Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 13: Message Passing
Spring Semester 2017

Assigned on: **23.05.2017** Due by: **30.05.2017**

Overview

This week's assignment is about message passing. Message passing enables parallel communication between independent processes that do not share memory, but where data is always explicitly transferred between processes. This assignment gives you a hands-on introduction on how to use message passing. In this exercise we will use MPI, the message passing interface library, which is the dominant message passing interface run on super computers with millions of cores. As not every student has a super computer and to remain within the Java language environment, we will be using in this exercise MPJ-Express, an implementation of MPI for Java.
Note: For performance evaluations MPI experiments should be run with high-performance MPI implementations (e.g., MVAPICH) available for C and Fortran.

Exercise 1 – Hello World

For this exercise you are asked to implement a simple "Hello World" program with MPJ Express (http://mpj-express.org/). MPJ express is already part of assignment13.zip, but to use it in eclipse a specific "Run Configuration" must be set up, where the VM arguments are set to "-jar libs/starter.jar -np 2" and the environment variable MPJ_HOME is set to "mpj".

The basic programming constructs you need for writing the "Hello world" program are:

```
public class MPI extends java.lang.Object {
        static Intracomm                COMM_WORLD
        static java.lang.String[]       Init(java.lang.String[] argv)
        static void                     Finalize()
}

public class public class Intracomm {
        public int                      Rank() throws MPIException
        public int                      Size() throws MPIException
}
```

Implement a new class `HelloWorld` which initializes an MPI object, retrieves the 'rank' and 'size' of the current process, and prints in each process:

```
Hi from <rank>
Running with size <size> processes
```

Run this program with 4 concurrent ranks. The expected output is:

```
Hi from <3>
Running with size <4> processes
Hi from <0>
Hi from <2>
Running with size <4> processes
Running with size <4> processes
Hi from <1>
Running with size <4> processes
```

Exercise 2 – Ping Pong

Implement a new MPI program `PingPong` that is expected to be run with two ranks. "Rank 0" initializes an integer data value to 'zero' and then sends this data value to process "Rank 1". "Rank 1" increments the received item by one and sends it back to "Rank 0". "Rank 0" again increments the data item and sends it back to "Rank 1". This process is implemented in a loop with `Iterations` iterations.

```
public class public class Intracomm {

        public void Send(java.lang.Object buf,
                         int offset,
                         int count,
                         Datatype datatype,
                         int dest,
                         int tag) throws MPIException

        public Status Recv(java.lang.Object buf,
                           int offset,
                           int count,
                           Datatype datatype,
                           int source,
                           int tag) throws MPIException
}
```

Run your program and measure (use `System.nanoTime`) how long a single "ping-pong" takes on your system? What is the minimal, maximal, median, and mean time of a "ping-pong"?

Run your program again and this time transfer instead a single integer an array of size 128, 1024, 1024^2. How does the size of the data affect the time of a ping-pong? Which property are you measuring with a "ping-pong"? Does the property change with the size of the data transferred?

Exercise 3 – Sieve of Eratosthenes

In this exercise, you distribute the computation of prime numbers using MPI. The Sieve of Eratosthenes computes prime numbers within a range [0, MaxNumber[as follows. First, an array is allocated which keeps track for each number if this number is possibly prime. Initially it is assumed all numbers might be prime (except of 0 and 1). The algorithm preserves the invariant that the first number in the array that is marked as possibly prime is indeed prime. In the initial state, this number if '2', which is the smallest prime number. To eliminate numbers that are not prime, we iterate over the list of numbers and cross out all multiples of the current prime number, then look for the next prime number (smallest number marked as possibly prime), and cross again all numbers out that are multiples of the current prime number. This process continues until a prime number that is larger than $sqrt(MaxNumber)$ is reached. In this exercise, we are only interested in how many prime numbers exist, not their actual values.

Parallelize the sieve of Erathosthenes by assigning each process a partition of the overall number array. Each process initializes its partition of the number array and marks numbers that are multiples of a given prime only within its partition of the number array. To obtain the number of prime numbers, each process counts the primes in its partition of the number array. These individual results are the *reduced* to a single value that is reported by "Rank 0".

The following functions might be useful:

```
public class public class Intracomm {

        public void Bcast(java.lang.Object buf,
                          int offset,
                          int count,
                          Datatype type,
                          int root) throws MPIException

        public void Reduce(java.lang.Object sendbuf,
                           int sendoffset,
                           java.lang.Object recvbuf,
                           int recvoffset,
                           int count,
                           Datatype datatype,
                           Op op,
                           int root) throws MPIException
}
```

Note: Assume $sqrt(MaxNumber)$ still belongs to "Rank 0". This allows Rank 0 to distribute the work to all other processes.

Measure the performance of sequential and parallel execution of your MPISieve implementation. How does performance change with the number of processors? What did you expect to see? Can you explain your results? What would you expect to see on a distributed memory super computer?

Bonus: Can you run your algorithm on Euler with 40 nodes? (You either need to install Java and MPJ on Euler or you need to port your algorithm to C and MPI.) What speedup do you measure?

Submission

In order for us to grade your exercises and give you feedback, you need to submit your code to the Subversion repository. You will find detailed instructions on how to install and set-up Eclipse for use with Subversion in Exercise 1.

Once you have completed the skeleton, commit it to SVN in a directory named `assignment13` by following the steps described below. The questions that require written answers should all be recorded in a single file named `report.pdf` and placed in the base directory of your project (i.e., in folder `assignment13`).

- **Check-in your project for the first time**
  - Right click your created project called **assignment13**.
  - In the menu go to **Team**, then click **Share Project**.
  - In the dialog that now appears, select **SVN** as a repository type, then click **Next**.
  - In case you have submitted Exercise 1, choose **Use existing repository location** and select the pre-defined URL in the dialog that should look like this

https://svn.inf.ethz.ch/svn/vechev/pprog17/students/NETHZ_USERNAME
Click Finish. Otherwise follow the steps in Exercise 1 to set-up a repository location.

- **Commit changes in your project**
  - Now that your project is connected with the SVN server, you need to make sure that every time you change your code or your report, at the end you submit it to the SVN server as well.
  - Right click your project called **assignment13**.
  - In the menu go to **Team**, then click **Commit**.
  - In the Comment field, enter a comment that summarizes your changes.
  - Then, click on **Ok**.