Big Data Analytics

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Computer Science
University of Hildesheim, Germany

A. Parallel Computing / 2. Message Passing Interface (MPI)

Syllabus

Tue.	4.4.	(1)	0. Introduction
			A. Parallel Computing
Tue.	11.4.	(2)	A.1 Threads
Tue.	18.4.	(3)	A.2 Message Passing Interface (MPI)
Tue.	2.5.	(4)	A.3 Graphical Processing Units (GPUs)
			B. Distributed Storage
Tue.	9.5.	(5)	B.1 Distributed File Systems
Tue.	16.5.	(6)	B.2 Partioning of Relational Databases
Tue.	23.5.	(7)	B.3 NoSQL Databases
			C. Distributed Computing Environments
Tue.	30.5.	(8)	C.1 Map-Reduce
Tue.	6.6.	—	— Pentecoste Break —
Tue.	13.6.	(9)	C.2 Resilient Distributed Datasets (Spark)
			D. Distributed Machine Learning Algorithms
Tue.	20.6.	(10)	D.1 Distributed Stochastic Gradient Descent
Tue.	27.6.	(11)	D.2 Distributed Matrix Factorization
Tue.	4.7.	(12)	D.3 Alternating Direction Method of Multipliers (ADMM)

Outline

1. MPI Basics

2. Point to Point Communication

3. Collective Communication

4. One-sided Communication

Outline

1. MPI Basics

2. Point to Point Communication

3. Collective Communication

4. One-sided Communication

The MPI Standard

- A standard for parallel and distributed computing
- Authored by a consortium of academics and industry.
  - MPI 1.0 (1994; 236 pages)
  - MPI 2.0 (1998)
  - MPI 3.0 (2012)
  - MPI 3.1 (2015; 868 pages)

The MPI Standard

- A standard for parallel and distributed computing
- Authored by a consortium of academics and industry.
  - MPI 1.0 (1994; 236 pages)
  - MPI 2.0 (1998)
  - MPI 3.0 (2012)
  - MPI 3.1 (2015; 868 pages)
- Basic concept:
  - Processes run in parallel
  - Processes synchronize and exchange data by **passing messages** from one to another.
- http://www.mpi-forum.org/docs/

OpenMPI / Compile and Run

- OpenMPI: an open source reference implementation of MPI
  - http://www.openmpi.org
  - support C++, C and Fortran
    - see MPI4Py for Python, http://pythonhosted.org/mpi4py/
- compile programs with

```
1 mpijavac Hello.java
```

  - just runs javac with mpi.jar in the classpath.
- run programs with

```
1 mpirun java Hello
```

  - option **-np N**: to start N copies in parallel
  - option **-H h1,h2,h3**: to start processes on hosts h1,h2 and h3.
  - to run on other hosts, one needs:
    - password-less ssh login to the compute host from the submit host
    - openmpi installed on both hosts

Java MPI Skeleton

- **MPI**: service class with static methods and constants:
  - **Init(args)**: initialize the MPI system
  - **Finalize()**: shutdown the MPI system
  - **COMM_WORLD**: default communicator (class **Intracomm**)

Java MPI Skeleton

- **MPI**: service class with static methods and constants:
  - **Init(args)**: initialize the MPI system
  - **Finalize()**: shutdown the MPI system
  - **COMM_WORLD**: default communicator (class **Intracomm**)

- The communicator allows interactions with other processes:
  - **getSize()**: number of processes in this group.
  - **getRank()**: id of this process (between 0 and size-1).
  - synchronize
  - exchange data

Java MPI Skeleton

- **MPI**: service class with static methods and constants:
  - **Init(args)**: initialize the MPI system
  - **Finalize()**: shutdown the MPI system
  - **COMM_WORLD**: default communicator (class **Intracomm**)

- The communicator allows interactions with other processes:
  - **getSize()**: number of processes in this group.
  - **getRank()**: id of this process (between 0 and size-1).
  - synchronize
  - exchange data

- **MPIException**: thrown if anything goes wrong.

Java MPI Skeleton

```
1  import mpi.*;
2
3  public class ... {
4      public static void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = comm.getRank(),
8              num_workers = comm.getSize();
9          ...
10         MPI.Finalize();
11     }
12 }
```

Hello World MPI

```
1  import mpi.*;
2
3  class Hello {
4      static public void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = comm.getRank(),
8              num_workers = comm.getSize();
9          System.out.println("Hello world from worker " + worker + " of " + num_workers);
10         MPI.Finalize();
11     }
12 }
```

Hello World MPI

```
1  import mpi.*;
2
3  class Hello {
4      static public void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = comm.getRank(),
8              num_workers = comm.getSize();
9          System.out.println("Hello world from worker " + worker + " of " + num_workers);
10         MPI.Finalize();
11     }
12 }
```

Output

```
Hello world from worker 2 of 4
Hello world from worker 0 of 4
Hello world from worker 3 of 4
Hello world from worker 1 of 4
```

Outline

1. MPI Basics

2. Point to Point Communication

3. Collective Communication

4. One-sided Communication

Blocking vs. Non-blocking

- Send and Receive have to occur paired at a source and a destination process.
- Blocking Send and Receive:
  - process waits/blocks until data was sent or received.
- Non-blocking Send and Receive:
  - returns from the call immediately.
  - returns a **Status** object that can be used to
    - get information if the data has arrived already.
    - get the data.
    - wait/block for the data.
- Blocking and non-blocking send's and receive's can be mixed on both sides.

Blocking Send and Receive

Communicator methods:

- **void send(Object buf, int count, Datatype type, int dest, int tag)**
- **Status recv(Object buf, int count, Datatype type, int source, int tag)**
- buffers for different types from **java.nio**
  - **MPI.new<Type>Buffer(int length)**: create a buffer
- **Datatype**: **MPI.INT**, **MPI.DOUBLE**, etc.
- dest/source: ID/rank of the destination/source process.
- tag: ID to distinguish different messages.
- both may throw an **MPIException**

Computing Pi / Sequential

```
1  public class Pi {
2      public static void main(String[] args) {
3          long N = 100000000;
4
5          long N_circle = 0;
6          for (long i = 0; i < N; ++i) {
7              double x = Math.random(), y = Math.random();
8              if (x*x+y*y <= 1)
9                  ++N_circle;
10         }
11         double pi = N_circle * 4.0 / N;
12         System.out.println("pi ~ " + pi);
13     }
14 }
```

Computing Pi / Parallel

```
1  import mpi.*;
2  public class Pi_MPI {
3      public static void main(String[] args)
4          throws MPIException {
5          MPI.Init(args);
6          int worker = MPI.COMM_WORLD.getRank(),
7              num_workers = MPI.COMM_WORLD.getSize();
8
9          long N = 100000000;
10         long N_worker = N / num_workers;
11
12         long N_circle = 0;
13         for (long i = 0; i < N_worker; ++i) {
14             double x = Math.random(), y = Math.random();
15             if (x*x+y*y <= 1)
16                 ++N_circle;
17         }
18
19         Comm comm = MPI.COMM_WORLD;
20         if (worker != 0) {
21             long[] data = { N_circle };
22             comm.send(data, 1, MPI.LONG, 0, 0);
23         } else {
24             long[] data = new long[1];
25             for (int w = 1; w < num_workers; ++w) {
26                 Status status = comm.recv(data, 1, MPI.LONG, MPI.ANY_SOURCE, 0);
27                 N_circle += data[0];
28             }
29         }
31         if (worker == 0) {
32             double pi = N_circle * 4.0
33                 / (N_worker * num_workers);
34             System.out.println("pi ~ " + pi);
35         }
36
37         MPI.Finalize();
38     }
39 }
```

Computing Pi / Parallel

```
1  import mpi.*;
2  public class Pi_MPI {
3      public static void main(String[] args)
4          throws MPIException {
5          MPI.Init(args);
6          int worker = MPI.COMM_WORLD.getRank(),
7              num_workers = MPI.COMM_WORLD.getSize();
8
9          long N = 100000000;
10         long N_worker = N / num_workers;
11
12         long N_circle = 0;
13         for (long i = 0; i < N_worker; ++i) {
14             double x = Math.random(), y = Math.random();
15             if (x*x+y*y <= 1)
16                 ++N_circle;
17         }
18
19         Comm comm = MPI.COMM_WORLD;
20         if (worker != 0) {
21             long[] data = { N_circle };
22             comm.send(data, 1, MPI.LONG, 0, 0);
23         } else {
24             long[] data = new long[1];
25             for (int w = 1; w < num_workers; ++w) {
26                 Status status = comm.recv(data, 1, MPI.LONG, MPI.ANY_SOURCE, 0);
27                 N_circle += data[0];
28             }
29         }
30
31         if (worker == 0) {
32             double pi = N_circle * 4.0
33                 / (N_worker * num_workers);
34             System.out.println("pi␣~␣" + pi);
35         }
36
37         MPI.Finalize();
38     }
39 }
```

implementation	runtime [s]
sequential	3.915
parallel (using 4 cores)	1.429

Non-blocking Communication

Examples:

- **Request iSend(Buffer buf, int count, Datatype type, int dest, int tag)**
- **Request iRecv(Buffer buf, int count, Datatype type, int source, int tag)**
- **Request** allows to inspect progress on the operation.
  - **testStatus()** tests if operation has been completed
    - returns **null**, if not,
      and a **Status** object, if so.
  - **waitStatus()** waits until operation has been completed
    - returns a **Status** object

Blocking Communication / Example

```
1  import mpi.*;
2
3  public class BlockingComm {
4      public static void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = MPI.COMM_WORLD.getRank(),
8              num_workers = MPI.COMM_WORLD.getSize();
9
10         for (int i = 0; i < 10; ++i) {
11             int dur = (int)Math.round(Math.random()*1000);
12             System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣begin");
13             try {
14                 Thread.sleep(dur);
15             } catch (InterruptedException ex) { break; }
16             System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣end");
17
18             if (worker != 0) {
19                 long[] data = { dur };
20                 comm.send(data, 1, MPI.LONG, 0, 0);
21             } else {
22                 long[] data = new long[1];
23                 long totaldur = dur;
24                 for (int w = 1; w < num_workers; ++w) {
25                     Status status = comm.recv(data, 1, MPI.LONG, MPI.ANY_SOURCE, 0);
26                     int count = status.getCount(MPI.LONG);
27                     totaldur += data[0];
28                 }
29                 System.out.println("--␣total␣round␣" + i + ":␣" + totaldur + "␣ms");
30             }
31         }
```

Blocking Communication / Example

```
1  import mpi.*;
2
3  public class BlockingComm {
4      public static void main(String[] args) throws MPIExceptio
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = MPI.COMM_WORLD.getRank(),
8              num_workers = MPI.COMM_WORLD.getSize();
9
10         for (int i = 0; i < 10; ++i) {
11             int dur = (int)Math.round(Math.random()*1000);
12             System.out.println("worker␣" + worker + ",␣round␣"
13             try {
14                 Thread.sleep(dur);
15             } catch (InterruptedException ex) { break; }
16             System.out.println("worker␣" + worker + ",␣round␣"
17
18             if (worker != 0) {
19                 long[] data = { dur };
20                 comm.send(data, 1, MPI.LONG, 0, 0);
21             } else {
22                 long[] data = new long[1];
23                 long totaldur = dur;
24                 for (int w = 1; w < num_workers; ++w) {
25                     Status status = comm.recv(data, 1, MPI.LONG
26                     int count = status.getCount(MPI.LONG);
27                     totaldur += data[0];
28                 }
29                 System.out.println("--␣total␣round␣" + i + ":␣"
30             }
31         }
```

Output

```
worker 1, round 0: 343 ms begin
worker 2, round 0: 487 ms begin
worker 0, round 0: 664 ms begin
worker 3, round 0: 281 ms begin
worker 3, round 0: 281 ms end
worker 3, round 1: 708 ms begin
worker 1, round 0: 343 ms end
worker 1, round 1: 621 ms begin
worker 2, round 0: 487 ms end
worker 2, round 1: 137 ms begin
worker 2, round 1: 137 ms end
worker 2, round 2: 609 ms begin
worker 0, round 0: 664 ms end
- total round 0: 1775 ms
worker 0, round 1: 242 ms begin
worker 0, round 1: 242 ms end
worker 1, round 1: 621 ms end
worker 1, round 2: 794 ms begin
worker 3, round 1: 708 ms end
worker 3, round 2: 342 ms begin
- total round 1: 1708 ms
worker 0, round 2: 908 ms begin
.
.
.
```

Non-blocking Communication / Example

```
1  import mpi.*;
2  import java.nio.LongBuffer;
3
4  public class NonblockingComm {
5      public static void main(String[] args)
6          throws MPIException {
7          MPI.Init(args);
8          Comm comm = MPI.COMM_WORLD;
9          int worker = MPI.COMM_WORLD.getRank(),
10             num_workers = MPI.COMM_WORLD.getSize();
11         LongBuffer data = MPI.newLongBuffer(1);
12         Request req = null;
13         if (worker == 0)
14             req = comm.iRecv(data, 1, MPI.LONG, MPI.ANY_SOURCE, 0);
15
16         for (int i = 0; i < 10; ++i) {
17             int dur = (int)Math.round(Math.random()*1000);
18             System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣begin");
19             try {
20                 Thread.sleep(dur);
21             } catch (InterruptedException ex) { break; }
22             System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣end");
23
24             if (worker != 0) {
25                 data.put(0, dur);
26                 comm.send(data, 1, MPI.LONG, 0, 0);
27             } else {
28                 long totaldur = dur;
29                 Status status = req.testStatus();
30                 while (status != null) {
31                     totaldur += data.get(0);
32                     req = comm.iRecv(data, 1,
33                                      MPI.LONG,
34                                      MPI.ANY_SOURCE,
35                                      0);
36                     status = req.testStatus();
37                 }
38                 System.out.println("--␣total␣0-round
39                                    + i + ":␣" + totald
40             }
41         }
42         MPI.Finalize();
43     }
44 }
```

Non-blocking Communication / Example

```
1  import mpi.*;
2  import java.nio.LongBuffer;
3
4  public class NonblockingComm {
5      public static void main(String[] args)
6          throws MPIException {
7          MPI.Init(args);
8          Comm comm = MPI.COMM_WORLD;
9          int worker = MPI.COMM_WORLD.getRank(),
10             num_workers = MPI.COMM_WORLD.getSize();
11         LongBuffer data = MPI.newLongBuffer(1);
12         Request req = null;
13         if (worker == 0)
14             req = comm.iRecv(data, 1, MPI.LONG, MPI.ANY_SOURCE
15
16         for (int i = 0; i < 10; ++i) {
17             int dur = (int)Math.round(Math.random()*1000);
18             System.out.println("worker␣" + worker + ",␣round␣"
19             try {
20                 Thread.sleep(dur);
21             } catch (InterruptedException ex) { break; }
22             System.out.println("worker␣" + worker + ",␣round␣"
23
24             if (worker != 0) {
25                 data.put(0, dur);
26                 comm.send(data, 1, MPI.LONG, 0, 0);
27             } else {
28                 long totaldur = dur;
29                 Status status = req.testStatus();
30                 while (status != null) {
31                     totaldur += data.get(0);
32
33
34    URCE,
35
36    ;
37
38    ␣0-round
39    + totalc
40
41
42
43
44
```

Output

```
worker 0, round 0: 509 ms begin
worker 2, round 0: 117 ms begin
worker 3, round 0: 188 ms begin
worker 1, round 0: 821 ms begin
worker 2, round 0: 117 ms end
worker 2, round 1: 633 ms begin
worker 3, round 0: 188 ms end
worker 3, round 1: 693 ms begin
worker 0, round 0: 509 ms end
- total 0-round 0: 814 ms
worker 0, round 1: 660 ms begin
worker 2, round 1: 633 ms end
worker 2, round 2: 559 ms begin
worker 1, round 0: 821 ms end
worker 1, round 1: 285 ms begin
worker 3, round 1: 693 ms end
worker 3, round 2: 24 ms begin
worker 3, round 2: 24 ms end
worker 3, round 3: 447 ms begin
worker 1, round 1: 285 ms end
worker 1, round 2: 652 ms begin
worker 0, round 1: 660 ms end
- total 0-round 1: 3116 ms
worker 0, round 2: 805 ms begin
.
.
.
```

Outline

1. MPI Basics

2. Point to Point Communication

3. Collective Communication

4. One-sided Communication

Collective Communication

Communicator methods:

- send buffer to all processes.
- **void bcast(Object buf, int count, Datatype type, int root)**
- root: ID/rank of the sending process.
- **bcast** does the right thing for all processes:
  - for root: send the local buffer buf.
  - for all others: receive into the local buffer buf.

  There is no extra Receive operation required (or allowed).

Broadcast / Example

```
1  import mpi.*;
2
3  public class ExBroadcast {
4      public static void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = MPI.COMM_WORLD.getRank(),
8              num_workers = MPI.COMM_WORLD.getSize();
9          int msg_len = 6;
10         char[] msg = new char[msg_len];
11
12         if (worker == 0)
13             msg = new char[] { 'H', 'e', 'l', 'l', 'o', '!'};
14         comm.bcast(msg, msg_len, MPI.CHAR, 0);
15         System.out.println("@" + worker + ": msg = " + new String(msg));
16
17         MPI.Finalize();
18     }
19 }
```

Collective Communication

Communicator methods:

- aggregate buffers from all processes.
- **reduce(Object sendbuf, Object recvbuf, int count, Datatype type, Op op, int root)**
- **reduce(Object buf, int count, Datatype type, Op op, int root)**
- **reduce** acts differently at different processes:
  - for non-root: send buffer sendbuf to root.
  - for root: aggregate received buffers into recvbuf using operation op
    - **MPI.SUM**, **MPI.PROD**: sum/product of values.
    - **MPI.MAX**, **MIN**: maximum/minimum value.
    - **MPI.MAXLOC**, **MINLOC**: argmax, argmin.
    - also user defined functions

Reduce / Example

```
1  import mpi.*;
2
3  public class Pi_MPI2 {
4      public static void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          int worker = MPI.COMM_WORLD.getRank(),
7              num_workers = MPI.COMM_WORLD.getSize();
8
9          long N = 100000000;
10         long N_worker = N / num_workers;
11
12         long N_circle = 0;
13         for (long i = 0; i < N_worker; ++i) {
14             double x = Math.random(), y = Math.random();
15             if (x*x+y*y <= 1)
16                 ++N_circle;
17         }
18
19         Comm comm = MPI.COMM_WORLD;
20         long[] data = { N_circle };
21         comm.reduce(data, 1, MPI.LONG, MPI.SUM, 0);
22
23         if (worker == 0) {
24             N_circle = data[0];
25             double pi = N_circle * 4.0 / (N_worker * num_workers);
26             System.out.println("pi ~ " + pi);
27         }
28
29         MPI.Finalize();
30     }
31 }
```

Example: Nearest Neighbor

Examples:

- Search nearest neighbor
  - stop to consider a candidate once its **partial distance** (computed on first k attributes) exceeds minimum distance so far.
  - special case: search best match for edit distance
  - here we stay with the simpler Euclidean distance

Nearest Neighbor / Sequential

```
1  public class NearestNeighbor {
2      public static void main(String[] args) {
3          int N = 1000000, M = 100;
4          double[][] data = new double[N][M];
5          for (int n = 0; n < N; ++n)
6              for (int m = 0; m < M; ++m)
7                  data[n][m] = 2*Math.random() - 1;
8          int num_queries = 100;
9
10         double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
11         int[] nn = new int[num_queries];
12         for (int n1 = 0; n1 < num_queries; ++n1) {
13             for (int n2 = num_queries; n2 < N; ++n2) {
14                 double dist = 0;
15                 for (int m = 0; m < M; ++m)
16                     dist += (data[n1][m] - data[n2][m]) * (data[n1][m] - data[n2][m]);
17                 if (dist < dist_min) {
18                     dist_min = dist;
19                     n_min = n2;
20                 }
21             }
22             nn[n1] = n_min;
23         }
24         System.out.println("done");
25     }
26 }
```

Nearest Neighbor / Sequential with Partial Distances

```
1  public class NearestNeighbor_PD {
2      public static void main(String[] args) {
3          int N = 1000000, M = 100;
4          double[][] data = new double[N][M];
5          for (int n = 0; n < N; ++n)
6              for (int m = 0; m < M; ++m)
7                  data[n][m] = 2*Math.random() - 1;
8          int num_queries = 100;
9          int delta_M = (int) Math.ceil(M/10);
10
11         double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
12         int[] nn = new int[num_queries];
13         for (int n1 = 0; n1 < num_queries; ++n1) {
14             for (int n2 = num_queries; n2 < N; ++n2) {
15                 double dist = 0;
16                 for (int m0 = 0; m0 < M; m0 += delta_M) {
17                     int m1 = Math.min(M, m0 + delta_M);
18                     for (int m = m0; m < m1; ++m)
19                         dist += (data[n1][m] - data[n2][m]) * (data[n1][m] - data[n2][m]);
20                     if (dist > dist_min)
21                         break;
22                 }
23                 if (dist < dist_min) {
24                     dist_min = dist;
25                     n_min = n2;
26                 }
27             }
28             nn[n1] = n_min;
29         }
30         System.out.println("done");
31     }
32 }
```

Nearest Neighbor / Parallel

```
1  import mpi.*;
2  public class NearestNeighbor_PD_par {
3      public static void main(String[] args)
4          throws MPIException {
5          MPI.Init(args);
6          Comm comm = MPI.COMM_WORLD;
7          int worker = MPI.COMM_WORLD.getRank(),
8              num_workers = MPI.COMM_WORLD.getSize();
9
10         int N = 1000000, M = 100;
11         int N_worker = (int) Math.ceil(N/num_workers);
12         double[][] data = new double[N_worker][M];
13         for (int n = 0; n < N_worker; ++n)
14             for (int m = 0; m < M; ++m)
15                 data[n][m] = 2*Math.random() - 1;
16         int num_queries = 100;
17         int N_start = worker == 0? num_queries : 0;
18         int delta_M = (int) Math.ceil(M/10);
19         double[] query = new double[M];
20
21         double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
22         int[] nn = new int[num_queries];
23         for (int n1 = 0; n1 < num_queries; ++n1) {
24             if (worker == 0)
25                 System.arraycopy(data[n1], 0, query, 0, M);
26             comm.bcast(query, M, MPI.DOUBLE, 0);
27
28             for (int n2 = N_start; n2 < N_worker; ++n2) {
29                 double dist = 0;
30                 for (int m0 = 0; m0 < M; m0 += delta_M) {
```

Nearest Neighbor / Parallel

```
31                 int m1 = Math.min(M, m0 + delta_M);
32                 for (int m = m0; m < m1; ++m)
33                     dist += (query[m] - data[n2][m]) * (query[m] - data[n2][m]);
34                 if (dist > dist_min)
35                     break;
36             }
37             if (dist < dist_min) {
38                 dist_min = dist;
39                 n_min = n2;
40             }
41         }
42         if (worker != 0) {
43             double[] msg = { n_min, dist_min };
44             comm.send(msg, 2, MPI.DOUBLE, 0, 0);
45         } else {
46             double[] msg = new double[2];
47             for (int w = 1; w < num_workers; ++w) {
48                 comm.recv(msg, 2, MPI.DOUBLE, MPI.ANY_SOURCE, 0);
49                 double dist_min_w = msg[1]; int n_min_w = (int) msg[0];
50                 if (dist_min_w < dist_min) {
51                     dist_min = dist_min_w;
52                     n_min = n_min_w;
53                 }
54             }
55         }
56         nn[n1] = n_min;
57     }
58     System.out.println("done");
59     MPI.Finalize();
60   }
61 }
```

Nearest Neighbor / Parallel

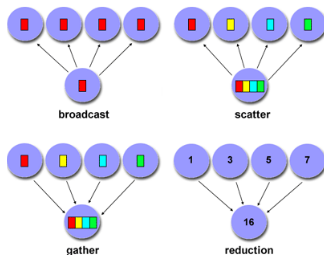
```
31                 int m1 = Math.min(M, m0 + delta_M);
32                 for (int m = m0; m < m1; ++m)
33                     dist += (query[m] - data[n2][m]) * (query[m] - data[n2][m]);
34                 if (dist > dist_min)
35                     break;
36             }
37             if (dist < dist_min) {
38                 dist_min = dist;
39                 n_min = n2;
40             }
41         }
42         if (worker != 0) {
43             double[] msg = { n_min, dist_min };
44             comm.send(msg, 2, MPI.DOUBLE, 0, 0);
45         } else {
46             double[] msg = new double[2];
47             for (int w = 1; w < num_workers; ++w) {
48                 comm.recv(msg, 2, MPI.DOUBLE, MPI.ANY_SOURCE, 0);
49                 double dist_min_w = msg[1]; int n_min_w = (int) msg[0];
50                 if (dist_min_w < dist_min) {
51                     dist_min = dist_min_w;
52                     n_min = n_min_w;
53                 }
54             }
55         }
56         nn[n1] = n_min;
57     }
58     System.out.println("done");
59     MPI.Finalize();
60   }
61 }
```

Runtimes

implementation	runtime [s]
sequential	18.1
sequential + PD	12.7
parallel	7.1
(using 4 cores)	

More Collective Communication Operations

- **Scatter**:
  - Distribute parts of a buffer to different processes.
- **Gather**:
  - Collect parts of a buffer from different processes.



https://computing.llnl.gov/tutorials/mpi/

Outline

1. MPI Basics

2. Point to Point Communication

3. Collective Communication

4. One-sided Communication

One-sided Communication (1/2)

- exchange data not with paired Send/Receive operations, but with unpaired/one-sided Get/Put operations.
- requires shared data to be explicitly marked (**window**)
- to create, use constructor
  Win(Buffer base, int size, int dispUnit, Info info, Comm comm)
  - dispUnit: (usually 1)
  - **Info**: various window settings (often **MPI.INFO_NULL**)
  - **Comm**: the communicator used.

One-sided Communication (2/2)

Win objects:

- **put(Buffer origin, int orgCount, Datatype orgType, int targetRank, int targetDisp, int targetCount, Datatype targetType)**
- **get(Buffer origin, int orgCount, Datatype orgType, int targetRank, int targetDisp, int targetCount, Datatype targetType)**
- **put** transfers data from local buffer origin to the shared buffer at process targetRank.
- **get** transfers data from shared buffer at process targetRank to local buffer origin.
- targetDisp: offset in target buffer.
- **fence(int assertion)** starts and ends fenced synchronization, i.e., a phase where data is exchanged between processes.
- **free()**: release the shared data window.

One-sided Put / Example

```
1  import mpi.*;
2  import java.nio.IntBuffer;
3
4  public class ExOnesided3 {
5      public static void main(String[] args) throws MPIException {
6          MPI.Init(args);
7          int worker = MPI.COMM_WORLD.getRank(),
8              num_workers = MPI.COMM_WORLD.getSize();
9          Comm comm = MPI.COMM_WORLD;
10         int len = num_workers;
11         int len_worker = worker == 0? len : 0;
12         IntBuffer buf = MPI.newIntBuffer(len_worker);
13         Win win = new Win(buf, len_worker, 1, MPI.INFO_NULL, comm);
14
15         win.fence(0);
16         if (worker == 0)
17             buf.put(0, 100);
18         else {
19             IntBuffer data = MPI.newIntBuffer(1);
20             data.put(0, 100 + 2 * worker);
21             win.put(data, 1, MPI.INT, 0, worker, 1, MPI.INT);
22         }
23         win.fence(0);
24
25         if (worker == 0)
26             for (int w = 0; w < num_workers; ++w)
27                 System.out.println("buf[" + w + "] = " + buf.get(w));
28
29         win.free();
30         MPI.Finalize();
31     }
```

One-sided Get / Example

```
import mpi.*;
import java.nio.IntBuffer;

public class ExOnesided_Get {
    public static void main(String[] args) throws MPIException {
        MPI.Init(args);
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();
        Comm comm = MPI.COMM_WORLD;
        int len = num_workers;
        int len_worker = worker == 0? len : 0;
        IntBuffer buf = MPI.newIntBuffer(len_worker);
        Win win = new Win(buf, len_worker, 1, MPI.INFO_NULL, comm);

        win.fence(0);
        if (worker == 0)
            for (int i = 0; i < len; ++i)
                buf.put(i, 100 + 2*i);
        win.fence(0);

        IntBuffer data = MPI.newIntBuffer(len);
        if (worker != 0)
            win.get(data, len, MPI.INT, 0, 0, len, MPI.INT);
        win.fence(0);

        if (worker != 0) {
            String s = "@" + worker + ":␣";
            for (int i = 0; i < len; ++i)
                s += "" + data.get(i) + "␣";
            System.out.println(s);
        }
```

Further MPI capabilities

- Datatypes
- Process creation
- Shared memory
  - i.e., interactions between processes and threads
- Groups and contexts
- Process topologies
- Parallel I/O

Summary (1/3)

- The **Message Passing Interface (MPI)** allows processes to
  - exchange data and
  - synchronize,

  also processes running distributed.

- The most simple way to execute a distributed program is to start a program in several copies in parallel
  - as different processes
  - possibly distributed, on different machines
    - **submit host**: machine the program have been submitted on.
    - **compute hosts**: machines the program is actually running.
  - The MPI runtime sets up a **communicator** that enables processes to send messages to each other.
    - the process ID (called **rank**) is used to assign different roles to different processes
      (e.g., process 0 often is a "master").

Summary (2/3)

- The most simple communication is between two processes (**point to point**, paired).
  - a **message** is a buffer of given element type and size.
  - one process **sends** such a buffer
  - another process **receives** such a buffer
  - **blocking communication**:
    - both processes wait until communication is completed.
  - **non-blocking communication**:
    - sending/receiving is done in parallel to process execution.
    - a request object allows a process to inspect the state and result of such a non-blocking communication operation.

Summary (3/3)

- **Collective communication** allows more complex communication schemes to be implemented and executed more efficiently.
  - one-to-all, all-to-one, all-to-all
  - same data to/from all: **broadcast** and **reduce**
  - different data to/from all: **scatter** and **gather**
- **One-sided communication** allows to access remote data without cooperative synchronization by the remote process.
  - shared data has to be wrapped into a **window**.
  - **get**/**put** can access remote data.
  - synchronization in the most simple case done by defining exchange epochs (**fence**).