Proposed Parallel Architecture for Matrix Triangularization with Diagonal Loading

Charles M. Rader

*This work is sponsored by Defense Advanced Research Projects Agency, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Sept. 29, 2004

The Matrix Triangularization Problem

A common task in adaptive signal processing is as follows: We have a set of N training vectors, each with M components. These constitute a matrix X and we need the Cholesky factor, T, of its correlation matrix $R=XX^h + \lambda I$.

Usually $N >> M$.

The cost of the computation is of the order of M^2N. If M^2N is large, we will need some parallel computation to keep up with a real time requirement.

The Matrix Triangularization Problem

A common approach is to premultiply the N by M matrix X by each of a sequence of Householder matrices, one after the other. Most of the operations required are adds and multiplies, and it is straightforward to perform many adds and multiplies in parallel, but the algorithm also requires a few divisions and square roots. These interfere with the efficiency of the use of a parallel array of multipliers and adders.

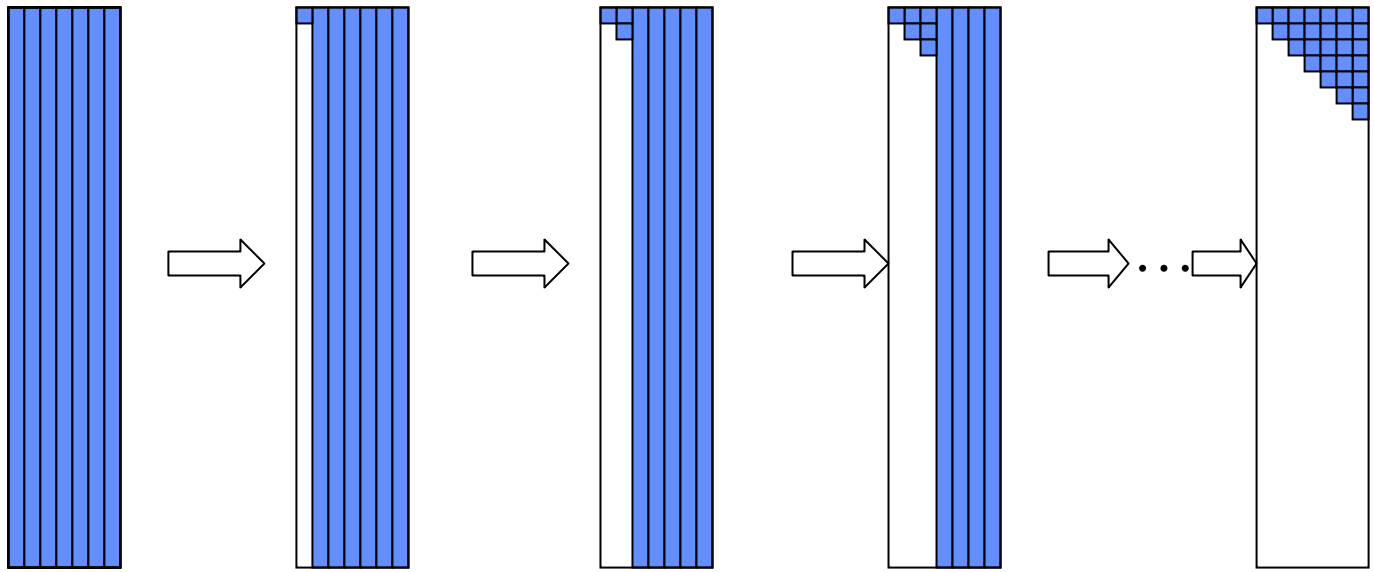
The Matrix Triangularization Problem

This talk is about an architecture which might be suitable to realize using FPGAs. We have in mind problems with M $\approx$ 20 and N $\approx$ 100.

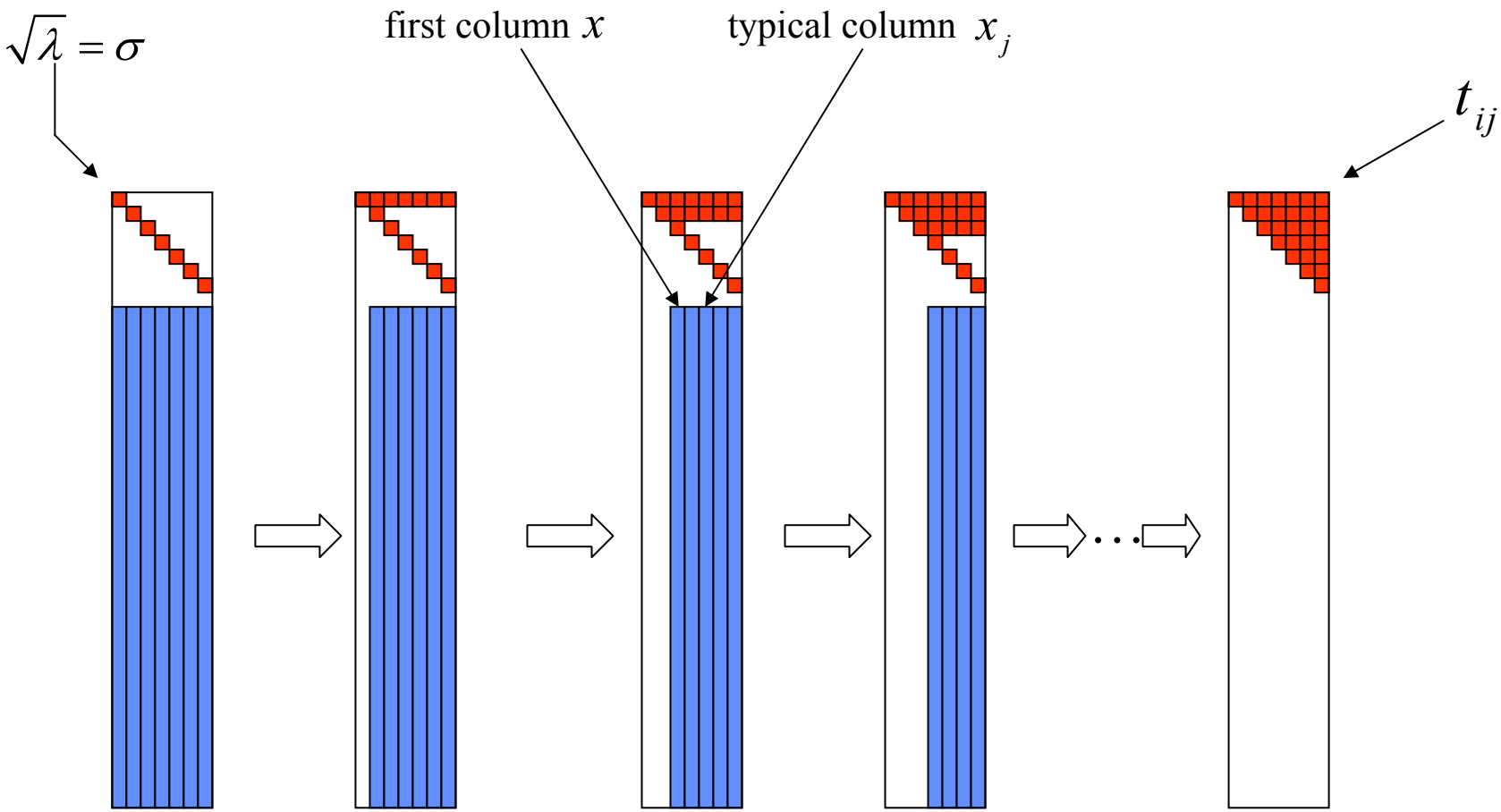
FPGAs are now available with approximately 100 built-in multipliers and with the capability to create a similar number of adders. Hence about ten FPGAs should be able to perform about 1000 multiply-adds in parallel.

Our architecture should use these 100 multipliers and adders with near 100% efficiency and we desire that all the FPGAs be identical (and, indeed, might later be replaced by custom ASICs.

Steps to triangularization using unitary matrix premultiplications



Steps to triangularization with diagonal loading



The Math of Zeroing a Column

$$\phi = x^h x, \phi_j = x^h x_j$$ N operations per column

$$t_{ii} = \xi = \sqrt{\phi + \lambda}, \mu = 1 / \xi$$

$$\theta = \frac{1}{\xi(\xi - \sigma)}$$

$$\beta_j = \theta \phi_j$$ 1 operation per column

$$x_j^{'} = x_j - \beta_j x$$ N operations per column

$$t_{ij} = \mu \phi_j$$ 1 operation per column

$$\phi = x^h x, \phi_j = \boxed{x^h x_j}$$

$$t_{ii} = \xi = \sqrt{\phi + \lambda}$$

$$\theta = \frac{1}{\xi(\xi - \sigma)}$$

N multiplications at once
$\text{conj}(x_i) \bullet x_{ij}$; i=1,…,N
(times the number of columns)

$$\beta_j = \theta\phi_j$$

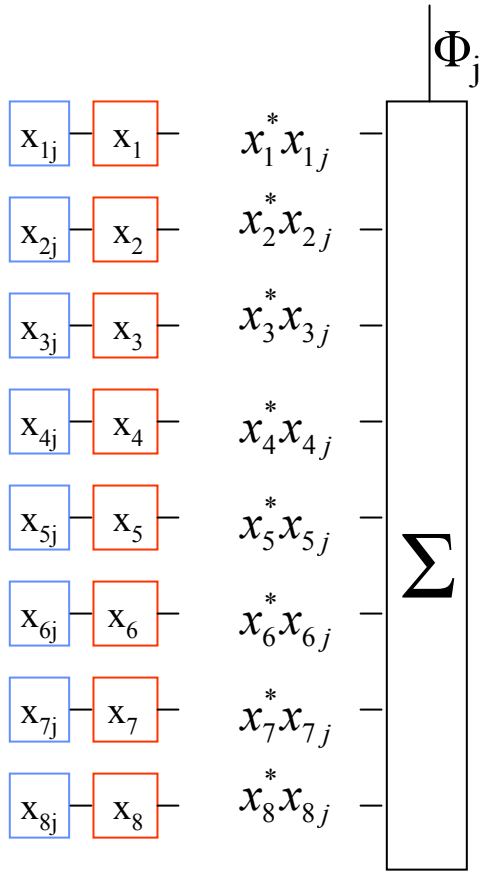
$$x'_j = x_j - \boxed{\beta_j x}$$

$$t_{ij} = \mu\phi_j$$

N multiplications at once
$\beta_j \bullet x_i$; i=1,…,N
(times the number of columns minus 1)

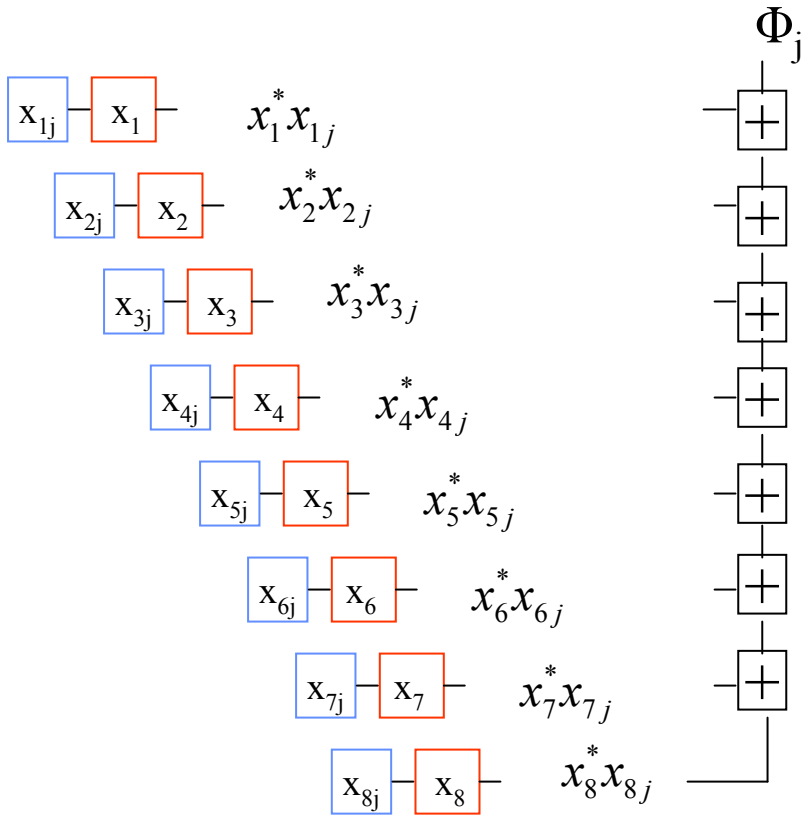
Front End Processing

Φ-processor computes Φ_j

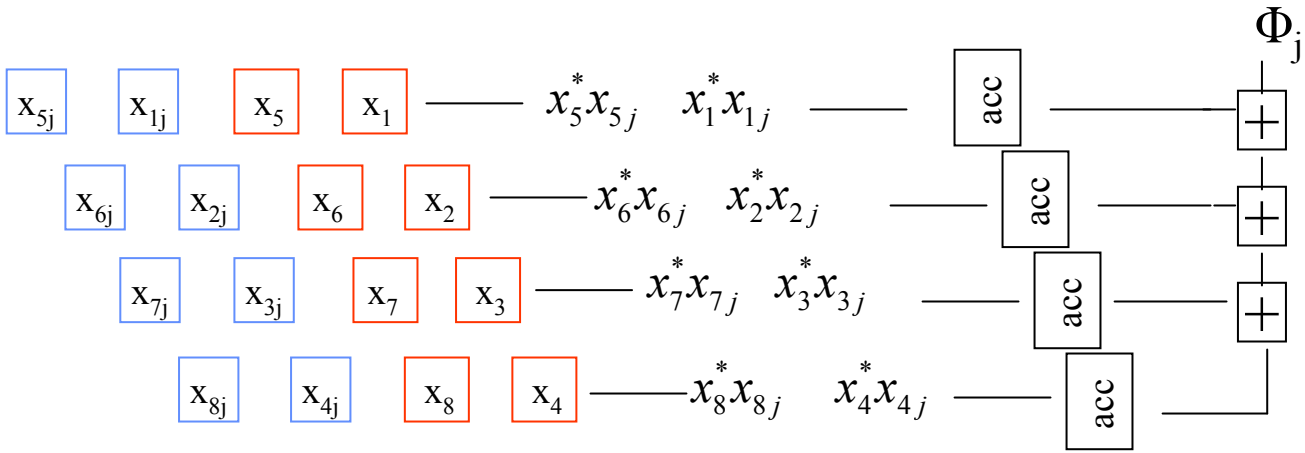


Skewed Front End Processing

Computes Φ_j



Series-parallel Front End Processing



Computing ξ and θ

$$t_{ii} = \xi = \sqrt{\phi + \lambda}$$

Saved as part of the answer

$$\theta = \frac{1}{\xi(\xi - \sigma)}$$

Used in output processor

- **These are needed before output processing can begin.**
- **They are relatively complicated computations and will be computed slowly.**
- **Our aim is to organize the algorithm so that the slow computation of ξ and θ can be buried.**

Output processor

For each column

$$\beta_j = \theta\phi_j$$

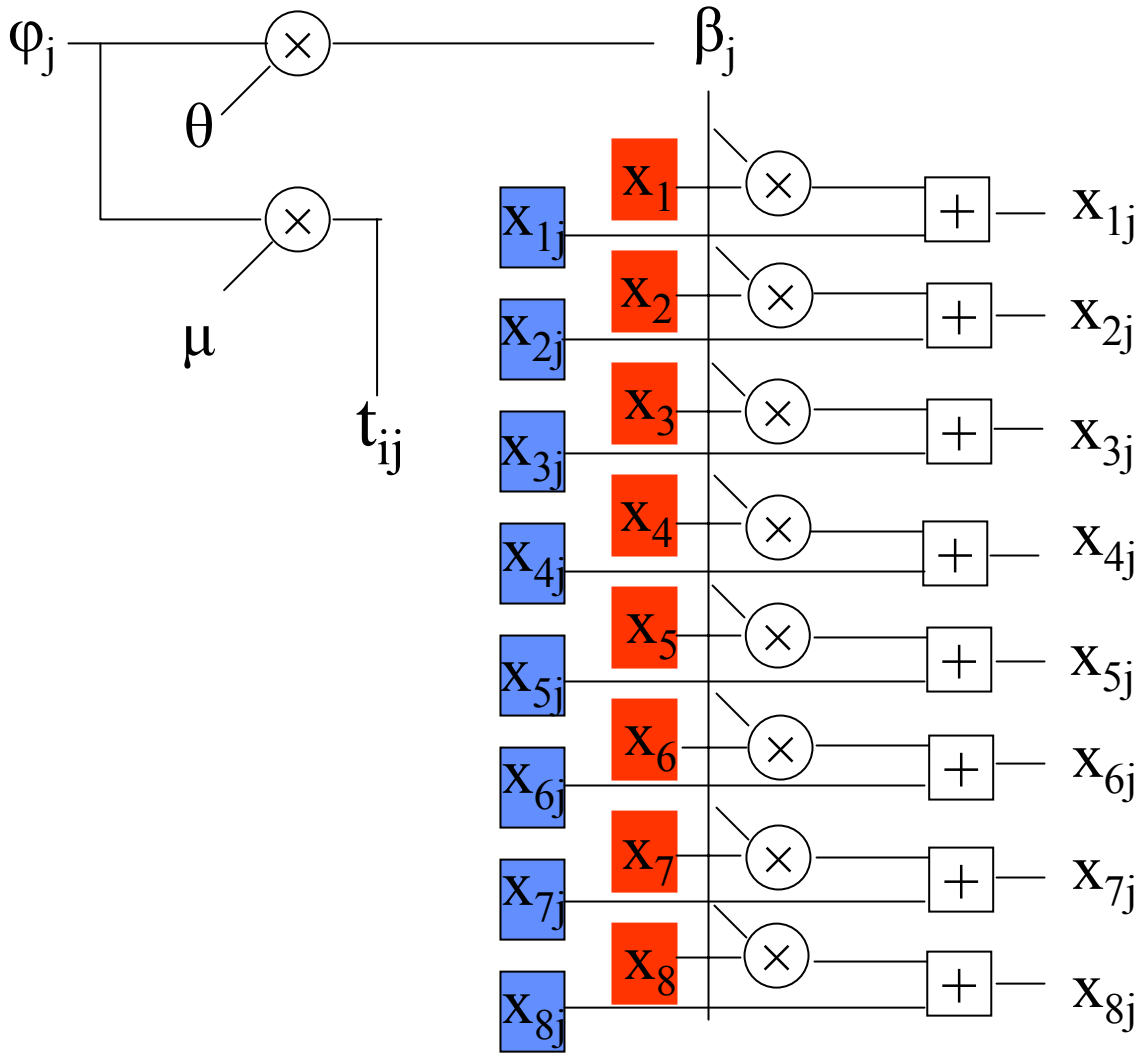
Compute and broadcast to all multipliers

$$x'_j = x_j - \beta_j x$$

One complex multiply per element. The first column was saved and the current column streams by.

$$t_{ij} = \mu\phi_j$$

Output Processor



Overview of the column elimination

- There are M columns. The process that eliminates column i accepts M-i+1columns in their normal order and spits out M-i columns.
- Elements move only horizontally and are involved in arithmetic only with other elements on the same horizontal level.
- Sums propagate upward -- ξ , θ, and β_j are computed at the upper edge of the processor.
- β_j must travel upward from the bottom edge to where it is needed by a multiplier.

Many Processors

- Let us define a *virtual super-processor.* Its job is to zero out one column.
- Let τ be the time separation between successive columns presented to the processor input.
- τ is also the time required to do the multiplies needed for Φ_j and is the time multiply x by β_j.
- Then the time that column j spends inside the virtual super-processor is K τ and most of this is waiting for the computation of ξ, θ, and β_j. (We'll determine K later.)
- We desire that the *virtual super-processor* whose job is to zero out column i+1 be ready for column j as soon as it is computed by the previous virtual super-processor.

When do columns get where?

virtual superprocessor	first column enters	column j enters	last column enters
1	0	(j-1) τ	(M-1) τ
2	(K+1) τ	(K+j-1) τ	(K+M-1) τ
3	(2K+2) τ	(2K+j-1) τ	(2K+M-1) τ
i	((i-1)K+(i-1)) τ	((i-1)K+j-1) τ	((i-1)K+M-1) τ

Super-processor sharing

virtual superprocessor	first column enters	column j enters	last column enters
i	((i-1)K+(i-1)) τ	((i-1)K+j-1) τ	((i-1)K+M-1) τ
M+1-i	((M–i)K+(M-i)) τ	((M-i)K+j-1) τ	((M-i)K+M-1) τ

These two virtual super-processors together process M+1 columns, independent of i, so it is tempting to combine them into one actual super-processor. (M/2) actual super-processors are needed for the whole triangularization.

Make Super-processors Identical

virtual superprocessor	first column enters	column j enters	last column enters
i	((i-1)K+(i-1)) τ	((i-1)K+j-1) τ	((i-1)K+M-1) τ
M+1-i	((M–i)K+(M-i)) τ	((M-i)K+j-1) τ	((M-i)K+M-1) τ

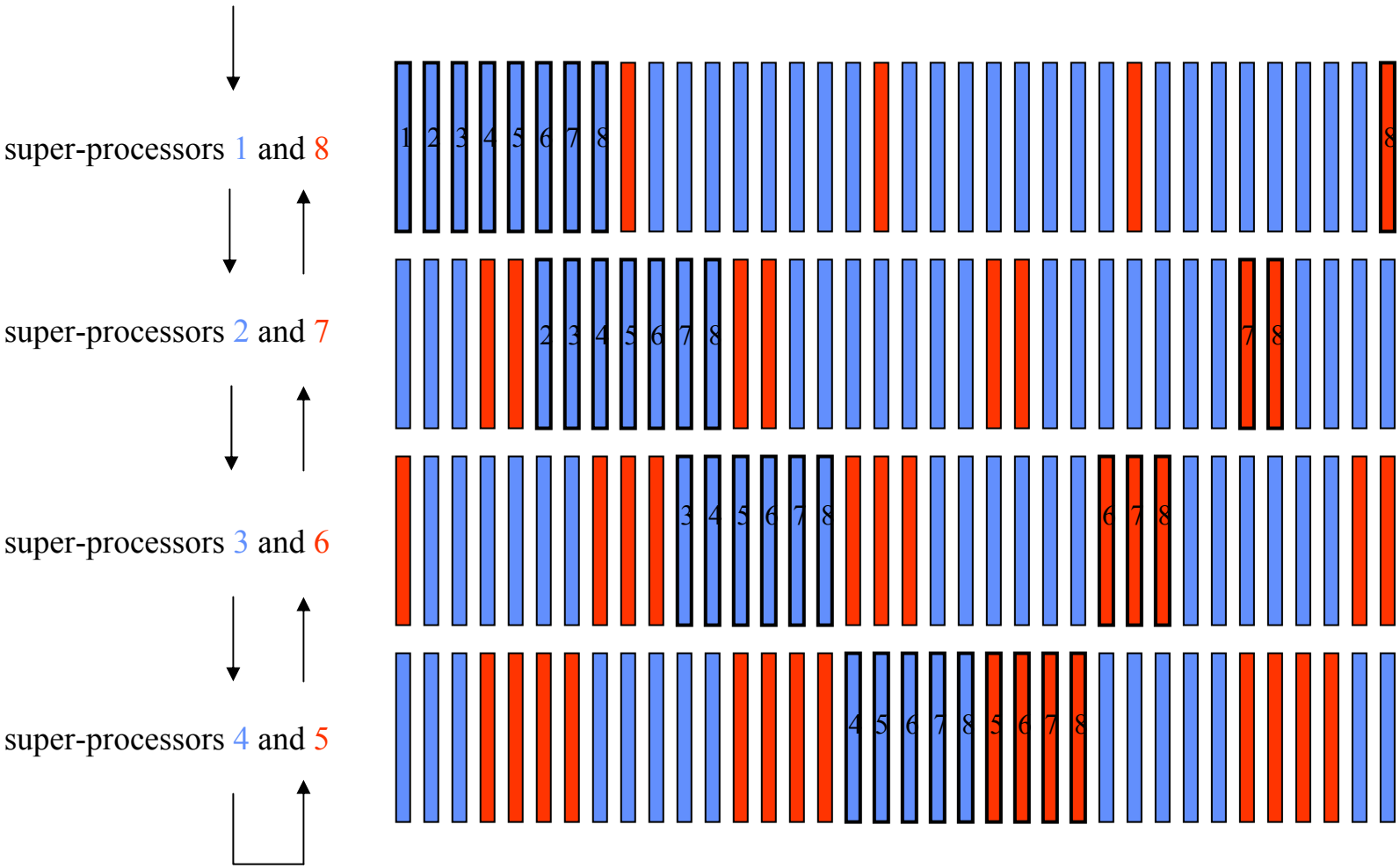
When actual super-processor i accepts its last column from actual super-processor i-1, in the next interval it is ready to accept the first column from actual super-processor i+1, but that must be from an earlier triangularization problem. The M/2 super-processors begin a new triangularization problem every (M+1) τ.

If that column is to be ready, we require $(i-1)K + M \equiv (M-i)(K+1)$ modulo M+1
or

$$(2K+1)i \equiv (M+1)K \equiv 0 \quad \text{modulo M+1}$$

So we choose K to make 2K+1 = M+1, K=M/2

Column Timing



A Dose of Reality

- Problems with word length and scaling
- Problems with input and output

Problems with word length and scaling

The N elements of input column i have the same total energy as the i elements of the final output for that column, so some element might have dynamic range expansion of up to $\sqrt{N}$.

So we might need floating point. (This is not a result of the architecture – it is intrinsic to the problem.) FPGAs come with efficient built-in multipliers, but not built-in floating point. We don't know how many floating point multipliers and adders we can get in a single FPGA.

Problems with input and output

Our architecture has negligible internal control, but requires that data arrive from multiple problems at just the right time, including skewing.

Several problems are active at once and late t-elements from one problem get delivered to the customer after the early t-elements from later problems.

So we will need an interface that transfers data for several “customers” to and from the processing array.

Summary

We've presented principles for an architecture suitable for realizing matrix triangularization with highly parallel use of multipliers and adders. Identical parts are used and internal control is negligible.

Parallelism comes from working on many independent problems at once. The waiting time for square roots and divisions is buried and does not reduce the efficiency of the use of multipliers.

The architecture will only become practical when FPGAs can realize large numbers of floating point adders and multipliers.