Technische Universität München

Assignment 4: MPI Collectives and MPI-IO

Programming of Super Computers

Friedrich Menhorn, Benjamin Rüth, Erik Wannerberg
Team 12

February 2, 2016

Contents

Optimizations

Sending the dimensions of the matrix to all processes:

Original code:

```
// send dimensions to all peers
if(rank == 0) {
    int i;
    for(i = 1; i < size; i++){
        MPI_Send(matrices_a_b_dimensions
             , 4, MPI_INT, i, 0,
             cartesian_grid_communicator
             );
    }
} else {
    MPI_Recv(matrices_a_b_dimensions, 4,
          MPI_INT, 0, 0,
         cartesian_grid_communicator, &
         status);
}
```

With collective operation:

```
// send dimensions to all peers
MPI_Bcast(matrices_a_b_dimensions, 4,
     MPI_INT, 0,
     cartesian_grid_communicator);
```

Optimizations (contd.)

Distribute matrix blocks among all processes:

Original code:

```
// send a block to each process
if(rank == 0) {
    int i;
    for(i = 1; i < size; i++){
        MPI_Send(A_array, ...);
        MPI_Send(B_array, ...);
    }
    for(i = 0; i < A_local_block_size; i
         ++){
        A_local_block[i] = A_array[i];
    }
    for(i = 0; i < B_local_block_size; i
         ++){
        B_local_block[i] = B_array[i];
    }
} else {
    MPI_Recv(A_local_block, ...);
    MPI_Recv(B_local_block, ...);
}
```

With collective operation:

```
// send a block to each process
MPI_Scatter(A_array, A_local_block_size,
      MPI_DOUBLE, A_local_block,
     A_local_block_size, MPI_DOUBLE, 0,
      cartesian_grid_communicator);
MPI_Scatter(B_array, B_local_block_size,
      MPI_DOUBLE, B_local_block,
     B_local_block_size, MPI_DOUBLE, 0,
      cartesian_grid_communicator);
```

Optimizations (contd.)

Distribute matrix blocks among all processes and do initial rotation of blocks:

Original code:

```
// send a block to each process
...
// fix initial arrangements before the
    core algorithm starts
if(coordinates[0] != 0){
    MPI_Sendrecv_replace(A_local_block,
        ...);
}
if(coordinates[1] != 0){
    MPI_Sendrecv_replace(B_local_block,
        ...);
}
```

With collective operation:

```
// send a block to each process and fix
     initial arrangements before the
     core algorithm starts
int *send_count= (int *) malloc(...);
int *displs_A = (int *) malloc(...);
int *displs_B = (int *) malloc(...);
int displ_coord_A_col = ...
int displ_coord_B_row = ...
int displs_local_A = ...
int displs_local_B = ...

MPI_Gather(&displs_local_A, ...);
MPI_Gather(&displs_local_B, ...);

for(i = 0; i < size; i++){
send_count[i] = A_local_block_size;
}

MPI_Scatterv(A_array, ...);
MPI_Scatterv(B_array, ...);
```

Optimizations (contd.)

Collect results after computation loop:

Original code:

```
// get C parts from other processes at
      rank 0
if(rank == 0) {
    for(i = 0; i < A_local_block_rows *
         B_local_block_columns; i++){
        C_array[i] = C_local_block[i];
    }
    int i;
    for(i = 1; i < size; i++){
        MPI_Recv(....);
    }
} else {
    MPI_Send(...);
}
```

With collective operation:

```
// get C parts from other processes at
     rank 0
int C_local_block_size =
     A_local_block_size;
MPI_Gather(C_local_block,
     C_local_block_size, MPI_DOUBLE,
     C_array, C_local_block_size,
     MPI_DOUBLE, 0,
     cartesian_grid_communicator);
```

Optimizations (contd.)

Expectations

- **Performance:**
  Higher performance due to highly optimized MPI collectives (might also consider network topology etc.) vs. naive implementation via `MPI_Send/MPI_Recv`.
- **Scaling:**
  Efficient implementation of Broadcast, Gather, Scatter use treelike distribution of values, $\mathcal{O}(n) \to \mathcal{O}(\log(n))$. This means we also benefit with respect to scalability.

Optimizations (contd.)

Readability and Maintainability

- ⊕ Sending dimensions with `MPI_Bcast`
- ⊕ Distributing matrix blocks `MPI_Scatter`
- ⊖ Distributing matrix blocks and initial alignment with `MPI_Scatterv`
  - → introduces complicated offsets
  - → hard to see what actually happens (distribution & rotation)
- ⊕ Collecting results using `MPI_Gather`

Performance Measurements

Show some measurements for falsification/verification of our guess

Data Sieving and 2-Phase IO

What is ”Data Sieving” and ”2-Phase IO”? How do they help improve IO performance?

Scalability of original application

- Was the original implementation scalable in terms of IO performance?
- Was the original implementation scalable in terms of RAM storage?

Impact of parallel IO–Reading Data

How much of the communication in the application was replaced with MPI-IO operations?

Original code (line 55-106):

```
if (rank == 0){
   int row, column;
   if ((fp = fopen (argv[1], "r")) != NULL){
       fscanf(fp, "%d %d\n", &matrices_a_b_dimensions[0], &matrices_a_b_dimensions[1]);
       A = (double **) malloc (matrices_a_b_dimensions[0] * sizeof(double *));
       for (row = 0; row < matrices_a_b_dimensions[0]; row++){
            A[row] = (double *) malloc(matrices_a_b_dimensions[1] * sizeof(double));
            for (column = 0; column < matrices_a_b_dimensions[1]; column++)
                fscanf(fp, "%lf", &A[row][column]);
        }
        fclose(fp);
    } else {
        if(rank == 0) fprintf(stderr, "error opening file for matrix A (%s)\n", argv[1]);
        MPI_Abort(MPI_COMM_WORLD, -1);
    }
    /* Here same for B matrix */

    // need to check that the multiplication is possible given dimensions
    /* Checks for right dimensions */
}
```

Impact of parallel IO–Reading Data

How much of the communication in the application was replaced with MPI-IO operations?

Using MPI IO:

```
MPI_Info readInfo;
MPI_Info_create(&readInfo);
int ierr;
int blocksize_A[2] = {A_rows, A_columns*characters_per_number};
MPI_Datatype datatype_blocks_A;
int subsize_A[2] = {A_local_block_rows, A_local_block_columns*characters_per_number};
int array_of_starts_A[2] = {coordinates[0]*A_local_block_rows,((coordinates[1] + coordinates[0])
      % sqrt_size)*A_local_block_columns*characters_per_number};    //This shifts the starting
      coordinate according to the original shift in Cannon's algorithm!
ierr = MPI_Type_create_subarray(2, blocksize_A, subsize_A, array_of_starts_A, MPI_ORDER_C,
      MPI_CHAR, &datatype_blocks_A);
MPI_Type_commit(&datatype_blocks_A);
MPI_File mpi_file_matrixA;
ierr=MPI_File_open(MPI_COMM_WORLD, argv[1], MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN,
      MPI_INFO_NULL, &mpi_file_matrixA);
if (ierr!=MPI_SUCCESS) {printf(" Cannot open file\n");}
MPI_File_set_view(mpi_file_matrixA, A_file_header_size, MPI_CHAR, datatype_blocks_A, "native",
      readInfo);

MPI_File_read_all(mpi_file_matrixA, A_local_block_read, A_local_block_size *
      characters_per_number, MPI_CHAR, MPI_STATUS_IGNORE);

MPI_File_close(&mpi_file_matrixA);
```

Impact of parallel IO–Writing Data

How much of the communication in the application was replaced with MPI-IO operations?

Original code (line 55-106):

```
int blocksize_C[2] = {A_rows, B_columns};
MPI_Datatype datatype_blocks_C;
int subsize_C[2] = {A_local_block_rows, B_local_block_columns};
int array_of_starts_C[2] = {coordinates[0]*A_local_block_rows, coordinates[1]*
      B_local_block_columns};    //This shifts the starting coordinate according to the original
       shift in Cannon's algorithm!

ierr = MPI_Type_create_subarray(2, blocksize_C, subsize_C, array_of_starts_C, MPI_ORDER_C,
      MPI_DOUBLE, &datatype_blocks_C);

MPI_Type_commit(&datatype_blocks_C);
MPI_File mpi_file_matrixC;
ierr=MPI_File_open(MPI_COMM_WORLD, output_filename, MPI_MODE_WRONLY | MPI_MODE_CREATE |
      MPI_MODE_UNIQUE_OPEN, MPI_INFO_NULL, &mpi_file_matrixC);
if (ierr!=MPI_SUCCESS) {printf(" Cannot open file\n");}
MPI_File_set_view(mpi_file_matrixC, 0, MPI_DOUBLE, datatype_blocks_C, "native", writeInfo);

MPI_File_write_all(mpi_file_matrixC, C_local_block, C_local_block_size, MPI_DOUBLE,
      MPI_STATUS_IGNORE);

MPI_File_close(&mpi_file_matrixC);
```