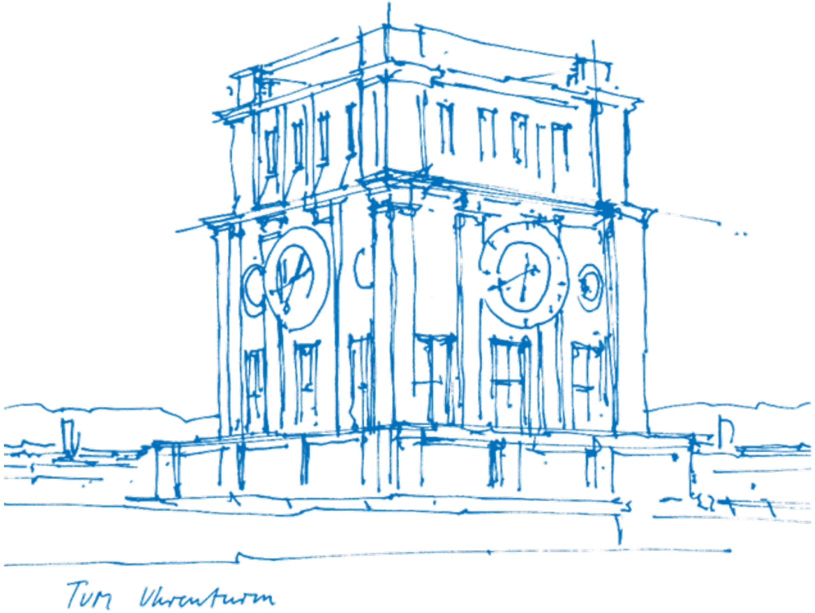
Parallel Programming Tutorial - Introduction to MPI

M.Sc. Amir Raoofy

Technical University of Munich

30. Mai 2018



Organizational notes

Organization

- Please evaluate the course between June 6th to 20th
  - You will get 15 minutes in lecture on June 18th to evaluate the course.
- We will have no tutorial next week. next tutorial is on Monday June 11th.
- The deadline for the first MPI assignment is June 11th.

Solution for Assignment 6

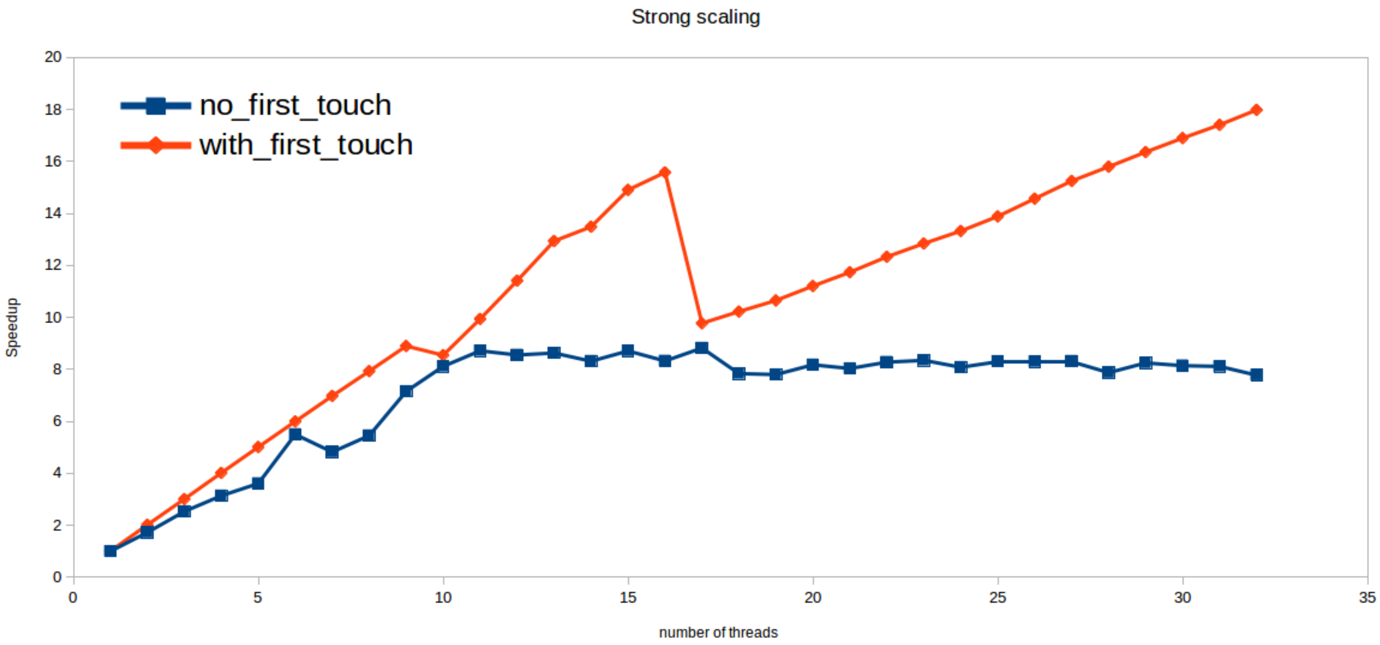
Solution for OpenMP, First touch

```
template<int SIZE>
inline void time_step(double a[SIZE + 2][SIZE + 2], double b[SIZE + 2][SIZE + 2], int n)
{
    if (n % 2 == 0)
    {
        #pragma omp parallel for schedule(static)
        for (int i = 1; i < SIZE + 1; i++)
            for (int j = 1; j < SIZE + 1; j++)
                b[i][j] = (a[i + 1][j] + a[i - 1][j] + a[i][j - 1] + a[i][j + 1]) / 4.0;
    }
    else
    {
        #pragma omp parallel for schedule(static)
        for (int i = 1; i < SIZE + 1; i++)
            for (int j = 1; j < SIZE + 1; j++)
                a[i][j] = (b[i + 1][j] + b[i - 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0;
    }
}
```

Solution for OpenMP, First touch (Cont.)

```
template<int SIZE>
inline void initialize(double a[SIZE + 2][SIZE + 2], double b[SIZE + 2][SIZE + 2])
{
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < SIZE + 2; i++)
        for (int j = 0; j < SIZE + 2; j++)
        {
            a[i][j] = 0.0;
            b[i][j] = 0.0;
        }
}
```

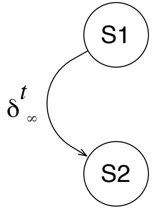
Solution for OpenMP, Scaling on the server



Loop Transformations, remaining parts

Loop Fusion I

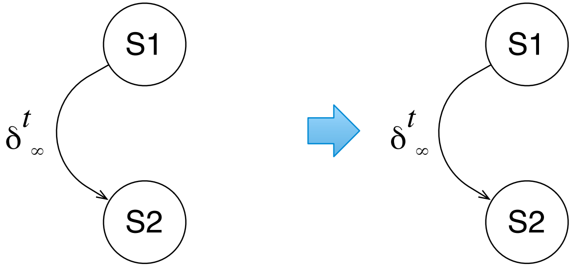
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion I

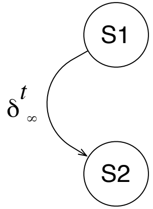
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion II - Fusion preventing Dependency

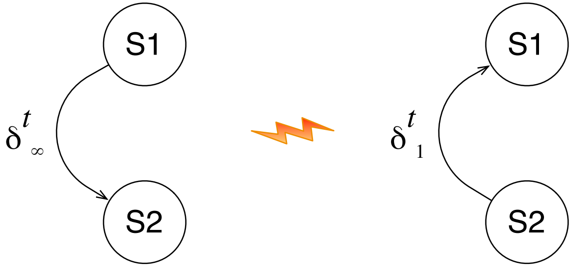
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i+1) + B(i)
}
```



Loop Fusion II - Fusion preventing Dependency

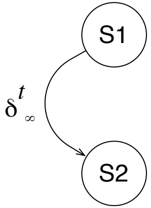
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i+1) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
    S2:   C(i) = A(i+1) + B(i)
}
```



Loop Fusion III - Parallelism inhibiting Dependency

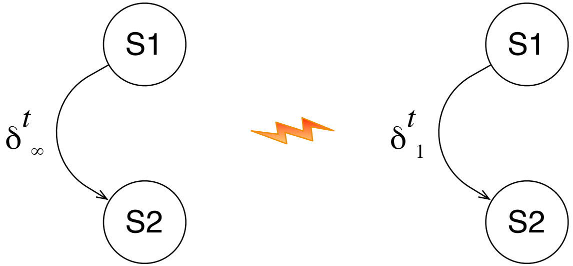
```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion III - Parallelism inhibiting Dependency

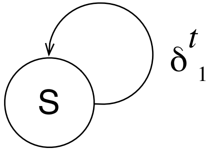
```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
    S2:   C(i)   = A(i) + B(i)
}
```



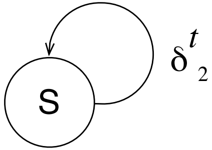
Loop Interchange

```
for (i=1; i<n; i++) {
    for(j=1; j<m; j++) {
        S:    A(i+1,j) = A(i,j) + B(i,j)
    }
}
```



⟺

```
for (j=1; j<m; j++) {
    for(i=1; i<n; i++) {
        S:    A(i+1,j) = A(i,j) + B(i,j)
    }
}
```



MPI: Message Passing Interface

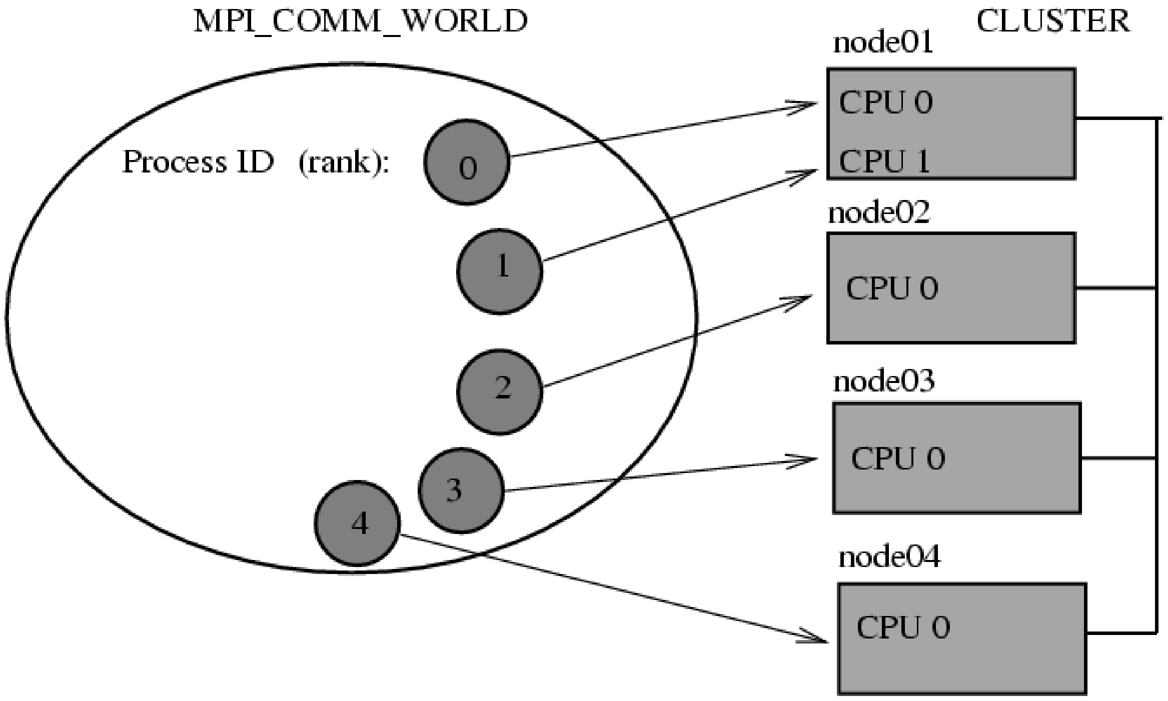
MPI: Message Passing Interface

- MPI is an API specification
- implemented as Library + Tools (compiler wrapper, documentation, deamon)
- most common implementations: Open MPI and MPICH
- Using MPI you can write applications on distributed memory (also shared memory systems).
- Communications is done by sending messages.
- Types of operations: point-to-point or collectives and one-sided
- SPMD programming model (Most common)
- Typically, a single Program(source), is started as (multiple) processes on local or remote machines and each processes works on local data.
- Each process in a communicator is identified by its rank (id)
- Work distribution can be done using the rank.
- All data is private. If data has be accessed by another process, it has to be sent to this process.

MPI: Message Passing Interface

- MPI runtime handles the startup of all processes and takes care about the enumeration of the processes (ranks).
- Distribution of processes to machines can be configured, but this is not part of the exercise. You will work on a shared memory machine, but there is no difference working on a remote machine, except for the performance issues.
- Debugging is difficult with MPI, even worse then OpenMP or Pthreads, because of multiple processes. It's, however, more deterministic then OpenMP or Pthreads, since you have do everything explicitly.
- This makes writing MPI applications time consuming.
- Debugging can be done by printf(). MPI takes care that everything is printed on your terminal. An alternative is attaching a debugger to the running processes.
- There are also commercial MPI debuggers (e.g., totalview) and plugin for Eclipse called Parallel Tools Platform (PTP)

MPI: Overview



MPI: Installation Ubuntu

- $ sudo apt-get install libcr-dev libopenmpi-dev openmpi-bin openmpi-doc
- OR
- $ sudo apt-get install libcr-dev mpich2 mpich2-doc

MPI: Hello world!

```c
#include <mpi.h>
#include <stdio.h>

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

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

MPI: Compilation & Exectuion

```
$ mpicc mpi_hello.c -o hello
$ mpirun -np 2 ./hello
Hello world from process 0 of 2
Hello world from process 1 of 2
```

MPI: Message Passing Interface

- Some MPI calls are blocking, e. g. MPI_Send, MPI_Recv
- This is important to know, to avoid deadlocks!
- Send doesn't block until message is received, but only until data is copied into internal buffer if there is enough space.

Example 1; What is the problem with this code?

```
int main (int argc, char* argv[])
{
  int rank, size, tmp;

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */

  MPI_Send(&rank, 1, MPI_INT, mod(rank+1,size), 0,
           MPI_COMM_WORLD);
  MPI_Recv(&tmp, 1, MPI_INT, mod(rank-1,size), 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

Example 2; What is the problem with this code?

```
int main (int argc, char* argv[])
{
  int rank, size, tmp;
  ...

  if(rank == 0)
  { MPI_Recv(&tmp, 1, MPI_INT, mod(rank-1,size), 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE); }
    MPI_Send(&rank, 1, MPI_INT, mod(rank+1,size), 0,
    MPI_COMM_WORLD); }
  else
  { MPI_Send(&rank, 1, MPI_INT, mod(rank+1,size), 0,
    MPI_COMM_WORLD);
    MPI_Recv(&tmp, 1, MPI_INT, mod(rank-1,size), 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE); }

  MPI_Finalize();
  return 0;
}
```

Example 3: Does this always work?

```
int main (int argc, char* argv[])
{
  int rank, size, tmp;

  MPI_Init(&argc, &argv); /* starts MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process id */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* number processes */

  MPI_Sendrecv(&rank, 1, MPI_INT, mod(rank+1,size), 0,
                &tmp, 1, MPI_INT, mod(rank-1,size), 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

Assignment 8

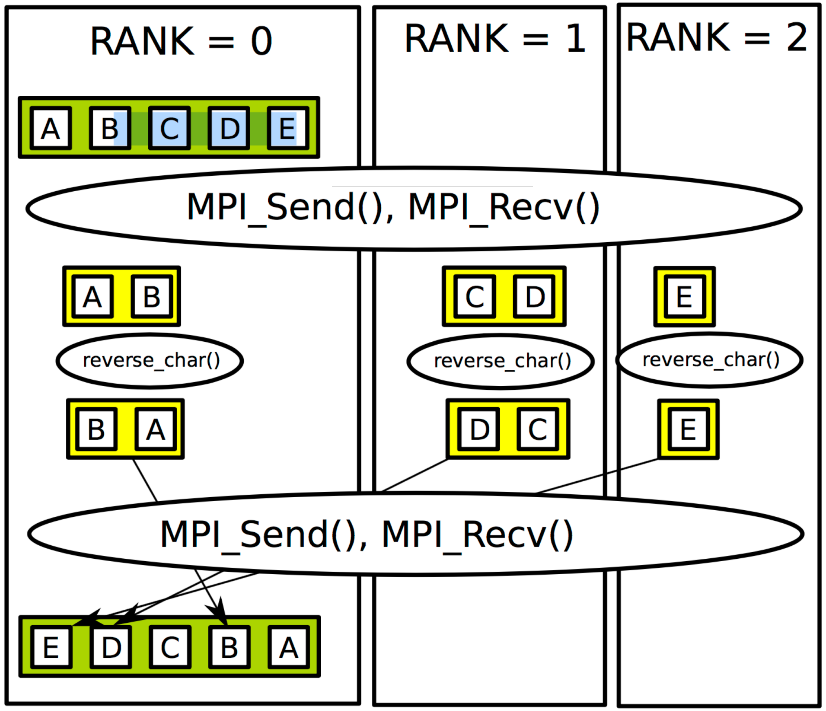
Assignment: Reversing with MPI

- Task: Reversing a (huge) char buffer with MPI
- Input e.g.: ”This is a simple string that should be printed in reverse order”
- Output: ”redro esrever ni detnirp eb dluohs taht gnirts elpmis a si sihT”
- Has to work with any number of processes (np $<$ number of chars)
- You can reuse the reverse function for local computation

Assignment: Reversing with MPI

- 3 steps necessary to parallelize the application
  - Distribute array from rank 0 to all ranks using `MPI_Send()` and `MPI_Recv()`
  - Call provided reverse function on the local part of the array
  - Send local part of the array back to rank 0 and store it directly at the right position
- Implement `scatterv` first and make sure that it is working correctly. You can use the provided print function to print the char buffer
- Use only the following MPI routines for communication: `Send()`, `Recv()`
- MPI template of the assignment will be provided

Assignment: Reversing with MPI


RANK = 0
RANK = 1
RANK = 2
A B C D E
MPI_Send(), MPI_Recv()
A B
C D
E
reverse_char()
reverse_char()
reverse_char()
B A
D C
E
MPI_Send(), MPI_Recv()
E D C B A