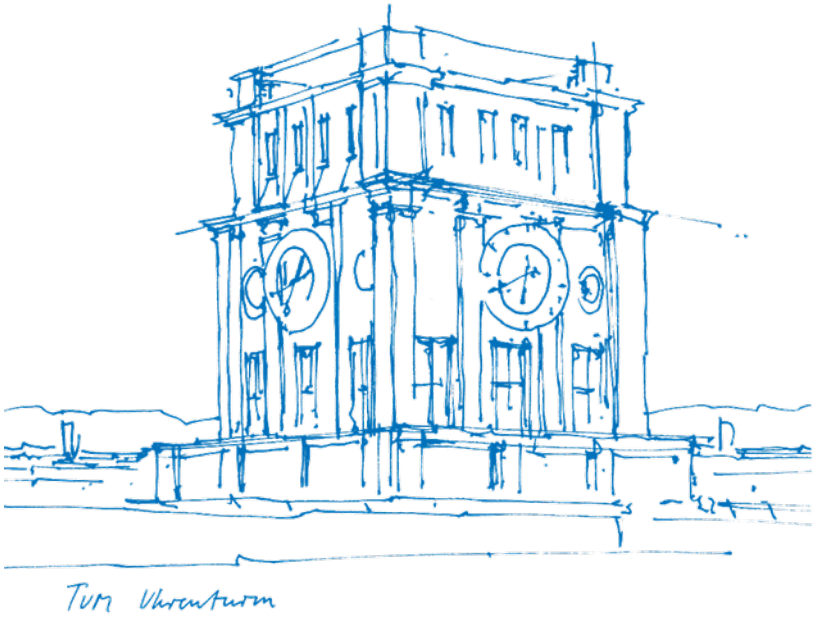
Parallel Programming Tutorial - MPI 1

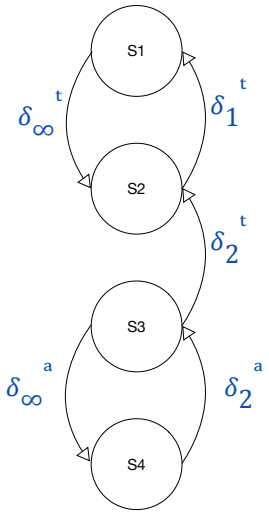
M.Sc. Andreas Wilhelm
Technichal University Munich
July 3rd, 2017



Solution for Assignment 6

Solution for Loop Distribution

```
for (int j = 1; j < N; j++) {
   for (int i = 1; i < N; i++) {
S1:   a[i][j]     = a[i][j] * b[i][j];
S2:   b[i][j + 1] = 2 * a[i][j] * c[i - 1][j];
S3:   c[i][j]     = 3 * d[i][j];
S4:   d[i][j]     = 2 * c[i + 1][j];
   }
}
```



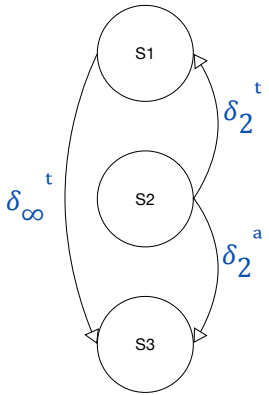
Solution

```
#pragma omp parallel num_threads(num_threads)
{
   for (int i = 1; i < N; i++) {
      #pragma omp for
      for (int j = 1; j < N; j++) {
         c[i][j]   = 3 * d[i][j];
         d[i][j]   = 2 * c[i + 1][j];
      }
   }

   for (int j = 1; j < N; j++) {
      #pragma omp for
      for (int i = 1; i < N; i++) {
         a[i][j]   = a[i][j] * b[i][j];
         b[i][j + 1] = 2 * a[i][j] * c[i - 1][j];
      }
   }
}
```

Solution for Loop Alignment

```
for (int i = 1; i < N; i++) {
   for (int j = 1; j < N; j++) {
S1:    a[i][j] = 3 * b[i][j];
S2:    b[i][j + 1] = c[i][j] * c[i][j];
S3:    c[i][j - 1] = a[i][j] * d[i][j];
   }
}
```



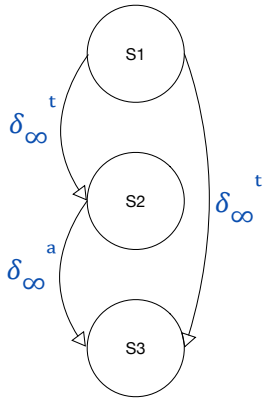
Solution

```
#pragma omp parallel num_threads(num_threads)
{
   #pragma omp for schedule(dynamic)
   for (int i = 1; i < N; i++) {
      int j = 1;
      a[i][j] = 3 * b[i][j];
      c[i][j - 1] = a[i][j] * d[i][j];

      for (j = 2; j < N; j++) {
         b[i][j] = c[i][j - 1] * c[i][j - 1];
         a[i][j] = 3 * b[i][j];
         c[i][j - 1] = a[i][j] * d[i][j];
      }
      j = N;
      b[i][j] = c[i][j - 1] * c[i][j - 1];
   }
}
```

Solution for Loop Fusion

```
for (int i = 1; i < N; i++) {
   for (int j = 1; j < N; j++) {
S1:   a[i][j] = 2 * b[i][j];
S2:   d[i][j] = a[i][j] * c[i][j];
   }
}
for (int j = 1; j < N; j++) {
   for (int i = 1; i < N; i++) {
S3:   c[i][j - 1] = a[i][j - 1] - a[i][j + 1];
   }
}
```



Solution

```
#pragma omp parallel for num_threads(num_threads)
for (int i = 1; i < N; i++) {
  int j = 1;
  a[i][j] = 2 * b[i][j];
  d[i][j] = a[i][j] * c[i][j];
  c[i][j - 1] = a[i][j - 1] - 2 * b[i][j + 1];

  for (j = 2; j < N - 1; j++) {
     a[i][j] = 2 * b[i][j];
     d[i][j] = a[i][j] * c[i][j];
     c[i][j - 1] = 2 * b[i][j - 1] - 2 * b[i][j + 1];
  }

  j = N - 1;
  a[i][j] = 2 * b[i][j];
  d[i][j] = a[i][j] * c[i][j];
  c[i][j - 1] =  2 * b[i][j - 1] - a[i][j + 1];
}
```

MPI: Message Passing Interface

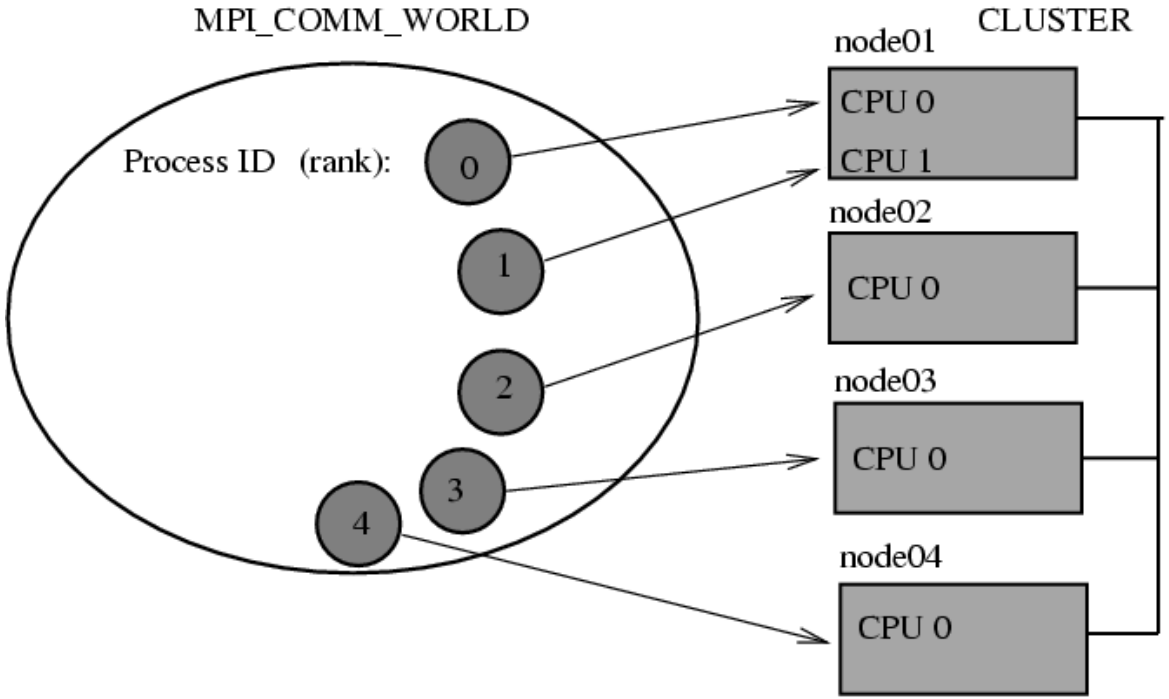
MPI: Message Passing Interface

- Library + Tools (compiler wrapper, documentation, deamon)
- Enables writing applications on distributed memory and shared memory systems.
- Communications is done by sending messages.
- Two types of operations: point-to-point or collectives
- SPMD programming model
- Single Program(source), is started as (multiple) processes on local or remote machines. 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 a another process, it has to be send 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 locally with MPI, but theres no difference working on a remote machine, except of performance.
- Debugging is nasty 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 (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
- Only install one of these two MPI libraries

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

- Most 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.

MPI: 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_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;
}
```

MPI: Does this always work?

```
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;
}
```

MPI: 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;
}
```

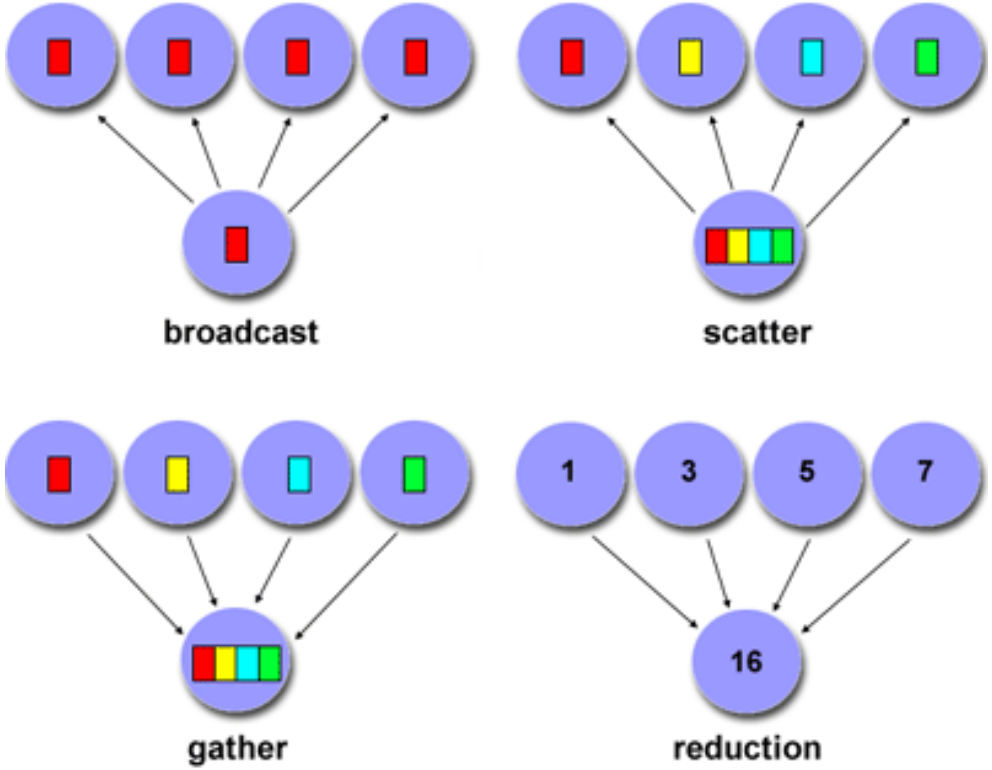
MPI_Sendrecv()

- Send and Recv are called as if by two independent threads!
- No deadlock that can be caused by the order of these two calls.
- Deadlocks can still occur, if Send and Recv signature doesn't match.
- MPI tag and source has to match.
- MPI status can be ignored by using MPI_STATUS_IGNORE.
- MPI_ANY_TAG and MPI_ANY_SOURCE can be used if tag and source are of no interest.
- Use the source identifier to make sure that you send and receive from the right process.

MPI: Important Functions

- MPI_Send(), MPI_Recv(), MPI_Sendrecv()
- asynchronous versions: MPI_Isend(), MPI_Irecv(), MPI_Wait(), MPI_Test()
- Collective Operations:
  - MPI_Broadcast()
  - MPI_Reduce()
  - MPI_Scatter() – MPI_Scatterv() only for assignment
  - MPI_Gather() – MPI_Gatherv() only for assignment
  - MPI_Barrier()

MPI: Important Collectives



MPI_Scatterv() and MPI_Gatherv()

- Like MPI_Scatter() and MPI_Gather() but can work on sparse / unregular data
- Two additional parameters, pointer to arrays (one element per rank) that hold the number of elements and the index of elements to scatter or gather.
- Take a look at the online documentation for further details.

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_Scatterv()`
  - 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 correclty. You can use the provided print function to print the char buffer
- Use only the following MPI Routines: `Scatterv()`, `Send()`, `Recv()`
- MPI template of the assignment will be provided

Assignment: Reversing with MPI