ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Triangular systems

- a matrix L is lower triangular if all entries above the main diagonal are zero: $L_{ij} = 0$ for $i < j$.
- a matrix U is upper triangular if all entries below the main diagonal are zero: $U_{ij} = 0$ for $i > j$.
- triangular systems appear frequently
  - most direct methods of solving linear systems of equation start with a reduction of the matrix of coefficients into triangular form
  - they are also used as preconditioners in iterative methods

Forward substitution

- the lower triangular system $Lx = b$ can be solved by *forward substitution*

$$x_i = \left(b_i - \sum_{j=1}^{i-1} L_{ij} x_j \right) / L_{ii} \tag{1}$$

- that can be implemented as

Algorithm 1: FORWARDSUBSTITUTION(L, b)

```
1 for $j = 1$ to $n$ do
2     $x_j = b_j / L_{jj}$
3     for $i = j + 1$ to $n$ do
4         $b_i = b_i - L_{ij} x_j$
```

- with roughly n^2 multiply-adds

Backward substitution

- the upper triangular system $Ux = b$ can be solved by *backward substitution*

$$x_i = \left(b_i - \sum_{j=i+1}^{n} U_{ij} x_j \right) / U_{ii} \tag{2}$$

- that can be implemented as

Algorithm 2: BACKWARDSUBSTITUTION(L, b)

```
1  for $j = n$ to 1 do
2      $x_j = b_j / U_{jj}$
3      for $i = 1$ to $j - 1$ do
4          $b_i = b_i - U_{ij} x_j$
```

- with roughly n^2 multiply-adds
- the two algorithms are very similar, so focus on lower triangular systems

Sequential implementations

- there are two possible ways to arrange the forward substitution loops

```
1 for $j = 1$ to $n$ do
2     $x_j = b_j / L_{jj}$
3     for $i = j + 1$ to $n$ do
4         $b_i = b_i - L_{ij} x_j$
```

- immediate update
- data driven
- fan out

```
1 for $i = 1$ to $n$ do
2     for $j = 1$ to $i - 1$ do
3         $b_i = b_i - L_{ij} x_j$
4     $x_i = b_i / L_{ii}$
```

- delayed update
- demand driven
- fan in

Parallelization

- label the fine grain tasks as (i, j) with $i, j = 1, \ldots, n$
  - for $i = 2, \ldots, n$ and $j = 1, \ldots, i-1$, task (i, j)
    - stores L_{ij}
    - computes $L_{ij} x_j$
  - for $i = 1, \ldots, n$, task (i, i)
    - stores L_{ii} and b_i
    - collects the sum $t_i = \sum_{j=1}^{i-1} L_{ij} x_j$
    - computes and stores $x + i = (b_i - t_i)/L_{ii}$
- this arrangement yields a two dimensional triangular grid of $n(n+1)/2$ fine grain tasks
- with the following communication patterns
  - for $j = 1, \ldots, n-1$, task (j, j) broadcasts x_j to tasks (i, j), $i = j+1, \ldots, n$
  - for $i = 1, \ldots, n$, task (i, i) collects the sum reduction of $L_{ij} x_j$ from tasks $(i, j), j = 1, \ldots, i-1$

Parallel implementation

- here is the program for task (i,j) with $i \geq j$

```
if $i = j$ then
    if $i > 1$ then
        recv sum reduction $t$
    else
        $t = 0$
    broadcast $x_i$ to tasks $(k,i)$ and $(i,k)$, $k = i+1,\ldots,n$
else
    recv $x_j$
    $t = L_{ij} x_j$
    send $t$ for reduction across tasks $(i,k)$, $k = 1,\ldots,i-1$ to task $(i,i)$
```

- for properly pipelined communication, this algorithm can be implemented in $\mathcal{O}(n)$, but it requires $\mathcal{O}(n^2)$ tasks
- if there are many b to solve for, the tasks can be working on multiple systems at the same time
- coarsening strategies can manage the number of tasks and improve the balance between computation and communication

Coarsening by rows

- for one dimensional coarsening into n/p rows
    - there is no need to communicate to perform the reductions, but also no parallelism
    - vertical broadcasts are still needed to move the components of x

```
for $j = 1$ to $n$ do
    if $j \in myrows$ then
        $x_j = b_j / L_{jj}$
        broadcast $x_j$ to other tasks
    else
        recv $x_j$
    for $i \in myrows,\ i > j$ do
        $b_i = b_i - L_{ij} / x_j$
```

Observations on coarsening by rows

- load balance:
  - tasks become idle after the solution components corresponding to their last row are computed, and there is progressively more work as row number increases
  - if a task holds a contiguous block of rows, it may become idle before mush of the calculation is finished
- both concurrency and load balance may be improved by assigning rows to tasks in more creative ways
  - cyclically: assign row j to task $j \mod p$
  - block cyclically
  - reflectively
- overall execution speed depends on the ability to overlap communication with the computation of successive steps

Coarsening by columns

- for one dimensional coarsening into n/p rows
  - there is no need to broadcast the components of x vertically, but also no parallelism in computing the products
  - horizontal exchanges are still required for the sum reductions that accumulate the inner products

```
for $i = 1$ to $n$ do
    t = 0
    for $j \in mycolumns, j < i$ do
        $t = t + L_{ij} x_j$
    if $i \in mycolumns$ then
        recv reduction of $t$
        $x_i = (b_i - t)/L_{ii}$
    else
        send $t$ for reduction across all tasks
```

Observations on coarsening by columns

- load balance:
  - tasks are idle until solution component corresponding to their first column is computed, and there is progressively less work as column number increases
  - if a task holds a contiguous block of columns, it may remain idle for most of the calculation
- both concurrency and load balance may be improved by assigning columns to tasks in more creative ways
  - cyclically: assign column j to task $j \mod p$
  - block cyclically
  - reflectively
- overall execution speed depends on the ability to overlap communication with the computation of successive steps

Wavefront algorithms

- fan out and fan in algorithm share many characteristics
  - parallelism comes from partitioning and distributing the work of the inner loop
  - while the outer loop is serial
  - they work on one component of the solution at a time, although one can partially pipeline successive steps
- *wavefront* algorithms exploit parallelism in the outer loops by explicitly working on multiple components of the solution at the same time
- consider the one dimensional fan out algorithm
  - it appears there is no opportunity for parallelism: after the owner of column j computes x_j, the updated components of b cannot shared with other tasks because they have no access to column j
  - however, the task that owns column j could finish only a fraction of the work, say the first s components, and pass them on to the task that owns column $j + 1$, before continuing on to the next s components
  - the task that owns column $j + 1$ receives the first s components of b, it can compute x_{j+1}, begin a fraction of the remaining updates and forward its contributions to the next task

Wavefront implementation

- we need two new features
  - a vector z that accumulates the updates to b
  - the notion of a *segment* that contains no more than s consecutive components of z

```
for $j \in mycolumns$ do
    for $k = 1$ to $number\ of\ segments$ do
        recv $segment$
        if $k = 1$ then
            $x_j = (b_j - z_j)/L_{jj}$
            $segment = segment - \{z_j\}$
        for $z_i \in segment$ do
            $z_i = z_i + L_{ij}x_j$
        if $length(segment) > 0$ then
            send $segment$ to task owning column $j + 1$
```

Block coarsening

- coalesce $(n/\sqrt{p}) \times (n/\sqrt{p})$ fine grain tasks in to a coarse grain block
- resulting in an algorithm with features from both column and row coarsening
  - both vertical broadcasts and horizontal reductions are required to communicate solution components and accumulate inner products
- the naïve implementation assigns contiguous blocks of rows and columns to coarse tasks
  - poor concurrency and efficiency
  - almost half the tasks are idle
- cyclic assignment, with L_{ij} assigned to task $(i \mod \sqrt{p}, j \mod \sqrt{p})$, yields p non-null tasks so the full grid of tasks is active
  - again, the obvious implementation where we loop over successive solution components to perform horizontal reductions and vertical broadcasts has limited concurrency, since computation of each component involves only one task row and one task column
  - improve by computing solution components in groups of $\sqrt{p}$, which enables all tasks to perform the updates concurrently

Iterative methods

- iterative methods start with an initial guess for the vector x and improve until some desired accuracy is achieved
  - no upper bound on the number of iterations to the exact solution
  - in practice, establish an error measure such as $||b - Ax|| < \epsilon$
  - particularly good with sparse matrices because sparsity is preserved
- we will take a quick look at
  - Jacobi
  - Gauss-Seidel
  - successive over-relaxation (SOR)
  - conjugate gradient

The Jacobi method

- given an initial guess $x^{(0)}$, the Jacobi method iterates by

$$x_i^{(k+1)} = \left(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)} \right) / A_{ii} \tag{3}$$

  which can be expressed as

$$x^{(k+1)} = D^{-1} \left(b - (L + U) x^{(k)} \right) \tag{4}$$

  for D, L, and U respectively diagonal, upper and lower triangular matrices
- the method requires
  - non-zero diagonal entries, usually achievable by permuting rows
  - extra storage for the x iterates
- convergence is neither guaranteed nor fast, but the components of $x^{(k)}$ are decoupled form each other so they can be computed in parallel

The Gauss-Seidel method

- the convergence rate can be improved by using the components of $x^{(k+1)}$ as soon as they become available

$$x_i^{(k+1)} = \left(b_i - \sum_{j<i} A_{ij} x_j^{(k+1)} - \sum_{j>i} A_{ij} x_j^{(k)} \right) / A_{ii} \tag{5}$$

or

$$x^{(k+1)} = (D+L)^{-1} \left(b - U x^{(k)} \right) \tag{6}$$

- this method also requires non-zero diagonal entries, but no extra storage for *x* since the values can be written in place
  - but this coupling reduces the parallelism opportunities
- convergence is about twice as fast as Jacobi, and guaranteed to converge under weaker conditions
  - e.g. positive definite symmetric *A*
  - but may still be too slow for practical purposes

Successive over-relaxation

- SOR uses the weighted average of the current iterate and the next Gauss-Seidel iterate

$$x^{(k+1)} = (1-\omega)x^{(k)} + \omega x_{GS}^{(k+1)} \tag{7}$$

  where ω is relaxation parameter chosen to accelerate convergence
  - $\omega > 1$ gives over-relaxation, $\omega < 1$ gives under-relaxation
  - $\omega = 1$ is pure Gauss-Seidel
  - the method diverges unless $0 < \omega < 2$
- with optimal value of ω, the convergence rate is an order of magnitude faster than Gauss-Seidel
  - but the optimal ω is difficult to find in general
- this method suffers from reduced parallelism as well
  - but allowing each process to use its most current value, rather than waiting for the latest update, leads to *asynchronous* over-relaxation
  - but the stochastic nature complicates the convergence analysis

Conjugate gradients

- another approach is to observe that, if A is a positive definite matrix, the quadratic form
$$\phi(x) = \frac{1}{2} x^T A x - x^T b \tag{8}$$
is minimized by the solution to the linear system $Ax = b$
- this optimization problem is solved by iterates
$$x^{(k+1)} = x^{(k)} + \omega s^{(k)} \tag{9}$$
where ω is a search parameter chosen to minimize the *objective function* $\phi(x^{(k)} + \omega s^{(k)})$ along the direction $s^{(k)}$
- *steepest descent* if obtained when $s^{(k)} = -\nabla \phi(x)$

Conjugate gradients for linear systems

- for the special case of the quadratic problem in Eq. 8
  - the residual vector is the negative gradient

$$r = b - Ax = -\nabla\phi \tag{10}$$

  - the optimal line search parameter is given by

$$\omega = \frac{r^T s}{s^T A s} \tag{11}$$

  - successive search directions can be made orthogonal to A by a simple three-term recurrence relation
- leading to the following conjugate gradient algorithm for linear systems

Conjugate gradient method

Algorithm 3: CONJUGATEGRADIENT(A, b)

```
1 $x_0 = \text{initial guess}$
2 $r_0 = b - Ax_o$
3 $s_0 = r_0$
4 for $k \in \{0, 1, 2, \ldots\}$ do
5     $\omega_k = \frac{r_k^T r_k}{s_k^T A s_k}$
6     $x_{k+1} = x_k + \omega_k s_k$
7     $r_{k+1} = r_k = \omega_k A s_k$
8     $\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
9     $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
```

Observations on conjugate gradient

- the conjugate gradient method is widely used because
  - determination of the orthogonal search directions is accomplished using a simple three step recurrence relation
  - at any given iteration, the accumulated error is *minimal* over the space spanned by the search vectors
  - which implies that the method will produce the exact solution after a finite number of steps
  - in practice, roundoff error spoils orthogonality, so the method is used iteratively
- at each iteration the error is reduced on average by a factor of

$$(\sqrt{\kappa}-1)/(\sqrt{\kappa}+1) \tag{12}$$

where

$$\kappa = \mathrm{cond}(A) = ||A|| \cdot ||A^{-1}|| = \frac{\lambda_{\mathrm{max}}}{\lambda_{\mathrm{min}}} \tag{13}$$

- convergence is rapid if A is well-conditioned, but arbitrarily slow for ill-conditioned matrices
- also depends on the clustering of the eigenvalues of A

Parallelization of iterative methods

- all these methods are composed of basic operations
  - vector updates
  - inner products
  - matrix-vector multiplications
  - solutions to triangular systems
- in parallel, both data and operations are partitioned among multiple tasks
- additional communication is required to compute the convergence criterion
- these methods require additional storage for a variety of vectors
  - they are dense, even if A is sparse
  - they are typically partitioned uniformly among processes
  - so vector updates require no communication, but their inner products require reductions
- as we have seen, there are many strategies for partitioning the matrix A