Universidade do Minho

Engenharia de Sistemas de Computação

Mestrado Integrado em Engenharia Informática

Bernardo Silva - a77230

Francisco Lira - a73909

TP3

Análise/Avaliação de uma aplicação C/C++

Conteúdo

1 Introdução

Neste trabalho vamos realizar a análise de uma aplicação escrita em C/C++ utilizando traçado dinâmico (DTrace). Para isso estamos a utilizar os programas realizados pelo nosso grupo na disciplina de Paradigmas da Computação Paralela.

O programa que foi realizado corresponde a um problema de dissipação de calor, no qual em cada iteração do problema a temperatura de cada ponto numa matriz é representado pela média da temperatura dos pontos rodeantes nas direções cardeais (Norte, Sul, Este, Oeste) e pela sua própria temperatura.

O código referente a estes programas será disponibilizado em anexo.

2 Implementação com PThreads

Na implementação com pthreads utilizou-se uma barreira tal como na versão original openmp na qual cada thread fica com um bloco de linhas da matriz, correspondente à divisão do número de linhas pelo número de threads.

Retirou-se os cálculos matriciais da função main, de forma a que estes possam ser atribuídos como função a cada thread.

3 Implementação com C++11

Uma das dificuldades da implementação em C++11 é o facto de a biblioteca threads não conter nenhuma implementação de barreira, tendo sido apenas implementada na versão C++20, e como tal tive de escrever a minha própria implementação de uma barreira com mutexes.

```
class Barrier {
public:
    explicit Barrier(std::size_t iCount) :
      mThreshold(iCount),
      mCount(iCount),
      mGeneration(0) {
    }

    void Wait() {
        std::unique_lock<std::mutex> lLock{mMutex};
        auto lGen = mGeneration;
        if (!--mCount) {
            mGeneration++;
            mCount = mThreshold;
            mCond.notify_all();
        } else {
```

```
            mCond.wait(lLock, [this, lGen] {
                return lGen != mGeneration;
            });
        }
    }

private:
    std::mutex mMutex;
    std::condition_variable mCond;
    std::size_t mThreshold;
    std::size_t mCount;
    std::size_t mGeneration;
};
```

4 Pontas de proba USDT

De forma a marcar fins de iterações e obter informações sobre os programas a serem utilizados, foram implementadas algumas prontas de proba definidas pelo utilizador, as quais foram incluídas no programa:

- matrix_generation(int): Retornada quando se inicia a geração das matrizes. Retorna o tamanho das matrizes que serão usadas.
- start_calc(): Retornada quando se acaba a geração e alocação das matrizes e quando se iniciam os cálculos em si.
- start_iteration(): Retornada quando se inicia uma iteração do cálculo da dispersão de calor.
- start_copy(): Retornada quando se acabam os cálculos da dispersão de calor e se está a copiar os dados de uma matriz para a outra de forma a se prosseguir para a próxima iteração.
- end_iteration(int): Retornada no fim de cada iteração. Como resultado de retorno é enviada a iteração corrente.
- end_calc(): Retornada no fim de todas as iterações do programa.

De forma a garantir que não são retornadas mais pontas de proba USDT do que o ideal, definiu-se que apenas o master thread (o thread com ranque 0) poderá ativar a ponta de proba. Esta ativação apenas será realizada diretamente a seguir às barreiras implementadas, de tal forma que exista o mínimo de disparidade entre os resultados.

5 D script

Para interpretação das pontas de proba foi escrito um D script que gera texto baseado nas pontas de proba descritas acima, além de outras provenientes dos diversos providers.

Para todas as pontas de proba que não são USDT colocou-se uma cláusula referente ao execname do programa, para garantir que a ponta de proba é ativada pelo mesmo, como por exemplo:

```
/execname = "pthreads"/
```

Em geral, nas probes USDT definidas guarda-se o tempo ao qual elas foram disparadas de forma a se poder gerar médias, máximos e mínimos em relação a tempos de iteração, por exemplo.

Em relação a sondas pré-definidas:

- **syscall::open*:entry** - É impresso o caminho do ficheiro que foi aberto pela aplicação. É utilizado normlamente para observar o caminho do ficheiro no qual se guarda o resultado da matriz final.
- **syscall::pwrite*:entry** - É impresso o caminho do ficheiro no qual se começou a escrever.
- **syscall::pwrite*:return** - É impresso o caminho do ficheiro quando este é fechado.
- **sched:::on-cpu** - Imprime quando um thread começou a correr.
- **sched:::off-cpu** - Imprime quando um thread parou de correr.
- **sched:::sleep** - Guarda numa estrutura o tempo em que cada thread adormece.
- **sched:::wakeup** - Utiliza o valor obtido anteriormente por sched:::sleep de forma a medir quanto tempo os threads estiveram a dormir.
- **lockstat:::adaptive-block** - Guarda quantas vezes os threads foram bloqueados por barreiras etc.
- **proc:::exec** - É impresso o pid de um processo quando este inicia a sua execução.
- **proc:::exec-failure** - É impresso o pid de um processo quando este termina sem sucesso a sua execução.
- **proc:::exec-success** - É impresso o pid de um processo quando este termina com sucesso a sua execução.

A sonda dtrace:::END imprime no final da execução do programa os dados referentes à execução do programa, os quais foram guardados pelas sondas anteriormente. Estes resultados podem ser vistos na seguinte probe:

```
dtrace:::END
{
printf("---------------------- Final Report ----------------------\n");
printf("Generated Matrices with size:           %dx%d\n",m_size,m_size);
printf("Time spent generating matrices:         %d\n",m_gen_time);
printf("Time spent running the main algorithm:  %d\n",alg_time);
printf("Iteration time:\n");
printa("    Average:                            %@d\n",@avg_it_time);
printa("    Maximum:                            %@d\n",@max_it_time);
printf("Calculation time:\n");
printa("    Average:                            %@d\n",@avg_calc_time);
printa("    Maximum:                            %@d\n",@max_calc_time);
printf("Copying time:\n");
printa("    Average:                            %@d\n",@avg_copy_time);
printa("    Maximum:                            %@d\n",@max_copy_time);
printa("Total number of threads locked:         %@d\n",@blocks);
printa("Time spent sleeping by thread %d        %@d\n",@sleep);
}
```

6 Resultados Obtidos

Como nota antes de mostrar os resultados obtidos é de salientar que a máquina Solaris disponibilizada foi partilhada pelos vários alunos e desta maneira os resultados obtidos em relação a tempos de execução não são necessáriamente corretos ou viáveis.

6.1 Implementação sequencial

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Program stopped calculating
Opened the matrix file: /dev/dtrace/helper
---------------------- Final Report ----------------------
Generated Matrices with size:            1024x1024
Time spent generating matrices:          58667157
Time spent running the main algorithm:   4669361532
Iteration time:
    Average:                             4646320
    Maximum:                             23738109
Calculation time:
    Average:                             3341146
    Maximum:                             21251183
Copying time:
    Average:                             1305173
    Maximum:                             2486926
Time spent sleeping by thread 1         5985298640
```

6.2 Implementação OpenMP

6.2.1 2 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Time running: 3.374516
Program stopped calculating
Opened the matrix file: /dev/dtrace/helper
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         266562042
Time spent running the main algorithm:  3374472491
Iteration time:
    Average:                            3203098
    Maximum:                            153854472
Calculation time:
    Average:                            2123330
    Maximum:                            99820044
Copying time:
    Average:                            1079768
    Maximum:                            151743270
Total number of threads locked:         7
Time spent sleeping by thread 2        1499299485
Time spent sleeping by thread 1        3748823173
```

6.2.2 4 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Program stopped calculating
Press ENTER to start the program: Time running: 3.628665
Opened the matrix file: /dev/dtrace/helper
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         25847052
Time spent running the main algorithm:  3628511571
Iteration time:
    Average:                            2605295
    Maximum:                            111157740
Calculation time:
    Average:                            1802231
    Maximum:                            109995746
```

```
Copying time:
    Average:                               803063
    Maximum:                               38741696
Total number of threads locked:            18
Time spent sleeping by thread 1           2115218981
Time spent sleeping by thread 3           2181612902
Time spent sleeping by thread 2           2284874073
Time spent sleeping by thread 4           2322364484
```

6.2.3 8 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Program stopped calculating
Press ENTER to start the program: Time running: 2.840380
Opened the matrix file: /dev/dtrace/helper
---------------------- Final Report ----------------------
Generated Matrices with size:              1024x1024
Time spent generating matrices:            80519955
Time spent running the main algorithm:     2840337993
Iteration time:
    Average:                               1843545
    Maximum:                               21369525
Calculation time:
    Average:                               1062121
    Maximum:                               11639497
Copying time:
    Average:                               781423
    Maximum:                               18660511
Total number of threads locked:            15
Time spent sleeping by thread 7           1972132727
Time spent sleeping by thread 3           1984181907
Time spent sleeping by thread 6           1991912581
Time spent sleeping by thread 2           1992744452
Time spent sleeping by thread 1           2015936313
Time spent sleeping by thread 8           2023470676
Time spent sleeping by thread 5           2069151421
Time spent sleeping by thread 4           2267338987
```

6.3 Implementação MPI

Infelizmente não me foi possível correr a versão MPI com sucessos, pois estava a obter um erro que não consegui resolver a tempo da entrega do relatório.

6.4 Implementação com PThreads

6.4.1 2 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Opened the matrix file: /dev/dtrace/helper
Program stopped calculating
Opened the matrix file: /usr/lib/locale/en_US.UTF-8/LC_MESSAGES/solaris_linkers.mo
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         56728854
Time spent running the main algorithm:  2601083594
Iteration time:
    Average:                            2583421
    Maximum:                            47259391
Calculation time:
    Average:                            1836347
    Maximum:                            45679083
Copying time:
    Average:                            747074
    Maximum:                            2292989
Total number of threads locked:         1
Time spent sleeping by thread 3        631840690
Time spent sleeping by thread 2        982669178
Time spent sleeping by thread 1        9702239149
```

6.4.2 4 threads

```
Tracer is ready!
Opened the matrix file: result.txt
Opened the matrix file: /dev/dtrace/helper
Press ENTER to start the program: Opened the matrix file: /usr/lib/locale/en_US.UTF-
Opened the matrix file: /dev/dtrace/helper
Program stopped calculating
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         135345753
Time spent running the main algorithm:  2377443323
Iteration time:
    Average:                            2360002
    Maximum:                            137274737
Calculation time:
    Average:                            1423624
```

```
    Maximum:                                112654588
Copying time:
    Average:                                936377
    Maximum:                                50734609
Total number of threads locked:             3
Time spent sleeping by thread 1            66865
Time spent sleeping by thread 5            495512152
Time spent sleeping by thread 4            562690185
Time spent sleeping by thread 3            581875649
Time spent sleeping by thread 2            745362976
```

6.4.3 8 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Opened the matrix file: /usr/lib/locale/en_US.UTF-
Opened the matrix file: /dev/dtrace/helper
Program stopped calculating
---------------------- Final Report ----------------------
Generated Matrices with size:               1024x1024
Time spent generating matrices:             80505414
Time spent running the main algorithm:      2804115343
Iteration time:
    Average:                                2779589
    Maximum:                                162960841
Calculation time:
    Average:                                1413562
    Maximum:                                123956787
Copying time:
    Average:                                1366026
    Maximum:                                162335914
Total number of threads locked:             5
Time spent sleeping by thread 8            320962189
Time spent sleeping by thread 9            448724426
Time spent sleeping by thread 5            538287898
Time spent sleeping by thread 2            549681876
Time spent sleeping by thread 4            575055364
Time spent sleeping by thread 3            627603164
Time spent sleeping by thread 6            713475331
Time spent sleeping by thread 7            921782004
Time spent sleeping by thread 1            3267530171
```

6.5 Implementação em C++11

6.5.1 2 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Program stopped calculating
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: /usr/lib/locale/en_US.UTF-8/LC_MESSAGES/solaris_linkers.mo
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         332075828
Time spent running the main algorithm:  2984016980
Iteration time:
    Average:                            2949100
    Maximum:                            156551851
Calculation time:
    Average:                            1891426
    Maximum:                            33125169
Copying time:
    Average:                            1057673
    Maximum:                            153113578
Total number of threads locked:         4
Time spent sleeping by thread 3        821964159
Time spent sleeping by thread 2        1135193402
Time spent sleeping by thread 1        4299053232
```

6.5.2 4 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Program stopped calculating
Opened the matrix file: /usr/lib/locale/en_US.UTF-8/LC_MESSAGES/solaris_linkers.mo
Press ENTER to start the program: Opened the matrix file: /dev/dtrace/helper
---------------------- Final Report ----------------------
Generated Matrices with size:           1024x1024
Time spent generating matrices:         46885006
Time spent running the main algorithm:  2744923028
Iteration time:
    Average:                            2728525
    Maximum:                            194461527
Calculation time:
    Average:                            1885908
```

```
    Maximum:                              193656004
Copying time:
    Average:                              842616
    Maximum:                              101989370
Total number of threads locked:           5
Time spent sleeping by thread 1          46001
Time spent sleeping by thread 5          859894303
Time spent sleeping by thread 3          1122221533
Time spent sleeping by thread 4          1193721878
Time spent sleeping by thread 2          1359281778
```

6.5.3 8 threads

```
Tracer is ready!
Opened the matrix file: /dev/dtrace/helper
Opened the matrix file: result.txt
Press ENTER to start the program: Opened the matrix file: /usr/lib/locale/en_US.UTF-
Opened the matrix file: /dev/dtrace/helper
Program stopped calculating
---------------------- Final Report ----------------------
Generated Matrices with size:             1024x1024
Time spent generating matrices:           34491684
Time spent running the main algorithm:    2541628137
Iteration time:
    Average:                              2520711
    Maximum:                              152327165
Calculation time:
    Average:                              1192145
    Maximum:                              80788941
Copying time:
    Average:                              1328566
    Maximum:                              151038845
Total number of threads locked:           13
Time spent sleeping by thread 1          246946368
Time spent sleeping by thread 3          1468760810
Time spent sleeping by thread 8          1549090749
Time spent sleeping by thread 7          1610508844
Time spent sleeping by thread 4          1616914297
Time spent sleeping by thread 5          1643953608
Time spent sleeping by thread 6          1665085987
Time spent sleeping by thread 2          1768867876
Time spent sleeping by thread 9          1776747141
```

7 Anexos

7.1 Implementação sequencial

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "heattimer.h"

#define N_MAX 1000
#define MAT_SIZE 1024
#define M_SIZE MAT_SIZE + 2

int main()
{

    printf("Press ENTER to start the program: ");
    scanf("*");

    FILE *file = fopen("result.txt", "w+");

    if(HEATTIMER_QUERY_MATRIX_GENERATION_ENABLED())
        HEATTIMER_QUERY_MATRIX_GENERATION(MAT_SIZE);

    int **G1, **G2;

    G1 = (int **)malloc(sizeof(int *) * M_SIZE);
    G2 = (int **)malloc(sizeof(int *) * M_SIZE);

    for (int i = 0; i < M_SIZE; i++)
    {
        G1[i] = (int *)malloc(sizeof(int) * M_SIZE);
        G2[i] = (int *)malloc(sizeof(int) * M_SIZE);
    }

    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
        {
            G1[i][j] = 0;
            G2[i][j] = 0;
        }
    }
```

```
//Filling the lower line of the matrix with the highest heat
for (int i = 0; i < M_SIZE; i++)
{
    G1[i][0] = 0xffffff; //Hexcode ffffff
}

if(HEATTIMER_QUERY_START_CALC_ENABLED())
    HEATTIMER_QUERY_START_CALC();

for (int it = 0; it < N_MAX; it++)
{
    if(HEATTIMER_QUERY_START_ITERATION_ENABLED())
        HEATTIMER_QUERY_START_ITERATION();

    for (int i = 1; i < M_SIZE - 1; i++)
    {
        for (int j = 1; j < M_SIZE - 1; j++)
        {
            G2[i][j] = (G1[i - 1][j] + G1[i + 1][j] + G1[i][j - 1] +
        }
    }

    if(HEATTIMER_QUERY_START_COPY_ENABLED())
        HEATTIMER_QUERY_START_COPY();

    //Copiar G2 para G1
    for (int i = 1; i < M_SIZE - 1; i++)
    {
        for (int j = 1; j < M_SIZE - 1; j++)
        {
            G1[i][j] = G2[i][j];
        }
    }

    if(HEATTIMER_QUERY_END_ITERATION_ENABLED())
        HEATTIMER_QUERY_END_ITERATION(it);
}

if(HEATTIMER_QUERY_END_CALC_ENABLED())
    HEATTIMER_QUERY_END_CALC();

//Prints results to a file
for (int i = 0; i < M_SIZE; i++)
{
```

```
        for (int j = 0; j < M_SIZE; j++)
            fprintf(file, "%d|", G1[i][j]);
    }
    fprintf(file, "\n");
}
```

7.2 Implementação OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include "heattimer.h"

#define N_MAX 1000
#define N_THREADS 8
#define MAT_SIZE 1024
#define M_SIZE MAT_SIZE + 2

int main()
{
    printf("Press ENTER to start the program: ");
    scanf("*");

    FILE *file = fopen("result.txt", "w+");

    if(HEATTIMER_QUERY_MATRIX_GENERATION_ENABLED())
        HEATTIMER_QUERY_MATRIX_GENERATION(MAT_SIZE);

    int **G1, **G2;

    G1 = (int **)malloc(sizeof(int *) * M_SIZE);
    G2 = (int **)malloc(sizeof(int *) * M_SIZE);

    for (int i = 0; i < M_SIZE; i++)
    {
        G1[i] = (int *)malloc(sizeof(int) * M_SIZE);
        G2[i] = (int *)malloc(sizeof(int) * M_SIZE);
    }

    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
        {
            G1[i][j] = 0;
            G2[i][j] = 0;
        }
    }

    //Filling the lower line of the matrix with the highest heat
```

```
for (int i = 0; i < M_SIZE; i++)
{
    G1[i][0] = 0xffffff; //Hexcode ffffff
}

double start_time = omp_get_wtime();

omp_set_num_threads(N_THREADS);

if(HEATTIMER_QUERY_START_CALC_ENABLED())
    HEATTIMER_QUERY_START_CALC();

for (int it = 0; it < N_MAX; it++)
{

    #pragma omp parallel
    {
            #pragma omp master
        {
            if(HEATTIMER_QUERY_START_ITERATION_ENABLED())
                HEATTIMER_QUERY_START_ITERATION();
        }

        #pragma omp for schedule(static)
        for (int i = 1; i < M_SIZE - 1; i++)
        {
            for (int j = 1; j < M_SIZE - 1; j++)
            {
                G2[i][j] = (G1[i - 1][j] + G1[i + 1][j] + G1[i][j - 1
            }
        }

        #pragma omp master
        {
            if(HEATTIMER_QUERY_START_COPY_ENABLED())
                HEATTIMER_QUERY_START_COPY();
        }

        //Copiar G2 para G1
        #pragma omp for schedule(static)
        for (int i = 1; i < M_SIZE - 1; i++)
        {
            for (int j = 1; j < M_SIZE - 1; j++)
            {
```

```
                    G1[i][j] = G2[i][j];
                }
            }

            #pragma omp master
                {
                if(HEATTIMER_QUERY_END_ITERATION_ENABLED())
                    HEATTIMER_QUERY_END_ITERATION(it);
            }
        }
    }

    if(HEATTIMER_QUERY_END_CALC_ENABLED())
        HEATTIMER_QUERY_END_CALC();

    double end_time = omp_get_wtime();

    printf("Time running: %lf\n", end_time - start_time);

    //Prints results to a file
    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
            fprintf(file, "%d|", G1[i][j]);
    }
    fprintf(file, "\n");
}
```

7.3 Implentação MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include "heattimer.h"

#define N_MAX 1000
#define N_MACHINES 8
#define MAT_SIZE 1024
#define M_SIZE (MAT_SIZE + 2)

int main(int argc, char *argv[])
{
    printf("Press ENTER to start the program: ");
    scanf("*");

    FILE *file = fopen("result.txt", "w+");

    int rank;
    int i_division = MAT_SIZE / N_MACHINES;
    int MACH_MAT_SIZE = i_division * M_SIZE;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double start_time = MPI_Wtime();
    int left_send_buffer[M_SIZE], right_send_buffer[M_SIZE], left_recv_bu
    int *final_send_buffer = (int *)malloc(sizeof(int) * M_SIZE * i_divis
    int *final_result_buffer = (int *)malloc(sizeof(int) * M_SIZE * MAT_S
    if (final_send_buffer == NULL || final_result_buffer == NULL)
    {
        printf("NULL, not enough memory\n");
    }
    MPI_Request left_send_request, right_send_request, left_recv_request,

    if(rank == 0){
        if(HEATTIMER_QUERY_MATRIX_GENERATION_ENABLED())
            HEATTIMER_QUERY_MATRIX_GENERATION(MAT_SIZE);
    }

    int **G1 = (int **)malloc(sizeof(int *) * i_division);
    int **G2 = (int **)malloc(sizeof(int *) * i_division);
    for (int i = 0; i < i_division; i++)
```

```
{
    G1[i] = (int *)malloc(sizeof(int) * M_SIZE);
    G2[i] = (int *)malloc(sizeof(int) * M_SIZE);

    for (int j = 0; j < M_SIZE; j++)
    {
        G1[i][j] = 0;
        G2[i][j] = 0;
    }

    G1[i][0] = 0xffffff; //Hexcode ffffff
    G2[i][0] = 0xffffff;
}

for (int j = 0; j < M_SIZE; j++)
{
    left_recv_buffer[j] = 0;
    right_recv_buffer[j] = 0;
}

if(rank == 0){
    if(HEATTIMER_QUERY_START_CALC_ENABLED())
        HEATTIMER_QUERY_START_CALC();
}
for (int it = 0; it < N_MAX; it++)
{

    if(HEATTIMER_QUERY_START_ITERATION_ENABLED())
        HEATTIMER_QUERY_START_ITERATION();

    //Computes the parts with no dependencies
    for (int i = 1; i < i_division - 1; i++)
    {
        for (int j = 1; j < M_SIZE - 1; j++)
        {
            G2[i][j] = (G1[i - 1][j] + G1[i + 1][j] + G1[i][j - 1] +
        }
    }

    //Waits to receive the left buffer
    if (it != 0 && rank != 0)
    {
        MPI_Wait(&left_recv_request, MPI_STATUS_IGNORE);
    }
```

```
for (int j = 1; j < M_SIZE - 1; j++)
{
    G2[0][j] = (left_recv_buffer[j] + G1[1][j] + G1[0][j - 1] + G
}

//Waits to receive the right buffer
if (it != 0 && rank != N_MACHINES - 1)
{
    MPI_Wait(&right_recv_request, MPI_STATUS_IGNORE);
}
for (int j = 1; j < M_SIZE - 1; j++)
{
    G2[i_division - 1][j] = (G1[i_division - 2][j] + right_recv_b
}

//Guarantees the buffers have been sent
if (it != 0)
{
    if (rank != 0)
    {
        MPI_Wait(&left_send_request, MPI_STATUS_IGNORE);
    }
    if (rank != N_MACHINES - 1)
    {
        MPI_Wait(&right_send_request, MPI_STATUS_IGNORE);
    }
}
//Copies the column to the buffer
for (int j = 0; j < M_SIZE; j++)
{
    if (rank != 0)
        left_send_buffer[j] = G1[0][j];
    if (rank != N_MACHINES - 1)
        right_send_buffer[j] = G1[i_division - 1][j];
}

//Sends and receives assynchonously the buffers
if (rank != 0)
{
    MPI_Isend(left_send_buffer, M_SIZE, MPI_INT, rank - 1, 0, MPI
    MPI_Irecv(left_recv_buffer, M_SIZE, MPI_INT, rank - 1, 0, MPI
}
if (rank != N_MACHINES - 1)
{
```

```
        MPI_Isend(right_send_buffer, M_SIZE, MPI_INT, rank + 1, 0, MP
        MPI_Irecv(right_recv_buffer, M_SIZE, MPI_INT, rank + 1, 0, MP
    }

    if(HEATTIMER_QUERY_START_COPY_ENABLED())
        HEATTIMER_QUERY_START_COPY();

    //Copies from G2 to G1
    for (int i = 0; i < i_division; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
        {
            G1[i][j] = G2[i][j];
        }
    }

    if(HEATTIMER_QUERY_END_ITERATION_ENABLED())
        HEATTIMER_QUERY_END_ITERATION(it);
}

for (int i = 0; i < i_division; i++)
{
    for (int j = 0; j < M_SIZE; j++)
    {
        final_send_buffer[i * M_SIZE + j] = G1[i][j];
    }
}

//Rank 0 gathers results from all other ranks
//int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
MPI_Gather(final_send_buffer, MACH_MAT_SIZE, MPI_INT, final_result_bu



if (rank == 0)
{
    if(HEATTIMER_QUERY_END_CALC_ENABLED())
        HEATTIMER_QUERY_END_CALC();

    //Creates the final matrix to output the result
    int **FINAL_MAT = (int **)malloc(sizeof(int *) * M_SIZE);
    for (int i = 0; i < M_SIZE; i++)
    {
        FINAL_MAT[i] = (int *)malloc(sizeof(int) * M_SIZE);
```

```
        }
        for (int j = 0; j < M_SIZE; j++)
        {
            FINAL_MAT[0][j] = 0;
            FINAL_MAT[M_SIZE - 1][j] = 0;
        }
        FINAL_MAT[0][0] = 0xffffff;
        FINAL_MAT[M_SIZE - 1][0] = 0xffffff;

        //Copies the result from the receive buffer to the result matrix
        for (int i = 0; i < MAT_SIZE; i++)
        {
            for (int j = 0; j < M_SIZE; j++)
            {
                FINAL_MAT[i + 1][j] = final_result_buffer[i * M_SIZE + j]
            }
        }

        double end_time = MPI_Wtime();

        printf("Total time: %lf seconds\n", end_time - start_time);

        //Prints results to a file
        for (int i = 0; i < M_SIZE; i++)
        {
            for (int j = 0; j < M_SIZE; j++)
            {
                fprintf(file, "%d|", FINAL_MAT[i][j]);
            }
            fprintf(file, "\n");
        }
    }

    MPI_Finalize();
}
```

7.4 Implementação com PThreads

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <stdint.h>
#include "heattimer.h"

#define N_MAX 1000
#define MAT_SIZE 1024
#define M_SIZE MAT_SIZE + 2
#define N_THREADS 8
#define M_DIV MAT_SIZE / N_THREADS

int **G1, **G2;
pthread_barrier_t barrier;

void *heat_dispersion(void* tnum_void){
    int tnum = (intptr_t) tnum_void;
    for (int it = 0; it < N_MAX; it++)
    {

        if(tnum == 0)
            if(HEATTIMER_QUERY_START_ITERATION_ENABLED())
                HEATTIMER_QUERY_START_ITERATION();

        for (int i = (tnum * M_DIV) + 1; i < ((tnum + 1) * M_DIV) + 1; i+
        {
            for (int j = 1; j < M_SIZE - 1; j++)
            {
                G2[i][j] = (G1[i - 1][j] + G1[i + 1][j] + G1[i][j - 1] +
            }
        }

        pthread_barrier_wait(&barrier);

        if(tnum == 0)
            if(HEATTIMER_QUERY_START_COPY_ENABLED())
                    HEATTIMER_QUERY_START_COPY();

        for (int i = (tnum * M_DIV) + 1; i < ((tnum + 1) * M_DIV) + 1; i+
        {
            for (int j = 1; j < M_SIZE - 1; j++)
```

```
            {
                G1[i][j] = G2[i][j];
            }
        }

        pthread_barrier_wait(&barrier);

        if(tnum == 0)
            if(HEATTIMER_QUERY_END_ITERATION_ENABLED())
                HEATTIMER_QUERY_END_ITERATION(it);
    }
}

int main()
{

    printf("Press ENTER to start the program: ");
    scanf("*");

    FILE *file = fopen("result.txt", "w+");

    if(HEATTIMER_QUERY_MATRIX_GENERATION_ENABLED())
        HEATTIMER_QUERY_MATRIX_GENERATION(MAT_SIZE);

    G1 = (int **)malloc(sizeof(int *) * M_SIZE);
    G2 = (int **)malloc(sizeof(int *) * M_SIZE);

    for (int i = 0; i < M_SIZE; i++)
    {
        G1[i] = (int *)malloc(sizeof(int) * M_SIZE);
        G2[i] = (int *)malloc(sizeof(int) * M_SIZE);
    }

    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
        {
            G1[i][j] = 0;
            G2[i][j] = 0;
        }
    }

    for (int i = 0; i < M_SIZE; i++)
    {
```

```
        G1[i][0] = 0xffffff; //Hexcode ffffff
    }

    pthread_t* thread_handles = (pthread_t*) malloc(N_THREADS * sizeof(pt
    pthread_barrier_init(&barrier, (pthread_barrierattr_t*) NULL, N_THREA

    if(HEATTIMER_QUERY_START_CALC_ENABLED())
        HEATTIMER_QUERY_START_CALC();

    for(int thread = 0; thread < N_THREADS; thread++){
        pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
                      heat_dispersion, (void*) (intptr_t) thread);
    }

    for(int thread = 0; thread < N_THREADS; thread++){
        pthread_join(thread_handles[thread],NULL);
    }

    free(thread_handles);
    pthread_barrier_destroy(&barrier);

    if(HEATTIMER_QUERY_END_CALC_ENABLED())
        HEATTIMER_QUERY_END_CALC();

    //Prints results to a file
    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
            fprintf(file, "%d|", G1[i][j]);
    }
    fprintf(file, "\n");
}
```

7.5 Implementação em C++11

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <condition_variable>
#include "heattimer.h"

#define N_MAX 1000
#define MAT_SIZE 1024
#define M_SIZE MAT_SIZE + 2
#define N_THREADS 8
#define M_DIV MAT_SIZE / N_THREADS

class Barrier {
public:
    explicit Barrier(std::size_t iCount) :
      mThreshold(iCount),
      mCount(iCount),
      mGeneration(0) {
    }

    void Wait() {
        std::unique_lock<std::mutex> lLock{mMutex};
        auto lGen = mGeneration;
        if (!--mCount) {
            mGeneration++;
            mCount = mThreshold;
            mCond.notify_all();
        } else {
            mCond.wait(lLock, [this, lGen]
            { return lGen != mGeneration; });
        }
    }

private:
    std::mutex mMutex;
    std::condition_variable mCond;
    std::size_t mThreshold;
    std::size_t mCount;
    std::size_t mGeneration;
};
```

```
void heat_dispersion(int tnum, int** G1, int** G2, Barrier *br){
    for (int it = 0; it < N_MAX; it++)
    {
        if(tnum == 0)
            if(HEATTIMER_QUERY_START_ITERATION_ENABLED())
                HEATTIMER_QUERY_START_ITERATION();

        for (int i = (tnum * M_DIV) + 1; i < ((tnum + 1) * M_DIV) + 1; i+
        {
            for (int j = 1; j < M_SIZE - 1; j++)
            {
                G2[i][j] = (G1[i - 1][j] + G1[i + 1][j] + G1[i][j - 1] +
            }
        }

        br->Wait();

        if(tnum == 0)
            if(HEATTIMER_QUERY_START_COPY_ENABLED())
                    HEATTIMER_QUERY_START_COPY();

        for (int i = (tnum * M_DIV) + 1; i < ((tnum + 1) * M_DIV) + 1; i+
        {
            for (int j = 1; j < M_SIZE - 1; j++)
            {
                G1[i][j] = G2[i][j];
            }
        }

        br->Wait();

        if(tnum == 0)
            if(HEATTIMER_QUERY_END_ITERATION_ENABLED())
                    HEATTIMER_QUERY_END_ITERATION(it);
    }
}

int main()
{
    char c{};
    std::cout << "Press ENTER to start the program: ";
    scanf("*");
```

```
FILE *file = fopen("result.txt", "w+");

if(HEATTIMER_QUERY_MATRIX_GENERATION_ENABLED())
    HEATTIMER_QUERY_MATRIX_GENERATION(MAT_SIZE);

int **G1, **G2;

G1 = (int **)malloc(sizeof(int *) * M_SIZE);
G2 = (int **)malloc(sizeof(int *) * M_SIZE);

for (int i = 0; i < M_SIZE; i++)
{
    G1[i] = (int *)malloc(sizeof(int) * M_SIZE);
    G2[i] = (int *)malloc(sizeof(int) * M_SIZE);
}

for (int i = 0; i < M_SIZE; i++)
{
    for (int j = 0; j < M_SIZE; j++)
    {
        G1[i][j] = 0;
        G2[i][j] = 0;
    }
}

//Filling the lower line of the matrix with the highest heat
for (int i = 0; i < M_SIZE; i++)
{
    G1[i][0] = 0xffffff; //Hexcode ffffff
}

std::thread threads[N_THREADS];

Barrier br(N_THREADS);

if(HEATTIMER_QUERY_START_CALC_ENABLED())
    HEATTIMER_QUERY_START_CALC();

for(int i = 0; i < N_THREADS; i++){
    threads[i] = std::thread(heat_dispersion,i,G1,G2,&br);
}

for(int i = 0; i < N_THREADS; i++){
    threads[i].join();
```

```
    }

    if(HEATTIMER_QUERY_END_CALC_ENABLED())
        HEATTIMER_QUERY_END_CALC();

    //Prints results to a file
    for (int i = 0; i < M_SIZE; i++)
    {
        for (int j = 0; j < M_SIZE; j++)
            fprintf(file, "%d|", G1[i][j]);
    }
    fprintf(file, "\n");
}
```

7.6 DScript (Exemplo para todos os programas com exceção da versão MPI)

```
#!/usr/sbin/dtrace -qs

uint64_t m_size;
uint64_t m_gen_time;
uint64_t alg_time;
uint64_t sleep_time[id_t];

self int iteration_start;
self int copy_start;
self string write_path;
self int asleep;

this int it_time;
this int copy_time;
this int calc_time;

dtrace:::BEGIN
{
    printf("Tracer is ready!\n");
}

heattimer*:::query-matrix_generation
{
    m_gen_time = timestamp;
    m_size = arg0;
}

heattimer*:::query-start_calc
{
    m_gen_time = timestamp - m_gen_time;
    alg_time = timestamp;
}

heattimer*:::query-start_iteration
{
    self->iteration_start = timestamp;
}

heattimer*:::query-start_copy
{
    self->copy_start = timestamp;
```

```
}

heattimer*:::query-end_iteration
{
    this->it_time = timestamp - self->iteration_start;
    this->copy_time = timestamp - self->copy_start;
    this->calc_time = this->it_time - this->copy_time;
    /*printf("Iteration %d finished on PROCESS: %d, THREAD: %d\n\tTime spent on calc
           arg0,
       pid,
           tid,
           this->calc_time,
           this->copy_time,
           this->it_time);*/

    @avg_calc_time = avg(this->calc_time);
    @max_calc_time = max(this->calc_time);
    @avg_copy_time = avg(this->copy_time);
    @max_copy_time = max(this->copy_time);
    @avg_it_time = avg(this->it_time);
    @max_it_time = max(this->it_time);
}

heattimer*:::query-end_calc
{
    printf("Program stopped calculating\n");
    alg_time = timestamp - alg_time;
}

syscall::open*:entry
/execname == "pthreads"/
{
    self->open_path = copyinstr(arg1);
    printf("Opened the matrix file: %s\n",self->open_path);
}

syscall::pwrite*:entry
/execname == "pthreads"/
{
    self->write_path = copyinstr(arg1);
    printf("Started writing in file: %s\n",self->write_path);
}

syscall::pwrite*:return
```

```
/execname == "pthreads"/
{
    printf("Finished writing in file: %s\n",self->write_path);
}

sched:::on-cpu
/execname == "pthreads"/
{
    /*printf("Thread %d started running\n",tid);*/
}

sched:::off-cpu
/execname == "pthreads"/
{
    /*printf("Thread %d stopped running\n",tid);*/
}

sched:::sleep
/execname == "pthreads"/
{
    sleep_time[tid] = timestamp;
}

sched:::wakeup
/execname == "pthreads" && sleep_time[tid] != 0/
{
    @sleep[tid] = sum(timestamp - sleep_time[tid]);
    sleep_time[tid] = 0;
}

lockstat:::adaptive-block
/execname == "pthreads"/
{
    @blocks = count();
}

proc:::exec
/execname == "pthreads"/
{
    printf("Process %d started executing\n",pid);
}

proc:::exec-failure
/execname == "pthreads"/
```

```
{
    printf("Process %d exectued unsuccessfully\n",pid);
}

proc:::exec-success
/execname == "pthreads"/
{
    printf("Process %d executed correctly\n",pid);
}


dtrace:::END
{
    printf("---------------------- Final Report ----------------------\n");
    printf("Generated Matrices with size:            %dx%d\n",m_size,m_size);
    printf("Time spent generating matrices:          %d\n",m_gen_time);
    printf("Time spent running the main algorithm:   %d\n",alg_time);
    printf("Iteration time:\n");
    printa("    Average:                             %@d\n",@avg_it_time);
    printa("    Maximum:                             %@d\n",@max_it_time);
    printf("Calculation time:\n");
    printa("    Average:                             %@d\n",@avg_calc_time);
    printa("    Maximum:                             %@d\n",@max_calc_time);
    printf("Copying time:\n");
    printa("    Average:                             %@d\n",@avg_copy_time);
    printa("    Maximum:                             %@d\n",@max_copy_time);
    printa("Total number of threads locked:          %@d\n",@blocks);
    printa("Time spent sleeping by thread %d         %@d\n",@sleep);
}
```