Parallel Numerical Algorithms

Chapter 8 – Triangular Systems

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

1. Triangular Systems
2. Parallel Algorithms
3. Wavefront Algorithms
4. Cyclic Algorithms

Triangular Matrices

- Matrix $\boldsymbol{L}$ is *lower triangular* if all entries above its main diagonal are zero, $\ell_{ij} = 0$ for $i < j$
- Matrix $\boldsymbol{U}$ is *upper triangular* if all entries below its main diagonal are zero, $u_{ij} = 0$ for $i > j$
- Triangular matrices are important because triangular linear systems are easily solved by successive substitution
- Most direct methods for solving general linear systems first reduce matrix to triangular form and then solve resulting equivalent triangular system(s)
- Triangular systems are also frequently used as preconditioners in iterative methods for solving linear systems

Forward Substitution

For lower triangular system $\boldsymbol{Lx} = \boldsymbol{b}$, solution can be obtained by *forward substitution*

$$x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij}\, x_j\right) / \ell_{ii}, \quad i = 1, \ldots, n$$

for $j = 1$ **to** n
  $x_j = b_j / \ell_{jj}$ { compute soln component }
  for $i = j + 1$ **to** n
    $b_i = b_i - \ell_{ij} x_j$ { update right-hand side }
  end
end

Back Substitution

For upper triangular system $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b}$, solution can be obtained by *back substitution*

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij}\, x_j\right) / u_{ii}, \quad i = n, \ldots, 1$$

for $j = n$ **to** 1
  $x_j = b_j / u_{jj}$ { compute soln component }
  for $i = 1$ **to** $j - 1$
    $b_i = b_i - u_{ij} x_j$ { update right-hand side }
  end
end

Solving Triangular Systems

- Forward or back substitution requires about $n^2/2$ multiplications and similar number of additions, so model serial time as

$$T_1 = t_c\, n^2/2$$

  where t_c is cost of paired multiplication and addition (we ignore cost of n divisions)

- We will consider only lower triangular systems, as analogous algorithms for upper triangular systems are similar

Loop Orderings for Forward Substitution

for $j = 1$ **to** n
    $x_j = b_j / \ell_{jj}$
    for $i = j + 1$ **to** n
        $b_i = b_i - \ell_{ij}\, x_j$
    end
end

- right-looking
- immediate-update
- data-driven
- fan-out

for $i = 1$ **to** n
    for $j = 1$ **to** $i - 1$
        $b_i = b_i - \ell_{ij}\, x_j$
    end
    $x_i = b_i / \ell_{ii}$
end

- left-looking
- delayed-update
- demand-driven
- fan-in

Parallel Algorithm

Partition

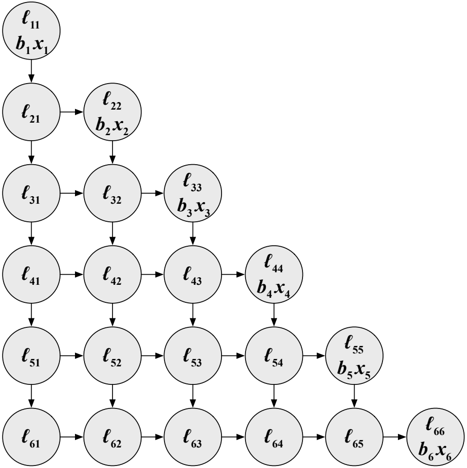
- For $i = 2, \ldots, n$, $j = 1, \ldots, i-1$, fine-grain task (i, j) stores ℓ_{ij} and computes product $\ell_{ij}\, x_j$
- For $i = 1, \ldots, n$, fine-grain task (i, i) stores ℓ_{ii} and b_i, collects sum $t_i = \sum_{j=1}^{i-1} \ell_{ij}\, x_j$, and computes and stores $x_i = (b_i - t_i)/\ell_{ii}$

yielding 2-D triangular array of $n\,(n+1)/2$ fine-grain tasks

Communicate

- For $j = 1, \ldots, n-1$, task (j, j) broadcasts x_j to tasks (i, j), $i = j+1, \ldots, n$
- For $i = 2, \ldots, n$, sum reduction of products $\ell_{ij}\, x_j$ across tasks (i, j), $j = 1, \ldots, i$, with task (i, i) as root

Fine-Grain Tasks and Communication



Fine-Grain Parallel Algorithm

if $i = j$ **then**
  $t = 0$
  if $i > 1$ **then**
    recv sum reduction of t across tasks (i, k), $k = 1, \ldots, i$
  end
  $x_i = (b_i - t)/\ell_{ii}$
  broadcast x_i to tasks (k, i), $k = i + 1, \ldots, n$
else
  recv broadcast of x_j from task (j, j)
  $t = \ell_{ij}\, x_j$
  reduce t across tasks (i, k), $k = 1, \ldots, i$
end

Fine-Grain Algorithm

- If communication is suitably pipelined, then fine-grain algorithm can achieve $\Theta(n)$ execution time, but uses $\Theta(n^2)$ tasks, so it is inefficient
- If there are multiple right-hand-side vectors $\boldsymbol{b}$, then successive solutions can be pipelined to increase overall efficiency
- Agglomerating fine-grain tasks yields more reasonable number of tasks and improves ratio of computation to communication

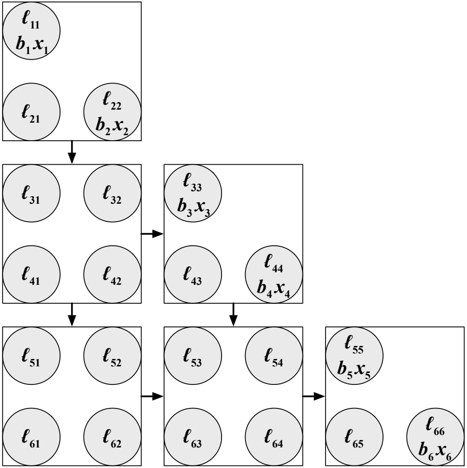
Agglomeration

Agglomerate

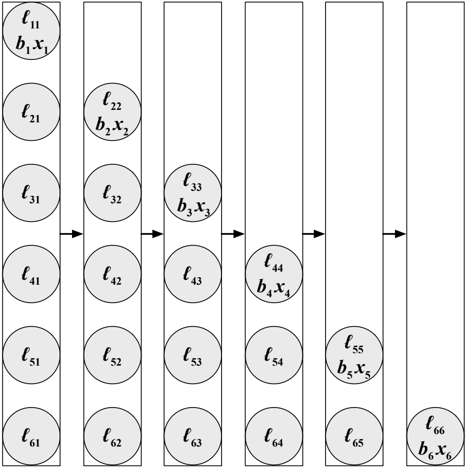
With $n \times n$ array of fine-grain tasks, natural strategies are

- 2-D: combine $k \times k$ subarray of fine-grain tasks to form each coarse-grain task, yielding $(n/k)^2$ coarse-grain tasks
- 1-D column: combine n fine-grain tasks in each column into coarse-grain task, yielding n coarse-grain tasks
- 1-D row: combine n fine-grain tasks in each row into coarse-grain task, yielding n coarse-grain tasks

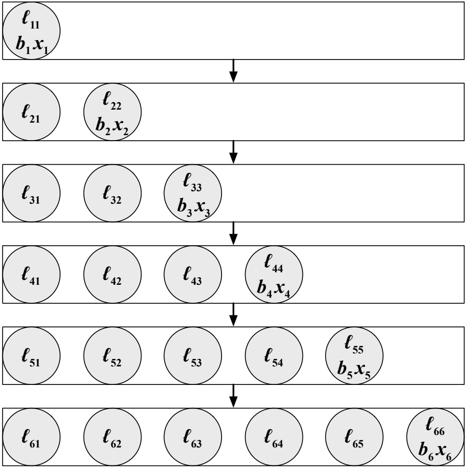
2-D Agglomeration



1-D Column Agglomeration



1-D Row Agglomeration

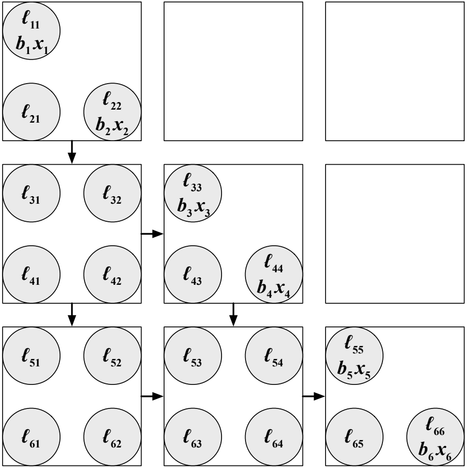


Mapping

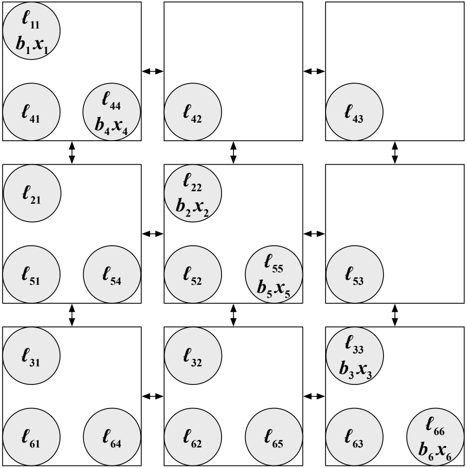
Map

- 2-D: assign $(n/k)^2/p$ coarse-grain tasks to each of p processes using any desired mapping in each dimension, treating target network as 2-D mesh
- 1-D: assign n/p coarse-grain tasks to each of p processes using any desired mapping, treating target network as 1-D mesh

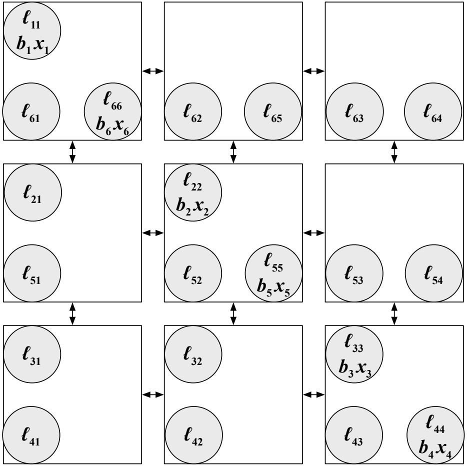
2-D Agglomeration, Block Mapping



2-D Agglomeration, Cyclic Mapping



2-D Agglomeration, Reflection Mapping



2-D Algorithm

- For 2-D agglomeration with $(n/\sqrt{p}\,) \times (n/\sqrt{p}\,)$ subarray of fine-grain tasks per process, both vertical broadcasts and horizontal sum reductions are required to communicate solution components and accumulate inner products, respectively
- If each process holds contiguous block of rows and columns, we obtain block version of original fine-grain algorithm, with poor concurrency and efficiency
- Moreover, this approach yields only $(p + \sqrt{p}\,)/2$ non-null processes, wasting almost half of 2-D mesh of processors

2-D Algorithm

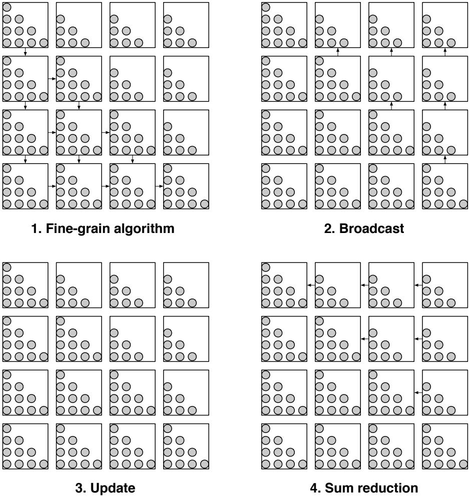
- Cyclic assignment of rows and columns to processes yields p non-null processes, so full 2-D mesh can be utilized
- But obvious implementation, looping over successive solution components and performing corresponding horizontal sum reductions and vertical broadcasts, still has limited concurrency because computation for each component involves only one process row and one process column
- Better algorithm can be obtained by computing solution components in groups of $\sqrt{p}$, which permits all processes to perform resulting updating concurrently

2-D Algorithm

Each step of resulting algorithm has four phases

1. Computation of next $\sqrt{p}$ solution components by processes in lower triangle using 2-D fine-grain algorithm
2. Broadcast of resulting solution components vertically from processes on diagonal to processes in upper triangle
3. Computation of resulting updates (partial sums in inner products) by all processes
4. Horizontal sum reduction from processes in upper triangle to processes on diagonal

2-D Algorithm



2-D Algorithm

- Total time required is approximately

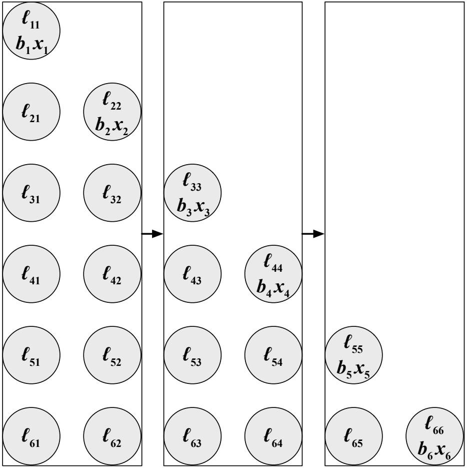
$$T_p = t_c\, n^2/(2p) + (4(t_s + t_w) + 5\, t_c)\, n$$

- To determine isoefficiency function, set

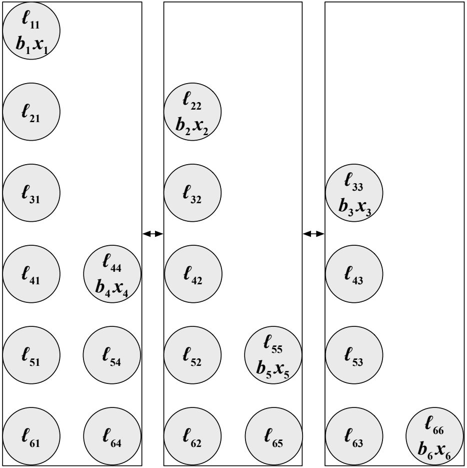
$$t_c\, n^2/2 \approx E\,(t_c\, n^2/2 + (4(t_s + t_w) + 5\, t_c)\, p\, n)$$

  which holds for large p if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

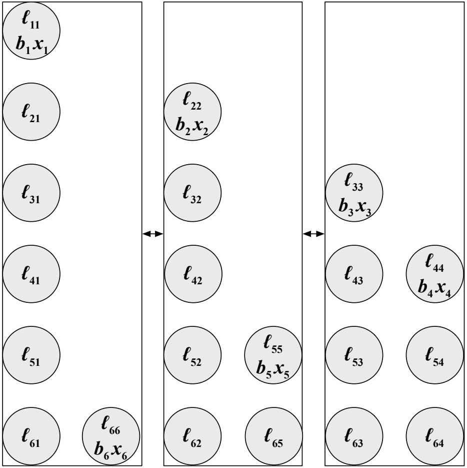
1-D Column Agglomeration, Block Mapping



1-D Column Agglomeration, Cyclic Mapping



1-D Column Agglomeration, Reflection Mapping



1-D Column Algorithm

- For 1-D agglomeration with n/p columns of fine-grain tasks per process, vertical broadcasts of components of x are unnecessary because any given matrix column is entirely contained in only one process
- But there is also no parallelism in computing products resulting from given component of x
- Horizontal communication is required for sum reductions to accumulate inner products

1-D Column Fan-in Algorithm

for $i = 1$ **to** n
    $t = 0$
    for $j \in$ *mycols*, $j < i$,
        $t = t + \ell_{ij}\, x_j$
    end
    if $i \in$ *mycols* **then**
        recv sum reduction of t
        $x_i = (b_i - t)/\ell_{ii}$
    else
        reduce t across processes
    end
end

1-D Column Algorithm

- Each process remains idle until solution component corresponding to its first column is computed
- If each process holds contiguous block of columns, it may remain idle through most of computation
- Moreover, number of products computed involving each component of x declines with increasing column number
- Concurrency and load balance can be improved by assigning columns to processes in cyclic manner
- Other mappings may also be useful, such as block-cyclic or reflection

1-D Column Algorithm

- If successive steps (outer loop) are overlapped, then approximate execution time is

$$T_p = t_c\,(n^2 + 2n(p-1))/(2p) + (t_s + t_w)\,(n-1)$$

ignoring cost of additions in sum reductions

- Without such overlapping, term representing communication cost is multiplied by factor of
  - $p-1$ for 1-D mesh
  - $2(\sqrt{p}-1)$ for 2-D mesh
  - $\log p$ for hypercube

representing path length for sum reduction

1-D Column Algorithm

- To determine isoefficiency function, set

$$t_c\, n^2/2 \approx E\,(t_c\,(n^2 + 2n(p-1))/2 + (t_s + t_w)\,p\,(n-1))$$

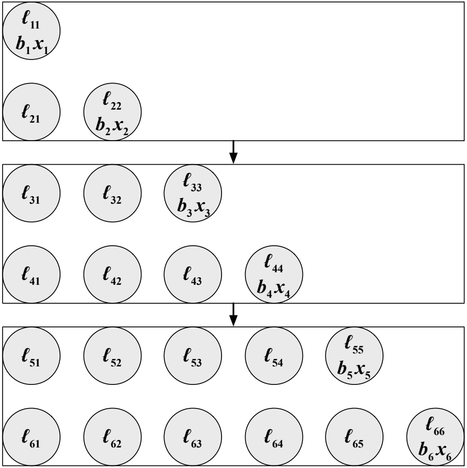
  which holds for large p if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

- Without overlapping of successive steps, isoefficiency function becomes
  - p^4 for 1-D mesh
  - p^3 for 2-D mesh
  - $p^2(\log p)^2$ for hypercube

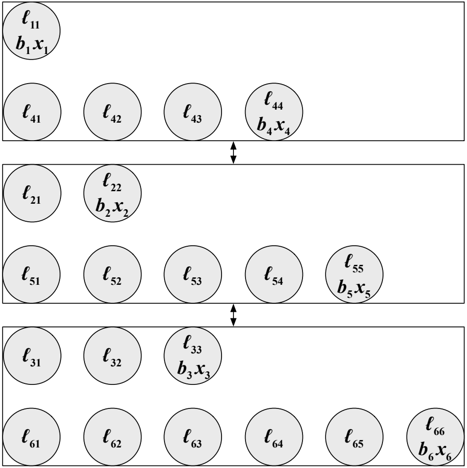
1-D Column Algorithm

- Overlap achievable is strongly affected by network topology and mapping of rows to processes
- For example, cyclic mapping on ring network permits almost complete overlap, whereas hypercube permits much less overlap
- Overlap of successive steps can potentially be enhanced by “compute ahead” strategy
- Process owning column i could compute most of its contribution to inner product for step $i + 1$ while waiting for contributions from other processes in step i, thereby avoiding being bottleneck for next step (because it will be last to complete step i)

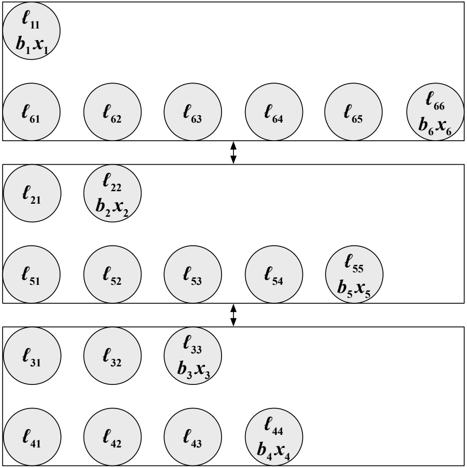
1-D Row Agglomeration, Block Mapping



1-D Row Agglomeration, Cyclic Mapping



1-D Row Agglomeration, Reflection Mapping



1-D Row Algorithm

- For 1-D agglomeration with n/p rows of fine-grain tasks per process, communication for horizontal sum reductions across process rows is unnecessary because any given matrix row is entirely contained in only one process
- But there is also no parallelism in computing these sums
- Vertical broadcasts are required to communicate components of x

1-D Row Fan-out Algorithm

for $j = 1$ **to** n
    if $j \in$ *myrows* **then**
        $x_j = b_j / \ell_{jj}$
    end
    broadcast x_j
    for $i \in$ *myrows*, $i > j$,
        $b_i = b_i - \ell_{ij}\, x_j$
    end
end

1-D Row Algorithm

- Each process falls idle as soon as solution component corresponding to its last row has been computed
- If each process holds contiguous block of rows, it may become idle long before overall computation is complete
- Moreover, computation of inner products across rows requires successively more work with increasing row number
- Concurrency and load balance can be improved by assigning rows to processes in cyclic manner
- Other mappings may also be useful, such as block-cyclic or reflection

1-D Row Algorithm

- If successive steps (outer loop) are overlapped, then approximate execution time is

$$T_p = t_c\,(n^2 + 2n(p-1))/(2p) + (t_s + t_w)\,(n-1)$$

- Without such overlapping, term representing communication cost is multiplied by factor of
  - $p-1$ for 1-D mesh
  - $2(\sqrt{p}-1)$ for 2-D mesh
  - $\log p$ for hypercube

  representing path length for broadcast

1-D Row Algorithm

- To determine isoefficiency function, set

$$t_c\, n^2/2 \approx E\,(t_c\,(n^2 + 2n(p-1))/2 + (t_s + t_w)\,p\,(n-1))$$

  which holds for large p if $n = \Theta(p)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

- Without overlapping of successive steps, isoefficiency function becomes
  - p^4 for 1-D mesh
  - p^3 for 2-D mesh
  - $p^2(\log p)^2$ for hypercube

1-D Row Algorithm

- Overlap achievable is also strongly affected by network topology and mapping of rows to processes
- For example, cyclic mapping on ring network permits almost complete overlap, whereas hypercube permits much less overlap
- Overlap of successive steps can potentially be enhanced by “send ahead” strategy
- At step j, process owning row $j+1$ could compute x_{j+1} and broadcast it *before* completing remaining updating due to x_j

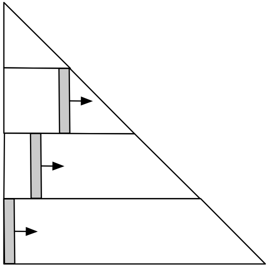
Wavefront Algorithms

- Fan-out and fan-in algorithms derive their parallelism from inner loop, whose work is partitioned and distributed across processes, while outer loop is serial
- Conceptually, fan-out and fan-in algorithms work on only one component of solution at a time, though successive steps may be pipelined to some degree
- Wavefront algorithms exploit parallelism in outer loop explicitly by working on multiple components of solution simultaneously

1-D Column Wavefront Algorithm

- 1-D column fan-out algorithm seems to admit no parallelism: after process owning column j computes x_j, resulting updating of right-hand side cannot be shared with other processes because they have no access to column j
- Instead of performing all such updates immediately, however, process owning column j could complete only first s components of update vector and forward them to process owning column $j+1$ *before* continuing with next s components of update vector, etc.
- Upon receiving first s components of update vector, process owning column $j+1$ can compute x_{j+1}, begin further updates, forward its own contributions to next process, etc.

1-D Column Wavefront Algorithm



To formalize wavefront column algorithm we introduce

- z : vector in which to accumulate updates to right-hand-side
- *segment* : set containing at most s consecutive components of z

1-D Column Wavefront Algorithm

for $j \in$ *mycols*
    for $k = 1$ **to** # segments
        recv *segment*
        if $k = 1$ **then**
            $x_j = (b_j - z_j)/\ell_{jj}$
            segment $=$ *segment* $- \{z_j\}$
        end
        for $z_i \in$ *segment*
            $z_i = z_i + \ell_{ij}\, x_j$
        end
        if $|$*segment*$| > 0$ **then**
            send *segment* to process owning column $j + 1$
        end
    end
end

1-D Column Wavefront Algorithm

- Depending on segment size, column mapping, communication-to-computation speed ratio, etc., it may be possible for all processes to become busy simultaneously, each working on different component of solution
- Segment size is adjustable parameter that controls tradeoff between communication and concurrency
- "First" segment for given column shrinks by one element after each component of solution is computed, disappearing after s steps, when next segment becomes "first" segment, etc.

1-D Column Wavefront Algorithm

- At end of computation only one segment remains and it contains only one element
- Communication volume declines throughout algorithm
- As segment length s increases, communication start-up cost decreases but computation cost increases, and vice versa as segment length decreases
- Optimal choice of segment length s can be predicted from performance model

1-D Column Wavefront Algorithm

- Approximate execution time is

$$T_p = ((t_s/s) + t_w + t_c)\,(n^2 + np + s(s-1)p^2)/(2p)$$

  where s is segment length

- To determine isoefficiency function, set

$$t_c\, n^2/2 \approx E\,(((t_s/s) + t_w + t_c)\,(n^2 + np + s(s-1)p^2)/2)$$

  which holds for large p if $n = \Theta(p)$, assuming s is constant, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n^2)$

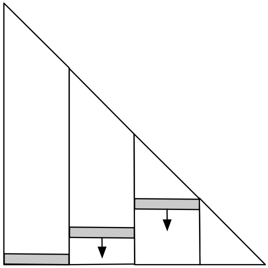
1-D Row Wavefront Algorithm

- Wavefront approach can also be applied to 1-D row fan-in algorithm
- Computation of ith inner product cannot be shared because only one process has access to row i of matrix
- Thus, work on multiple components must be overlapped to attain any concurrency
- Analogous approach is to break solution vector x into segments that are pipelined through processes

1-D Row Wavefront Algorithm

- Initially, process owning row 1 computes x_1 and sends it to process owning row 2, which computes resulting update and then x_2
- This process continues (serially at this early stage) until s components of solution have been computed
- Henceforth, receiving processes forward any full-size segments *before* they are used in updating
- Forwarding of currently incomplete segment is delayed until next component of solution is computed and appended to it

1-D Row Wavefront Algorithm



1-D Row Wavefront Algorithm

for $i \in$ *myrows*
    for $k = 1$ **to** # $segments - 1$
        recv *segment*
        send *segment* to process owning row $i + 1$
        for $x_j \in$ *segment*
            $b_i = b_i - \ell_{ij}\, x_j$
        end
    end
    recv *segment*   /* last may be empty */
    for $x_j \in$ *segment*
        $b_i = b_i - \ell_{ij}\, x_j$
    end
    $x_i = b_i / \ell_{ii}$
    segment $=$ *segment* $\cup \{x_i\}$
    send *segment* to process owning row $i + 1$
end

1-D Row Wavefront Algorithm

- Instead of starting with full set of segments that shrink and eventually disappear, segments appear and grow until there is a full set of them
- It may be possible for all processes to be busy simultaneously, each working on different segment
- Segment size is adjustable parameter that controls tradeoff between communication and concurrency, and optimal value of segment length s can be predicted from performance model
- Performance analysis and resulting performance model are more complicated than for 1-D column wavefront algorithm, but performance and scalability for 1-D row wavefront algorithm are nevertheless similar

Cyclic Algorithms

- In wavefront algorithms, each segment is sent up to s times and may pass through same process repeatedly, depending on mapping of rows or columns
- Cyclic algorithms are somewhat similar to wavefront algorithms, but they minimize communication by exploiting cyclic mapping of rows or columns
- Instead of having variable number of segments of adjustable length, cyclic algorithms circulate single segment of length $p-1$

1-D Column Cyclic Algorithm

- In cyclic 1-D column algorithm, segment of size $p-1$, containing partially accumulated components of update vector z, passes from process to process, one step for each column of matrix, cycling through all $p-1$ other processes before returning to any given process

- At step j, process owning column j receives segment from process owning column $j-1$ and uses its first element (which has accumulated all necessary prior updates) to compute x_j

- Task owning column j then modifies segment by deleting first element, updating remaining elements, and appending new element to begin accumulation of z_{j+p-1}

1-D Column Cyclic Algorithm

- Segment is then sent to process owning column $j+1$, where similar procedure is repeated
- After forwarding modified segment, process owning column j then computes remaining updates resulting from x_j, which will be needed when segment returns to this process again
- Updating while segment is elsewhere provides all concurrency, since computations on segment are serial

1-D Column Cyclic Algorithm

for $j \in$ *mycols*
    recv *segment*
    $x_j = (b_j - z_j - t_j)/\ell_{jj}$
    segment $=$ *segment* $- \{z_j\}$
    for $z_i \in$ *segment*
        $z_i = z_i + t_i + \ell_{ij}\, x_j$
    end
    $z_{j+p-1} = t_{j+p-1} + \ell_{j+p-1,j}\, x_j$
    segment $=$ *segment* $\cup \{z_{j+p-1}\}$
    send *segment* to process owning column $j+1$
    for $i = j+p$ **to** n
        $t_i = t_i + \ell_{ij}\, x_j$
    end
end

1-D Column Cyclic Algorithm

- Segment must pass through *all* other processes before returning to any given process, so correctness depends on use of cyclic mapping
- Maps naturally to 1-D torus (ring) network, but since only one pair of processes communicates at any given time, also works well with bus network
- Attains minimum possible volume of interprocessor communication to solve triangular system using column-oriented algorithm

1-D Column Cyclic Algorithm

- For $n \leq p\,(t_p + p)$, where

$$t_p = (t_s + t_w(p-1))/t_c$$

  is cost, measured in flops, of sending message of length $p-1$, execution time is determined by segment cycle time, so that

$$T_p = t_c(n\,(t_p + p) - p\,(p-1)/2 - t_p)$$

- For $n > p\,(t_p + p)$, execution time is dominated by cost of updating, so that

$$T_p = t_c((n^2 + n\,p)/(2p) + p\,((t_p + p)^2 - t_p - p + 1)/2 \; - \; t_p)$$

1-D Column Cyclic Algorithm

- Two-phase behavior complicates scalability analysis, but tradeoff point between phases for n as function of p grows like p^2, so isoefficiency function is at least $\Theta(p^4)$
- Performance of both phases can be improved
  - Segment cycle time can be reduced by breaking segment into smaller pieces and pipelining them through processes
  - Updating work can be reorganized, deferring excessive work until later cycles, to obtain more even distribution throughout computation

1-D Row Cyclic Algorithm

- 1-D row cyclic algorithm is similar, except processes are agglomerated by rows and segment contains $p-1$ components of solution $\boldsymbol{x}$
- At step i, process owning row i receives segment from process owning row $i-1$ and uses components of $\boldsymbol{x}$ it contains to complete ith inner product, so that x_i can then be computed
- Task then modifies segment by deleting first element and appending new element x_i just computed
- Segment is then sent to process owning row $i+1$, where similar procedure is repeated

1-D Row Cyclic Algorithm

- After forwarding modified segment, process then computes partial inner products that use components of segment, which will be further accumulated when segment returns to this process again
- Latter computations, which take place while segment passes through other processes, provide concurrency in algorithm, because computations on segment itself are serial
- Again, correctness of algorithm depends on use of cyclic mapping
- Performance and scalability are similar to those for 1-D column cyclic algorithm, although details differ

1-D Row Cyclic Algorithm

for $i \in$ *myrows*
    recv *segment*
    for $x_j \in$ *segment*
        $b_i = b_i - \ell_{ij}\, x_j$
    end
    $x_i = b_i / \ell_{ii}$
    segment $=$ *segment* $- \{x_{i-p}\} \cup \{x_i\}$
    send *segment* to process owning row $i + 1$
    for $m \in$ *myrows*, $m > i$,
        for $x_j \in$ *segment*
            $b_m = b_m - \ell_{mj}\, x_j$
        end
    end
end

References

- R. H. Bisseling and J. G. G. van de Vorst, Parallel triangular system solving on a mesh network of Transputers, *SIAM J. Sci. Stat. Comput.* 12:787-799, 1991

- S. C. Eisenstat, M. T. Heath, C. S. Henkel, and C. H. Romine, Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Stat. Comput.* 9:589-600, 1988

- M. T. Heath and C. H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Stat. Comput.* 9:558-588, 1988

- N. J. Higham, Stability of parallel triangular system solvers, *SIAM J. Sci. Comput.* 16:400-413, 1995

References

- G. Li and T. F. Coleman, A parallel triangular solver for a distributed-memory multiprocessor, *SIAM J. Sci. Stat. Comput.* 9:485-502, 1988
- G. Li and T. F. Coleman, A new method for solving triangular systems on distributed-memory message-passing multiprocessors, *SIAM J. Sci. Stat. Comput.* 10:382-396, 1989
- C. H. Romine and J. M. Ortega, Parallel solution of triangular systems of equations, *Parallel Computing* 6:109-114, 1988
- E. E. Santos, On designing optimal parallel triangular solvers, *Information and Computation* 161:172-210, 2000