Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Exercises, Chapter/Topic 3

Submitted 2017-02-16
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Lars Karlsson / Mikael Ränner

Contents

1 Introduction

This report is part of the mandatory coursework. It describes the solutions for several chosen exercises from the course book [**?**].

2 3.2 - Generalization of algorithm for trapezoidal rule

Two functions to adapt the *trapezoidal rule* for `calc\_local\_a` and `calc\_local\_b` were written and tested with the source code from the book (*mpi_trap.c*).

```
double calc_local_a(int my_rank, double a, double b, int n, int comm_sz){
  double local_a = 0;
  double h = 0;
  int local_n = 0;
  int rest_n  = 0;

  h = (b-a)/n;
  local_n = n/comm_sz;

  rest_n = n%comm_sz;

  if(my_rank < rest_n){
    local_a = a + my_rank*local_n*h + my_rank*h;
  } else {
    local_a = a + my_rank*local_n*h + rest_n*h;
    local_a += (my_rank-rest_n) * h;
  }

  return local_a;

}


double calc_local_b(int my_rank, double a, double b, int n, int comm_sz){
  double h;
  int local_n;

  h = (b-a)/n;
  local_n = n/comm_sz;

  if (my_rank == (comm_sz-1)){
    return a + my_rank+1*local_n*h;
  } else {
    return calc_local_a(my_rank+1, a, b, n, comm_sz);
  }
}
```

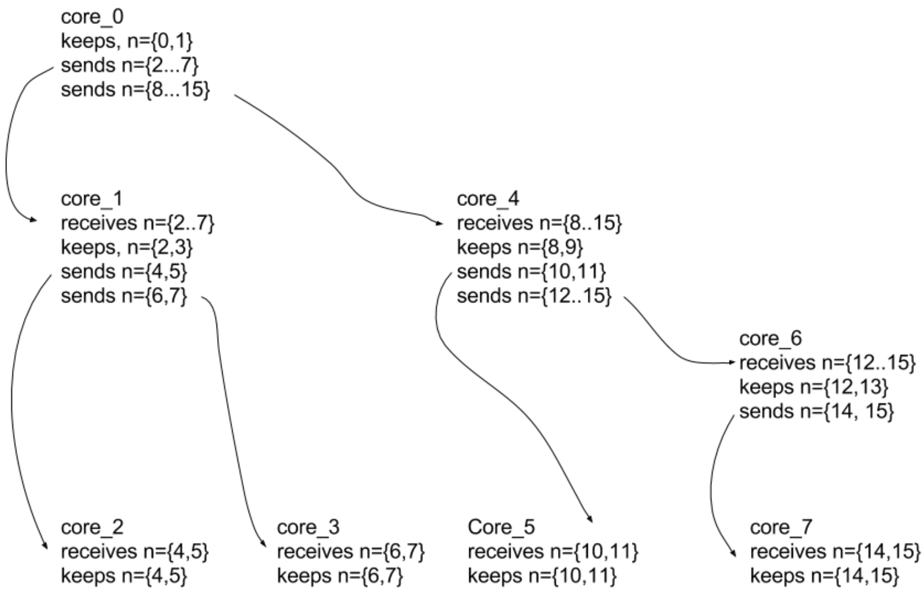

Figure 1: *This graph shows a tree based implementation of scatter for comm_sz = 8 and n = 16.*

3 3.6 - Array distributions

Block distribution

Block distribution can be obtained by $b = \lfloor i \div p \rfloor$ where b is the block number, i the index of n and p is the number of processes. This solution is however not fair. An improved, fair expression can be devised using a ternary operator:

$$i < n \bmod p \times \lceil n \div p \rceil ? \lfloor i \div \lceil n \div p \rceil \rfloor : n \mod p + \lfloor (i - n \mod p \times \lceil n \div p \rceil) \div \lfloor n \mod p \rfloor \rfloor$$

Cyclic distribution

Cyclic distribution is described by $b = i \mod p$ with b as block number i as index of n and p as number of processes.

Block cyclic distribution

Block cyclic distribution can be expressed as $b = \lfloor i \div l \rfloor mod p$ where b is block index, i index of n, l block length and p number of processes.

4 3.8 - Tree-structured algorithms for scatter and gather

5 3.9 - Vector scaling and dot product

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

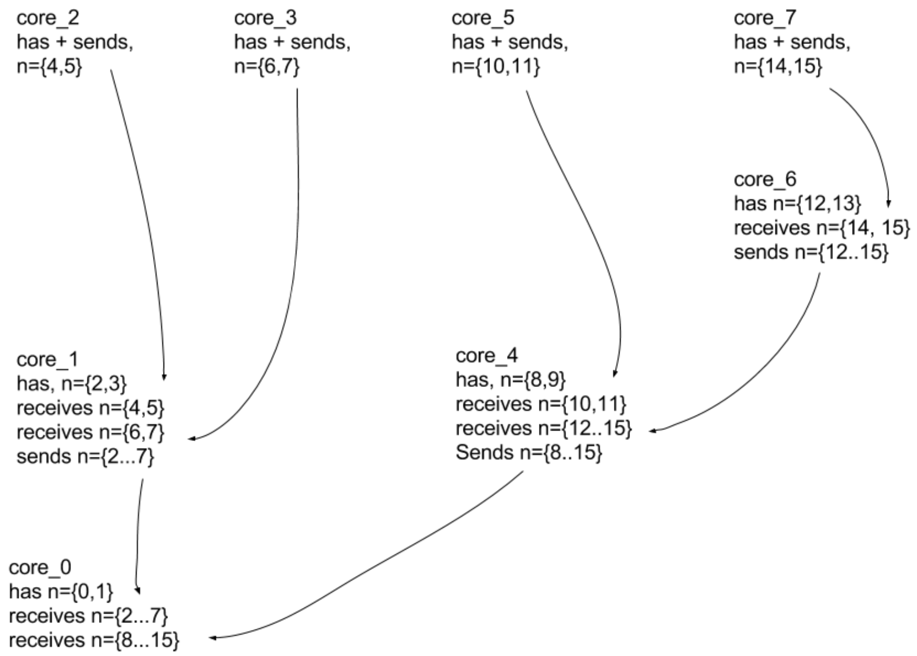

Figure 2: *This graph shows a tree based implementation of gather for comm_sz = 8 and n = 16.*

4(6)

```
#include <mpi.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>


int main(int argc, char *argv[]) {
  int my_rank, comm_sz;
  int n, local_n, local_dotp_sum = 0, scalar, result_dot;
  int* local_vec1;
  int* local_vec2;
  int* vector1;
  int* vector2;

  /* Initializing */
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  srand(time(NULL));

  /* Obtaining Data */
  if(my_rank==0 && argc > 1){

    if(strcmp(argv[1], "r") == 0){
      printf("using random data, vector length = %d\n", 100*comm_sz);
      n = 100*comm_sz;
      vector1 = (int *) malloc(100*comm_sz * sizeof(int));
      vector2 = (int *) malloc(100*comm_sz * sizeof(int));

      for(int i = 0; i < n;i++){
vector1[i] = rand() % 1000;
vector2[i] = rand() % 1000;
      }
      scalar = rand() % 1000;
    }

  } else if (my_rank==0){

    printf("enter vector length\n");
    scanf("%d", &n);
    printf("enter integer vector 1\n");

    vector1 = (int *) malloc(n * sizeof(int));
    vector2 = (int *) malloc(n * sizeof(int));

    for(int i = 0; i < n; i++){
      scanf("%d", &vector1[i]);
    }
```

```
    printf("enter integer vector 2\n");

    for(int i = 0; i < n; i++){
      scanf("%d", &vector2[i]);
    }

    printf("enter integer scalar\n");
    scanf("%d", &scalar);
  }

  /* Distribute Data */
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  local_n = n/comm_sz;
  local_vec1 = (int*) malloc(local_n * sizeof(int));
  local_vec2 = (int*) malloc(local_n * sizeof(int));

  MPI_Bcast(&scalar, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Scatter(vector1, local_n, MPI_INT, local_vec1, local_n, MPI_INT, 0, MPI_COMM_WORLD)
  MPI_Scatter(vector2, local_n, MPI_INT, local_vec2, local_n, MPI_INT, 0, MPI_COMM_WORLD)

  /* Calculations */
  /* Calculate Dot Product */
  for(int i = 0;  i< local_n; i++){
    local_vec2[i]*=local_vec1[i];
  }

  /* Calculate vector-scalar product */
  for(int i = 0; i< local_n; i++){
    local_vec1[i]*=scalar;
  }

  /* Summing for dot product */
  for(int i = 0; i < local_n; i++){
    local_dotp_sum += local_vec2[i];
  }

  /* Collect Data */
  MPI_Gather(local_vec1, local_n, MPI_INT, vector1, local_n, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Reduce(&local_dotp_sum, &result_dot, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  /* Results */
  if(my_rank == 0){
    printf("dot product = %d\n", result_dot);

    printf("vector-scalar product = ");
    for(int i = 0; i < n;i++){
```

```
      printf("%d ", vector1[i]);
    }
    printf("\n");
  }

  /* Clean up */
  if(my_rank==0){
    free(vector1);
    free(vector2);
  }

  free(local_vec1);
  free(local_vec2);

  MPI_Finalize();

  return 0;
}
```

6 3.11 - Prefix sums

takes a qhile to solve requires programming

7 3.13 - Generalization of vector scaling and dot product

8 3.16 - Diagram for a butterfly implementation of allgather

9 3.18 - Derived data types

takes a while to solve requires programming

10 3.20 - Pack and unpack

requires programming

11 3.21 - Matrix-vector multiplication

takes a while to solve requires programming requires testing

12 3.22 - Timing the trapezoidal rule

takes a while to solve Requires programming requires testing

13 3.27 - Speedup and efficienciy of odd-even sort