Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. A Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with MPI_Allgather
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with mpi4py

3. A Multithreaded Julia Program
   - strip partitioning and reduce barriers

MCS 572 Lecture 29
Introduction to Supercomputing
Jan Verschelde, 17 March 2023
Parallel Iterative Methods for Linear Systems

1. **Jacobi iterations**
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. **A Parallel Implementation with MPI**
   - the sequential program
   - `gather-to-all with MPI_Allgather`
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with `mpi4py`

3. **A Multithreaded Julia Program**
   - strip partitioning and reduce barriers
a fixed point formula

We want to solve $Ax = b$ for $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, for very large $n$.

Consider $A = L + D + U$, where

- $L = [\ell_{i,j}]$, $\ell_{i,j} = a_{i,j}$, $i > j$, $\ell_{i,j} = 0$, $i \leq j$. $L$ is lower triangular.
- $D = [d_{i,j}]$, $d_{i,i} = a_{i,i} \neq 0$, $d_{i,j} = 0$, $i \neq j$. $D$ is diagonal.
- $U = [u_{i,j}]$, $u_{i,j} = a_{i,j}$, $i < j$, $u_{i,j} = 0$, $i \geq j$. $U$ is upper triangular.

Then we rewrite $Ax = b$ as

$$Ax = b \iff (L + D + U)x = b$$
$$\iff Dx = b - Lx - Ux$$
$$\iff Dx = Dx + b - Lx - Ux - Dx$$
$$\iff Dx = Dx + b - Ax$$
$$\iff x = x + D^{-1}(b - Ax).$$

The fixed point formula $x = x + D^{-1}(b - Ax)$ is well defined if $a_{i,i} \neq 0$. 
the Jacobi iterative method

The fixed point formula \( x = x + D^{-1}(b - Ax) \) leads to

\[
x^{(k+1)} = x^{(k)} + D^{-1} \left( b - Ax^{(k)} \right), \quad k = 0, 1, \ldots
\]

Writing the formula as an algorithm:

```
Input: A, b, x^{(0)}, \epsilon, N.
Output: x^{(k)}, k is the number of iterations done.

for k from 1 to N do
    \Delta x := D^{-1}(b - Ax^{(k)})
    x^{(k+1)} := x^{(k)} + \Delta x
    exit when (\|\Delta x\| \leq \epsilon)
end for.
```
cost and convergence

Counting the number of operations in

\[
\text{for } k \text{ from } 1 \text{ to } N \text{ do}
\]
\[
\Delta x := D^{-1}(b - A x^{(k)})
\]
\[
x^{(k+1)} := x^{(k)} + \Delta x
\]
\[
\text{exit when } (\|\Delta x\| \leq \epsilon)
\]
end for.

we have a cost of \(O(Nn^2)\), \(O(n^2)\) for \(A x^{(k)}\), if \(A\) is dense.

**Theorem (convergence of the Jacobi method)**

The Jacobi method converges for strictly row-wise or column-wise diagonally dominant matrices, i.e.: if

\[
|a_{i,i}| > \sum_{j \neq i} |a_{i,j}| \quad \text{or} \quad |a_{i,i}| > \sum_{j \neq i} |a_{j,i}|, \quad i = 1, 2, \ldots, n.
\]
Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. a Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with MPI_Allgather
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with mpi4py

3. a Multithreaded Julia Program
   - strip partitioning and reduce barriers
parallel version of Jacobi iterations

for \( k \) from 1 to \( N \) do
\[
\Delta x := D^{-1}(b - Ax^{(k)})
\]
\[
x^{(k+1)} := x^{(k)} + \Delta x
\]
exit when (\( \|\Delta x\| \leq \epsilon \))
end for.

To run the code above with \( p \) processors:

- The \( n \) rows of \( A \) are distributed evenly (e.g.: \( p = 4 \)):

\[
D \times \begin{bmatrix}
\Delta x[0] \\
\Delta x[1] \\
\Delta x[2] \\
\Delta x[3]
\end{bmatrix} = \begin{bmatrix}
b[0] \\
b[1] \\
b[2] \\
b[3]
\end{bmatrix} - \begin{bmatrix}
\end{bmatrix} \times \begin{bmatrix}
x[0, (k)] \\
x[1, (k)] \\
x[2, (k)] \\
x[3, (k)]
\end{bmatrix}
\]

- Synchronization is needed for (\( \|\Delta x\| \leq \epsilon \)).
butterfly synchronization

For \( \| \cdot \| \), use \( \| \Delta \mathbf{x} \|_1 = |\Delta x_1| + |\Delta x_2| + \cdots + |\Delta x_n| \).
communication and computation stages

The communication stages:
- At the start, every node must have $x^{(0)}$, $\epsilon$, $N$,
  - a number of rows of the matrix $A$; and
  - the corresponding part of the right hand side vector $b$.
- After each update $n/p$ elements of $x^{(k+1)}$ must be scattered.
- The butterfly synchronization takes $\log_2(p)$ steps.

The scattering of $x^{(k+1)}$ can coincide with the butterfly synchronization.

The computation effort: $O(n^2/p)$ in each stage.
Jacobi iterations
- derivation of the formulas
- parallel version with butterfly synchronization

2. a Parallel Implementation with MPI
- the sequential program
- gather-to-all with MPI_Allgather
- the parallel program
- analysis of the computation and communication cost
- collective communications with mpi4py

3. a Multithreaded Julia Program
- strip partitioning and reduce barriers
the test system

For the dimension $n$, we consider the diagonally dominant system:

$$
\begin{bmatrix}
  n+1 & 1 & \cdots & 1 \\
  1 & n+1 & \cdots & 1 \\
  \vdots & \vdots & \ddots & \vdots \\
  1 & 1 & \cdots & n+1 \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix}
= 
\begin{bmatrix}
  2n \\
  2n \\
  \vdots \\
  2n \\
\end{bmatrix}.
$$

The exact solution is $x$: for $i = 1, 2, \ldots, n$, $x_i = 1$.

We start the Jacobi iteration method at $x^{(0)} = 0$.

Parameters: $\epsilon = 10^{-4}$ and $N = 2n^2$. 
running the program

$ time ./jacobi 1000
  0 : 1.998e+03
  1 : 1.994e+03
...
8405 : 1.000e-04
8406 : 9.982e-05
computed 8407 iterations
error : 4.986e-05

real 0m42.411s
user 0m42.377s
sys 0m0.028s
void run_jacobi_method
  ( int n, double **A, double *b,
    double epsilon, int maxit,
    int *numit, double *x );

/*
 * Runs the Jacobi method for A*x = b.
 *
 * ON ENTRY:
 *  n    the dimension of the system;
 *  A    an n-by-n matrix A[i][i] /= 0;
 *  b    an n-dimensional vector;
 *  epsilon  accuracy requirement;
 *  maxit  maximal number of iterations;
 *  x     start vector for the iteration.
 *
 * ON RETURN:
 *  numit number of iterations used;
 *  x     approximate solution to A*x = b. */
void run_jacobi_method
( int n, double **A, double *b,
  double epsilon, int maxit,
  int *numit, double *x )
{

double *dx,*y;
dx = (double*) calloc(n,sizeof(double));
y = (double*) calloc(n,sizeof(double));
int i,j,k;

for(k=0; k<maxit; k++) { ... } /* main loop */

*numit = k+1;
free(dx); free(y);
}
for(k=0; k<maxit; k++)
{
    double sum = 0.0;
    for(i=0; i<n; i++)
    {
        dx[i] = b[i];
        for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
        dx[i] /= A[i][i];
        y[i] += dx[i];
        sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
    }
    for(i=0; i<n; i++) x[i] = y[i];
    printf("%3d : %.3e\n",k,sum);
    if(sum <= epsilon) break;
}
Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. A Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with MPI_Allgather
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with mpi4py

3. A Multithreaded Julia Program
   - strip partitioning and reduce barriers
Gathering the four elements of a vector to four processors:

\[
\begin{align*}
P_0 & : 1 & 0 & 0 & 0 \\
P_1 & : 0 & 2 & 0 & 0 \\
P_2 & : 0 & 0 & 3 & 0 \\
P_3 & : 0 & 0 & 0 & 4
\end{align*}
\]
The syntax of the gather-to-all command is

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype, comm)
```

where the parameters are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements received from any process</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
running `use_allgather`

```bash
$ mpirun -np 4 ./use_allgather
data at node 0 : 1 0 0 0
data at node 1 : 0 2 0 0
data at node 2 : 0 0 3 0
data at node 3 : 0 0 0 4
data at node 3 : 1 2 3 4
data at node 0 : 1 2 3 4
data at node 1 : 1 2 3 4
data at node 2 : 1 2 3 4
$```
```c
int i, j, p;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &i);
MPI_Comm_size(MPI_COMM_WORLD, &p);
{
    int data[p];
    for (j = 0; j < p; j++) data[j] = 0;
    data[i] = i + 1;
    printf("data at node %d : ", i);
    for (j = 0; j < p; j++) printf(" %d", data[j]);
    printf("\n");
    MPI_Allgather(&data[i], 1, MPI_INT,
                   data, 1, MPI_INT, MPI_COMM_WORLD);
    printf("data at node %d : ", i);
    for (j = 0; j < p; j++) printf(" %d", data[j]);
    printf("\n");
}
```
Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. A Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with MPI_Allgather
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with mpi4py

3. A Multithreaded Julia Program
   - strip partitioning and reduce barriers
running `jacobi_mpi`

```bash
$ time mpirun -np 10 ./jacobi_mpi 1000
...
8405 : 1.000e-04
8406 : 9.982e-05
computed 8407 iterations
error : 4.986e-05
```

real 0m5.617s
user 0m45.711s
sys 0m0.883s

**Recall the run with the sequential program:**

real 0m42.411s
user 0m42.377s
sys 0m0.028s

**Speedup:** \( \frac{42.411}{5.617} = 7.550 \).
void run_jacobi_method
( int id, int p,
  int n, double **A, double *b,
  double epsilon, int maxit,
  int *numit, double *x )
{
  double *dx,*y;
  dx = (double*) calloc(n,sizeof(double));
  y = (double*) calloc(n,sizeof(double));
  int i,j,k;
  double sum[p];
  double total;
  int dnp = n/p;
  int istart = id*dnp;
  int istop = istart + dnp;
the main loop in `jacobi_mpi.c`

```c
for(k=0; k<maxit; k++)
{
    sum[id] = 0.0;
    for(i=istart; i<istop; i++)
    {
        dx[i] = b[i];
        for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
        dx[i] /= A[i][i];
        y[i] += dx[i];
        sum[id] += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
    }
    for(i=istart; i<istop; i++) x[i] = y[i];
}
```

Introduction to Supercomputing (MCS 572) Parallel Iterative Methods L-29 17 March 2023 24/37
MPI_Allgather(&x[istart], dnp, MPI_DOUBLE, x, dnp, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Allgather(&sum[id], 1, MPI_DOUBLE, sum, 1, MPI_DOUBLE, MPI_COMM_WORLD);

total = 0.0;
for (i=0; i<p; i++) total += sum[i];
if (id == 0) printf("%3d : %.3e\n", k, total);
if (total <= epsilon) break;

*numit = k+1;
free(dx);
Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. a Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with \texttt{MPI\_Allgather}
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with \texttt{mpi4py}

3. a Multithreaded Julia Program
   - strip partitioning and reduce barriers
Computing $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$ with $p$ processors costs

$$t_{\text{comp}} = \frac{n(2n + 3)}{p}.$$ 

We count $2n + 3$ operations because of
- one $-$ and one $\times$ when running over the columns of $A$; and
- one $/$, one $+$ for the update and one $+$ for the $\| \cdot \|_1$.

The communication cost is

$$t_{\text{comm}} = p \left( t_{\text{startup}} + \frac{n}{p} t_{\text{data}} \right).$$

In the examples, the time unit is the cost of one arithmetical operation. Then the costs $t_{\text{startup}}$ and $t_{\text{data}}$ are multiples of this unit.
finding the $p$ with the minimum total cost

The computation, communication, and total cost for $p$ from 2 to 32, for 1 iteration, $n = 1,000$, $t_{\text{startup}} = 10,000$, and $t_{\text{data}} = 50$. 
investigating the scalability

The computation, communication, and total cost for \( p \) from 16 to 256, for 1 iteration, \( n = 10,000 \), \( t_{\text{startup}} = 10,000 \), and \( t_{\text{data}} = 50 \).
Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. A Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with `MPI_Allgather`
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with `mpi4py`

3. A Multithreaded Julia Program
   - strip partitioning and reduce barriers
a parallel matrix-vector product

Copied from the MPI for Python documentation:

```python
from mpi4py import MPI
import numpy

def matvec(comm, A, x):
    m = A.shape[0]  # local rows
    p = comm.Get_size()
    xg = numpy.zeros(m*p, dtype='d')
    comm.Allgather([x, MPI.DOUBLE],
                    [xg, MPI.DOUBLE])
    y = numpy.dot(A, xg)
    return y
```

Parallel Iterative Methods for Linear Systems

1. Jacobi iterations
   - derivation of the formulas
   - parallel version with butterfly synchronization

2. A Parallel Implementation with MPI
   - the sequential program
   - gather-to-all with MPI_Allgather
   - the parallel program
   - analysis of the computation and communication cost
   - collective communications with mpi4py

3. A Multithreaded Julia Program
   - strip partitioning and reduce barriers
strip partitioning

If the dimension of the matrix is a multiple of the number of threads, for some matrix $A$ and vectors $x, y$:

```plaintext
nt = nthreads()
size = 10
dim = nt * size

@threads for i=1:nt
    tdx = threadid()
    idxstart = 1 + (tdx-1)*size
    idxend = tdx*size
    @inbounds y[idxstart:idxend] = A[idxstart:idxend, :] * x
end
```
reduce barriers

using Base.Threads
using SyncBarriers

nt = nthreads()
nb = [k for k=1:nt]
barrier = reduce_barrier(+, Int, nt)
s = 0
@threads for i=1:nt
    tdx = threadid()
    global s = reduce!(barrier[tdx], nb[tdx])
end
println("The sum of ", nb, " is ", s, ".")

The output of julia -t 4 mtreduce.jl is

The sum of [1, 2, 3, 4] is 10.
a multithreaded method of Jacobi

In a multithreaded Jacobi method, with \( p \) threads:

1. The \( i \)-th thread
   - computes the \( i \)-th strip of the update \( \Delta x_i \),
   - updates the \( i \)-th strip of \( x_i \) with \( \Delta x_i \),
   - computes the norm of the \( i \)-th update \( \|\Delta x_i\| \).

2. Given \( (\|\Delta x_1\|, \|\Delta x_2\|, \ldots, \|\Delta x_p\|) \), a reduce barrier computes

\[
\|\Delta x\|_1 = \|\Delta x_1\| + \|\Delta x_2\| + \cdots + \|\Delta x_p\|
\]

and that \( \|\Delta x\|_1 \) is used by every thread as the stop criterion.

See the Julia program \texttt{mtjacobi.jl}, posted at the course web site.
three runs on pascal

time julia -t 2 mtjacobi.jl 8000
number of iterations : 40
the error : 1.968107347290746e-5

real 0m15.390s
user 11m35.441s
sys 4m51.916s

$ time julia -t 4 mtjacobi.jl 8000
number of iterations : 20
the error : 2.3621495916454325e-5

real 0m5.400s
user 2m13.138s
sys 1m18.059s

$ time julia -t 8 mtjacobi.jl 8000
number of iterations : 39
the error : 1.7918058060438385e-5

real 0m5.400s
user 2m10.425s
sys 1m13.413s
We covered §6.3.1 in the book of Wilkinson and Allen. *Because of its slow convergence, the Jacobi method is seldomly used.*

**Exercises:**

1. Use mpi4py or MPI.jl for the parallel Jacobi method. Compare with the C version to demonstrate the correctness.

2. Use OpenMP to write a parallel version of the Jacobi method. Do you observe a better speedup than with MPI?

3. The power method to compute the largest eigenvalue of a matrix $A$ uses the formulas $y := Ax^{(k)}; x^{(k+1)} := y/\|y\|$. Describe a parallel implementation of the power method.

4. Consider the formula for the total cost of the Jacobi method for an $n$-dimensional linear system with $p$ processors. Derive an analytic expression for the optimal value of $p$. What does this expression tell about the scalability?