Distributed Memory Parallel Programming

- Vast majority of clusters are homogeneous
  - Necessitated by the complexity of maintaining heterogeneous resources
- Most problems can be divided into constant chunks of work upfront
  - Often based on Geometric Domain Decomposition
  - No necessity for a master process to manage work load
  - Each process can calculate the work distribution by themselves
- Efficiency, Portability, Usability primary goals
About MPI

- MPI defines interfaces how to send data to a process and how to receive data from a process.
  - Not a protocol
  - Does not enforce a particular implementation
- Until the early 90’s:
  - All vendors of parallel hardware had their own message passing library
  - Some public domain message passing libraries available
  - High efforts for end-users to move code from one architecture to another
- June 1994: Version 1.0 of MPI presented by the MPI Forum
- 1997: MPI 2.0 - adding new functionality to MPI
- 2012: MPI 3.0 released in Nov. 2012

Simple Example (I)

MPI command to start process

name of the application to start

number of processes to be started

Number of processes which have been started

Rank of the 2nd process

Rank of the 1st process

Connected to shank.cs.uh.edu
**Simple example (II)**

MPICH starts the application `t1`
- two times (as specified with the `-np` argument)
- on two currently available processors of the parallel machine
- telling one process that his `rank` is 0
- and the other that his `rank` is 1

```bash
$shark> mpirun -np 2 ./t1
```

**Simple Example (III)**

```c
#include "mpi.h"

int main ( int argc, char **argv )
{
  int rank, size;

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

  printf ("MPI Hello World from process %d job size %d\n",
          rank, size);

  MPI_Finalize ();
  return (0);
}
```
MPI summary (I)

- **mpirun** starts the required number of processes
- every process has a unique identifier (**rank**) which is between 0 and \(n-1\)
  - no identifiers are duplicate, no identifiers are left out
- all processes which have been started by **mpirun** are organized in a process group (**communicator**) called **MPI_COMM_WORLD**
- **MPI** assumes a static configuration
  - number of processes can not change
  - participating processes can not change

### Sending Data

<table>
<thead>
<tr>
<th>Data element which shall be send</th>
<th>Number of elements which shall be send</th>
<th>Data Type of the element which shall be send</th>
</tr>
</thead>
<tbody>
<tr>
<td>---snip---</td>
<td>---snip---</td>
<td>---snip---</td>
</tr>
</tbody>
</table>

```c
MPI_Send (&s_local, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

- Process group containing all processes started by **mpirun**
- Rank of processes in the process group **MPI_COMM_WORLD** to which the message shall be sent
- a user defined integer (tag) for uniquely identifying a message
Receiving Data

- Buffer to use for receiving the data
- Number of elements to be received
- Data type of the element to be received
- Process group
- Rank of processes in the process group which sent the message
- Status information about the message
- A user defined integer (tag) for uniquely identifying a message

```c
MPI_Recv (&s_local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

Static work distribution

- Example: block column distribution of an $N \times N$ matrix
- Each of the $p$ processes holds $N/p$ rows of the matrix

```c
#define DIM 9
double *A;
int rank, nprocs;
...
MPI_Comm_rank ( MPI_COMM_WORLD, &rank);
MPI_Comm_size ( MPI_COMM_WORLD, &nprocs);

A = malloc ( sizeof(double) * DIM * DIM/nprocs );
fp = open ("matrix.out", O_RDONLY);
lseek ( fp, rank * DIM * DIM/nprocs*sizeof(double), SEEK_SET));
read ( fp, &A[0][0], sizeof(double)*DIM*DIM/nprocs);
...
```
Further features of MPI

- **Performance**
  - asynchronous point-to-point operations
  - collective operations
  - derived data-types
  - parallel I/O
  - hints

- **Usability**
  - process grouping functions
  - environmental and process management
  - error handling
  - object attributes
  - language bindings

Collective operation

- **All process of a process group have to participate in the same operation**
  - process group is defined by a communicator
  - all processes have to provide the same arguments
  - for each communicator, you can have one collective operation ongoing at a time

- **Collective operations are abstractions for often occurring communication patterns**
  - eases programming
  - enables low-level optimizations and adaptations to the hardware infrastructure
MPI_Bcast

```c
MPI_Bcast (void *buf, int cnt, MPI_Datatype dat, 
            int root, MPI_Comm comm);
```

- The process with the rank root distributes the data stored in buf to all other processes in the communicator comm.
- Data in buf is identical on all processes after the bcast operation
- Compared to point-to-point operations no tag, since you cannot have several ongoing collective operations

MPI Word count

- Each process determines the number of lines and the offset to the number of lines it has to read
  - holds a temporary list of <word, #occurrences> as a result of this operation
    - Dynamic load balancing through shared file pointer operations would be possible
- Parallel Sorting of the temporary list across processes
- Each process is assigned a subset of the sorted list of words to perform the summation of the occurrences
- Writing of the final output file
Parallel Sorting

- Given:
  - Problem size $N$
  - Number of processes $p$
- Each process holds initially $N' = N/p$ elements of the problem
- Goal: have all elements sorted across all processes
- Prominent Algorithm: PSRS (Parallel Sorting through Regular Sampling)

Parallel Sorting through Regular Sampling

Algorithm:
1. Local sort operation by each process
2. Each process selects $p-1$ pivot points, by separating the list into $N$ sections and use the $i$th element as the local pivot point, with $i = k \times N'/p$, $k = 1, p-1$
3. Gather pivot points by all processes
4. Sort pivot points and select global pivot points using the same approach as in step 2
5. Distribute global pivot points to all processes
6. Redistribute data across processes such that process 0 holds all elements $< 1^{st}$ global pivot, process 1 all elements $\geq 1^{st}$ global pivot point but $< 2^{nd}$ global pivot point etc.
7. Local sort of the data
### 1. Local sort operation on each process

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>47</td>
<td>3</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>89</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>22</td>
<td>65</td>
<td>59</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>49</td>
</tr>
</tbody>
</table>

**N = 27**

**p = 3**

**N' = 9**

### 2. Each process selects p-1 local pivot points

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>22</td>
<td>24</td>
<td>45</td>
</tr>
<tr>
<td>23</td>
<td>28</td>
<td>49</td>
</tr>
<tr>
<td>31</td>
<td>29</td>
<td>51</td>
</tr>
<tr>
<td>40</td>
<td>65</td>
<td>59</td>
</tr>
<tr>
<td>47</td>
<td>89</td>
<td>61</td>
</tr>
</tbody>
</table>

### 3. Gather all pivot points

- **19**
- **31**
- **18**
- **20**
- **30**
- **51**

### 4a. Sort pivot points

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
</tbody>
</table>

### 4b. Determine global pivot points

- All elements <20 to process 0
- All elements >=20 but < 31 to process 1
- All elements >=31 to process 2
6. Redistribution of data across processes based on global pivot points (20, 31)

7. Local sort operation on each process
MPI version of PSRS

Algorithm:
1. Local sort operation by each process
2. Each process selects \( p-1 \) pivot points, by separating the list into \( N \) sections and use the \( i^{th} \) element as the local pivot point, with \( i = k \cdot N' / p \), \( k = 1, p-1 \)
3. Gather pivot points by all processes \( \rightarrow \text{MPI\_Gather()} \)
4. Sort pivot points and select global pivot points using the same approach as in step 2
5. Distribute global pivot points to all processes \( \rightarrow \text{MPI\_Bcast()} \)
6. Redistribute data across processes such that process 0 holds all elements < 1st global pivot, process 1 all elements >= 1st global pivot point but < 2nd global pivot point etc.
7. Local sort of the data

Performance of PSRS
- Sorting an array of edges in an MPI parallel image segmentation code
K-means clustering: MPI parallel version

Algorithm

One process: set C to initial value

broadcast C to all processes

while m has changed

\[ c_{\text{temp}}^k = 0, \quad k = 1, \ldots, n \]

\[ \text{count}_k = 0, \quad k = 1, \ldots, n \]

for each \( i \in I \)

\[ m(i_j) = \min \text{ distance } (i_j, c_k), \quad k = 1, \ldots, n \]

\[ c_{\text{temp}}^k += j \]

\[ \text{count}_k ++ \]

end

Determine global sum of all \( c_{\text{temp}} \) across all processes

Determine global sum \( \text{count} \) across all processes

recompute C based on \( c_{\text{temp}} \) and \( \text{count} \)

Reduction operations

- Perform simple calculations (e.g. caculate the sum or the product) over all processes in the communicator
- MPIReduce
  - \( \text{outbuf} \) has to be provided by all processes
  - result is only available at \( \text{root} \)
- MPIAllreduce
  - result available on all processes
Predefined reduction operations

- MPI_SUM: sum
- MPI_PROD: product
- MPI_MIN: minimum
- MPI_MAX: maximum
- MPI_LAND: logical and
- MPI_LOR: logical or
- MPI_LXOR: logical exclusive or
- MPI_BAND: binary and
- MPI_BOR: binary or
- MPI_BXOR: binary exclusive or
- MPI_MAXLOC: maximum value and location
- MPI_MINLOC: minimum value and location

Reduction operations on vectors

- Reduce operation is executed element wise on each entry of the array

<table>
<thead>
<tr>
<th>Rank 0 inbuf</th>
<th>Rank 1 inbuf</th>
<th>Rank 2 inbuf</th>
<th>Rank 3 inbuf</th>
<th>Rank 0 outbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>26</td>
</tr>
</tbody>
</table>

- Reduction of 5 elements with root = 0

```c
MPI_Reduce (inbuf, outbuf, 5, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```
K-means clustering parallel performance

- K-means clustering in an MPI parallel image segmentation code

![Graph showing execution time vs. number of processes for K-means clustering.](image)

Summary

- Data Decomposition represents a very general parallel programming paradigm
  - More generic than Master-Worker
  - Has been shown to scale to ~100,000 processes
- MPI mostly designed for static data distribution
- Collective operations a powerful feature
- MPI is not designed for failure-prone environments
  - Either all MPI processes finish correctly, or none
  - Difficult to extend to support process failures