coordinate. The difference is that AddByHashApply follows with only one stage of reading the repartitioned input matrices and writing a result matrix, whereas AddByCoordUnionV2 has three stages that each do approximately that amount of work.

n	m	$n * m$	**what**
16	4	64	hash input by coord
128	0.50	64	merge
64	0.75	48	compute sum, rehash
8	4	32	merge sum
		208	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 7: Breakdown of work in matrix addition method AddByHashmanyApply.

6.5 Method AddByHashmanyApply

If more than eight compute nodes are available, matrix addition could be accellerated by partitioning the matrixes into more parts. To test this idea, we modify method AddByHashApply to partition into $numParts^2$ parts. We call this method "AddByHashmanyApply". Figure 7 shows an example listing and the resulting execution plan. The method concludes by repartitioning the result matrix back to the standard number of parts.

Table 7 summarizes the breakdown of work in this matrix addition method, given eight compute nodes.

Compared to method AddByHashApply, the initial hash partition vertices each take about four times as long. This appears to be due to the overhead of creating 64 intermediate files instead of 8 intermediate files, even though the total amount written is the same.

The following merge vertices are eight times as many but with one-eighth as much work to do each. However, there is a large variance in how fast they run, with some taking a few seconds and some taking almost a minute. This seems to be due to how much contention was present at the time each vertex ran.

The addition vertices are eight times as many but with one-eighth as much work to do each.

The final merge vertices to bring the result matrix back to the standard number of parts were not needed in method AddByHashPartition.

```
public static IQueryable<Tile> AddByHashmanyApply (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  var n = numParts * numParts;
  A = A.HashPartition(a => a.coord, n);
  B = B.HashPartition(b => b.coord, n);
  return A
    .Apply(B, (aa,bb) => LocalAdd(aa,bb))
    .HashPartition(c => c.coord, numParts);
}

[DistributiveOverConcat]
public static IEnumerable<Tile> LocalAdd (
  IEnumerable<Tile> A,
  IEnumerable<Tile> B)
{
  return A.Concat(B)
    .GroupBy(c => c.coord)
    .Select(cc => cc.Aggregate((x,y) => x + y));
}
```

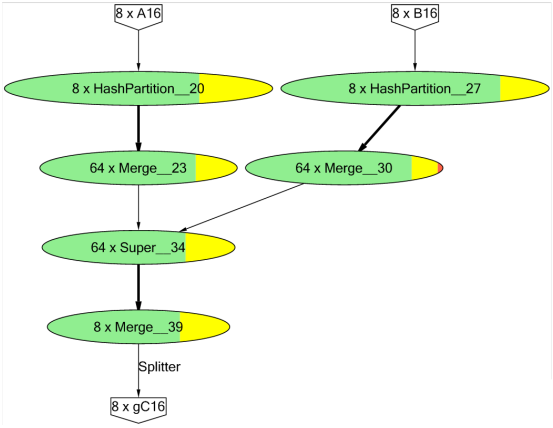


Figure 7: Matrix addition method AddByHashmanyApply.

Because of the additional work of partitioning the input matrixes more finely and then at the end repartitioning the result back to the standard number of partitions, this method performs significantly worse that AddByHashApply on eight compute nodes. Table 3 shows the performance with various numbers of compute nodes. Although the job time improves with increasing numbers of compute nodes, the repartitioning overheads prevent this method from ever being competitive with the simpler method AddByHashApply.

7 Multiplication

Ignoring refinements such as Strassen's algorithm [6], multiplication of $n * n$ matrices is theoretically an n^3 operation, requiring n^3 multiplications and $(n-1) * n^2$ additions.

For a tile of 4096 by 4096 doubles, our measured tile addition time is 1.24 seconds and tile multiplication time is 3.23 seconds. Given a matrix of 16 by 16 tiles it theoretically should take one compute node about 300 minutes to multiply two matrices. With 8 compute nodes it should theoretically take about 38 minutes.

Of course, these theoretical times are not achievable, because it takes time to read and write the matrices on disk and also time to send tiles around to the right place. For matrix multiplication, sending tiles around efficiently becomes particularly important, because you have to get column-row matching pairs of input tiles together to multiply, and then you have to get same-coordinate product tiles together to aggregate.

Next we present various methods for multiplying large matrices using DryadLINQ.

7.1 Method MulByJoinAgg

An elegant method to multiply two matrices is based on joining the two collections of tiles, forming all pairs in which a tile from the first collection has a column number that matches the row number of a tile from the second collection. The tiles in each pair are then multiplied and the entire result grouped by coordinate and each group summed up. We call this method "MulByJoinAgg". Figure 8 shows an example listing. If an uncompact representation of the result were acceptable, we could even skip the group and aggregate steps.

```
public static IQueryable<Tile> MulByJoinAgg (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  return A
    .Join(B, a => a.col, b => b.row, (a,b) => a * b)
    .GroupBy(c => c.coord)
    .Select(cc => cc.Aggregate((x,y) => x + y));
}
```

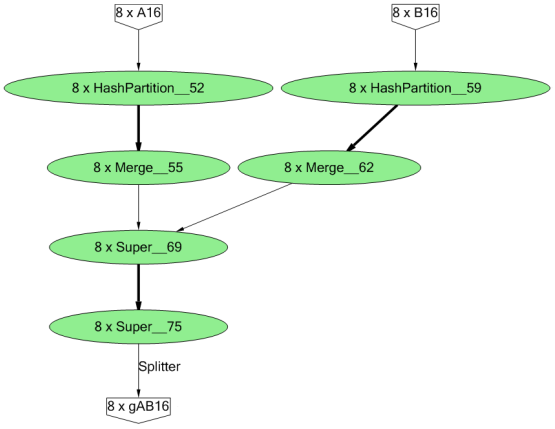


Figure 8: Matrix multiplication method MulByJoinAgg.

Method MulByJoinAgg perfectly employs the semantics of Join in performing multiplication on matrices represented as a possibly sparse, possibly uncompact collection of tiles. If a coordinate is omitted from one of the input matrices, the Join will form no product, but then omitting a coordinate means that the value of the tile is zero and the product would be zero anyway. If a coordinate appears multiple times in one of the input matrices, the Join will form multiple products, which works out to the correct result because multiplication distributes over addition.

Unfortunately, method MulByJoinAgg is completely impractical for large matrices. The problem arises in computing the Join. DryadLINQ repartitions A by column and B by row, so that tiles that need to be joined will appear in corresponding parts. Then DryadLINQ runs a local join on one compute node for each pair of corresponding parts. The local join reads all of its input A part into memory and then reads through its input B part one tile at a time, for

each B tile finding all of the matching A tiles and computing their product.

The first problem (in the current version of DryadLINQ) is that the local join buffers all of its results in memory until it is completely done, before the compute node goes on to the grouping and aggregating actions. When processing dense matrices according to our configuration parameters, the local join will have two complete columns of tiles from A to match against two complete rows of tiles from B. Joining these tiles produces 1024 product tiles, which would require 137 GB of memory to store. This far exceeds the 16 GB physical memory capacity of one compute node.

However, even if DryadLINQ could aggressively feed the result of the local join into the grouping and aggregating actions, there would still be a second problem. Observe that the local join produces product tiles covering all 256 distinct coordinates. Hence, even if there were aggressive local aggregation, the result of the local aggregation would require 34 GB to store, which still exceeds the 16 GB physical memory capacity of one compute node.

So, in order to multiply large matrices, method MulByJoinAgg is not going to work. Instead, we will have to broadcast some of the input so that distributed aggregation can take place.

7.2 Method MulByAstreamBcols

One method for large matrix multiplication that works is to partition matrix B by columns, store a small number of columns in each compute node, and then stream all of matrix A through each compute node, computing all possible product tiles and aggregating them on the fly.

Because of the way matrix multiplication works, each product tile $a * b$ lies in the same column as the right argument tile b. Since each compute node operates with matrix B tiles from a small number of columns, only a small number of columns of product tiles will be produced for local aggregation. Because each compute node joins the entire matrix A with entire columns from B, the result of local aggregation will in fact be complete for the product columns that it has.

This method requires coordinating the entire input A with each part of input B, which can be accomplished by using Apply with a LeftDistributiveOverConcat local multiplication subroutine. We call this method "MulByAstreamBcols". Figure 9 shows an example listing and the resulting execution plan.

To simplify coding the local subroutine, we employ a LazyTiles class that manages a lazy array of tiles. Appendix C contains an example implementation of the LazyTiles class.

For our configuration parameters, running the MulByAstreamBcols execution plan takes about 160 minutes. Note that this is faster than the theoretical time of 300 minutes it would take one compute node to perform this multiplication, disregarding data movement overhead.

Inspecting the plan, we see that DryadLINQ partitions matrix B by columns in a hash partition stage followed by a merge stage. For some reason, the hash partition vertices perform better than we saw when measuring matrix repartition in Section 5, taking only about 2 minutes each rather than almost 4 minutes each. Perhaps the files were laid out differently on disk leading to less disk arm interference. However, the merge vertices performed worse, taking about 4 minutes each rather than 3 minutes.

In order to feed the entire collection of tiles for matrix A into each local multiplication vertex, DryadLINQ merges all of the parts of B into a single, 34 GB intermediate file. Actually, in a case like this, when preparing for a LeftDistributiveOverConcat apply, DryadLINQ creates a separate merged file for each of the apply vertices. So there are eight vertices in the merge stage, each reading all of the parts and creating separate 34 GB intermediate files.

These merge vertices each takes about 33 minutes to run. Each merge vertex reads 256 tiles, almost all from remote disks, and writes 256 tiles into its single output file. Calculating from the tile operation performance numbers in Table 2, this ought to take about 28 minutes, but of course there is contention, since all of the vertices in the stage are reading from the same set of input files.[2]

Then an apply stage contains the vertices that run the local multiplication method. Each of these vertices reads two input files: the first containing a roughly 4 GB part of matrix B and the second containing a 34 GB copy of the entire matrix A. One would hope that the Dryad job scheduler would prefer to schedule the vertex on the com-

[2]Actually, the contention is not as bad as it might be, because DryadLINQ employs a clever trick in this case. Each of the merge vertices picks a random permutation of its input files and reads them in that order.

```
public static IQueryable<Tile> MulByAstreamBcols (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  B = B.HashPartition(b => b.col, numParts);
  return B.Apply(A, (bb,aa) => LdocMul(aa,bb));
}

[LeftDistributiveOverConcat]
public static IEnumerable<Tile> LdocMul (
  IEnumerable<Tile> A,
  IEnumerable<Tile> B)
{
  // Accumulate all tiles from input A into memory.
  var DA = new LazyTiles(tileSize).Acc(A);

  // Read through all tiles from input B. For each
  // B tile, multiply all of the A tiles by it that
  // are in the same column as the B tile's row.
  // Accumulate the products into the proper
  // coordinates in the output product C.
  var DC = new LazyTiles(tileSize);
  foreach (var b in B)
    foreach (var a in DA.EnCol(b.row))
      DC.Acc(a * b);

  // Wherever there is a tile in the product,
  // output it.
  return DC.En();
}
```

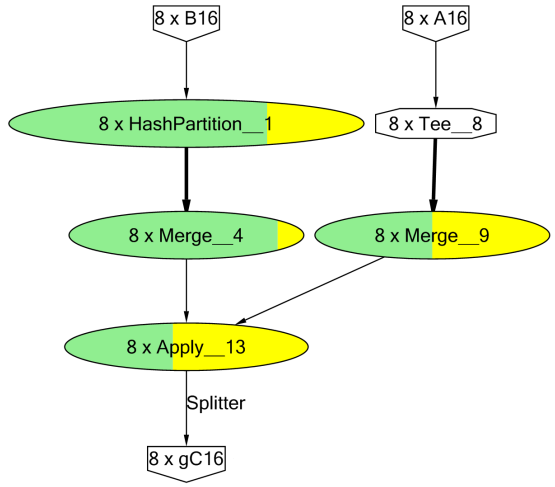


Figure 9: Matrix multiplication method MulByAstreamBcols.

n	m	$n*m$	**what**
8	2	16	hash matrix B
8	4	32	merge matrix B
8	33	264	merge matrix A to 1 file
8	55	440	compute product
		752	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 8: Breakdown of work in matrix multiplication method MulByAstreamBcols.

pute node that has the local disk containing the 34 GB copy of matrix A. Inspecting the detailed performance logs reveals that indeed this does usually happen. However, sometimes it does not, and in those cases the vertex takes much longer to run.

Each of the local multiplication vertices reads 32 tiles of matrix B and 256 tiles of matrix A. It performs 512 tile multiplications, 480 tile additions, and writes 32 product tiles as output. Calculating from the tile operation performance numbers in Table 2, when matrix A is local and matrix B is remote the vertex ought to take about 43 minutes, which is close to the observed average of about 50 minutes for this case. When matrix A is remote and matrix B is local the vertex ought to take about 62 minutes, which is close to the observed average of about 70 minutes for this case.

In our trial runs, the vertex scheduler got the preferred locality about three-quarters of the time, so the weighted average comes out to about 55 minutes for each of the local multiplication vertices.

Table 8 summarizes the breakdown of work in this matrix multiplication method. Recall that theoretically only 300 vertex*minutes of work is needed to perform the actual multiplication. Everything else is overhead.

Adding up the run times in each stage, we get an estimated total job time of 94 minutes. This compares with the measured total job time of 160 minutes. The substantial excess can be explained by two facts. First, there

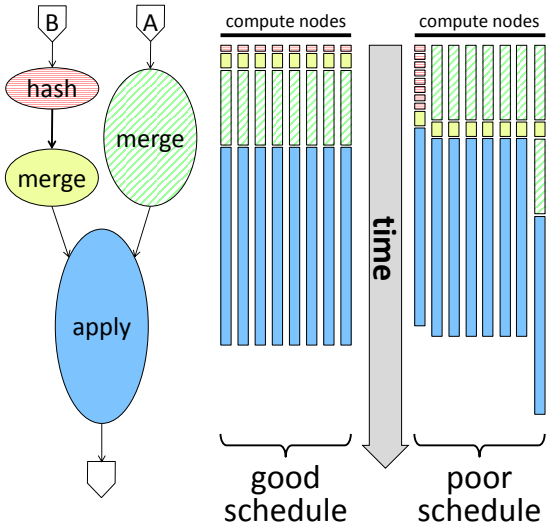


Figure 10: Good and poor vertex schedules for executing matrix multiplication method MulByAstreamBcols.

Each stage contains eight vertices. Eight compute notes are available. The illustration ignores locality, assuming that the runtime of a vertex is independent of which compute node runs it.

is a variance in individual compute node performance, and the actual job time depends on the worst case critical path. Note, for example, the substantial difference in the run time of the local multiplication vertex depending on whether matrix A is local or not.

Second, the current Dryad job scheduler uses a general-purpose heuristic that in this case tends to make a poor assignment of compute nodes to vertices. Given eight compute nodes to execute the MulByAstreamBcols plan, in which each stage has exactly eight vertices, it turns out that making a good assignment of compute nodes to vertices is critical to achieving the earliest completion time. Figure 10 illustrates a good schedule and a poor schedule. The good schedule retires the plan by stages in topological order.

The current Dryad scheduler does not follow that approach. Instead, when it has a free compute node, it picks a runnable vertex from any stage using a general-purpose heuristic. Presumably the heuristic works well in the general case, but it performs poorly with the MulByAstreamBcols plan, producing a result somewhat like that shown for the poor schedule in Figure 10. Typically, the final 40 minutes or more of the job execution consist in waiting for the last local multiplication vertex to complete.

As shown in Table 3, running the MulByAstreamBcols plan on more than eight compute nodes produces an improved run time. Actually, as can be seen by comparing the execution plan with the breakdown of work in Table 8, there not much vertex parallelism that can be exploited by additional compute nodes. Instead, the improved run time is almost entirely due to enabling the scheduler not to get trapped into a poor schedule.

Method MulByAstreamBcols partitions B by column and streams all of matrix A through each compute node. Obviously, there is a dual method that partitions A by rows and streams all of matrix B through each compute node. Everything being equal, the performance of the two methods would be identical. One could choose which alternative to employ based on which of matrix A or B was larger or whether one of the matrices was already partitioned in the required manner.

7.3 Method MulByAmwhBcolsApplySide

Although method MulByAstreamBcols is fairly straightforward, merging matrix A into a single file and then reading the whole thing takes a lot of time. Another approach would be to partition matrix A by rows, partition matrix B by columns, and then send each possible pair of parts to a vertex for local multiplication and aggregation.

Because of the way matrix multiplication works, the product tile $a * b$ lies in the same row as the left argument tile a and in the same column as the right argument tile b. Since each vertex operates with complete rows from A and complete columns from B, the result of local aggregation will in fact be complete for whatever product coordinates the vertex has responsibility.

Unfortunately, DryadLINQ provides no mechanism for coordinating an interaction of partitions in the required fashion. There is a proposed extension, sometimes called FrankFork, which would extract each of the partitions of a data set into a separate IQueryable. Suitable employment of FrankFork and Concat (also currently unimplemented) could accomplish many novel varieties of coordination.

However, even without being able to access the parts of a partition directly, each individual part can be computed using Where to filter for the desired records. This work-around is of course considerably less efficient, since the entire input data set has to be processed for each part extracted.[3]

So we can partition matrix B by columns and then coordinate the columns of B with a computed partition of the rows of matrix A. We use an Apply on matrix B with a DistributiveOverConcat local multiplcation subroutine that receives its A rows via a side-channel. We call this method "MulByAmwhBcolsApplySide". Figure 11 shows an example listing of this method and Figure 12 shows the resulting execution plan.

The method iterates through a loop using Where to extract rows from matrix A corresponding to a partitioning of matrix A by rows. The renaming of the loop control variable "lp" to a loop body local variable "p" is a technical detail required to avoid a particular brain damage in the way LINQ expressions work.

In each iteration of the loop, we use Apply with a DistributiveOverConcat subroutine to coordinate each partition of B with a broadcast copy of the extracted rows from A sent in as a DryadLINQ "side-channel". DryadLINQ arranges for each vertex to have access to a copy of the side channel data set.

The resulting IQueryables are collected up and then presented to an N-ary Apply which consolidates them into a single data set with the standard number of parts.

To simplify coding the local subroutines, we employ a LazyTiles class that manages a lazy array of tiles. Appendix C contains an example implementation of the LazyTiles class.

For our configuration parameters, running the MulByAmwhBcolsApplySide execution plan takes about 116 minutes.

Inspecting the plan, we see that DryadLINQ partitions matrix B by columns in a hash partition stage folllowed by a merge stage. Each hash partition vertex takes about 1 minute to run and each merge vertex takes about 4 minutes to run. Although they read and write the same number of

```
public static
IQueryable<Tile> MulByAmwhBcolsApplySide (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  B = B.HashPartition(b => b.col, numParts);
  var QC = new Queue<IQueryable<Tile>>();
  for (long lp = 0; lp < numParts; lp++)
  {
    var p = lp;
    var Aix = A.Where(a => a.row % numParts == p);
    QC.Enqueue(B.Apply(Bxj => DocMul(Aix, Bxj)));
  }
  var C = QC.Dequeue();
  return C.Apply(QC.ToArray(), cca => Mcat(cca));
}

[DistributiveOverConcat]
public static IEnumerable<Tile> DocMul (
  IEnumerable<Tile> Aix,
  IEnumerable<Tile> Bxj)
{
  var DA = new LazyTiles(tileSize).Acc(Aix);
  var DC = new LazyTiles(tileSize);
  foreach (var b in Bxj)
    foreach (var a in DA.EnCol(b.row))
      DC.Acc(a * b);
  return DC.En();
}

[DistributiveOverConcat]
public static IEnumerable<Tile> Mcat (
  IEnumerable<Tile>[] cca)
{
  var DC = new LazyTiles(tileSize);
  foreach (var cc in cca) DC.Acc(cc);
  return DC.En();
}
```

Figure 11: Matrix multiplication method MulByAmwhBcolsApplySide.

[3]One could employ multiple stages of extraction: for example, first splitting the entire collection into two parts, then splitting each part into two subparts, and so on. We do not further investigate this idea. The ideal solution would be to adopt FrankFork, which would obviate this work-around.

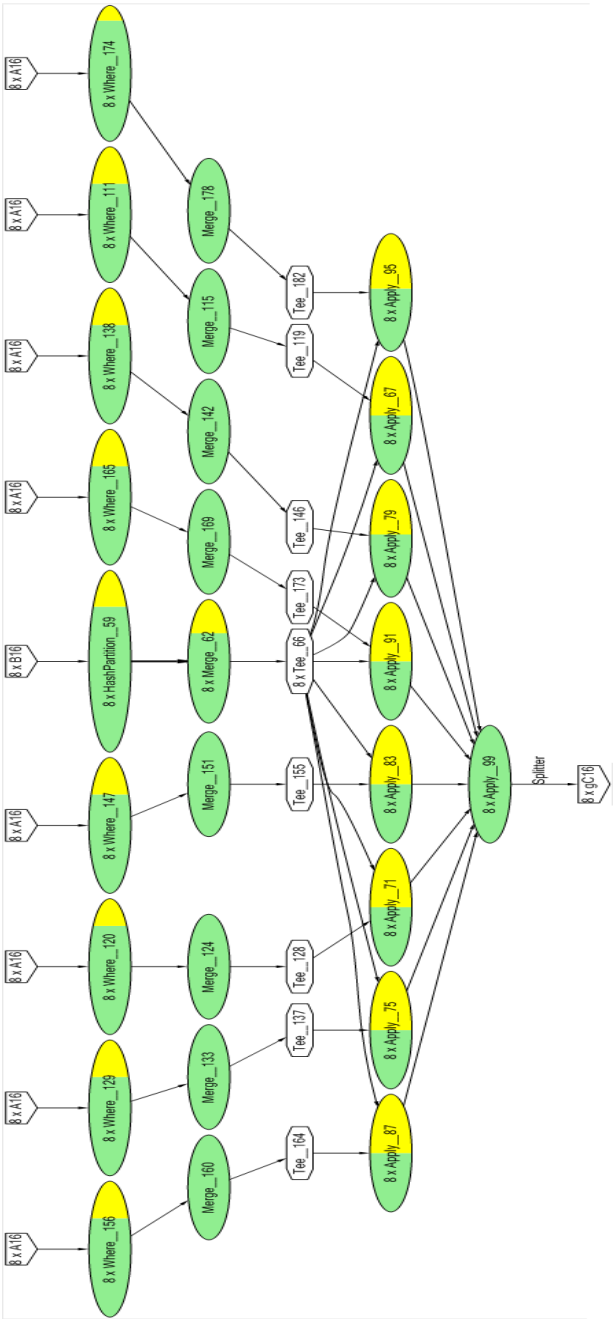


Figure 12: Execution plan for matrix multiplication method MulByAmwhBcolsApplySide.

tiles, with ideal scheduling each hash partition reads all of its tiles locally, whereas each merge vertex has to read almost all of its tiles remotely. Hence the difference in run times.

The matrix A is fed into a vector of where stages, each of which extracts a different part. Each where vertex takes about 30 seconds to run, reading 32 tiles, usually from its local disk, and writing 4 tiles.

Each where stage is followed by a merge stage which merges the outputs into a single file in preparation for feeding as a side channel. Each of these merge vertices takes about 2 minutes to run, reading 32 tiles, almost always from a remote disk, and writing 32 tiles.

Each merge stage is followed by a "Tee" node that indicates that the single file merge output is consumed by each of the vertices in the following apply stage. Each of these apply stages also gets a copy of the repartitioned matrix B, but in this case each part is connected to a separate vertex in the stage. Each vertex in these apply stages takes about 10 minutes to run, reading 32 tiles of matrix A and 32 tiles of matrix B, usually half from the local disk and half remote, performing 64 tile multiplications and 60 tile additions, and writing 4 tiles. Depending on how the vertices are scheduled, there is likely to be contention on reading the parts of matrix A or reading the parts of matrix B.

The apply stages all feed into a final apply stage that consolidates the results. Each vertex in this final apply stage takes about 5 minutes to run, reading 32 tiles, almost all remotely, and writing 32 tiles.

Table 9 summarizes the breakdown of work in this matrix multiplication method. If the scheduling were perfect and all vertices ran at the average rate, with 8 compute nodes the job should finish in about 96 minutes. Measured performance is about 116 minutes.

With eight compute nodes, MulByAmwhBcolsApplySide runs a little quicker than MulByAstreamBcols, even though its plan is much more complicated. Since MulByAmwhBcolsApplySide contains many more vertices that run for much shorter periods of time than MulByAstreamBcols, it has many more checkpoints, and generally much less work would be lost should a compute node fail. This could become important at larger matrix sizes.

Another advantage of MulByAmwhBcolsApplySide is that it contains much more vertex parallelism than MulByAstreamBrows. Hence if more compute nodes are

n	m	$n * m$	**what**
8	1	8	partition matrix B
8	4	32	merge matrix B
64	0.5	32	where matrix A rows
8	2	16	merge matrix A rows
64	10	640	compute product
8	5	40	merge product
		768	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 9: Breakdown of work in matrix multiplication method MulByAmwhBcolsApplySide.

available, it can complete sooner. With 64 compute nodes available, the MulByAmwhBcolsApplySide plan takes about 40 minutes to run.[4] This is a factor of 3 faster than with 8 compute nodes, at the cost of using 8 times as many compute nodes. Table 3 shows the job times for various numbers of compute nodes. The speedup is sub-linear because using more compute nodes results both in more contention and in many vertices having to read remotely.

7.4 Method MulByAmwhBcolsApplyLdoc

Instead of using a side-channel to broadcast parts of matrix A into stages partitioned by columns of matrix B, we could instead use a two-argument Apply with a LeftDistributiveOverConcat local multiplication subroutine. We call this method "MulByAmwhBcolsApplyLdoc". Figure 13 shows an example listing and Figure 14 the resulting execution plan.

Inspecting the plan, we see that it differs from the MulByAmwhBcolsApplySide plan mainly in how the merge stages for the rows of matrix A are connected to the apply

[4]With 64 compute nodes available, unfortunately it turns out that the spurious Merge node that merges the entire matrix B into a single file is, in some runs, on the critical path to finishing the job. This vertex alone takes about 34 minutes to run.

```
public static
IQueryable<Tile> MulByAmwhBcolsApplyLdoc (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  B = B.HashPartition(b => b.col, numParts);
  var QC = new Queue<IQueryable<Tile>>();
  for (long lp = 0; lp < numParts; lp++)
  {
    var p = lp;
    var Aix = A.Where(a => a.row % numParts == p);
    QC.Enqueue(B.Apply(Aix,
      (Bxj,Aix0) => LdocMul(Aix0, Bxj)));
  }
  var C = QC.Dequeue();
  return C.Apply(QC.ToArray(), cca => Mcat(cca));
}

[LeftDistributiveOverConcat]
public static IEnumerable<Tile> LdocMul (
  IEnumerable<Tile> Aix,
  IEnumerable<Tile> Bxj)
{
  var DA = new LazyTiles(tileSize).Acc(Aix);
  var DC = new LazyTiles(tileSize);
  foreach (var b in Bxj)
    foreach (var a in DA.EnCol(b.row))
      DC.Acc(a * b);
  return DC.En();
}

[DistributiveOverConcat]
public static IEnumerable<Tile> Mcat (
  IEnumerable<Tile>[] cca)
{
  var DC = new LazyTiles(tileSize);
  foreach (var cc in cca) DC.Acc(cc);
  return DC.En();
}
```

Figure 13: Matrix multiplication method MulByAmwhBcolsApplyLdoc.

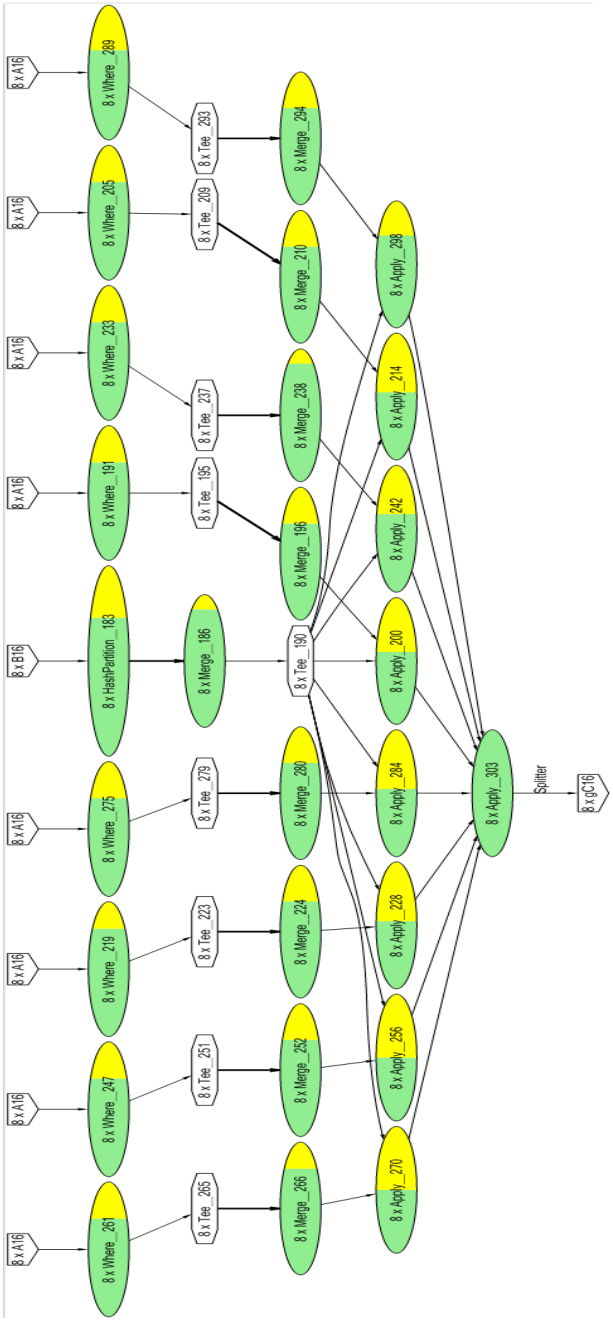


Figure 14: Execution plan for matrix multiplication method MulByAmwhBcolsApplyLdoc.

n	m	$n * m$	**what**
8	1	8	partition matrix B
8	4	32	merge matrix B
64	0.5	32	where matrix A rows
64	2	128	merge matrix A rows
64	10	640	compute product
8	5	40	merge product
		880	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 10: Breakdown of work in matrix multiplication method MulByAmwhBcolsApplyLdoc.

stages that perform the multiply. In the MulByAmwhBcolsApplySide plan, each merge stage produces one output that is broadcast to all of the vertices in the following apply stage. In the MulByAmwhBcolsApplyLdoc plan, on the other hand, each merge stage is replicated eight times to produce individual copies for each of the vertices in its following apply stage.

The result is more total work to be done, but it moves the broadcast contention from the apply stage inputs back to the merge stage inputs.

Table 10 summarizes the breakdown of work in this matrix multiplication method. If the scheduling were perfect and all vertices ran at the average rate, with 8 compute nodes the job should finish in about 110 minutes. Measured performance is about 133 minutes.

Compared with method MulByAmwhBcolsApplySide, this method spends more work making separate copies of the parts for matrix A, and yet does not manage to run the product vertices any faster. The problem is that 64 different files are produced containing the parts of matrix A and yet there are only 8 compute nodes on which to store them. Absent an ideal schedule, multiple product vertices often end up contending on the same remote disk. So there is still about the same amount of contention for reading the parts of A and the parts of B as there was in method MulByAmwhBcolsApplySide. Table 3 shows the

job times for various numbers of compute nodes.

7.5 Method MulByAmwhBcolsSelect

A limitation in method MulByAmwhBcolsApplySide is that it fails to fully exploit data parallelism. In the Apply stages that multiply and accumulate the product each compute node performs all operations in serial for the A rows and B columns that are input to that node. That is a problem with using Apply in DryadLINQ: if there is data parallelism, you have to program it explictly.

One approach to get around this limitation is to recode the Apply into a GroupBy that groups the partitioned B matrix by column and then a SelectMany that computes the result of multiplying that group by a partition of rows from matrix A. We call this method "MulByAmwhBcolsSelect". Figure 15 shows an example listing and Figure 16 the resulting execution plan. The overall structure of the code and the plan look very much the same as for the MulByAmwhBcols method.

One hazard that is perhaps not obvious arises with grouping an entire row or column of the matrix in this way. For our configuration parameters, if DryadLINQ constructs a plan in which such a group is sent from one vertex to another, during execution of the job Dryad will discover that it needs a record size larger than 2 GB. This is larger than the capacity of the current Dryad, so the job will fail. Fortunately, the MulByAmwhBcolsSelect method creats the column group only for temporary use within individual vertices, so it does not have this issue.

Table 11 summarizes the breakdown of work in this matrix multiplication method. If the scheduling were perfect and all vertices ran at the average rate, with 8 compute nodes the job should finish in about 80 minutes. Measured performance is about 103 minutes. This is 13 minutes faster than running MulByAmwhBcolsApplySide and is entirely due to saving about 2 minutes in each of the multiply-accumulate vertices through processing two columns of matrix B in parallel.

With 64 compute nodes available, the MulByAmwhBcolsSelect plan takes about 40 minutes to run, basically the same as MulByAmwhBcolsApplySide.

```
public static
IQueryable<Tile> MulByAmwhBcolsSelect (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  B = B.HashPartition(b => b.col, numParts);
  var QC = new Queue<IQueryable<Tile>>();
  for (long lp = 0; lp < numParts; lp++)
  {
    var p = lp;
    var Aix = A.Where(a => a.row % numParts == p);
    var Cix = B
      .GroupBy(b => b.col)
      .SelectMany(Bxj => LocalMul(Aix, Bxj));
    QC.Enqueue(Cix);
  }
  var C = QC.Dequeue();
  return C.Apply(QC.ToArray(), cca => Mcat(cca));
}

public static IEnumerable<Tile> LocalMul (
  IEnumerable<Tile> Aix,
  IEnumerable<Tile> Bxj)
{
  var DA = new LazyTiles(tileSize).Acc(Aix);
  var DC = new LazyTiles(tileSize);
  foreach (var b in Bxj)
    foreach (var a in DA.EnCol(b.row))
      DC.Acc(a * b);
  return DC.En();
}

[DistributiveOverConcat]
public static IEnumerable<Tile> Mcat (
  IEnumerable<Tile>[] cca)
{
  var DC = new LazyTiles(tileSize);
  foreach (var cc in cca) DC.Acc(cc);
  return DC.En();
}
```

Figure 15: Matrix multiplication method MulByAmwhBcolsSelect.

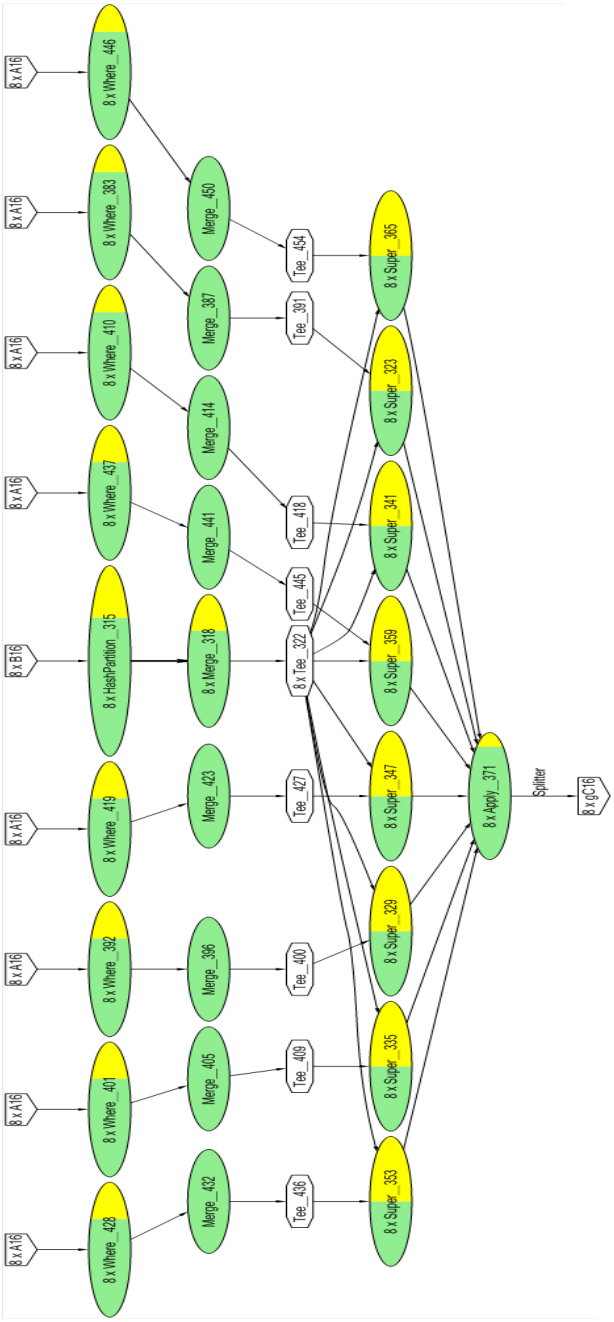


Figure 16: Execution plan for matrix multiplication method MulByAmwhBcolsSelect.

n	m	$n * m$	**what**
8	1	8	partition matrix B
8	4	32	merge matrix B
64	0.5	32	where matrix A rows
8	2	16	merge matrix A rows
64	8	512	compute product
8	5	40	merge product
		640	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 11: Breakdown of work in matrix multiplication method MulByAmwhBcolsSelect.

7.6 Method MulByAmwhBcolsSelectJoin

Method MulByAmwhBcolsSelect uses a LocalMul subroutine to handle computing the product result in the product vertices. This subroutine uses imperative code and the LazyTiles class. Instead, to be more in the flavor of DryadLINQ, we could write the local computation using the IEnumerable Join, GroupBy, and Aggregate methods. We call the resulting method "MulByAmwhBcolsSelectJoin". Figure 17 shows an example listing and Figure 18 the resulting execution plan. The overall structure of the code and the plan look very much the same as for the MulByAmwhBcols method.

Unfortunately, in the current version of DryadLINQ, the aggregation of the results does not start until the local Join is complete. With our configuration parameters, the grouping of matrix B by column results in two columns (4 GB) being stored in each product vertex. Bringing rows of matrix A in via a side-channel results in two rows being stored in each product vertex (4 GB). Each local Join joins one column of B against two rows of A, producing 32 product tiles, but since two columns of B are present in each product vertex, there are two local Joins running producing a total of 64 product tiles (9 GB). The peak memory requirement of 17 GB just barely exceeds the memory capacity of the compute node. As a consequence the com-

```
public static
IQueryable<Tile> MulByAmwhBcolsSelectJoin (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  B = B.HashPartition(b => b.col, numParts);
  var QC = new Queue<IQueryable<Tile>>();
  for (long lp = 0; lp < numParts; lp++)
  {
    var p = lp;
    var Aix = A.Where(a => a.row % numParts == p);
    var Cix = B
      .GroupBy(b => b.col)
      .SelectMany(Bxj => Bxj
        .Join(Aix, b => b.row, a => a.col,
          (b, a) => a * b)
        .GroupBy(c => c.coord)
        .Select(cc => cc.Aggregate((x, y) => x + y)));
    QC.Enqueue(Cix);
  }
  var C = QC.Dequeue();
  return C.Apply(QC.ToArray(), cca => Mcat(cca));
}

[DistributiveOverConcat]
public static IEnumerable<Tile> Mcat (
  IEnumerable<Tile>[] cca)
{
  var DC = new LazyTiles(tileSize);
  foreach (var cc in cca) DC.Acc(cc);
  return DC.En();
}
```

Figure 17: Matrix multiplication method MulByAmwhBcolsSelectJoin.

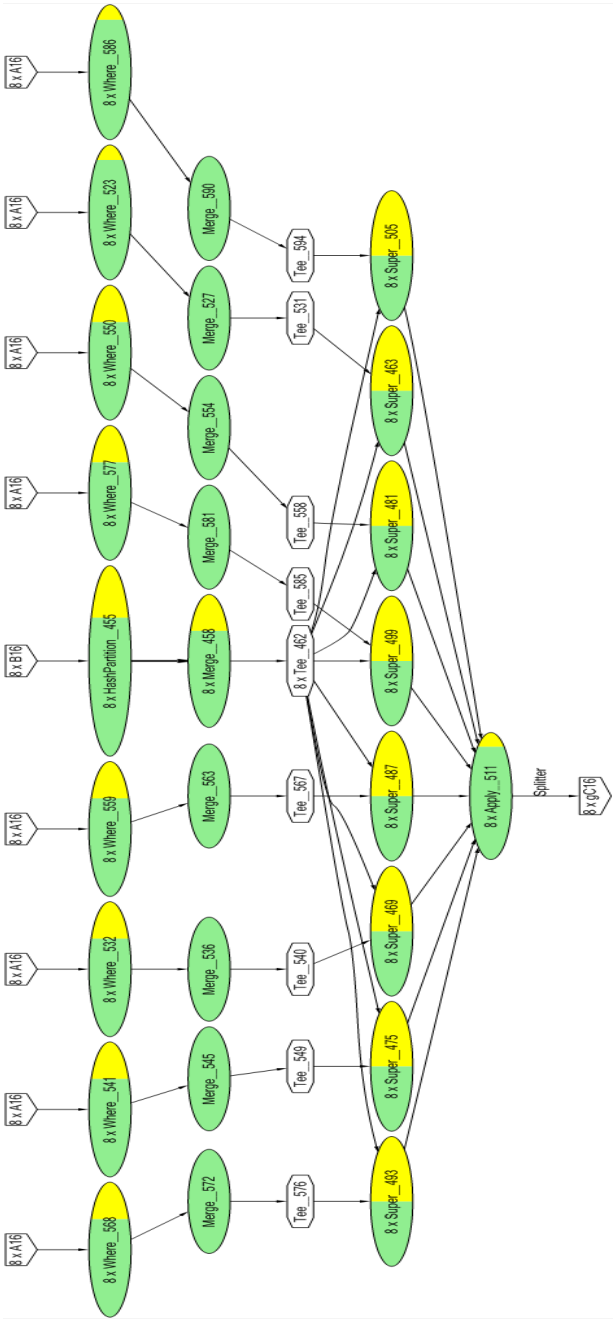


Figure 18: Execution plan for matrix multiplication method MulByAmwhBcolsSelectJoin.

n	m	$n * m$	**what**
8	1	8	partition matrix B
8	4	32	merge matrix B
64	0.5	32	where matrix A rows
8	2	16	merge matrix A rows
64	21	1344	compute product
8	5	40	merge product
		1472	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 12: Breakdown of work in matrix multiplication method MulByAmwhBcolsSelectJoin.

pute node goes into a short spasm of paging.

Table 12 summarizes the breakdown of work in this matrix multiplication method. Average vertex run times are identical to those for method MulByAmwhBcolsSelect except for the product vertices, which take about 21 minutes to run instead of 8. This is entirely due to paging. Hence this method is a bad alternative to MulByAmwhBcolsSelect.

7.7 Method MulByArowsBcolsEnv

Another approach to coordinating every row of matrix A with every column of matrix B takes the idea of stuffing tiles into envelopes that route the tile to the proper computation vertex. Sending copies of a tile to multiple vertices can be accomplished by stuffing copies of it into multiple envelopes. We call this method "MulByArowsBcolsEnv".

The advantage of this method is that it completely avoids any contention between multiple compute nodes reading from the same intermediate file. The disadvantage is that this comes at the price of considerably more data copying.

The idea is that we will have $numParts^2$ computation vertices, each responsible for multiplying a part of the rows of matrix A against a part of the columns of matrix B. Each vertex computes $1/numParts^2$ of the product matrix. For each of the tiles in matrix A and matrix B, we address envelopes and send a copy of the tile to each of the computation vertices that needs it.

To manage the envelopes, we use a TileEnv class. An example listing of the TileEnv class can be found in Appendix D. Since TileEnv is declared as Serializable, DryadLINQ can automatically arrange to send records of that type betweeen vertices, without any special code needed by the programmer.

Each envelope explicitly identifies which computation vertex is its destination. The MkPart routine calculates the part responsible for the product value at a given row and column. The SpreadOverCols routine takes a tile and copies it into envelopes addressed to each of the computation vertices responsible for any column combined with the tile's row. The SpreadOverRows routine does the analogous action for rows combined with the tile's column.

Using the TileEnv class, it is easy to implement matrix multiplication using envelopes. Figure 19 shows an example listing and Figure 20 the resulting execution plan.

Next we describe the performance of this matrix multiplication method given our configuration parameters.

The first step repartitions each input matrix into 64 parts. Inspecting the execution plan, each matrix has a HashPartition stage to accomplish this, and each of the vertices in these stages takes about 3 minutes to run. The purpose of this repartitioning step is to enable more parallelism in the next step.

The next step addresses envelopes, spreading copies of tiles from matrix A over the product columns and spreading copies of tiles from matrix B over the product rows. Inspecting the execution plan, each matrix has a Super stage that merges the HashPartition outputs, addresses envelopes, and repartitions the envelopes for the product matrix. Each of the vertices in these stages takes about 5 minutes to run.

The next step merges envelopes from the previous stage's outputs. Inspecting the execution plan, each matrix has a Merge stage that accomplishes this. Each of the vertices in these stages takes about 5 minutes to run.

So far all that has been accomplished is organizing data. The next step finally does some useful work, by taking the envelopes and computing the product. Inspecting the execution plan, there is a Super stage that uses an Apply with a DistributiveOverConcat subroutine to coordinate

```
public static
IQueryable<Tile> MulByArowsBcolsEnv (
  IQueryable<Tile> A,
  IQueryable<Tile> B)
{
  var n = numParts * numParts;
  var AE = A
    .HashPartition(a => a.coord, n)
    .SelectMany(a => TileEnv.SpreadOverCols(a))
    .HashPartition(ae => ae.part, n);
  var BE = B
    .HashPartition(b => b.coord, n)
    .SelectMany(b => TileEnv.SpreadOverRows(b))
    .HashPartition(be => be.part, n);

  return AE
    .Apply(BE, (AEix, BExj) => MulEnv(AEix, BExj))
    .HashPartition(c => c.coord, numParts);
}

[DistributiveOverConcat]
public static IEnumerable<Tile> MulEnv (
  IEnumerable<TileMux> AEix,
  IEnumerable<TileMux> BExj)
{
  var DA = new LazyTiles(tileSize)
    .Acc(AEix.Select(ae => ae.tile));

  var DB = new LazyTiles(tileSize)
    .Acc(BExj.Select(be => be.tile));

  var DC = new LazyTiles(tileSize);
  foreach (var a in DA.En())
    foreach (var b in DB.EnRow(a.col))
      DC.Acc(a * b);

  return DC.En();
}
```

Figure 19: Matrix multiplication method MulByArowsBcolsEnv.

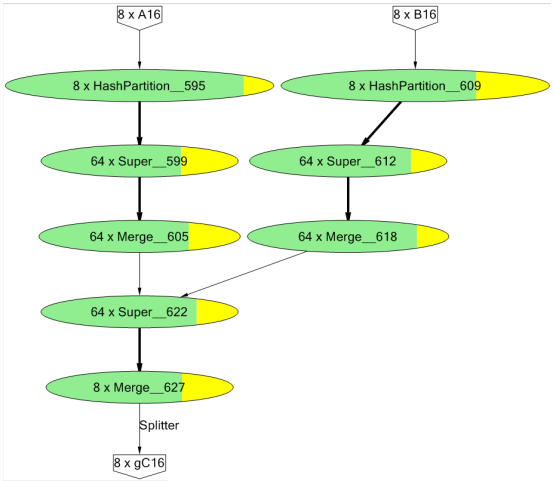


Figure 20: Execution plan for matrix multiplication method MulByArowsBcolsEnv.

the two input parts and compute the product by multiply and aggregate. This stage ends with a HashPartition to repartition the product back to 8 parts from 64. Each of the vertices in this stage takes about 10 minutes to run.

Finally, the last step merges product tiles back into 8 partitions. Inspecting the execution plan, there is a Merge stage the merges the HashPartition outputs from the previous stage. Each of the vertices in this stage takes about 4 minutes to run.

Table 13 summarizes the breakdown of work in this matrix multiplication method. Recall that theoretically only 300 vertex*minutes of work is needed to perform the actual multiplication. Everything else is overhead.

8 Lower-triangular solve

Lower-triangular solve (LTS) is the simplest instance of a division analog in matrix operations. The problem is as follows: given a lower-triangular matrix A and a product matrix B, find the matrix X such that $A * X = B$. Figure 21 gives an illustration.

If you just pick random values for the lower-triangular entries in matrix A, the solution is not likely to be numerically stable. However, the solution is numerically stable if A is strongly diagonally definite, that is, if the absolute

n	m	$n * m$	**what**
16	3	48	initial repartition
128	5	640	distribute envelopes
128	5	640	merge envelopes
64	10	640	compute product
8	4	32	merge product
		2000	total

n = *number of vertices.*
m = *average run time of each vertex in minutes.*

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 13: Breakdown of work in matrix multiplication method MulByArowsBcolsEnv.

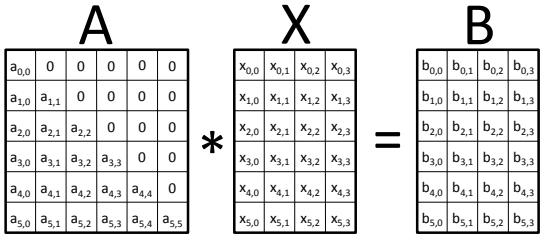


Figure 21: Lower-triangular solve (LTS) problem.

value of each element on the diagonal of A exceeds the sum of the absolute values of all other elements in its row. For testing our algorithms, we always used examples of matrix A that were strongly diagonally definite.

Since each column in the solution matrix X contributes only to the corresponding column in the product matrix B, it is possible to solve for each column of X independently. However, within each column, the rows have to be solved in order, since the value for each row depends on all the previous.

For square matrices of 16 x 16 tiles, matrix lower-triangular solve requires 256 tile lower-triangular solve operations, 1920 tile multiplications, and 1920 tile subtractions. Based on the times in Table 2, it theoretically should take one compute node about 154 minutes to perform a matrix lower-triangular solve. With 8 compute nodes it should theoretically take about 19 minutes.

Of course, these theoretical times are not achievable, because it takes time to read and write the matrices on disk and also time to send tiles around to the right place. The fact that the rows have to be computed in order adds a new level of complexity to solutions of this problem. This is in contrast with matrix multiplication, in which rows could be processed independently and in parallel.

Next we present various methods for performing lower-triangular solve on large matrices using DryadLINQ.

8.1 Method LtsByAorderBcols

One method to compute lower-triangular solve is to partition the product matrix B by columns, store each part in a solution vertex, and then stream row-by-row the entire lower-triangular matrix A into each solution vertex. Each solution vertex solves row-by-row for the entries in its columns of matrix X and, at the end, outputs its solved columns. The storage requirement is feasible since at any time each solution vertex has to store a part of the columns of matrix B, a part of the columns of matrix X, and a row of matrix A.

Actually, we observe that as soon as each entry in matrix X is computed, the corresponding entry in matrix B is no longer needed. So we can use the same storage for matrix X as is used for matrix B. With this refinement, the solution vertex only needs to hold storage for one matrix part for X and B combined, rather than two matrix parts.

The solution vertex uses an Apply with a LeftDistributiveOverConcat subroutine to coordinate one part of matrix B with the entire streamed matrix A. The next question is how to organize the row-by-row streaming of matrix A.

Our first idea was to group matrix A by row and then sort the groups. Unfortunately, this does not work with our configuration parameters, because DryadLINQ will construct an execution plan in which the groups have to be sent from one vertex to another as part of sorting and then streaming into the solution vertex. Since the last row in matrix A spans all the columns, it will need a record size larger than 2 GB, which exceeds the capacity of the current Dryad.

Our second idea was to sort the matrix A tiles by row and then have the solution vertex parse the stream into rows in an online fashion in order to compute the solution row-by-row. We call this method "LtsByAorderBcols0". Figure 22 shows a listing.

To manage the parsing, we use a RowScanInOrder class, listed in Appendix E. The static method RowScanInOrder.Do takes an IEnumerable of Tile sorted by row, parses it, and enumerates the rows. The method anticipates sorting methods that sort only into buckets of rows, where each bucket contains $stride$ rows. At this point, we just say $stride = 1$.

The main part of the lower-triangular solve method just partitions B by columns, sorts A by rows, and then coordinates each part of B with the entire sorted A.

Figure 23 shows the resulting execution plan.

Unfortunately, this method does not work. The problem occurs when DryadLINQ merges the multi-part sorted matrix A into one file in preparation for streaming it through the solution vertex. In spite of the fact that the collection is *sorted*, the current version of DryadLINQ picks a *random order* in which to merge the partitions. This, of course, makes the computation totally wrong. When we discovered this problem, we added the exception checks to RowScanInOrder.Do. These exceptions cause the solution vertex to fail which eventually causes the job to fail.

After some consultation with the DryadLINQ team, we determined that the only way to access a sorted collection as a query and keep it in order is to use one of the so-called "Partitioning Operators" Take, Skip, TakeWhile, or SkipWhile. None of the other operators guarantee to preserve

```
public static IQueryable<Tile> LtsByAorderBcols0 (
  IQueryable<Tile> A, long Arows,
  IQueryable<Tile> B)
{
  A = A.OrderBy(a => a.row);
  B = B.HashPartition(b => b.col,numParts);
  return B.Apply(A,
    (Bxj,SA) => LtsStream(Arows,SA,Bxj));
}

[LeftDistributiveOverConcat]
public static IEnumerable<Tile> LtsStream (
  long Arows,
  IEnumerable<Tile> SA,
  IEnumerable<Tile> Bxj)
{
  var DB = new LazyTiles(tileSize).Acc(Bxj);
  var cols = DB.En()
    .Select(b => b.col).Distinct().ToArray();

  var RSIO = RowScanInOrder.Do(Arows, 1, SA);
  foreach (var R in RSIO)
  {
    var i = R.row;
    var DA = R.DA;

    foreach (var j in cols)
    {
      // subtract [i,j] of A*X from B
      foreach (var a in DA.EnRow(i))
        if (a.col < i)
          foreach (var x in DB.EnCoord(a.col, j))
            DB.AccNeg(a * x);

      // solve for X[i,j]
      DB[i, j] = Tile.Lts(DA[i, i], DB[i, j]);
    }
  }
  return DB.En();
}
```

Figure 22: Matrix lower-triangular solve method LtsByAorderBcols0.
This method does not work.

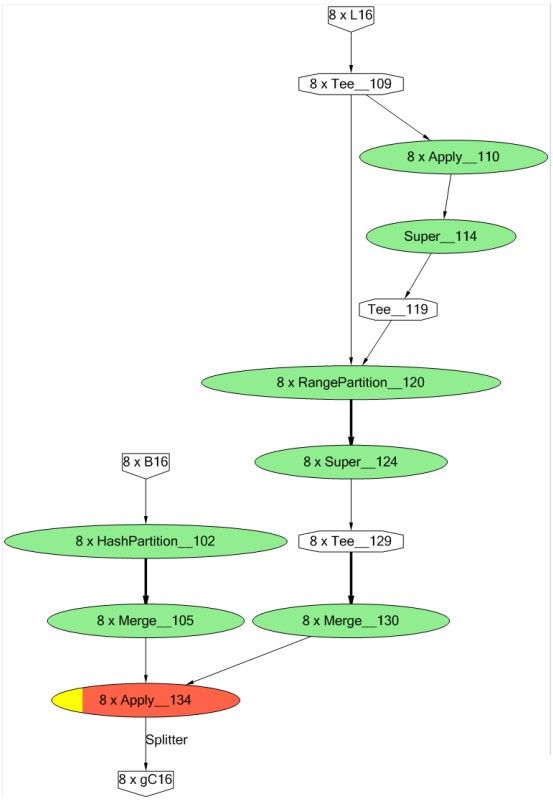


Figure 23: Execution plan for matrix lower-triangular solve method LtsByAorderBcols0.

order. So we added a Skip(0) to transform our multi-part sorted collection into a single part sorted collection. We call this method "LtsByAorderBcols". Figure 24 shows a listing and Figure 25 the resulting execution plan.

Next we describe the performance of this matrix lower-triangular solve method given our configuration parameters.

To sort a collection, DryadLINQ takes the approach of sampling the sorting key, repartitioning the collection into roughly equal piles based ranges computed from the sample, and then sorting within each range. The sampling is done first within each part, and then the samples combined to give an overall sample.

Inspecting the execution plan, sampling within each part of matrix A requires reading all the tiles in the part and then writing essentially a trivial amount of data. Since A is a lower-triangular matrix with the upper-triange zero tiles omitted, each part contains about 17 tiles. If the sampling vertex is lucky and can read its part from the local disk, it takes about 15 seconds. If the sampling vertex is unlucky and must read its part remotely, it takes about 2 minutes. The average run time is about 1 minute.

One vertex combines the samples and computes the range keys for repartitioning the collection. It takes less than a second to run.

The range keys are distributed to range partition vertices that repartition the matrix. Each range partition vertex reads and writes about 17 tiles. The average run time is about 1 minute, but again it depends greatly on whether the vertex is lucky or unlucky.

Next come merge vertices that read the outputs of the range partition vertices, sort their tiles by row, and then write out their part. Each vertex reads and writes about 17 tiles, with almost all of the reading being remote. The average run time is about 2 minutes.

In the final stage of preparing matrix A, a single skip vertex reads all of the sorted parts and writes a single file. It reads and writes 136 tiles. Almost all of the reads are remote and the run time is about 17 minutes.

The preparation for matrix B consists in repartitioning it by columns. The first stage consists of hash partition vertices that read and write about 32 tiles each. The average run time is about 2 minutes.

Next come merge vertices that read the output of the hash partition vertices. Each vertex reads and writes 32 tiles, with allmost all of the reading being remote. The

```
public static IQueryable<Tile> LtsByAorderBcols (
  IQueryable<Tile> A, long Arows,
  IQueryable<Tile> B)
{
  A = A.OrderBy(a => a.row)
       .Skip(0);
  B = B.HashPartition(b => b.col,numParts);
  return B.Apply(A,
    (Bxj,SA) => LtsStream(Arows,SA,Bxj));
}

[LeftDistributiveOverConcat]
public static IEnumerable<Tile> LtsStream (
  long Arows,
  IEnumerable<Tile> SA,
  IEnumerable<Tile> Bxj)
{
  var DB = new LazyTiles(tileSize).Acc(Bxj);
  var cols = DB.En()
    .Select(b => b.col).Distinct().ToArray();

  var RSIO = RowScanInOrder.Do(Arows, 1, SA);
  foreach (var R in RSIO)
  {
    var i = R.row;
    var DA = R.DA;

    foreach (var j in cols)
    {
      // subtract [i,j] of A*X from B
      foreach (var a in DA.EnRow(i))
        if (a.col < i)
          foreach (var x in DB.EnCoord(a.col, j))
            DB.AccNeg(a * x);

      // solve for X[i,j]
      DB[i, j] = Tile.Lts(DA[i, i], DB[i, j]);
    }
  }
  return DB.En();
}
```

Figure 24: Matrix lower-triangular solve method LtsByAorderBcols.

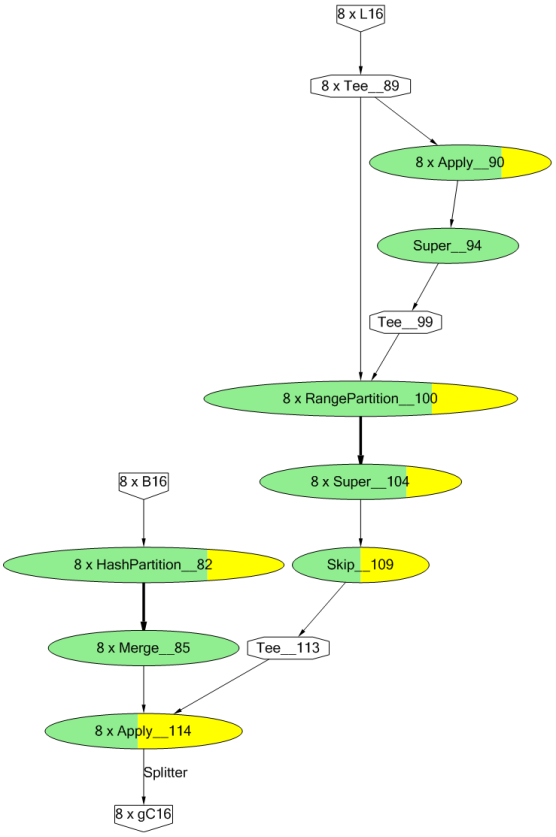


Figure 25: Execution plan for matrix lower-triangular solve method LtsByAorderBcols.

n	m	$n * m$	**what**
8	2	16	sample A by parts
1	0	0	combine samples
8	1	8	range partition A
8	2	16	merge A
8	2	16	hash partition B
8	4	32	merge B
8	45	360	compute solution
		448	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 14: Breakdown of work in matrix lower-triangular solve method LtsByAorderBcols.

average run time is about 4 minutes.

Finally, an apply stage combines the columns of matrix B with the sorted matrix A. Each vertex reads 32 tiles from matrix B and 136 tiles from the sorted matrix A; performs 32 tile ltsolve operations, 240 tile multiplications, and 240 tile subtractions; and writes 32 tiles of matrix X. In almost all cases reading from the sorted matrix A is remote and with contention. The average run time is about 45 minutes.

Table 14 summarizes the breakdown of work in this matrix lower-triangular solve method. Recall that theoretically only 154 vertex*minutes of work is needed to perform the actual computation. Everything else is overhead.

If all vertices ran at the average speed, with 8 compute nodes the job should be complete in about 56 minutes. Measured completion time is about 77 minutes. The extra time is due to variance in vertex completion time and the fact that in this execution plan, most stages depend on all vertices in the previous stage completing before any vertex can start. Hence, the job run time tends to be dominated by the worst case vertex time.

8.2 Method LtsByArangeBcols

One variation of method LtsByAorderBcols replaces the sorting step with an explict range partition. Whereas the sorting step samples the number of tiles in each row in order to get a partition with approximately the same number of tiles in each part, we can simply create partition separators to put approximately the same number of rows in each part. We call this method "LtsByArangeBcols". Figure 26 shows a listing and Figure 27 the resulting execution plan.

There is a little unfortunate complexity in setting up the range partition separators. The RangePartition operator specifies that it is "not possible to predict" whether keys that match a partition separator are sent to the lower bin or to the upper bin. Since we want to be able to direct where all the tiles go, we scale and offset the separators and keys so that there will be no ties.

As in method LtsByAorderBcols, to avoid the random partition order merge problem, we use a Skip(0) to reduce the partitioned collection to a single file.

The execution plan is simpler than for method LtsByAorderBcols, since the sampling stages are omitted. At first glance, this should save about two minutes of job run time, but it does not. With eight compute nodes, method LtsByAorderBcols and method LtsByArangeBcols complete in the same amount of time.

It turns out that in method LtsByArangeBcols, the vertices in the merge stage that follows the range partition are unbalanced. Because the range partition divides the lower-triangular matrix A evenly by rows, the number of tiles in each part are far from equal. The merge vertex with the least work has almost no work and the vertex with the most work has almost twice as much as average, or about two extra minutes of work. The skip vertex cannot start until all of the merge vertices are finished. The extra two minutes delay completely eats up the two minutes that were saved by omitting the sampling stages.

8.3 Method LtsByAmwhBcolsmwh

Streaming the entire matrix A through solution vertices causes these vertices to run for a long time. It might be better to break the computation up more so that more implicit checkpoints can be taken.

```
public static IQueryable<Tile> LtsByArangeBcols (
  IQueryable<Tile> A, long Arows,
  IQueryable<Tile> B)
{
  var stride = (Arows + numParts - 1) / numParts;
  var keys = Enumerable.Range(1, numParts - 1)
    .Select(i => i * stride * 2)
    .ToArray();

  A = A.RangePartition(a => a.row * 2 + 1, keys)
       .Skip(0);
  B = B.HashPartition(b => b.col,numParts);
  return B.Apply(A,
    (Bxj,SA) => LtsStream(Arows,stride,SA,Bxj));
}

[LeftDistributiveOverConcat]
public static IEnumerable<Tile> LtsStream (
  long Arows,
  long stride,
  IEnumerable<Tile> SA,
  IEnumerable<Tile> Bxj)
{
  var DB = new LazyTiles(tileSize).Acc(Bxj);
  var cols = DB.En()
    .Select(b => b.col).Distinct().ToArray();

  var RSIO = RowScanInOrder.Do(Arows, stride, SA);
  foreach (var R in RSIO)
  {
    var i = R.row;
    var DA = R.DA;

    foreach (var j in cols)
    {
      // subtract [i,j] of A*X from B
      foreach (var a in DA.EnRow(i))
        if (a.col < i)
          foreach (var x in DB.EnCoord(a.col, j))
            DB.AccNeg(a * x);

      // solve for X[i,j]
      DB[i, j] = Tile.Lts(DA[i, i], DB[i, j]);
    }
  }
  return DB.En();
}
```

Figure 26: Matrix lower-triangular solve method LtsByArangeBcols.

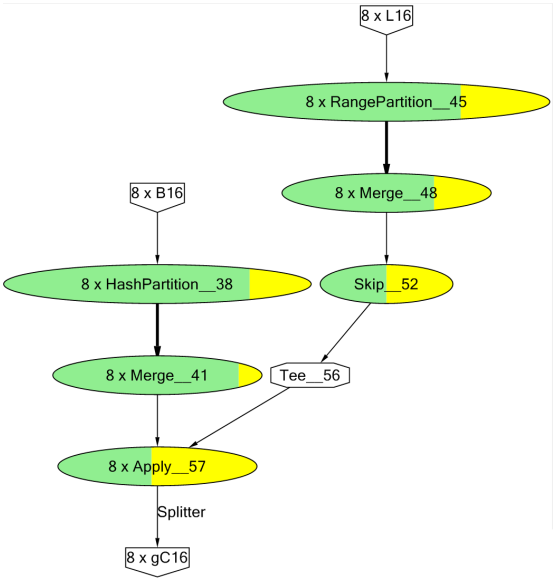


Figure 27: Execution plan for matrix lower-triangular solve method LtsByArangeBcols.

One way to do that is to compute rows of X step-by-step in stages. We start out with an empty solution matrix X. In each iteration, we feed the next few rows of the matrix X, the next few rows of the product B, and the current solution matrix X in to a stage and get as output the solution matrix X with the next few rows added. In each stage, the columns of B and X are partitioned across a number of vertexes.

We use Where operators to extract the next few rows of B and of A. The rows of B remain partitioned by columns. We use a side-channel to broadcast the rows of A to all vertices in the stage. We call this method "LtsByAmwhBcolsmwh". Figure 28 shows a listing and Figure 29 the resulting execution plan.

One tricky part is how the DistributiveOverConcat local lower-triangular solve subroutine determines which columns in B and X it is responsible for. One the first iteration, nothing is yet solved for matrix X, and if matrix B happened to have omitted zero tiles in the first few rows, there would be no way for the subroutine to know which columns to solve for. This is not a problem in the streaming methods because in those cases the subroutine has entire columns of B, and if the entire column omitted

then there is not need to solve for that column of X, since it is zero.

The way we solve this problem is to initialize X to have explicit zero tiles in row 0 for every non-empty column in B. The subroutine can then determine which columns it is responsible for by looking at row 0 in X.

Examining the execution plan for this method, the main feature is a central stem of apply stages, with rows of B being extracted and input on one side and rows of A being extracted and input on the other side. Each successive apply stage takes longer to execute than the previous, since it has to read and write all of the rows currently solved for X, which gets longer each time. In addition, the rows for A get longer and computing the next rows in X takes longer.

In spite of the greatly increased complexity in the execution plan, however, this method completes in about the same time as the streaming methods. Compared to the streaming methods, this method does not have to spend time sorting the lower-triangular matrix and writing it all into one file. However, that savings is lost in the time it takes to read and write the currently solved rows of X in each apply stage.

Table 3 shows the run time of this method for various numbers of compute nodes. The run times are all about the same. The critical path in this method is the central stem of apply stages, and these stages cannot use any more than 8 compute nodes.

8.4 Method LtsByAmwhmwhBcolsmwh

In computing lower-triangular solve, before solving for $x_{i,k}$, you first have to subtract the inner product $\sum_{j=0}^{i-1} a_{i,j} * x_{j,k}$ from $b_{i,}$. Method LtsByAmwhBcolsmwh computes this inner product in the vertex that solves for $x_{i,k}$. However, the needed values of $x_{j,k}$ are available earlier, some much earlier, and more parallelism can be exhibited by accumulating the inner product step-by-step in separate vertices.

Organizing the computation is somewhat tricky. As in method LtsByAmwhBcolsmwh, we iterate through rows of B and A, using Where operators to extract a batch of the next few rows. We adjust the row batch of B by subtracting the inner products. We use a Where operator to extract the diagonal batch of A, taking the same columns

```
public static IQueryable<Tile> LtsByAmwhBcolsmwh (
  IQueryable<Tile> A, long Arows,
  IQueryable<Tile> B)
{
  long stride = (Arows + numParts - 1) / numParts;
  A = A.HashPartition(a => a.col,numParts);
  B = B.HashPartition(b => b.col,numParts);
  X = B
    .GroupBy(b => b.col)
    .Select(g => new Tile(tileSize,0,g.Key));

  for (long p = 0; p < numParts; p++)
  {
    var fi = Math.Max((p + 0) * stride, 0);
    var ai = Math.Min((p + 1) * stride, Arows);
    var Aix = A.Where(a => fi <= a.row && a.row < ai);
    var Bix = B.Where(b => fi <= b.row && b.row < ai);
    X = X.Apply(Bix,
      (Xxj,Bij) => LtsChain(fi,ai,Aix,Xxj,Bij));
  }
  return X;
}

[DistributiveOverConcat]
public static IEnumerable<Tile> LtsChain (
  long fi, long ai,
  IEnumerable<Tile> Aix,
  IEnumerable<Tile> Xxj,
  IEnumerable<Tile> Bij)
{
  var DA = new LazyTiles(tileSize).Acc(Aix);
  var DX = new LazyTiles(tileSize).Acc(Xxj);
  var DB = new LazyTiles(tileSize).Acc(Bij);
  var cols = DX.EnRow(0).Select(x => x.col).ToArray();
  foreach (var j in cols)
  {
    for (var i = fi; i < ai; i++)
    {
      // subtract [i,j] of A*X from B
      foreach (var a in DA.EnRow(i))
        foreach (var x in DX.EnCoord(a.col,j))
          DB.AccNeg(a * x);

      // solve for X[i,j]
      DX[i, j] = Tile.Lts(DL[i, i], DB[i, j]);
    }
  }
  return DX.En();
}
```

Figure 28: Matrix lower-triangular solve method LtsByAmwhBcolsmwh.

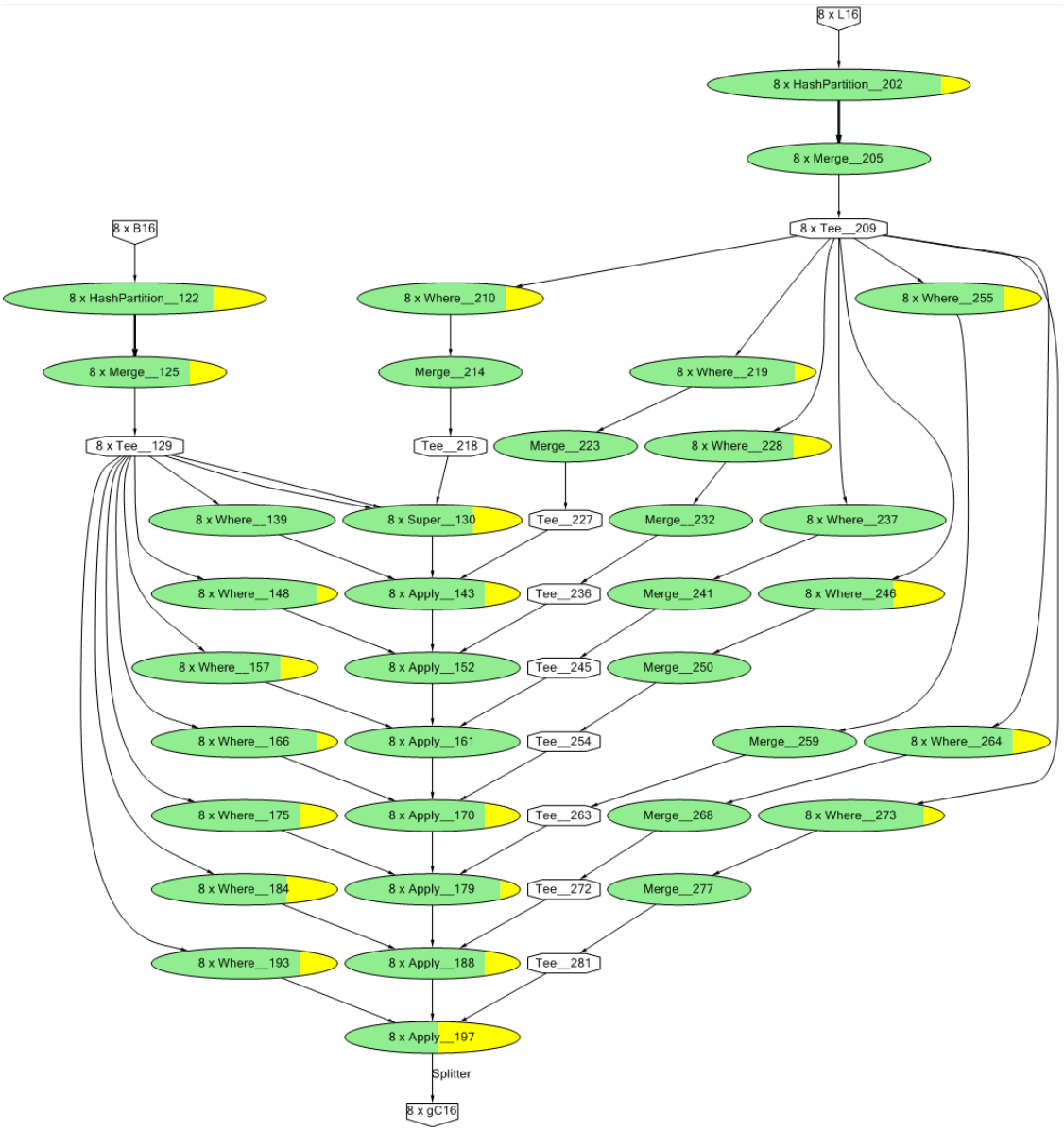


Figure 29: Execution plan for matrix lower-triangular solve method LtsByAmwhBcolsmwh.

as appear in the row batch. An apply stage, partitioned by columns in B, takes the adjusted row batch of B and a side-channel broadcast of the diagonal batch of A, and computes the solution X for this row batch.

Instead of combining the solution X into a single data set, we leave each row batch as a separate data set.

To adjust the row batch of B by subtracting the inner products, we iterate through columns of A within the current row batch. We use a Where operator to extract the next few columns, corresponding to a row batch already solved in X. An apply stage, partitioned by columns in B, takes the current adjusted row batch of B, the row batch of X, and a side-channel broadcast of the corresponding row-and-column batch of A, and updates the adjusted row batch of B.

At the end, we consolidate all of the row batches of X to make the final result.

We call this method "LtsByAmwhmwhBcolsmwh". Figures 30 and 31 show a listing and Figure 32 the resulting execution plan.

Examining the execution plan for this method reveals that it is too complicated to see what is going on. Figure 33 shows the logical structure of the method.

For our configuration parameters, detailed investigation of performance logs shows that the job spends about the first half hour repartitioning A and B and then extracting row and column batches of A.

There are 8 where stages that extract row batches of A. On average, each vertex in these stages takes about half a minute, reading 16 tiles and writing 2 tiles, but it varies greatly, since the higher-numbered rows have more tiles.

There are 36 where stages that extract column batches from the row batches of A. On average each vertex in these stages has very little to do and takes almost no time to run, since each of the column batches contains only 4 tiles for the entire stage.

There are 36 merge stages that merge the column batches into one file for broadcast. The vertex in these stages takes about half a minute to run, reading and writing 4 tiles.

After extracting row and column batches of A, the job starts running solution and adjustment stages, the solution stages working down the diagonal of A and the adjustment stages computing the inner product adjustment to B.

Each solution stage vertex takes about 1 minute to run, reading 3 tiles of A and 4 tiles of adjusted B, performing

```
public static
IQueryable<Tile> LtsByAmwhmwhBcolsmwh (
  IQueryable<Tile> A, long Arows,
  IQueryable<Tile> B)
{
  long stride = (Arows + numParts - 1) / numParts;
  A = A.HashPartition(a => a.col,numParts);
  B = B.HashPartition(b => b.col,numParts);

  var XX = new IQueryable<Tile>[numParts];

  // loop through each row batch
  for (long lpi = 0; lpi < numParts; lpi++)
  {
    var pi = lpi;
    var fi = Math.Max((pi + 0) * stride, 0);
    var ai = Math.Min((pi + 1) * stride, Arows);

    var Aix = A.Where(a => fi <= a.row && a.row < ai);
    var Bix = B.Where(b => fi <= b.row && b.row < ai);

    // loop through each col batch
    for (long lpj = 0; lpj < pi; lpj++)
    {
      var pj = lpj;
      var fj = Math.Max((pj + 0) * stride, 0);
      var aj = Math.Min((pj + 1) * stride, Arows);
      // Arows == Acols

      var Aij = Aix
        .Where(a => fj <= a.col && a.col < aj);
      var Xjx = XX[pj];

      Bix = Bix.Apply(Xjx,
        (Bik, Xjk) => SumT(Bik, Aij, Xjk));
    }

    var Aii = Aix
      .Where(a => fi <= a.col && a.col < ai);

    XX[pi] = Bix.Apply(Bik => LtsT(fi, ai, Aii, Bik));
  }

  var XQ = new Queue<IQueryable<Tile>>(XX);
  var X = XQ.Dequeue();
  if (XQ.Count > 0)
    X = X.Apply(XQ.ToArray(), xxa => Mcat(xxa));

  return X;
}
```

Figure 30: Matrix lower-triangular solve method LtsByAmwhmwhBcolsmwh (part 1).

```
[DistributiveOverConcat]
public static IEnumerable<Tile> SumT(
  IEnumerable<Tile> Bik,
  IEnumerable<Tile> Aij,
  IEnumerable<Tile> Xjk)
{
  var DB = new LazyTiles(tileSize).Acc(Bik);
  var DX = new LazyTiles(tileSize).Acc(Xjk);
  foreach (var a in Aij)
    foreach (var x in DX.EnRow(a.col))
      DB.AccNeg(a * x);
  return DB.En();
}

[DistributiveOverConcat]
public static IEnumerable<Tile> LtsT(
  long fi,
  long ai,
  IEnumerable<Tile> Aii,
  IEnumerable<Tile> Bik)
{
  var DX = new LazyTiles(tileSize);
  var DA = new LazyTiles(tileSize).Acc(Aii);
  var DB = new LazyTiles(tileSize).Acc(Bik);

  var kk = DB.En()
    .Select(b => b.col).Distinct().ToArray();

  foreach (var k in kk)
  {
    for (var i = fi; i < ai; i++)
    {
      // subtract [i,k] of A*X from B
      foreach (var a in DA.EnRow(i))
        foreach (var x in DX.EnCoord(a.col, k))
          DB.AccNeg(a * x);

      // solve for X[i,k]
      DX[i, k] = Tile.Lts(DA[i, i], DB[i, k]);
    }
  }
  return DX.En();
}

[DistributiveOverConcat]
public static IEnumerable<Tile> Mcat (
  IEnumerable<Tile>[] xxa)
{
  var DX = new LazyTiles(tileSize);
  foreach (var xx in xxa) DX.Acc(xx);
  return DX.En();
}
```

Figure 31: Matrix lower-triangular solve method LtsByAmwhmwhBcolsmwh (part 2).

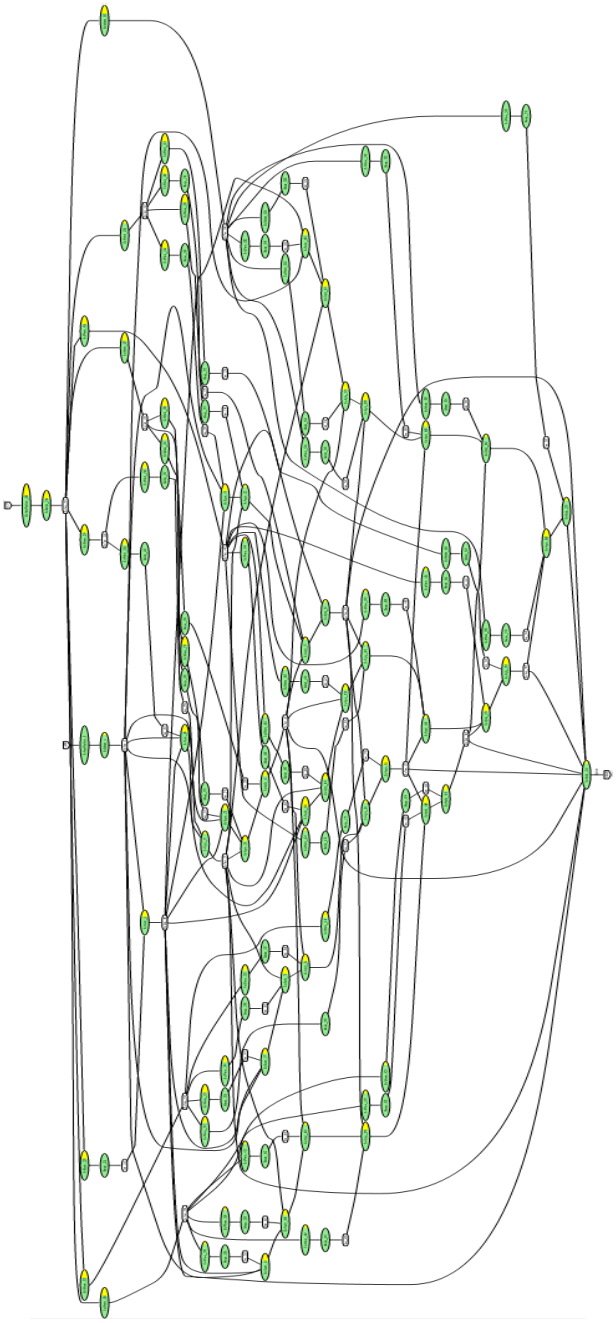


Figure 32: Execution plan for matrix lower-triangular solve method LtsByAmwhmwhBcolsmwh.

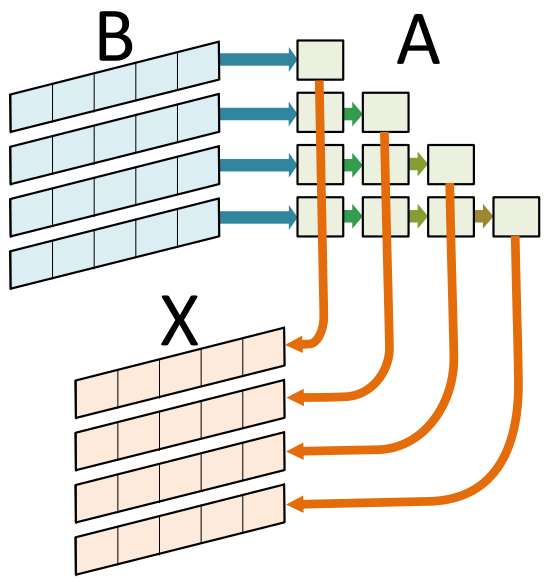


Figure 33: Logical structure of matrix lower-triangular solve method LtsByAmwhmwhBcolsmwh.

n	m	$n * m$	**what**
16	2	32	hash matrix A, B
16	4	64	merge matrix A, B
64	0.5	32	where A row
288	0.1	3	where A column in row
36	0.5	18	merge A column in row
8	3	24	where B row, first solution
56	3	168	where B row, first adjustment
56	1	56	subsequent solution
168	1	168	subsequent adjustment
8	4	32	consolidation
		597	total

n = number of vertices.
m = average run time of each vertex in minutes.

Tile size 4096 by 4096 doubles (134 MB) running on a quad-core AMD Opteron™ processor 2373 EE at 2.10 GHz with 2 processors and 16 GB memory. Matrix size 16 x 16 tiles (34 GB) partitioned into 8 parts. HPC Dryad Beta 3702 with custom serialization. Running on 8 compute nodes. Number of nodes does not count the job manager.

Table 15: Breakdown of work in matrix lower-triangular solve method LtsByAmwhmwhBcolsmwh.

four tile lower-triangular solve operations, two tile multiplications, and two tile subtractions, and writing 4 tiles of X.

Each adjustment stage vertex takes about 1 minute to run, reading 4 tiles of A, 4 tiles of X, and 4 tiles of adjusted B, performing eight tile multiplications and 8 tile subtractions, and writing 4 tiles of adjusted B.

However, there is a difference in the *first* solution stage on the diagonal and in the *first* adjustment stage in each row. In these first stages, DryadLINQ optimizes the computation by combining the solution or adjustment computation with a where that exracts the relevant row of B. So, in each first stage vertex, instead of reading 4 tiles of adjusted B it instead reads 32 tiles of B. This makes each first stage vertex take 3 minutes instead of 1 minute.

As it turns out, this DryadLINQ optimization *increases* the run time of the job, because the first adjustment stage vertices cannot start until the solution of the first row batch of X has been computed by the first solution stage. If the where had not been combined with the first adjustment stage, it could have been started earlier.

There is one first solution stage, seven subsequent solution stages, seven first inner product stages, and 21 subsequent inner product stages. Finally, the job performs a consolidation stage to consolidate the final result. A consolidation stage vertex takes about 4 minutes to run, reading and writing 32 tiles of X.

Table 15 summarizes the breakdown of work in this matrix lower-triangular solve method. Recall that theoretically only 154 vertex*minutes of work is needed to perform the actual computation. Everything else is overhead.

Table 3 shows the run time of this method for various number of compute nodes. Although with eight compute nodes the run time is significantly longer than for method LtsByAmwhBcolsmwh, with more compute nodes it improves and runs faster than method LtsByAmwhBcolsmwh, thus showing that it exhibits more parallelism. This would be especially important in situations where matrix B had few columns, and thus little parallelism would be available to methods that only exploited column independence.

References

[1] Message passing interface forum. http://www.mpi-forum.org/.

[2] A. Fitzgibbon and O. Williams. Coconut. http://research.microsoft.com/en-us/projects/coconut/.

[3] Intel. Intel® math kernel library. http://software.intel.com/en-us/articles/intel-mkl/.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[5] D. Pattee. Dryad beta program update, May 2011. http://blogs.technet.com/b/windowshpc/archive/2011/05/06/dryad-beta-update.aspx.

[6] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[7] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Operating Systems Design and Implementation*, pages 1–14, 2008.

A Coord class

```
[Serializable]
struct Coord : IEquatable<Coord>
{
  public long row;
  public long col;

  public Coord(long row,long col)
  {
    this.row = row;
    this.col = col;
  }

  public override int GetHashCode() ...
  // a good hash code combining row and col

  public bool Equals(Coord b)
  {
    return row == b.row && col == b.col;
  }
}
```

B Tile class

This is a naive example implementation of the Tile class. We actually used BLAS routines from the Intel® Math Kernel Libary [3], via the C# wrappers provided by Coconut [2], which required using the appropriate Coconut data structure to store the matrix. We also wrote custom Dryad serialization code to improve the performance of reading and writing tiles. None of these performance improvements are shown here.

```
[Serializable]
class Tile
{
  public int size;
  public Coord coord;
  public double[,] data;

  public long row { get { return coord.row; } }
  public long col { get { return coord.col; } }

  public Tile(int size,Coord coord)
  {
    this.size = size;
    this.coord = coord;
    this.data = new double[size,size];
  }

  public Tile(int size,long row,long col)
    : this(size,new Coord(row,col))
  {
  }

  public Tile(Tile b) : this(b.size,b.coord)
  {
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        this[i,j] = b[i,j];
  }

  public double this[int i,int j]
  {
    get { return data[i,j]; }
    set { data[i,j] = value; }
  }

  public static Tile operator +(Tile a,Tile b)
  {
    // add two tiles
    int size = a.size;
    var c = new Tile(size,a.coord);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        c[i,j] = a[i,j] + b[i,j];
```

```
    return c;
  }

  public static Tile operator -(Tile a,Tile b)
  {
    // subtract two tiles
    int size = a.size;
    var c = new Tile(size,a.coord);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        c[i,j] = a[i,j] - b[i,j];
    return c;
  }

  public static Tile operator *(Tile a,double s)
  {
    // scale tile
    int size = a.size;
    var c = new Tile(size,a.coord);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        c[i,j] = a[i,j] * s;
    return c;
  }

  public static Tile operator *(Tile a,Tile b)
  {
    // multiply two tiles
    int size = a.size;
    var c = new Tile(size,a.row,b.col);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        for (int k = 0; k < size; k++)
          c[i,j] += a[i,k] * b[k,j];
    return c;
  }

  public static Tile Lts(Tile a,Tile b)
  {
    // lower-triangular solve
    // find tile x such that a*x = b
    // where a is lower triangular
    int size = a.size;
    var x = new Tile(b);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++) {
        for (int k = 0; k < i; k++) {
          x[i,j] -= a[i,k] * x[k,j]
        }
        x[i,j] /= a[i,i]
      }
    return x;
  }
}
```

C LazyTiles class

```
class LazyTiles
{
  // A lazy array of tiles
  private int size;
  private Dictionary<Coord,Tile> dict;

  public LazyTiles(int size)
  {
    this.size = size;
    this.dict = new Dictionary<Coord,Tile>();
  }

  public Tile this[Coord c]
  {
    // access tile at given coordinate
    get
    {
      if (!dict.ContainsKey(c))
        dict[c] = new Tile(size,c);
      return dict[c];
    }
    set { dict[c] = value; }
  }

  public Tile this[long i,long j]
  {
    // access tile at given coordinate
    get { return this[new Coord(i,j)]; }
    set { this[new Coord(i,j)] = value; }
  }

  public LazyTiles Acc(Tile a)
  {
    // accumulate tile into the array
    if (!dict.ContainsKey(a.coord))
      this[a.coord] = a;
    else
      this[a.coord] += a;
    return this;
  }

  public LazyTiles Acc(IEnumerable<Tile> aa)
  {
    // accumulate sequence of tiles
    foreach (var a in aa) Acc(a);
    return this;
  }

  public LazyTiles AccNeg(Tile a)
  {
    // accumulate negative of tile into the array
    this[a.coord] -= a;
    return this;
  }

  public LazyTiles AccNeg(IEnumerable<Tile> aa)
  {
    // accumulate negative of sequence of tiles
    foreach (var a in aa) AccNeg(a);
    return this;
  }
```

```
  public IEnumerable<Tile> En()
  {
    // enumerate all tiles in the array
    return dict.Values;
  }

  public IEnumerable<Tile> EnRow(long i)
  {
    // enumerate all tiles in a given row
    return En().Where(a => a.row == i);
  }

  public IEnumerable<Tile> EnCol(long j)
  {
    // enumerate all tiles in a given column
    return En().Where(a => a.col == j);
  }

  public IEnumerable<Tile> EnCoord(Coord c)
  {
    // enumerate any tile at given coordinate
    if (dict.ContainsKey(c))
      yield return dict[c];
  }

  public IEnumerable<Tile> EnCoord(long i,long j)
  {
    // enumerate any tile at given coordinate
    return EnCoord(new Coord(i,j));
  }
}
```

D TileEnv class

```
[Serializable]
public class TileEnv
{
  // a tile inside an envelope; the envelope
  // is addressed to a particular part index
  public int part;
  public Tile tile;

  public TileEnv (int part, Tile tile)
  {
    this.part = part;
    this.tile = tile;
  }

  public static int MkPart(long r, long c)
  {
    // part index for a given row and column
    int rp = (int)(r % numParts);
    int cp = (int)(c % numParts);
    return rp + (cp * numParts);
  }

  public static IEnumerable<TileEnv>
  SpreadOverCols(Tile a)
  {
    // copy a tile into envelopes addressed to
    // part indexes for any column combined with
    // the tile's row
    for (int p = 0; p < numParts; p++)
      yield return new TileEnv(MkPart(a.row, p), a);
  }

  public static IEnumerable<TileEnv>
  SpreadOverRows(Tile b)
  {
    // copy a tile into envelopes addressed to
    // part indexes for any row combined with
    // the tile's column
    for (int p = 0; p < numParts; p++)
      yield return new TileEnv(MkPart(p, b.col), b);
  }
}
```

E RowScanInOrder class

```
public class RowScanInOrder
{
  // A LazyTiles that has valid tiles in the given row.
  public class Row
  {
    public long row;
    public LazyTiles DA;

    public Row(long row, LazyTiles DA)
    {
      this.row = row;
      this.DA = DA;
    }
  }

  // Assuming SA is sorted by parts, that is, first
  // come all tiles in rows [0..stride), then all tiles
  // in rows [stride..2*stride), and so on, enumerate
  // it by consecutive rows [0..rows).
  //
  // At any time, never store more than one part of
  // tiles plus a small constant number of tiles.
  //
  public static IEnumerable<Row> Do(
    long rows,
    long stride,
    IEnumerable<Tile> SA)
  {
    IEnumerator<Tile> Le = SA.GetEnumerator();
    bool notend = Le.MoveNext();
    long firstrow = 0;
    while (firstrow < rows)
    {
      long afterrow = Math.Min(firstrow + stride, rows);

      var DA = new LazyTiles(tileSize);
      while (notend && Le.Current.row < afterrow)
      {
        if (Le.Current.row < firstrow)
```

```
          throw new Exception("SA not sorted by parts");

        DA.Acc(Le.Current);
        notend = Le.MoveNext();
      }

      for (long row = firstrow; row < afterrow; row++)
        yield return new Row(row, DA);

      firstrow = afterrow;
    }
    if (notend)
      throw new Exception("SA row out of range");
  }
}
```