Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication

MCS 572 Lecture 8
Introduction to Supercomputing
Jan Verschelde, 9 September 2016
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
To turn a sequential algorithm into a parallel one, we distinguish between functional and domain decomposition:

**Functional decomposition:** distribute arithmetical operations among several processors.

Example: Monte Carlo simulations.

**Domain decomposition:** distribute data among several processors.

Example: Mandelbrot set computation.

Problem solving by parallel computers: the entire data set is often too large to fit into the memory of one computer.

Example: game tree for four in a row.
Divide and conquer used to solve problems:
- break the problem in smaller parts,
- solve the smaller parts,
- assemble the partial solutions.

Often, divide and conquer is applied in a recursive setting where the smallest nontrivial problem is the base case.

Examples in sorting: mergesort and quicksort.
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for \( \pi \)

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
summing numbers with divide and conquer

\[ \sum_{k=0}^{7} x_k = (x_0 + x_1 + x_2 + x_3) + (x_4 + x_5 + x_6 + x_7) \]

\[ = ((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7)) \]

With 4 processors, the summation of 8 numbers is done in 3 steps.
making partial sums

The size of the problem is \( n \), where \( S = \sum_{k=0}^{n-1} x_k \).

Assume we have 8 processors to make 8 partial sums:

\[
S = (S_0 + S_1 + S_2 + S_3) + (S_4 + S_5 + S_6 + S_7)
= ((S_0 + S_1) + (S_2 + S_3)) + ((S_4 + S_5) + (S_6 + S_7))
\]

where \( m = \frac{n - 1}{8} \) and \( S_i = \sum_{k=0}^{m} x_{k+im} \).

The communication pattern goes along divide and conquer:

- the numbers \( x_k \) are scattered in a *fan out* fashion,
- summing the partial sums happens in a *fan in* mode.
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
Algorithm: at step $k$, $2^k$ processors have data, and execute:

for $j$ from 0 to $2^k - 1$ do

processor $j$ sends $\frac{\text{data}}{2^{k+1}}$ to processor $j + 2^k$;

processor $j + 2^k$ receives $\frac{\text{data}}{2^{k+1}}$ from processor $j$. 
refining the algorithm

In fanning out, we want to use the same array for all nodes, and use only one send/recv statement.

Observe the bit patterns in nodes and data locations:

<table>
<thead>
<tr>
<th>node</th>
<th>step 0</th>
<th>step 1</th>
<th>step 2</th>
<th>step 3</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>[0...7]</td>
<td>[0...3]</td>
<td>[0...1]</td>
<td>[0]</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>[4...7]</td>
<td>[4...5]</td>
<td>[4]</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>[2...3]</td>
<td>[2]</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>[6...7]</td>
<td>[6]</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>[1]</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>[5]</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>[3]</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>[7]</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At step 3, the node with label in binary expansion $b_2 b_1 b_0$ has data starting at index $b_0 b_1 b_2$. 
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
on dual core Mac OS X with 8 processes

$ mpirun -np 8 /tmp/fan_out_integers
stage 0, d = 1 :
0 sends 40 integers to 1 at 40, start 40
1 received 40 integers from 0 at 40, start 40
stage 1, d = 2 :
0 sends 20 integers to 2 at 20, start 20
2 received 20 integers from 0 at 20, start 20
1 sends 20 integers to 3 at 60, start 60
3 received 20 integers from 1 at 60, start 60
stage 2, d = 4 :
0 sends 10 integers to 4 at 10, start 10
7 received 10 integers from 3 at 70, start 70
3 sends 10 integers to 7 at 70, start 70
4 received 10 integers from 0 at 10, start 10
1 sends 10 integers to 5 at 50, start 50
2 sends 10 integers to 6 at 30, start 30
6 received 10 integers from 2 at 30, start 30
5 received 10 integers from 1 at 50, start 50
run continued

data at all nodes:
1 has 10 integers starting at 40 with 40, 41, 42
2 has 10 integers starting at 20 with 20, 21, 22
7 has 10 integers starting at 70 with 70, 71, 72
5 has 10 integers starting at 50 with 50, 51, 52
0 has 10 integers starting at 0 with 0, 1, 2
6 has 10 integers starting at 30 with 30, 31, 32
3 has 10 integers starting at 60 with 60, 61, 62
4 has 10 integers starting at 10 with 10, 11, 12
MPI_Barrier to synchronize printing

To synchronize across all members of a group we apply

MPI_Barrier(comm)

where comm is the communicator (MPI_COMM_WORLD).

MPI_Barrier blocks the caller until all group members have called the statement.

The call returns at any process only after all group members have entered the call.
computing the offset

int parity_offset ( int n, int s );
/* returns the offset of node with label n
   * for data of size s based on parity of n */

int parity_offset ( int n, int s )
{
    int offset = 0;
    s = s/2;
    while(n > 0)
    {
        int d = n % 2;
        if(d > 0) offset += s;
        n = n/2;
        s = s/2;
    }
    return offset;
}
start of the main program

/* include headers omitted */
#define size 80       /* size of the problem */
#define tag 100       /* tag of send/recv */
#define v 1           /* verbose flag */

int main ( int argc, char *argv[] )
{
    int myid, p, s, i, j, d, b;
    int A[size];

    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if(myid == 0) /* manager initializes */
        for(i=0; i<size; i++) A[i] = i;
s = size;
for(i=0,d=1; i<3; i++,d*=2) /* A is fanned out */ {
    s = s/2;
    if(v>0) MPI_Barrier(MPI_COMM_WORLD);
    if(myid == 0)
        if(v > 0) printf("stage %d, d = %d :\n", i, d);
    if(v>0) MPI_Barrier(MPI_COMM_WORLD);
    for(j=0; j<d; j++)
    {
        b = parity_offset(myid, size);
```c
for(j=0; j<d; j++){
    b = parity_offset(myid,size);
    if(myid == j){
        if(v>0)
            printf("%d sends %d integers to %d at %d, \n start %d\n",j,s,j+d,b+s,A[b+s]);
        MPI_Send(&A[b+s],s,MPI_INT,j+d,tag,MPI_COMM_WORLD);
    }
    else if(myid == j+d){
        MPI_Recv(&A[b],s,MPI_INT,j,tag,MPI_COMM_WORLD,&status);
        if(v>0)
            printf("%d received %d integers from %d at %d, \n start %d\n",j+d,s,j,b,A[b]);
    }
}
```
the end of the program

}
if(v > 0) MPI_Barrier(MPI_COMM_WORLD);
if(v > 0) if(myid == 0) printf("data at all nodes :\n");
if(v > 0) MPI_Barrier(MPI_COMM_WORLD);
printf("%d has %d integers starting at %d with %d, %d, %d
", myid, size/p, b, A[b], A[b+1], A[b+2]);
MPI_Finalize();
return 0;
}
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for \( \pi \)

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
Algorithm: at step $k$, $2^k$ processors send results and execute:

for $j$ from 0 to $2^k - 1$ do
    processor $j + 2^k$ sends the result to processor $j$;
    processor $j$ receives the result from processor $j + 2^k$.

We run the algorithm for decreasing values of $k$: $k = 2, 1, 0$. 
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
the BBP algorithm for $\pi$

Computing $\pi$ to trillions of digits is a benchmark problem for supercomputers.

One of the remarkable discoveries made by the PSLQ Algorithm (PSLQ = Partial Sum of Least Squares, or integer relation detection) is a simple formula that allows to calculating any binary digit of $\pi$ without calculating the digits preceding it:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left( \frac{4}{8i + 1} - \frac{2}{8i + 4} - \frac{1}{8i + 5} - \frac{1}{8i + 6} \right).$$

BBP stands for Bailey, Borwein and Plouffe.

Instead of adding numbers, we concatenate strings.
some readings on calculations for $\pi$ 

  http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/.
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for π

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
nonblocking point-to-point communication

The `MPI_SEND` and `MPI_RECV` are `blocking`:
- The sender must wait till the message is received.
- The receiver must wait till the message is sent.

For synchronized computations, this is desirable.

To overlap the communication with the computation, we may prefer the use of `nonblocking` communication operations:
- `MPI_ISEND` for the Immediate send; and
- `MPI_Irecv` for the Immediate receive.

The status of the immediate send/receive
- can be queried with `MPI_TEST`; or
- we can wait for its completion with `MPI_WAIT`.
MPI_ISEND specification

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>address of the send buffer</td>
</tr>
<tr>
<td>count</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of each send buffer element</td>
</tr>
<tr>
<td>dest</td>
<td>rank of the destination</td>
</tr>
<tr>
<td>tag</td>
<td>message tag</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
<tr>
<td>request</td>
<td>communication request (output)</td>
</tr>
</tbody>
</table>

The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.
MPI_Irecv specification

\[
\text{MPI_Irecv} \ (\text{buf}, \ \text{count}, \ \text{datatype}, \ \text{source}, \\
\text{tag}, \ \text{comm}, \ \text{request})
\]

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>address of the receive buffer</td>
</tr>
<tr>
<td>count</td>
<td>number of elements in receive buffer</td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
</tr>
<tr>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
</tr>
<tr>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
<tr>
<td>request</td>
<td>communication request (output)</td>
</tr>
</tbody>
</table>

The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.
Data Partitioning

1. Data Partitioning
   - functional and domain decomposition

2. Parallel Summation
   - applying divide and conquer
   - fanning out an array of data
   - fanning out with MPI
   - fanning in the results

3. An Application
   - computing hexadecimal expansions for $\pi$

4. Nonblocking Point-to-Point Communication
   - immediate send and receive
   - query the status of a communication
waiting for a nonblocking communication

After the call to MPI_ISEND or MPI_Irecv, the request can be used to query the status of the communication or wait for its completion.

To wait for the completion of a nonblocking communication:

```
MPI_WAIT (request, status)
```

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>communication request</td>
</tr>
<tr>
<td>status</td>
<td>status object</td>
</tr>
</tbody>
</table>

To test the status of the communication:

```
MPI_TEST (request, flag, status)
```

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>communication request</td>
</tr>
<tr>
<td>flag</td>
<td>true if operation completed</td>
</tr>
<tr>
<td>status</td>
<td>status object</td>
</tr>
</tbody>
</table>
We started chapter 4 in the text book by Wilkinson and Allen.

Exercises:

1. Adjust the fanning out of the array of integers so it works for any number \( p \) of processors where \( p = 2^k \) for some \( k \). You may take the size of the array as an integer multiple of \( p \). To illustrate your program, provide screen shots for \( p = 8, 16, \) and 32.

2. Complete the summation and the fanning in of the partial sums, extending the program. You may leave \( p = 8 \).