# Big Data Analytics

Lars Schmidt-Thieme

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

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

# Syllabus

| | | |
|---|---|---|
| Tue. 10.4. | (1) | 0. Introduction |
| | | **A. Parallel Computing** |
| Tue. 17.4. | (2) | A.1 Threads |
| Tue. 24.4. | (3) | A.2 Message Passing Interface (MPI) |
| Tue. 1.5. | (4) | A.3 Graphical Processing Units (GPUs) |
| | | **B. Distributed Storage** |
| Tue. 8.5. | (5) | B.1 Distributed File Systems |
| Tue. 15.5. | (6) | B.2 Partioning of Relational Databases |
| Tue. 22.5. | — | — Pentecoste Break — |
| Tue. 29.5. | (7) | B.3 NoSQL Databases |
| | | **C. Distributed Computing Environments** |
| Tue. 5.6. | (8) | C.1 Map-Reduce |
| Tue. 12.6. | (9) | C.2 Resilient Distributed Datasets (Spark) |
| Tue. 19.6. | (10) | C.3 Computational Graphs (TensorFlow) |
| | | **D. Distributed Machine Learning Algorithms** |
| Tue. 26.6. | (11) | D.1 Distributed Stochastic Gradient Descent |
| Tue. 3.7. | (12) | D.2 Distributed Matrix Factorization |
| Tue. 10.7. | (13) | Questions and Answers |

# Outline

# Outline

# 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

```
mpijavac Hello.java
```

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

```
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.

# Hello World MPI

```
#!/usr/bin/env python3
from mpi4py import MPI

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

print('Hello␣world␣from␣worker␣{}␣of␣{}'.format(worker, num_workers))
```

# Hello World MPI

**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
```

```
#!/usr/bin/env python3
from mpi4py import MPI

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

print('Hello␣world␣from␣worker␣{}␣of␣{}'.format(worker, num_workers))
```

# 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

```
#!/usr/bin/env python3
from random import random

N = 10000000
N_circle = 0
for i in range(0,N):
    x = random(); y = random()
    if x*x + y*y <= 1:
        N_circle += 1
pi = N_circle * 4.0 / N
print('pi ~ {}'.format(pi))
```

# Computing Pi / Parallel

```
#!/usr/bin/env python3
from mpi4py import MPI
from random import random

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

N = 10000000
N_worker = round(N / num_workers)
N_circle = 0
for i in range(N_worker):
    x = random(); y = random()
    if x*x + y*y <= 1:
        N_circle += 1

if worker != 0:
    data = {'N_circle': N_circle}
    comm.send(data, dest=0)
else:
    for w in range(1, num_workers):
        data = comm.recv() # source=0)
        N_circle += data['N_circle']
    pi = N_circle * 4.0 / (num_workers * N_worker)
    print('pi ~ {}'.format(pi))
```

# Computing Pi / Parallel

```
#!/usr/bin/env python3
from mpi4py import MPI
from random import random

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

N = 10000000
N_worker = round(N / num_workers)
N_circle = 0
for i in range(N_worker):
    x = random(); y = random()
    if x*x + y*y <= 1:
        N_circle += 1

if worker != 0:
    data = {'N_circle': N_circle}
    comm.send(data, dest=0)
else:
    for w in range(1, num_workers):
        data = comm.recv() # source=0)
        N_circle += data['N_circle']
    pi = N_circle * 4.0 / (num_workers * N_worker)
    print('pi ~ {}'.format(pi))
```

| implementation | runtime [s] |
|---|---|
| sequential | 6.363 |
| parallel | 2.011 |
| (using 4 cores) | |

# 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

```python
#!/usr/bin/env python3
from mpi4py import MPI
from random import randrange
from time import sleep

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

for i in range(10):
    dur = randrange(1000)
    print('worker {} , round {}: {} ms begin'.format(worker, i, dur))
    sleep(dur/1000)
    print('worker {} , round {}: {} ms end'.format(worker, i, dur))

    if worker != 0:
        data = { 'dur': dur }
        comm.send(data, dest=0)
    else:
        totaldur = dur
        for w in range(1, num_workers):
            data = comm.recv()
            # int count = status.getCount(MPI.LONG);
            totaldur += data['dur']
        print('-- total round {}: {} ms'.format(i, totaldur))
```

# Blocking Communication / Example

```
#!/usr/bin/env python3
from mpi4py import MPI
from random import randrange
from time import sleep

comm = MPI.COMM_WORLD
worker = comm.Get_rank()
num_workers = comm.Get_size()

for i in range(10):
    dur = randrange(1000)
    print('worker␣{}␣,␣round␣{}:␣{}␣ms␣begin'.format(worker,
    sleep(dur/1000)
    print('worker␣{}␣,␣round␣{}:␣{}␣ms␣end'.format(worker, i,

    if worker != 0:
        data = { 'dur': dur }
        comm.send(data, dest=0)
    else:
        totaldur = dur
        for w in range(1, num_workers):
            data = comm.recv()
            # int count = status.getCount(MPI.LONG);
            totaldur += data['dur']
        print('--␣total␣round␣{}:␣{}␣ms'.format(i, totaldur))
```

## 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

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

public class NonblockingComm {
    public static void main(String[] args)
        throws MPIException {
        MPI.Init(args);
        Comm comm = MPI.COMM_WORLD;
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();
        LongBuffer data = MPI.newLongBuffer(1);
        Request req = null;
        if (worker == 0)
            req = comm.iRecv(data, 1, MPI.LONG, MPI.ANY_SOURCE, 0);

        for (int i = 0; i < 10; ++i) {
            int dur = (int)Math.round(Math.random()*1000);
            System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣begin");
            try {
                Thread.sleep(dur);
            } catch (InterruptedException ex) { break; }
            System.out.println("worker␣" + worker + ",␣round␣" + i + ":␣" + dur + "␣ms␣end");

            if (worker != 0) {
                data.put(0, dur);
                comm.send(data, 1, MPI.LONG, 0, 0);
            } else {
                long totaldur = dur;
                Status status = req.testStatus();
                while (status != null) {
                    totaldur += data.get(0);
                    req = comm.iRecv(data, 1,
                                     MPI.LONG,
                                     MPI.ANY_SOURCE,
                                     0);
                    status = req.testStatus();
                }
                System.out.println("--␣total␣0-round
                                   + i + ":␣" + totald
            }
        }
        MPI.Finalize();
    }
}
```

# Non-blocking Communication / Example

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

public class NonblockingComm {
    public static void main(String[] args)
        throws MPIException {
        MPI.Init(args);
        Comm comm = MPI.COMM_WORLD;
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();
        LongBuffer data = MPI.newLongBuffer(1);
        Request req = null;
        if (worker == 0)
            req = comm.iRecv(data, 1, MPI.LONG, MPI.ANY_SOURCE

        for (int i = 0; i < 10; ++i) {
            int dur = (int)Math.round(Math.random()*1000);
            System.out.println("worker␣" + worker + ",␣round␣"
            try {
                Thread.sleep(dur);
            } catch (InterruptedException ex) { break; }
            System.out.println("worker␣" + worker + ",␣round␣"

            if (worker != 0) {
                data.put(0, dur);
                comm.send(data, 1, MPI.LONG, 0, 0);
            } else {
                long totaldur = dur;
                Status status = req.testStatus();
                while (status != null) {
                    totaldur += data.get(0);
```

```
URCE,
;
␣0-round
+ totald
```

## 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

# 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

```
import mpi.*;

public class ExBroadcast {
    public static void main(String[] args) throws MPIException {
        MPI.Init(args);
        Comm comm = MPI.COMM_WORLD;
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();
        int msg_len = 6;
        char[] msg = new char[msg_len];

        if (worker == 0)
            msg = new char[] { 'H', 'e', 'l', 'l', 'o', '!'};
        comm.bcast(msg, msg_len, MPI.CHAR, 0);
        System.out.println("@" + worker + ": msg = " + new String(msg));

        MPI.Finalize();
    }
}
```

# 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

```
import mpi.*;

public class Pi_MPI2 {
    public static void main(String[] args) throws MPIException {
        MPI.Init(args);
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();

        long N = 100000000;
        long N_worker = N / num_workers;

        long N_circle = 0;
        for (long i = 0; i < N_worker; ++i) {
            double x = Math.random(), y = Math.random();
            if (x*x+y*y <= 1)
                ++N_circle;
        }

        Comm comm = MPI.COMM_WORLD;
        long[] data = { N_circle };
        comm.reduce(data, 1, MPI.LONG, MPI.SUM, 0);

        if (worker == 0) {
            N_circle = data[0];
            double pi = N_circle * 4.0 / (N_worker * num_workers);
            System.out.println("pi ~ " + pi);
        }

        MPI.Finalize();
    }
}
```

# 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

```
public class NearestNeighbor {
    public static void main(String[] args) {
        int N = 1000000, M = 100;
        double[][] data = new double[N][M];
        for (int n = 0; n < N; ++n)
            for (int m = 0; m < M; ++m)
                data[n][m] = 2*Math.random() - 1;
        int num_queries = 100;

        double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
        int[] nn = new int[num_queries];
        for (int n1 = 0; n1 < num_queries; ++n1) {
            for (int n2 = num_queries; n2 < N; ++n2) {
                double dist = 0;
                for (int m = 0; m < M; ++m)
                    dist += (data[n1][m] - data[n2][m]) * (data[n1][m] - data[n2][m]);
                if (dist < dist_min) {
                    dist_min = dist;
                    n_min = n2;
                }
            }
            nn[n1] = n_min;
        }
        System.out.println("done");
    }
}
```

# Nearest Neighbor / Sequential with Partial Distances

```
public class NearestNeighbor_PD {
    public static void main(String[] args) {
        int N = 1000000, M = 100;
        double[][] data = new double[N][M];
        for (int n = 0; n < N; ++n)
            for (int m = 0; m < M; ++m)
                data[n][m] = 2*Math.random() - 1;
        int num_queries = 100;
        int delta_M = (int) Math.ceil(M/10);

        double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
        int[] nn = new int[num_queries];
        for (int n1 = 0; n1 < num_queries; ++n1) {
            for (int n2 = num_queries; n2 < N; ++n2) {
                double dist = 0;
                for (int m0 = 0; m0 < M; m0 += delta_M) {
                    int m1 = Math.min(M, m0 + delta_M);
                    for (int m = m0; m < m1; ++m)
                        dist += (data[n1][m] - data[n2][m]) * (data[n1][m] - data[n2][m]);
                    if (dist > dist_min)
                        break;
                }
                if (dist < dist_min) {
                    dist_min = dist;
                    n_min = n2;
                }
            }
            nn[n1] = n_min;
        }
        System.out.println("done");
    }
}
```

# Nearest Neighbor / Parallel

```
import mpi.*;
public class NearestNeighbor_PD_par {
    public static void main(String[] args)
        throws MPIException {
        MPI.Init(args);
        Comm comm = MPI.COMM_WORLD;
        int worker = MPI.COMM_WORLD.getRank(),
            num_workers = MPI.COMM_WORLD.getSize();

        int N = 1000000, M = 100;
        int N_worker = (int) Math.ceil(N/num_workers);
        double[][] data = new double[N_worker][M];
        for (int n = 0; n < N_worker; ++n)
            for (int m = 0; m < M; ++m)
                data[n][m] = 2*Math.random() - 1;
        int num_queries = 100;
        int N_start = worker == 0? num_queries : 0;
        int delta_M = (int) Math.ceil(M/10);
        double[] query = new double[M];

        double dist_min = Double.POSITIVE_INFINITY; int n_min = -1;
        int[] nn = new int[num_queries];
        for (int n1 = 0; n1 < num_queries; ++n1) {
            if (worker == 0)
                System.arraycopy(data[n1], 0, query, 0, M);
            comm.bcast(query, M, MPI.DOUBLE, 0);

            for (int n2 = N_start; n2 < N_worker; ++n2) {
                double dist = 0;
                for (int m0 = 0; m0 < M; m0 += delta_M) {
```

# Nearest Neighbor / Parallel

```
                    int m1 = Math.min(M, m0 + delta_M);
                    for (int m = m0; m < m1; ++m)
                        dist += (query[m] - data[n2][m]) * (query[m] - data[n2][m]);
                    if (dist > dist_min)
                        break;
                }
                if (dist < dist_min) {
                    dist_min = dist;
                    n_min = n2;
                }
            }
            if (worker != 0) {
                double[] msg = { n_min, dist_min };
                comm.send(msg, 2, MPI.DOUBLE, 0, 0);
            } else {
                double[] msg = new double[2];
                for (int w = 1; w < num_workers; ++w) {
                    comm.recv(msg, 2, MPI.DOUBLE, MPI.ANY_SOURCE, 0);
                    double dist_min_w = msg[1]; int n_min_w = (int) msg[0];
                    if (dist_min_w < dist_min) {
                        dist_min = dist_min_w;
                        n_min = n_min_w;
                    }
                }
            }
            nn[n1] = n_min;
        }
        System.out.println("done");
        MPI.Finalize();
    }
}
```

# Nearest Neighbor / Parallel

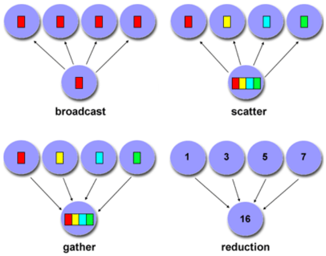
```
                        int m1 = Math.min(M, m0 + delta_M);
                        for (int m = m0; m < m1; ++m)
                            dist += (query[m] - data[n2][m]) * (query[m] - data[n2][m]);
                        if (dist > dist_min)
                            break;
                    }
                    if (dist < dist_min) {
                        dist_min = dist;
                        n_min = n2;
                    }
                }
                if (worker != 0) {
                    double[] msg = { n_min, dist_min };
                    comm.send(msg, 2, MPI.DOUBLE, 0, 0);
                } else {
                    double[] msg = new double[2];
                    for (int w = 1; w < num_workers; ++w) {
                        comm.recv(msg, 2, MPI.DOUBLE, MPI.ANY_SOURCE, 0);
                        double dist_min_w = msg[1]; int n_min_w = (int) msg[0];
                        if (dist_min_w < dist_min) {
                            dist_min = dist_min_w;
                            n_min = n_min_w;
                        }
                    }
                }
                nn[n1] = n_min;
            }
            System.out.println("done");
            MPI.Finalize();
        }
    }
```

**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

# 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

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

public class ExOnesided3 {
    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)
            buf.put(0, 100);
        else {
            IntBuffer data = MPI.newIntBuffer(1);
            data.put(0, 100 + 2 * worker);
            win.put(data, 1, MPI.INT, 0, worker, 1, MPI.INT);
        }
        win.fence(0);

        if (worker == 0)
            for (int w = 0; w < num_workers; ++w)
                System.out.println("buf[" + w + "] = " + buf.get(w));

        win.free();
        MPI.Finalize();
    }
```

# 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**).