

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ

РАСЧЕТНО-ГРАФИЧЕСКОЕ ЗАДАНИЕ

по дисциплине “Параллельное программирование”

на тему

**Разработка библиотеки отказоустойчивости для вычислительных
систем с массовым параллелизмом на примере одного приложения**

Выполнил студент _____ Марков В.А.
Ф.И.О.

Группы _____ МГ-165

Работу принял _____ д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1. Контрольные точки на уровне пользователя.....	4
2. Алгоритм выполнения программы при использовании контрольных точек	5
3. Реализация техники контрольных точек на уровне пользователя	6
ЗАКЛЮЧЕНИЕ	8
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	9
ПРИЛОЖЕНИЕ	10

ВВЕДЕНИЕ

Для существующих вычислительных систем петафлопсного уровня производительности и проектируемых систем эксафлопсного уровня существенное значение имеет показатель среднего времени безотказной работы. Разработка надежных высокопроизводительных систем на аппаратном уровне является трудноразрешимой задачей. В связи с этим, для организации вычислений на современных суперкомпьютерах необходимо развитие новых отказоустойчивых технологий, позволяющих с помощью программных решений корректно продолжать вычисления даже при отказе части оборудования [1].

Обеспечение отказоустойчивости на программном уровне классифицируется следующим образом:

- Протоколы репликации (replication protocols);
- Протоколы восстановления, основанных на откате (rollback recovery protocols);
- Протоколы само стабилизации (self stabilizing protocols);
- Логирование сообщений (message logging): оптимистичное, пессимистичное, обычное;
- Глобальные/локальные точки сохранения.

1 Контрольные точки на уровне пользователя

Техника контрольных точек (КТ) уровня пользователя (user-level checkpointing) задействует библиотеку (checkpointing library), предоставляя гибкий механизм реализации, в отличие от ограниченных возможностей КТ уровня ядра ОС.

В случае реализации отказоустойчивости на уровне пользователя, в контрольную точку входит только то, что явно укажет прикладной программист. В идеале, только те данные, которые необходимы для восстановления утерянной в результате сбоя информации. На уровне пользователя также можно контролировать время и частоту создания контрольных точек при выполнении программы, выбирая наиболее удобные моменты для их создания. Накладные расходы могут быть значительно уменьшены, однако, потребуется дополнительная работа прикладного программиста для реализации отказоустойчивости в приложении.

Преимущество такого подхода состоит в том, что КТ могут быть восстановлены на любом типе архитектуры. Тем не менее, КТ на уровне приложений требуют доступ к исходному коду пользователя и не поддерживают произвольное расположение контрольных точек. Таким образом, код пользователя должен быть инструментирован контрольными точками (часто вставляемых вручную программистом). В отличие от kernel-level checkpointing, user-level checkpointing не сигнализируются сигналами. Для того, чтобы контрольная точка сохранилась, ход выполнения программы должен достичь местоположения контрольной точки.

2. Алгоритм выполнения программы при использовании контрольных точек

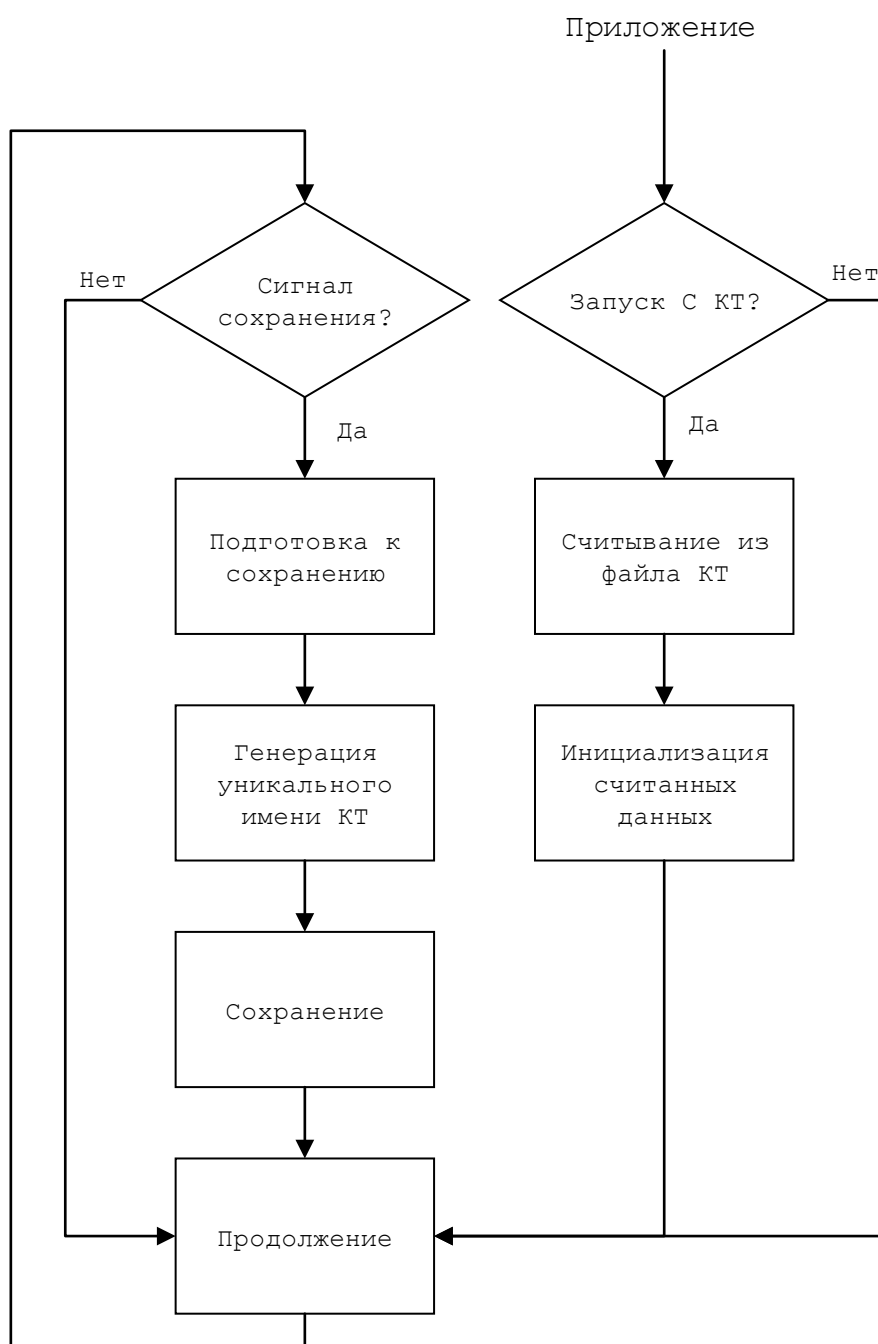


Рисунок 1 – Алгоритм работы КТ

3. Реализация техники контрольных точек на уровне пользователя

```
int main()
{
    // code . . .
    int idx = get_checkpoint_index();
    switch(idx)
    {
        Case 0: goto label_0;
        Case 1: goto label_1;
    }
    // code . . .
    label_0:
    func_1();
    label_1:
    func_2();
    // code . . .
    return 0;
}
```

Рисунок 2 – Пример реализации

```
/* ***** */
/* Global variables */
/* ***** */
extern double cpl_start_time;

extern void    **cpl_checkpoint_table;

extern int cpl_size;
extern int cpl_time;
extern int cpl_counter;

enum {
    CPL_CHECKPOINT_MODE = 0,
    CPL_RECOVERY_MODE   = 1
};

/* ***** */
/* Initializing checkpoint library macros */
/* ***** */

/*
 * Description:
 * size - checkpoints numbers
 * time - in seconds for timer
 * func - handler function
 */

#define CPL_INIT(size, time, func) \
    signal(SIGALRM, func); \
    cpl_size = time; \
    cpl_counter = 0; \
    cpl_size = size; \
    cpl_start_time = wtime(); \
    cpl_checkpoint_table = init_table_(cpl_size); \
```

```

//#define CPL_DEINIT() deinit_table_();

/*****
/* Declaration checkpoint macros */
*****/

/*
 * Description:
 * name - checkpoint_one, checkpoint_two, etc
 *       - phase_one, phase_two, etc
 *       - one, two, three, etc
 */

#define CPL_DECLARE_CHECKPOINT(name) \
    cpl_checkpoint_table[cpl_counter++] = name;

/*****
/* Control flow-macros */
*****/
#define CPL_GO_TO_CHECKPOINT(idx) \
    goto *cpl_checkpoint_table[idx];

#define CPL_SET_CHECKPOINT(checkpoint_name) \
    checkpoint_name :

/*****
/* Checkpoint-save macros */
*****/
#define CPL_FILE_OPEN(file, phase) \
    open_snapshot_file_(file, phase);

#define CPL_FILE_CLOSE(file) \
    close_snapshot_file_(file);

#define CPL_SAVE_SNAPSHOT(file, data, n, type) \
    write_to_snapshot_(file, data, n, type);

#define CPL_GET_SNAPSHOT(snapshot) \
    get_last_snapshot_(snapshot);

#define CPL_SAVE_STATE(checkpoint, user_save_callback) \
    user_save_callback(get_checkpoint_idx_by_name_(cpl_checkpoint_table, \
    cpl_size, checkpoint));\

```

Рисунок 3 – Реализованные макросы

ЗАКЛЮЧЕНИЕ

В результате выполнения расчетно-графического задания спроектирована и разработана библиотека отказоустойчивости для вычислительных систем с массовым параллелизмом на примере приложения, расчет теплопроводности двумерной пластины.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Обеспечение отказоустойчивости высокопроизводительных вычислений с помощью локальных контрольных точек URL:
<http://cyberleninka.ru/article/n/obespechenie-otkazoustoychivosti-vysokoproizvoditelnyh-vychisleniy-s-pomoschyu-lokalnyh-kontrolnyh-tochek>
2. Моделирование отказов в высокопроизводительных вычислительных системах в рамках стандарта MPI и его расширения ULFM URL:
http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=vyurv&paperid=1&option_lang=rus
3. Оптимизация времени создания и объёма контрольных точек восстановления параллельных программ URL:
http://vestnik.sibsutis.ru/uploads/1283929429_5076.pdf
4. Оптимальное сохранение контрольных точек на локальные устройства хранения URL: <http://2015.russianscdays.org/files/pdf/288.pdf>
5. Отказоустойчивая реализация метода молекулярной динамики на примере одного приложения URL: <http://ceur-ws.org/Vol-1576/058.pdf>
6. Building and using an Fault Tolerant MPI implementation URL:
<http://hpc.sagepub.com/content/18/3/353.abstract>
7. A Proposal for User-Level Failure Mitigation in the MPI-3 Standard URL:
http://icl.cs.utk.edu/news_pub/submissions/mpi3ft.pdf
8. A Comprehensive User-level Checkpointing Strategy for MPI Applications URL:
<https://pdfs.semanticscholar.org/4ec7/0cf657913023001279af685e601a0f85dcea.pdf>

ПРИЛОЖЕНИЕ

checkpoint_lib.c

```
/* **** */
/* C - check */
/* P - point */
/* L - library */
/* **** */
#include "checkpoint_lib.h"
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
/* **** */
/* Global variables */
/* **** */
double cpl_start_time = 0.0;
void **cpl_checkpoint_table;
int cpl_size = 0;
int cpl_time = 0;
int cpl_counter = 0;
static struct itimer cpl_timer;
/* **** */
/* Timer */
/* **** */
inline void timer_init_()
{
    cpl_timer.it_interval.tv_sec = cpl_time; // interval
    cpl_timer.it_interval.tv_usec = 0;
    cpl_timer.it_value.tv_sec = cpl_time; // time until next expiration
    cpl_timer.it_value.tv_usec = 0;

    setitimer(ITIMER_REAL, &cpl_timer, NULL);
}

inline void timer_stop_()
{
    cpl_timer.it_interval.tv_sec = 0;
    cpl_timer.it_value.tv_sec = 0;
}
/* **** */
/* Measure time */
/* **** */
inline double wtime_()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}
/* **** */
/* Additional internal mpi functions */
/* **** */
static int get_comm_rank__()
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    return rank;
}

static int get_comm_size__()
{
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    return size;
}
/* **** */
/* Work with files */
/* **** */
void open_snapshot_file_(MPI_File *snapshot, int phase)
{
    char file_name[256] = { 0 };
}
```

```

/*
 * 1_1_1.3456 => [PHASE_OF_CALCULATION]_[RANK]_[CHECKPOINT_TIME]
 *
 * PHASE_OF_CALCULATION - global state, need for 'goto'
 * RANK                  - process rank
 * CHECKPOINT_TIME       - each 'PHASE_OF_CALCULATION' could reach many times
 */
sprintf(file_name,"snapshot/%d_%d_%f", phase, get_comm_rank__(), wtime__() - cpl_start_time);
MPI_File_open( MPI_COMM_WORLD, file_name,
               MPI_MODE_CREATE|MPI_MODE_WRONLY,
               MPI_INFO_NULL, snapshot );
}

void close_snapshot_file_(MPI_File *snapshot)
{
    MPI_File_close(snapshot);
}

void write_to_snapshot_(MPI_File file, void *data, int n, MPI_Datatype type)
{
    MPI_Status status;
    MPI_File_write(file, data, n, type, &status);
}
/*****
/* Get last snapshot file
*****/
static int get_lastcheckpoint_rank__(char *file)
{
    int j, i, checkpoint_rank;

    char tmp_rank[10] = { 0 };

    for (j = 0, i = 2; i < strlen(file); i++, j++) {
        if (file[i] != '_') {
            tmp_rank[j] = file[i];
        } else {
            break;
        }
    }

    sscanf(tmp_rank, "%d", &checkpoint_rank);
    return checkpoint_rank;
}

static void get_lastcheckpoint_time__(char *a, char *b)
{
    int j, i;

    double x, y;

    char time_last[10] = { 0 };
    char time_cur[10] = { 0 };

    for (j = 0, i = 4; i < strlen(b); i++, j++) {
        time_last[j] = a[i];
        time_cur[j] = b[i];
    }

    sscanf(time_last, "%lf", &x);
    sscanf(time_cur, "%lf", &y);

    // Figure out which checkpoint is 'older'
    if (x < y) {
        strcpy(a, b);
    }
}

int get_last_snapshot_(char *last_checkpoint)
{
    int myrank = get_comm_rank__();

    DIR *dir;
    struct dirent *file;

    dir = opendir("./snapshot"); // open current directory
    if (dir) {
        while (file = readdir(dir)) {

```

```

        // Skip '.' '..' directory
        if (strlen(file->d_name) < 3) {
            continue;
        }
        if (file->d_name[1] != '_') {
            continue;
        }
        // Work only with myrank checkpoint
        if (myrank == get_lastcheckpoint_rank__(file->d_name)) {
            //printf("[DEBUG] Found for rank %d - %s\n", myrank, file->d_name);
            // Work only with greater checkpoint phase
            if (last_checkpoint[0] <= file->d_name[0]) {
                get_lastcheckpoint_time__(last_checkpoint, file->d_name);
            }
            //printf("[DEBUG] %s\n", last_checkpoint);
        }
        closedir(dir);
    } else {
        fprintf(stderr, "can't open current directory\n");
    }
    printf("Rankd %d, file %s, phase %d\n", myrank, last_checkpoint, last_checkpoint[0] - '0');
    return last_checkpoint[0] - '0';
}
/*****
/* Get checkpoint index by checkpoint name
*****/
int get_checkpoint_idx_by_name_(void **table, int size, void *name)
{
    int i;
    for (i = 0; i < size; i++) {
        if (table[i] == name) {
            break;
        }
    }
    return i;
}
/*****
/* Init
*****/
void **init_table_(int size)
{
    void **jump_table = (void**) malloc (sizeof(void*) * size);
    if (!jump_table) {
        fprintf(stderr, "[ERROR] can't allocate memory for CPL_GLOBAL_JUMP_TABLE\n");
        exit(1);
    }

    mkdir("snapshot", 0777);

    return jump_table;
}

#ifdef _CHECKPOINT_LIB_H_
#define _CHECKPOINT_LIB_H_
/*****
/* C - check
/* P - point
/* L - library
*****/

#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <mpi.h>
/*****
/* Global variables
*****/
extern double cpl_start_time;
extern void **cpl_checkpoint_table;
extern int cpl_size;
extern int cpl_time;
extern int cpl_counter;
enum {
    CPL_CHECKPOINT_MODE = 0,
    CPL_RECOVERY_MODE = 1

```

```

};

/*****
/* Initializing checkpoint library macros
*****/
#define CPL_INIT(size, time, func) \
    signal(SIGALRM, func); \
    cpl_size = time; \
    cpl_counter = 0; \
    cpl_size = size; \
    cpl_start_time = wtime_(); \
    cpl_checkpoint_table = init_table_(cpl_size); \
/*****
/* Declaration checkpoint macros
*****/
#define CPL_DECLARE_CHECKPOINT(name) \
    cpl_checkpoint_table[cpl_counter++] = name; \
/*****
/* Control flow-macros
*****/
#define CPL_GO_TO_CHECKPOINT(idx) \
    goto *cpl_checkpoint_table[idx]; \
#define CPL_SET_CHECKPOINT(checkpoint_name) \
    checkpoint_name : \
/*****
/* Checkpoint-save macros
*****/
#define CPL_FILE_OPEN(file, phase) \
    open_snapshot_file_(file, phase); \
#define CPL_FILE_CLOSE(file) \
    close_snapshot_file_(file); \
#define CPL_SAVE_SNAPSHOT(file, data, n, type) \
    write_to_snapshot_(file, data, n, type); \
#define CPL_GET_SNAPSHOT(snapshot) \
    get_last_snapshot_(snapshot); \
#define CPL_SAVE_STATE(checkpoint, user_save_callback) \
    user_save_callback(get_checkpoint_idx_by_name_(cpl_checkpoint_table, cpl_size, checkpoint)); \
/*****
/* Timer
*****/
#define CPL_TIMER_INIT() \
    timer_init_(); \
#define CPL_TIMER_STOP() \
    timer_stop_(); \
/*****
/* Prototypes
*****/
void **init_table_(int size);
double wtime_();
void timer_init_();
void timer_stop_();
void write_to_snapshot_(MPI_File file, void *data, int n, MPI_Datatype type);
int get_last_snapshot_(char *last_checkpoint);
int get_checkpoint_idx_by_name_(void **table, int size, void *name);
void open_snapshot_file_(MPI_File *snapshot, int phase);
void close_snapshot_file_(MPI_File *snapshot);
#endif /* _CHECKPOINT_LIB_H_ */

```

heat-2d.c

```

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <mpi.h>
#include <string.h>
#include "../checkpoint_lib/checkpoint_lib.h"
#define EPS 0.001
#define PI 3.14159265358979323846
#define NELEMS(x) (sizeof((x)) / sizeof((x)[0]))
#define IND(i, j) ((i) * (nx + 2) + (j))
/*****
/* Global variables
*****/
int is_time_to_save = 0;

```

```

int options      = CPL_CHECKPOINT_MODE;
//int options    = CPL_RECOVERY_MODE;
int niters      = 0;
int ny, nx;
double tttotal  = 0.0;
double thalo    = 0.0;
double treduce  = 0.0;
double *local_grid = NULL;
double *local_newgrid = NULL;
void *xmalloc(size_t nmemb, size_t size)
{
    void *p = calloc(nmemb, size);
    if (p == NULL) {
        fprintf(stderr, "No enough memory\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    return p;
}

int get_block_size(int n, int rank, int nprocs)
{
    int s = n / nprocs;
    if (n % nprocs > rank)
        s++;
    return s;
}

int get_sum_of_prev_blocks(int n, int rank, int nprocs)
{
    int rem = n % nprocs;
    return n / nprocs * rank + ((rank >= rem) ? rem : rank);
}
/*****
/* User defiend callbacks
/* Additional functions helps save and get data
*****/
inline static void user_save_callback(int phase)
{
    MPI_File local_snapshot;

    CPL_FILE_OPEN(&local_snapshot, phase);

    CPL_SAVE_SNAPSHOT(local_snapshot, &nx, 1, MPI_INT);
    CPL_SAVE_SNAPSHOT(local_snapshot, &ny, 1, MPI_INT);

    CPL_SAVE_SNAPSHOT(local_snapshot, local_grid, ((ny + 2) * (nx + 2)), MPI_DOUBLE);
    CPL_SAVE_SNAPSHOT(local_snapshot, &tttotal, 1, MPI_DOUBLE);
    CPL_SAVE_SNAPSHOT(local_snapshot, &thalo, 1, MPI_DOUBLE);
    CPL_SAVE_SNAPSHOT(local_snapshot, &treduce, 1, MPI_DOUBLE);
    CPL_SAVE_SNAPSHOT(local_snapshot, &niters, 1, MPI_INT);

    CPL_FILE_CLOSE(&local_snapshot);

    is_time_to_save = 0;
}
inline static int checkpoint_get(double *local_grid,
                                int size,
                                double *tttotal,
                                double *thalo,
                                double *treduce,
                                int *niters)
{
    int nx, ny;

    char last_checkpoint[] = { "0_0_0.000000" };
    int phase = CPL_GET_SNAPSHOT(last_checkpoint);

    char last_checkpoint_path[256] = { "snapshot/" };
    strcat(last_checkpoint_path, last_checkpoint);

    FILE * file = fopen(last_checkpoint_path, "rb");
    if (!file) {
        fprintf(stderr, "Can't read snapshot\n");
        exit(1);
    }

    // copy the file into the buffer:

```

```

fread(&nx, sizeof(int), 1, file);
fread(&ny, sizeof(int), 1, file);

if (size != ((ny + 2) * (nx + 2))) {
    fprintf(stderr, "Snapshot size not match\n");
    fclose(file);
    exit(1);
} else {
    fprintf(stderr, "Snapshot size match\n");
}

fread(local_grid, sizeof(double), size, file);
fread(tttotal, sizeof(double), 1, file);
fread(thalo, sizeof(double), 1, file);
fread(treduce, sizeof(double), 1, file);
fread(niters, sizeof(int), 1, file);

fclose(file);
return phase;
}

void time_handler(int sig)
{
    is_time_to_save = 1;
}

int main(int argc, char *argv[])
{
    /******
    /* Initialize checkpoint library
    /******
    int checkpoint_numbers = 2;
    int timers_time = 5;
    CPL_INIT(checkpoint_numbers, timers_time, time_handler);
    CPL_DECLARE_CHECKPOINT(&phase_one);
    CPL_DECLARE_CHECKPOINT(&phase_two);
    /******
    /* Local variables
    /******
    char procname[MPI_MAX_PROCESSOR_NAME];
    int commsize, rank, rankx, ranky, rows, cols, px, py, namelen;
    int coords[2];
    int dims[2] = { 0, 0 };
    int periodic[2] = { 0, 0 };
    int left, right, top, bottom;
    /******
    /* Initialize
    /******
    /* Initialize args, rank, commsize */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* Get processor name */
    MPI_Get_processor_name(procname, &namelen);
    /* Create 2D grid of processes: commsize = px * py */
    MPI_Comm cartcomm;
    MPI_Dims_create(commsize, 2, dims);
    px = dims[0];
    py = dims[1];
    /* Make start time point */
    tttotal = MPI_Wtime();
    if (px < 2 || py < 2) {
        fprintf(stderr, "Invalid number of processes %d: px %d and py %d must be greater than 1\n",
commsize, px, py);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &cartcomm);
    MPI_Cart_coords(cartcomm, rank, 2, coords);
    rankx = coords[0];
    ranky = coords[1];
    /******
    /* Broadcast command line arguments
    /******
    if (rank == 0) {
        rows = (argc > 1) ? atoi(argv[1]) : py * 100;
        cols = (argc > 2) ? atoi(argv[2]) : px * 100;
        if (rows < py) {
            fprintf(stderr, "Number of rows %d less then number of py processes %d\n", rows, py);
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);

```

```

    }
    if (cols < px) {
        fprintf(stderr, "Number of cols %d less then number of px processes %d\n", cols, px);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    int args[2] = { rows, cols };
    MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
} else {
    int args[2];
    MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
    rows = args[0];
    cols = args[1];
}
/*****
/* Allocate memory for local 2D subgrids with halo cells
/* [0..ny + 1][0..nx + 1]
*****/
ny = get_block_size(rows, ranky, py);
nx = get_block_size(cols, rankx, px);
local_grid = xmalloc((ny + 2) * (nx + 2), sizeof(*local_grid));
local_newgrid = xmalloc((ny + 2) * (nx + 2), sizeof(*local_newgrid));
// Fill boundary points:
// - left and right borders are zero filled
// - top border: u(x, 0) = sin(pi * x)
// - bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
double dx = 1.0 / (cols - 1.0);
int sj = get_sum_of_prev_blocks(cols, rankx, px);
if (ranky == 0) {
    // Initialize top border: u(x, 0) = sin(pi * x)
    for (int j = 1; j <= nx; j++) {
        // Translate col index to x coord in [0, 1]
        double x = dx * (sj + j - 1);
        int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * x);
    }
}
if (ranky == (py - 1)) {
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 1; j <= nx; j++) {
        // Translate col index to x coord in [0, 1]
        double x = dx * (sj + j - 1);
        int ind = IND(ny + 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * x) * exp(-PI);
    }
}
// Neighbours
MPI_Cart_shift(cartcomm, 0, 1, &left, &right);
MPI_Cart_shift(cartcomm, 1, 1, &top, &bottom);
// Left and right borders type
MPI_Datatype col;
MPI_Type_vector(ny, 1, nx + 2, MPI_DOUBLE, &col);
MPI_Type_commit(&col);

// Top and bottom borders type
MPI_Datatype row;
MPI_Type_contiguous(nx, MPI_DOUBLE, &row);
MPI_Type_commit(&row);
MPI_Request reqs[8];

if (options == CPL_CHECKPOINT_MODE) {
    CPL_SAVE_STATE(&phase_one, user_save_callback);
} else if (options == CPL_RECOVERY_MODE) {
    int phase = checkpoint_get(local_grid, ((ny + 2) * (nx + 2)), &ttotal, &thalo, &treduce,
&niters);
    // Jumping to checkpoint
    CPL_GO_TO_CHECKPOINT(phase);
}

CPL_SET_CHECKPOINT(phase_one);
CPL_TIMER_INIT();

is_time_to_save = 1;

for (;;) {
    niters++;

```



```

// Update interior points
for (int i = 1; i <= ny; i++) {
    for (int j = 1; j <= nx; j++) {
        local_newgrid[IND(i, j)] =
            (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1, j)] +
             local_grid[IND(i, j - 1)] + local_grid[IND(i, j + 1)]) * 0.25;
    }
}

// Check termination condition
double maxdiff = 0;
for (int i = 1; i <= ny; i++) {
    for (int j = 1; j <= nx; j++) {
        int ind = IND(i, j);
        maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
    }
}

// Swap grids (after termination local_grid will contain result)
double *p = local_grid;
local_grid = local_newgrid;
local_newgrid = p;

treduce -= MPI_Wtime();
MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
treduce += MPI_Wtime();

if (maxdiff < EPS) {
    break;
}

// Halo exchange:  $T = 4 * (a + b * (rows / py)) + 4 * (a + b * (cols / px))$ 
thalo -= MPI_Wtime();

MPI_Irecv(&local_grid[IND(0, 1)], 1, row, top, 0, cartcomm, &reqs[0]); // top
MPI_Irecv(&local_grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm, &reqs[1]); // bottom
MPI_Irecv(&local_grid[IND(1, 0)], 1, col, left, 0, cartcomm, &reqs[2]); // left
MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm, &reqs[3]); // right
MPI_Isend(&local_grid[IND(1, 1)], 1, row, top, 0, cartcomm, &reqs[4]); // top
MPI_Isend(&local_grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm, &reqs[5]); // bottom
MPI_Isend(&local_grid[IND(1, 1)], 1, col, left, 0, cartcomm, &reqs[6]); // left
MPI_Isend(&local_grid[IND(1, nx)], 1, col, right, 0, cartcomm, &reqs[7]); // right
MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
thalo += MPI_Wtime();
if (is_time_to_save)
    CPL_SAVE_STATE(&phase_one, user_save_callback);
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
CPL_SAVE_STATE(&phase_two, user_save_callback);
CPL_SET_CHECKPOINT(phase_two);
MPI_Type_free(&row);
MPI_Type_free(&col);
free(local_newgrid);
free(local_grid);
ttotal += MPI_Wtime();
if (rank == 0) {
    printf("# Heat 2D (mpi): grid: rows %d, cols %d, procs %d (px %d, py %d)\n", rows, cols,
           commsize, px, py);
}
printf("# P %4d (%2d, %2d) on %s: grid ny %d nx %d, total %.6f,"
       " mpi %.6f (%.2f) = allred %.6f (%.2f) + halo %.6f (%.2f)\n",
       rank, rankx, ranky, procname, ny, nx, ttotal, treduce + thalo,
       (treduce + thalo) / ttotal, treduce, treduce / (treduce + thalo),
       thalo, thalo / (treduce + thalo));

double prof[3] = { ttotal, treduce, thalo };
if (rank == 0) {
    MPI_Reduce(MPI_IN_PLACE, prof, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    printf("# procs %d : grid %d %d : niters %d : total time %.6f : "
           " mpi time %.6f : allred %.6f : halo %.6f\n",
           commsize, rows, cols, niters, prof[0], prof[1] + prof[2], prof[1], prof[2]);
} else {
    MPI_Reduce(prof, NULL, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}

```