Semi-automatic parallelization of iterative solvers

Oleg Batrashev
Distributed Systems Group
University of Tartu

February 5, 2010

Outline

- Domain introduction and motivation
  - scientific computing
  - iterative solvers
- Idea of semi-automatic parallelization
  - 1D Finite Difference with Jacobi solver
- Real examples that need parallelization
  - matrix-vector multiplication
  - preconditioners
- Static analysis
  - alternatives

Domain and motivation

Scientific Computing

- scientific computing - number crunching
  - process simulations, data analysis
  - speed is of most importance
  - methods and tools lag behind
- sparse linear systems in scientific computing
  - most physics simulations: weather forecast, air and fluid dynamics, structural mechanics
  - huge systems of linear equations: millions and billions of unknowns
  - *sparse*: most values in the matrix are zeros
  - general approach – *iterative solvers* with *preconditioners*
  - more easily parallelizable than *direct solvers*

Parallel iterative solvers

- need to solve $Ax = b$
- iterative solver
  - take initial approximation x_0 to the solution
  - in 5 to 100 iterations
    - using previous approximation x_i find next approximation x_{i+1}
- parallelize iterative solver (data parallelizm)
  - distribute A and b between the nodes
  - distribute x (each node is responsible for its own part of the vector)
  - intermediate vectors in each iteration “follow” x distribution

Conjugate Gradient method


```
x = np.zeros(b.shape)
r = b - A*x

it = 0
while np.sqrt(sum(r**2))>TOLERANCE and it<MAX_ITER:
    z = prec(r)

    rho = dot(r.T,z)
    if it==0:
        p = z
    else:
        beta = rho/rho_prev
        p = z + beta*p
    q = A*p
    alpha = rho/dot(p.T,q)
    x += alpha*p
    r -= alpha*q

    rho_prev = rho
    it += 1
```

Parallel CG

- matrix A and vectors b, x, p, r, q are distributed
- 2 operations are parallelized: vector dot product, matrix-vector multiplication
  - each requires synchornization and data exchange
  - communication pattern is static but only known at run-time
- *cg.py*: ~75 lines, ~20 is CG code
- *sparse.py*: ~76 lines, ~20 lines sparse matrix data structure and Ax code
- *parallel.py*: ~223 lines
  - ~129 is data preparation for parallel calculations
  - ~30 vector distribution/gather/parallel Ax/parallel dot product

Preconditioners for parallel CG

- Transformation to the original system: $M^{-1}Ax = M^{-1}b$
  - reduce the number of iterations
  - often implicitly
- "Preconditioner with robust coarse spaces", University of Bath, UK
  - 2 weeks to understand and implement reference version
  - optimization
  - parallelization

Idea of semi-automatic parallelization

The problem

- three vectors `x`, `y`, and `z`
- distribute elements of those vectors between processes
- `z = x+5*y` is trivial
- `sum(x)` and `dot(x,y)` are also trivial
- But not
  - `forall 1<i<N-1: z[i] = x[i-1]+y[i+1]`
  - `forall 1<i<N-1: z[inds[i]] = x[i]`
- these kind of relations are common

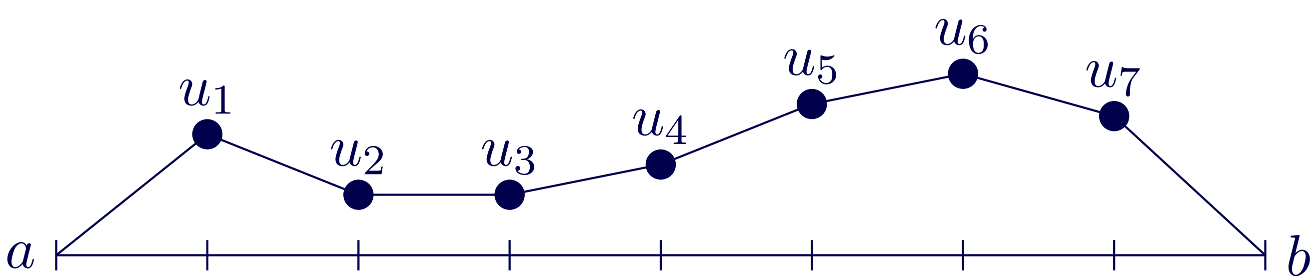
1D Finite Difference method

- need to solve $-\frac{\partial^2 u}{\partial x^2} = f(x)$, $a < x < b$, $u(a) = u(b) = 0$
- discretise $[a, b]$ into $n + 1$ even sections, $\Delta x = \frac{b-a}{n+1}$
- take unknowns $u_i \approx u(x_i)$, on the boundary $u_0 = u_{n+2} = 0$
- finite difference approximation for $\frac{\partial^2 u}{\partial x^2}$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x - \Delta x) - 2u(x) + u(x + \Delta x)}{\Delta x^2}$$

- for each $i = 1, \ldots, n$ get one linear equation

$$-u_{i-1} + 2u_i - u_{i+1} = \Delta x^2 f_i$$



- The system matrix

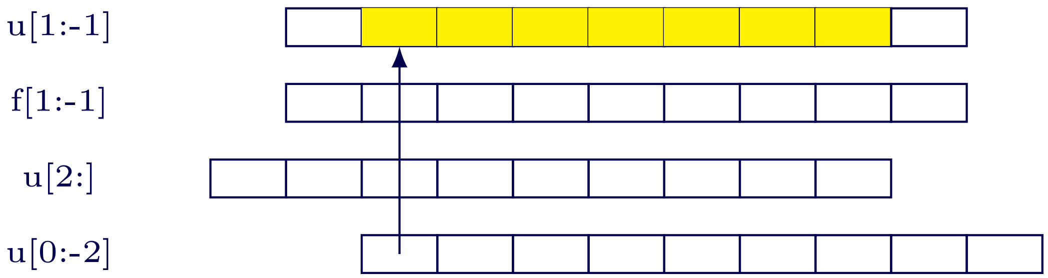
$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

- Jacobi method: iterative solver for $Au = b$

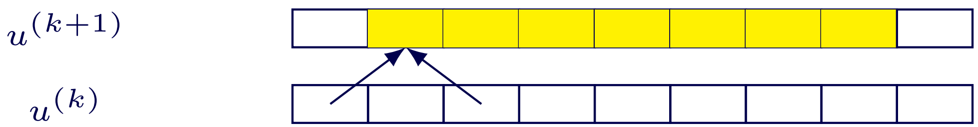
$$u_i^{(k+1)} = (b_i - \sum_{j=1, j \neq i}^{n} a_{i,j} u_i^{(k)})/a_{i,i} \quad i = 1, \ldots, n$$

Implicit implementation


```
for step in xrange(1000):
    u[1:-1] = (f[1:-1] + u[0:-2] + u[2:]) / 2.0
```

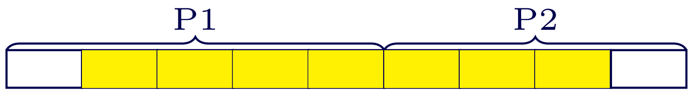


- $$u_i^{(k+1)} = (f_i + u_{i-1}^{(k)} + u_{i+1}^{(k)})/2$$

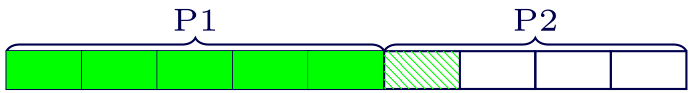


Parallelization (1)

- Distribute between 2 processes



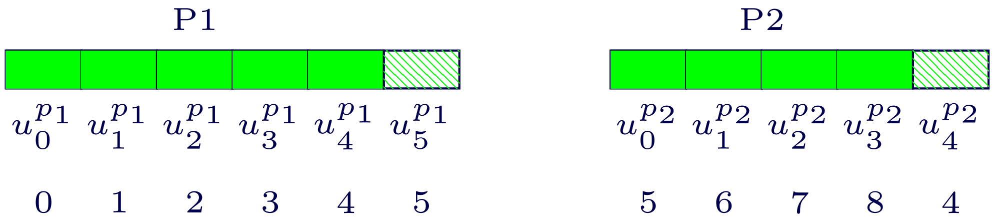
- $u_i^{(k+1)} = (f_i + u_{i-1}^{(k)} + u_{i+1}^{(k)})/2$
  - left-hand side determines where expression is evaluated
  - *ghost* values need to be received from other processes
- Local and ghost vector elements for process 1



- every iteration 1 value need to be sent from P1 to P2, and vice versa

Parallelization (2): reindexing

- store only local and ghost elements




```
for step in xrange(1000):
    u[1:-1] = (f[1:-1] + u[0:-2] + u[2:]) / 2.0
```

- reindexing slices with *index arrays*, for process 2 have
  - `1:-1` with `inds0=[0,1,2,3]`
  - `0:-2` with `inds1=[4,0,1,2]`
  - `2:` with `inds2=[1,2,3,4]`
- transform initial expression

```
u[inds0] = (f[inds0] + u[inds1] + u[inds2]) / 2.0
```

Semi-automatic parallelization (1)

- assume initial distribution of some vector is given $D_{\mathrm{x}} : I_{\mathrm{x}} \to P$ (domain decomposition)
- at compile time
  - find expressions that affect distribution and ghost values
  - collect pairs of slices, for each pair
    - $E(i, j)$ is a relation between indices of slices on LHS and RHS
      - `1:-1` to `0:-2`
  - modify them to use index arrays

Semi-automatic parallelization (2)

- at run-time
  - calculate ghost values from slice pairs
    - `y[...] = ... x[...] ...`
    - j is the index of ghost element for array x if

$$E(i,j) \bigwedge D_{\mathrm{y}}(i) = \mathrm{rank} \bigwedge D_{\mathrm{x}}(j) \neq \mathrm{rank}$$

  - create index arrays with ghost values

Real examples

Matrix-vector multiplication

- sparse matrix triple storage format – 3 arrays of size `nnz`
  - `irows` – row indices
  - `icols` – column indices
  - `vals` – matrix values
- matrix-vector multiplication $y = Ax$ (in vectorised form)

```
y[irows[:]] += x[icols[:]] * vals[:]
```

- calculate ghost values from both sides of expression
  - $I_x = I_y = I_0 \subset \mathbb{N}, \quad I_{\text{irows}} = I_{icols} = I_{vals} = I_1 \subset \mathbb{N}$
  - $D_0 : I_0 \to P \quad V_{\text{irows}} : I_1 \to I_0,$
  - i is the index of ghost element for array `icols` if
    - $D_y(V_{\text{irows}}(i)) = \text{rank} \bigwedge D_x(V_{\text{icols}}(i))) \neq \text{rank}$
  - $V_{\text{icols}}(i)$ is the index of ghost element for array x

First-level preconditioner

- preconditioning $z = Mr$
- without overlap
  - project $z^{(i)} = R^{(i)} z$ with projection matrices $R^{(i)}$
  - local matrices $A^{(i)} = R^{(i)} A \left(R^{(i)}\right)^T$, local preconditioners$M^{(i)} = \left(A^{(i)}\right)^{-1}$
  - total preconditioner $M = \sum_i \left(R^{(i)}\right)^T M^{(i)} R^{(i)}$
- with overlap
  - injection to the same element
  - not sum in total preconditioner

Coarse (second)-level preconditioner

- preconditioning $z = Mr$
- coarse grid on top of fine grid
- coarse nodes with unknowns r_c
- restrict $z_c = Rz$ with restriction matrix R
- coarse matrix $A_c = RAR^T$, coarse preconditioner $M_c = A_c^{-1}$
- preconditioner $M = R^T M_c R$

Static analysis

Why not a library

- usually 2 ways
  - ad-hoc parallel structures
    - parallel hash map
    - too limited
  - generalization of communication interfaces
    - local, ghost, border (overlap) values
    - still too limited – e.g. no map from coarse to fine vectors
    - requires a lot of code writing
- the other way: use some general rules
  - calculate how array elements are mapped based on non-parallel code

static analysis for communication

- communication and calculations
  - managed by different hardware
  - IO wait time
- with first and second level preconditioners
  1. values of second level preconditioners are send
  2. ghost values of first level preconditioner are sent
  3. first level-preconditiner is calculated with local values
  4. second level preconditioner is calculated
  5. first level-preconditiner is calculated with ghost values
- code is interleaved and messy

Summary

1. semi-automatic parallelization
   (a) assume distribution of some data is given
   (b) scan expressions and extract relations
   (c) apply algorithm that uses relations to find
       i. distribution of other data
       ii. communication pattern
   (d) transform the code
   - data dependencies, interprocedural analysis, alias analysis
2. optimize communication and calculation
   - send data early
   - data dependencies