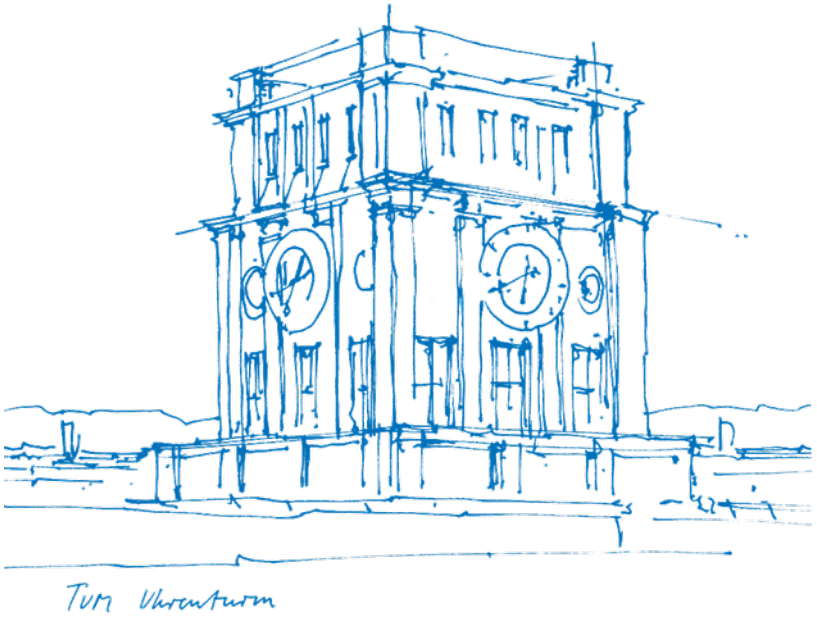
Parallel Programming Tutorial - Profiling tools

M.Sc. Amir Raoofy
Technical University of Munich
20. Juni 2018



Organization

Organization

- We will have enough explanation about the non-blocking communication homework today.
- Deadline for non-blocking communication homework is now extended to 03.07.2018.
- Those who submitted a solution without using MPI for reverse_str will not get a credit for the assignment.

- Reminder: Next week on Monday, we will have a lecture on optimization of sequential programs
  - By M.Sc. Alexis Engelke.

Solution for Assignment 9

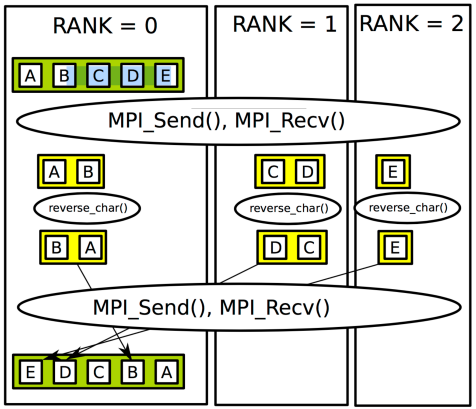
Solution for Reverse string using MPI

```
void reverse(char *str, int strlen)
{
    // get the rank and number of MPI processes
    int np, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int len, idx, map, len_r, idx_r, map_r;

    int n_extra = strlen%np;
    len = strlen/np + (rank<n_extra);
    idx = rank*len + n_extra*(rank>=n_extra);
    //Alocate memory
    char* temp = (char*) malloc(len*sizeof(char));

    ...
}
```



Solution for Reverse string using MPI (Cont.)

```
// Sending from rank 0
if (rank==0){
    // send the data to other ranks
    for (int r = 1; r < np; r++){
        len_r = strlen/np + (r<n_extra);
        idx_r = r*len_r + n_extra*(r>=n_extra);
        MPI_Send(&str[idx_r], len_r, MPI_CHAR, r,
        0, MPI_COMM_WORLD);
    }

    // copy data to a local temporary buffer
    memcpy(temp, str, len*sizeof(char));

}
else MPI_Recv(temp, len, MPI_CHAR, 0,
0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Reverse local strings in temp
reverse_str(temp, len);
```

```
// Send back reverted strings to rank 0
if (rank!=0){
    MPI_Send(temp, len, MPI_CHAR, 0,
    0, MPI_COMM_WORLD);
}
else{
    // receive the data back in rank 0
    for (int r = 1; r<np; r++) {
        len_r = strlen/np + (r<n_extra);
        idx_r = r*len_r + n_extra*(r>=n_extra);
        map_r = strlen - (idx_r + len_r - 1) - 1;
        MPI_Recv(&str[map_r], len_r, MPI_CHAR, r,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // copy the reverted string chunk back to the
    // appropriate location in rank 0
    map = strlen - (idx + len - 1) - 1;
    memcpy(str+map, temp, len*sizeof(char));
}
```

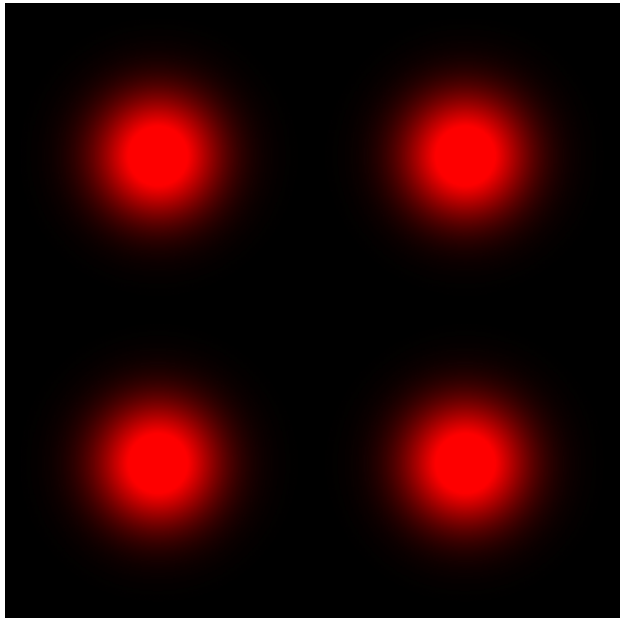
Solution for Reverse string using MPI (Cont.)

- Why did we use that for loop in helper.c in the implementation of reverse_str?
- Can you think of use of any collectives for the solution of reverse_str?
- Can you find a use-case for non-blocking communication in the solution of reverse_str?

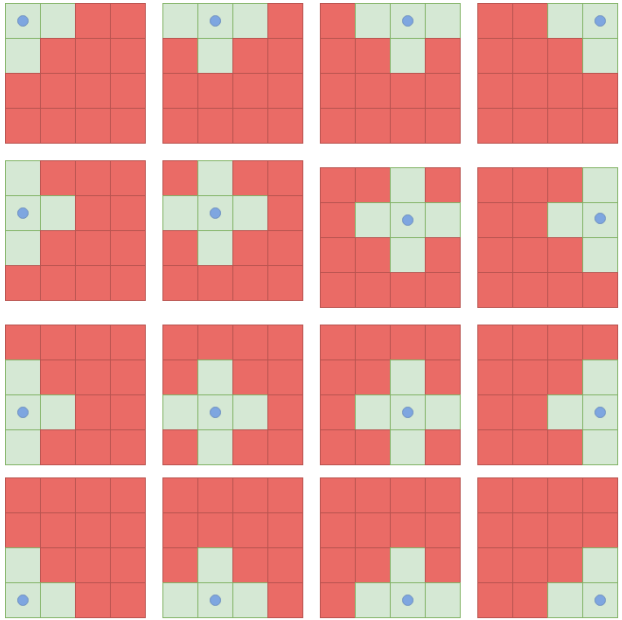
Hints for Assignment 10

Assignment 10 - Non-blocking communication

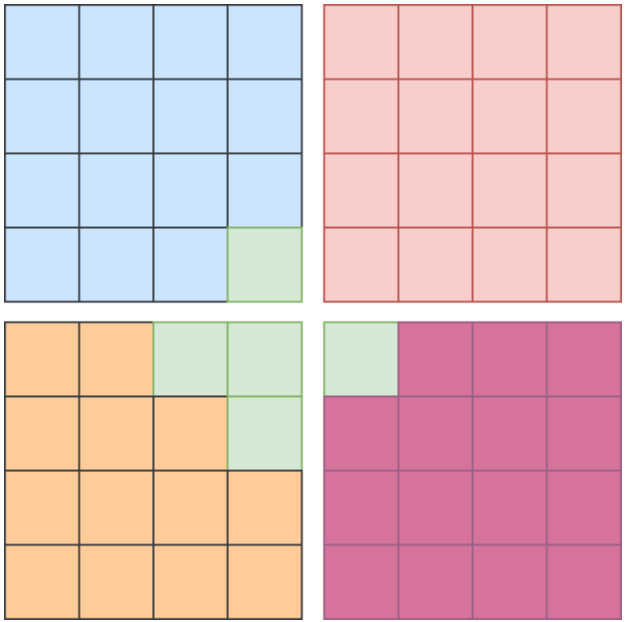
- 2d transient diffusion equation
- Problem domain is unit square with uniform mesh
- Finite differences are used for the discretization
- We use Jacobi iterative method to solve the equation
- Use non-blocking MPI communication to parallelize the solver
- The approach for parallelization is domain-decomposition
- You need to get a speedup of 12 on our server with 16 MPI processes



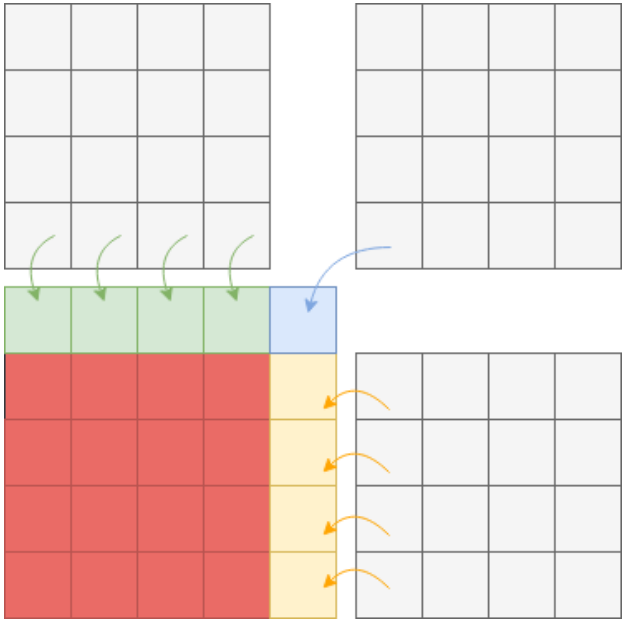
Assignment 10 - Non-blocking communication - stencil codes



Assignment 10 - Non-blocking communication - domain decomposition



Assignment 10 - Non-blocking communication - halo exchange



Assignment 10 - Non-blocking communication - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, unit_test, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
  - main function - argument handling + call initialization of arrays and main iteration loop
- heat.h
  - Headers and definition decelerations
- heat.c
  - Implements the function to initialize heat source locations
- heat_seq.h
  - Sequential version of `jacobi()` iterations.
- student/heat_par.h
  - Implement the parallel version in this file
- helper.h and helper.c
  - Declaration and implementation of helper functions, e.g., output writers

Assignment 10 - Non-blocking communication - Provided Files (Cont.)

- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.

build

- `>> make all`

run

- `>> mpirun -np n program <N> <energy_intensity> <niters> <iter_energy> <px> <py> <output_flag>`
  - `N`: problem size. calculation are done on a 2d, `N*N` grid.
  - `energy_intensity`: Intensity of heat sources.
  - `niters`: number of iterations of main loop.
  - `iter_energy`: number of iterations where heat sources are active.
  - `px` and `py`: number of MPI processes in x and y dimension, `n=px*py`.
  - `output_flag`: whether or not to create images of final output.

  - Example: `>> mpirun -np 8 ./student/heat_par 512 200 5000 50 4 2 1`

  - Note: your code should support px=py to get accepted to the server

Review from last tutorials

Quiz 1 - Which program will **always** have dead-lock?

```c
#define SIZE ....

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);
  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

```c
#define SIZE ....

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

Quiz 2 - Which program will **always** serialize communication?

```c
#define SIZE ....
int main (int argc, char* argv[])
{
  int rank, size;
  int message[SIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (rank==0){
  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);
  }else{
  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);
  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  MPI_Finalize();
  return 0;
}
```

```c
#define SIZE ....
int main (int argc, char* argv[])
{
  int rank, size;
  int message[SIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (rank==0){
  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);
  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }else{
  MPI_Recv(message, SIZE, MPI_INT, (rank+size-1)%size,
  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(message, SIZE, MPI_INT, (rank+1)%size,
  0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

Quiz 3 - What is the problem with this program?

```
#define SIZE ....

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

  MPI_Request req[2];

  MPI_Isend(message, SIZE, MPI_INT, (rank+1)%size, 0, MPI_COMM_WORLD, &req[0]);
  MPI_Irecv(message, SIZE, MPI_INT, (rank+size-1)%size, 0, MPI_COMM_WORLD, &req[1]);

  MPI_Waitall(2, &req, MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

Non-blocking collectives

Example - blocking collective

```
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double a[SIZE],b[SIZE],c[SIZE];
    srand(time(NULL));
    double sum_a=0, sum_b=0, sum_c=0;
    double avg_a=0, avg_b=0, avg_c=0;
    double min_a=RANGE, min_b=RANGE, min_c=RANGE;
    double max_a=-1, max_b=-1, max_c=-1;

    for (int i = 0; i < SIZE; ++i) { // init
^^I    a[i]=rand()%RANGE; b[i]=rand()%RANGE;
^^I    c[i]=rand()%RANGE;
    }
    for (int i = 0; i < SIZE; ++i) {
^^I    sum_a+=a[i]; // partail sums over array "a"
    }
    avg_a = sum_a / SIZE;
    MPI_Allreduce(&avg_a, &avg_a, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    avg_a/=size; // aggregate the average over all processes
```

Example - blocking collective (Cont.)

```
    for (int i = 0; i < SIZE; ++i) {
^^Ib[i]*=avg_a;
    }
    for (int i = 0; i < SIZE; ++i) {
^^Imin_b=MIN(min_b, b[i]);
^^Imax_b=MAX(max_b, b[i]);
    }
    MPI_Allreduce(&min_b, &min_b, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
    MPI_Allreduce(&max_b, &max_b, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

    for (int i = 0; i < SIZE; ++i) {
^^I    c[i]+=avg_a;
   ^^I    c[i]+=max_b/2.0;
   ^^I    c[i]+=min_b/2.0;
  ^^I    sum_c+=c[i];
    }
    avg_c = sum_c / SIZE;
    MPI_Allreduce(&avg_c, &avg_c, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    avg_c/=size;

    MPI_Finalize();
    return 0;
}
```

Example - non-blocking collective

```
^^I...

    for (int i = 0; i < SIZE; ++i) {
        min_b=MIN(min_b, b[i]);
        max_b=MAX(max_b, b[i]);
    }

    MPI_Request req_min, req_max;
    double temp_min = min_b, temp_max = max_b;
    MPI_Iallreduce(&temp_min, &min_b, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD, &req_min);
    MPI_Iallreduce(&temp_max, &max_b, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD, &req_max);

    for (int i = 0; i < SIZE; ++i) c[i]+=avg_a;
    MPI_Wait(&req_max, MPI_STATUS_IGNORE);

    for (int i = 0; i < SIZE; ++i) c[i]+=max_b/2.0;
    MPI_Wait(&req_min, MPI_STATUS_IGNORE);

    for (int i = 0; i < SIZE; ++i) c[i]+=min_b/2.0;
    for (int i = 0; i < SIZE; ++i) sum_c+=c[i];

    ...
```

mpiP; a lightweight MPI Profiling tool

mpiP - a lightweight MPI Profiling tool

- Open source; https://github.com/LLNL/mpiP
- Portable
- easy-to-use; single output file

Usage:

- Option1: add libmpip.a/.so to the link line
- Option2: set LD_PRELOAD to mpiP
- compile with -g for better accuracy

```
mpiP:
mpiP: mpiP: mpiP V3.4.2 (Build Jun 19 2018/10:51:26)
mpiP: Direct questions and errors to mpip-help@googlegroups.com
mpiP:
Before: THIS IS A SHORT TEST STRING
After : GNIRTS TSET TROHS A SI SIHT
Time: 0.241791 seconds
mpiP:
mpiP: Storing mpiP output in [./reverse_par.4.6461.1.mpiP].
mpiP:
```

Hands-on - Profiling first MPI homework

step 1: install mpiP:

```
>> git clone https://github.com/LLNL/mpiP.git
>> cd mpiP
>> ./configure
>> make all
>> make shared
```

step 2: open up the Makefile and apply the following changes:

```
change line#  4 to -> LDFLAGS = -lrt -I $(CURDIR) -L <path to libmpiP.so> -lmpiP -ldl -lm -lunwind
change line# 14 to -> CFLAGS += -g    ( only for this exercise; this is a bug in the Makefile ;-) )
```

step 3: compile your code

```
>> make
```

step 4: run

```
>> mpirun -np 4 ./student/reverse\_par "THIS IS A SHORT TEST STRING"
```

Output - Metadata

```
@ mpiP
@ Command : ./reverse_par THIS IS A SHORT TEST STRING
@ Version                  : 3.4.2
@ MPIP Build date          : Jun 19 2018, 13:25:37
@ Start time               : 2018 06 19 13:33:29
@ Stop time                : 2018 06 19 13:33:29
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 11521
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 lrr-laptop
@ MPI Task Assignment      : 1 lrr-laptop
@ MPI Task Assignment      : 2 lrr-laptop
@ MPI Task Assignment      : 3 lrr-laptop
```

Output - Overview

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0     0.0591     0.00458     7.75
   1     0.0593    0.000639     1.08
   2     0.0545    0.000648     1.19
   3     0.0546    0.000629     1.15
   *      0.228     0.00649     2.85
```

Output - Callsites

```
---------------------------------------------------------------------------
@--- Callsites: 4 ----------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line Parent_Funct             MPI_Call
  1   0 0x406b1f             28 reverse                  Recv
  2   0 0x406a8c             36 reverse                  Send
  3   0 0x406b90             43 reverse                  Recv
  4   0 0x406bba             51 reverse                  Send
```

Output - per Function Timing and Message Size

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time     App%    MPI%      Count    COV
Recv                    1       4.51     1.98   69.51          3   0.00
Recv                    3       1.84     0.81   28.32          3   0.02
Send                    4      0.077     0.03    1.19          3   0.14
Send                    2      0.064     0.03    0.99          3   0.00

---------------------------------------------------------------------------
@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
---------------------------------------------------------------------------
Call                 Site      Count      Total       Avrg  Sent%
Send                    2          3         20       6.67  50.00
Send                    4          3         20       6.67  50.00
```

Output - Callsite Time statistics

```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 8 ----------------------
---------------------------------------------------------------------------
```

Name	Site	Rank	Count	Max	Mean	Min	App%	MPI%
Recv	1	0	3	4.51	1.5	0.002	7.64	98.60
Recv	1	*	3	4.51	1.5	0.002	1.98	69.51
Recv	3	1	1	0.617	0.617	0.617	1.04	96.56
Recv	3	2	1	0.622	0.622	0.622	1.14	95.99
Recv	3	3	1	0.6	0.6	0.6	1.10	95.39
Recv	3	*	3	0.622	0.613	0.6	0.81	28.32
Send	2	0	3	0.054	0.0213	0.004	0.11	1.40
Send	2	*	3	0.054	0.0213	0.004	0.03	0.99
Send	4	1	1	0.022	0.022	0.022	0.04	3.44
Send	4	2	1	0.026	0.026	0.026	0.05	4.01
Send	4	3	1	0.029	0.029	0.029	0.05	4.61
Send	4	*	3	0.029	0.0257	0.022	0.03	1.19

Output - Callsite Message statistics

```
--------------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) --------------------
--------------------------------------------------------------------------------
```

Name	Site	Rank	Count	Max	Mean	Min	Sum
Send	2	0	3	7	6.667	6	20
Send	2	*	3	7	6.667	6	20
Send	4	1	1	7	7	7	7
Send	4	2	1	7	7	7	7
Send	4	3	1	6	6	6	6
Send	4	*	3	7	6.667	6	20

Other possibilities with mpiP

- You can change the parameters get better results.
  - More details
  - Reduce the size of output and also overheads
  - Change the stack trace length
  - Output paths

- You can use environment variables for changing the parameters
  - MPIP = "-c -o -k 4" (stack trace 4, include callsites)

- You can also limit the scope of profiling in the code,
  - MPI_Pcontrol(x)

Other profiling/tracing tools

- gprof: for profiling program executions; it uses call graphs
- mpiP: we had enough of it, right?
- Score-P: performance measurement tool for parallel codes
- Vampir: performance visualization and analysis tool
- Cube: performance visualizer for profiles based on Score-P
- Paraver also a performance visualization and analysis tool

- Are you interested in performance analysis of parallel codes and want to know more?
- Visit the course: "Parallel Program Engineering" offered by our chair
- There you see a lot of interesting topics including debugging, performance analysis, performance modeling and so on.

Assignment 11 - Profiling using mpiP

- Profiling a parallel classical Molecular Dynamic application (CoMD).
- Proxy code for MD developed by Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx).
- You can directly download the source code from https://github.com/ECP-copa/CoMD.
- But we only want to use mpiP to get an overview of the code.
- We only use the MPI version of CoMD.
- No programming this time.

What you need to do?

- Modify the Makefile to link against libmpiP.
- Compile the CoMD, download it from our server.
- Run the CoMD using the following command:
  - mpirun -np 16 ./bin/CoMD-mpi -i 4 -j 2 -k 2 -x 40 -y 40 -z 40
- Upload the output of the profiler to the server (.mpiP).