Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет по реализованной в рамках курса распределенные системы программе

Группа: 425
Студент: Повжик Юрий Максимович

Москва, 2025

Содержание

1. Исходные данные задачи

Исходной задачей является реализация метода релаксации матрицы.

При входе в программу каждый процесс вычисляет номера строк, в пределах которых он будет работать. Затем динамически выделяются два массива. Первый хранит исходные строки вместе с соседними, которые необходимы для вычислений. Нулевой и последний процесс используют лишь одну дополнительную строку.

Нужда в двух массивах объясняется необходимостью временного сохранения результата перед обменами с другими процессами полученными данными.

```
#define N (2*2*2*2*2*2+2)
```

```
struct two_row
{
    double data[N][N];
};

two_row* A;
two_row* B;
```

```
startrow = (my_rank * N) / my_size;
lastrow = ((my_rank + 1) * N) / my_size - 1;
n_rows = lastrow - startrow + 1;
int stop = 0;


if(startrow == 0 || lastrow == N-1)
    len = n_rows + 1;
else
    len = n_rows + 2;

A = new two_row[len];
B = new two_row[n_rows];
```

```
for(i=1; i <= len-2; i++)
for(j=1; j<=N-2; j++)
for(k=1; k<=N-2; k++)
{
    B[i-1].data[j][k]= (A[i-1].data[j][k]+A[i+1].data[j][k]
                +A[i].data[j-1][k]+A[i].data[j+1][k]
                +A[i].data[j][k-1]+A[i].data[j][k+1])/6.;
}

for(i=1; i <= len-2; i++)
for(j=1; j<=N-2; j++)
for(k=1; k<=N-2; k++)
{
    e = fabs(B[i-1].data[j][k] - A[i].data[j][k]);
    if(e > loceps)
        loceps = e;
    A[i].data[j][k] = B[i-1].data[j][k];
}
```

```
if(startrow != 0)
    MPI_Irecv(&A[0], N*N, MPI_DOUBLE, my_rank-1, 1235,
        main_comm, &request[0]);
if(lastrow != N - 1)
    MPI_Isend(&B[len-3], N*N, MPI_DOUBLE, my_rank+1, 1235
        main_comm, &request[2]);
if(lastrow != N - 1)
    MPI_Irecv(&A[len-1], N*N, MPI_DOUBLE, my_rank+1, 1236
        main_comm, &request[3]);
if(startrow  != 0)
    MPI_Isend(&B[0], N*N, MPI_DOUBLE, my_rank-1, 1236,
        main_comm, &request[1]);
```

2. Реализация отказоустойчивости

Для обработки ошибок предлагается использовать errhendler.

При смерти процесса мы создаем новый коммуникатор, восстанавливаем данные и продолжаем вычисления.

Для сохранения результатов предлагается каждые несколько итерация цикла записывать результаты файл и при поломке читать из него.

```
void write_to_file()
{
    MPI_File fh;
    int st = 1;
    if(startrow == 0)
    {
        st = 0;
    }


    MPI_File_open(main_comm, "my_file",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, startrow * sizeof(two_row), MPI_SEEK_SET);
    MPI_File_write_all(fh, &(A[st]), n_rows*N*N, MPI_DOUBLE,
                   MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
}
```

```
void read_from_file()
{
    MPI_File fh;
    MPI_Status status;
    int j;

    int count = startrow-1;
    if(startrow == 0)
    {
        count = 0;
    }

    MPI_File_open(main_comm, "my_file",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, count *sizeof(two_row), MPI_SEEK_SET);
    MPI_File_read_all(fh, A, len*N*N, MPI_DOUBLE, &status);
    MPI_File_close(&fh);
}
```

Для обработки ошибок в errhendler мы выбрасываем исключение, ловим его, заново перераспределяем данные и загружаем их в массив.

Для проверки работы отправляем процессу с рангом равным константе KILLED_PROCESS отправляем сигнал SIGKILL.

Живые делят между собой работу мертвого.

```
static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...)
{
    int err = *perr;
    char errstr[MPI_MAX_ERROR_STRING];
    int i, nf, len, eclass;
    MPI_Group group_c, group_f;
    int *ranks_gc, *ranks_gf;

    MPI_Error_class(err, &eclass);
    if( MPIX_ERR_PROC_FAILED != eclass ) {
        MPI_Abort(main_comm, err);
    }
    MPI_Comm_rank(main_comm, &my_rank);
    MPI_Comm_size(main_comm, &my_size);
    /* We use a combination of 'ack/get_acked' to obtain the list of failed processes.     */
    MPIX_Comm_failure_ack(main_comm);
    MPIX_Comm_failure_get_acked(main_comm, &group_f);
    MPI_Group_size(group_f, &nf);
    MPI_Error_string(err, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. %d found dead: { ", my_rank, my_size, errstr, nf);
    /* We use 'translate_ranks' to obtain the ranks of failed procs  in 'comm' communicator  */
    ranks_gf = (int*)malloc(nf * sizeof(int));
    ranks_gc = (int*)malloc(nf * sizeof(int));
    MPI_Comm_group(main_comm, &group_c);
    for(i = 0; i < nf; i++)
        ranks_gf[i] = i;
    MPI_Group_translate_ranks(group_f, nf, ranks_gf,
                              group_c, ranks_gc);
    for(i = 0; i < nf; i++)
        printf("%d ", ranks_gc[i]);
    printf("}\n");
    free(ranks_gf); free(ranks_gc);


    MPIX_Comm_shrink(main_comm, &main_comm);
    MPI_Comm_size(main_comm, &my_size);
    MPI_Comm_rank(main_comm, &my_rank);

    throw("one death");
}
```

Цикл с сохранением данных и убийством процесса.

```
write_to_file();
save_iter = 1;
int fst_kill = 1;
for(it=1; it<=itmax; it++)
{

    try
    {

        if(it%10 == 0)
        {
            write_to_file();
            save_iter = it;
        }
        if(it == DEAD_IT && fst_kill == 1)
        {
            fst_kill = 0;
            if(my_rank == KILLED_PROCESS) raise(SIGKILL);
            MPI_Barrier(main_comm);
        }
        eps = 1;
        relax();
        m_printf("it=%4i   eps=%f\n", it, eps);
        if(eps < maxeps) {
            stop = 1;
        }
        MPI_Bcast(&stop, 1, MPI_INT, 0, main_comm);
        if(stop) {
            break;
        }
    }
    catch(char const* e)
    {
        reset_param();
        read_from_file();
        it = save_iter-1;
    }
}
```

Перераспределение данных.

```
void reset_param()
{
    free(A);
    free(B);

    startrow = (my_rank * N) / my_size;
    lastrow = ((my_rank + 1) * N) / my_size - 1;
    n_rows = lastrow - startrow + 1;

    if(my_rank == 0 || my_rank == my_size-1)
        len = n_rows + 1;
    else
        len = n_rows + 2;


    A = new two_row[len];
    B = new two_row[n_rows];
}
```

3. Временные оценки

Наша цель – узнать, на сколько затратно по времени сохранение результатов работы на случай сбоя.

Рассмотрим зависимость времени работы без сохранения с работой с сохранением для различного числа процессов.

Количество процессов: время работы без сохранения; с сохранением

2: 0.255872; 0.345862
4: 0.203755; 0.378081
6: 0.157862; 0.390047

4. Заключение

Накладные расходы, вызванные промежуточным сохранением результатов, существенно увеличивают время работы.

Данной возможность стоит пользоваться, так как она обходится дешевле повторного запуска программы после сбоя.

Таким образом, не следует сохранять результаты слишком часто. Достаточно выбрать несколько контрольных точек на которым мы будем производить запись в файл.