Grupo 4 - Tiago Koji Castro Shibata (8988730), Adilson Torres Gregório de Souza (8988254)

Sistemas de Computação de Alto Desempenho

Aula 9

Relatório:
MPI

- **Introdução**

Em aula, estudamos o uso de MPI para sistemas distribuidos.

- **Configuração do sistema**

Testes foram realizados em um processador Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, com 4 núcleos. O sistema utilizado é Linux e os processos foram executados com *niceness* mínimo para que outros processos do sistema interferissem menos nos resultados.

Todos os programas foram compilados com *-O3 -DNDEBUG -Wall -Wextra -pedantic*.

- **Programas**

No **exercício 1**, implementamos uma versão com MPI para obter o valor máximo contido em um vetor de inteiros, onde o processo com rank=0 inicializa o vetor (inicializar com valores inteiros gerados randomicamente). Para análise comparativa foi realizada uma versão sequencial do problema.

Versão sequencial:

```c
/*
Valor máximo contido em um vetor.
 */
#include <stdio.h>
#include <time.h>

main(argc, argv)

int  argc;
char  *argv[];

{
    int vetor[100000],i,n;
    int max, max_parcial, max_total;
    struct timespec start_time, end_time;

    clock_gettime(CLOCK_MONOTONIC, &start_time);
    n=100000;

    for(i=0;i<n;i++)
      vetor[i]=rand();

    max_total = 0;
```

```
    for(i=0;(i<n);i++) {
      if (vetor[i] > max_total) {
        max_total = vetor[i];
      }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("RESULTADO=%d \nTime: %.5fms\n",max_total,
    (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec * 1000 +
1e-6 * start_time.tv_nsec));
    fflush(stdout);
       return(0);
}
```

Versão com MPI:

```
/*
Valor máximo contido em um vetor usando MPI_Send.
Numero de elementos do vetor: multiplo do número de processos.
 */

#include <mpi.h>
#include <stdio.h>
#include <time.h>

main(argc, argv)

int argc;
char *argv[];
{
    int n,n_nos, rank;
    MPI_Status status;
    int inicio,fim,vetor[100000],i,k;
    int max, max_parcial, max_total;
    struct timespec start_time, end_time;
/*
 * Initialize MPI.
 */
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    n=100000;
    k=n/n_nos;
    inicio=rank*k;
    fim=inicio+k;
    if (rank==0) {
      for(i=inicio;i<n;i++)
        vetor[i]=rand();
      for (i=1;i<n_nos;i++)
       {
         MPI_Send(&vetor[k*i],k,MPI_INT,i,10,MPI_COMM_WORLD);
         printf("rank=%d APOS SEND\n",rank);
         fflush(stdout);
       }
    }
```

```
    else {
      // sleep(5);
      MPI_Recv(vetor,k,MPI_INT,0,10,MPI_COMM_WORLD,&status);
      printf("rank=%d APOS RECV\n",rank);
      fflush(stdout);
    }
    max_parcial=0;
    for(i=0;(i<k);i++) {
      if (vetor[i] > max_parcial) {
        max_parcial = vetor[i];
      }
    }

    printf("rank=%d valor maximo parcial =%d\n",rank,max_parcial);
    fflush(stdout);
    if (rank==0) {
       max_total=max_parcial;
       for(i=1;i<n_nos;i++){
         MPI_Recv(&max,1,MPI_INT,MPI_ANY_SOURCE,11,MPI_COMM_WORLD,&status);
         if (max > max_total) {
           max_total = max;
         }
       }

       clock_gettime(CLOCK_MONOTONIC, &end_time);
       printf("RESULTADO=%d \nTime: %.5fms\n",max_total,
       (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));
       fflush(stdout);
       }
    else {
    printf("rank=%d max_parcial=%d\n",rank,max_parcial);
    fflush(stdout);
       MPI_Send(&max_parcial,1,MPI_INT,0,11,MPI_COMM_WORLD);
         }
       MPI_Finalize();
       return(0);
}
```

Na versão com MPI, foi considerado que o número elementos no vetor é múltiplo do números de processos. O processo com rank 0 inicia o vetor com valores aleatórios, e os outros processos recebendo o conjunto parcial do vetor e calcula o máximo parcial do processo, que é mandado para o processo com rank 0, que ao ter todos os máximos parciais verifica qual é o máximo total do vetor.

Tempos medidos:

Sequencial	Distribuido com mpi
1.58044ms	2.58381ms

Pelos tempos obtidos, foi possível perceber que o overhead do mpi para enviar e receber os dados entre os processos influenciou na performance do programa. Isso demonstra que uma das considerações que precisa é considerar o número de mensagens e o custo da operação em cada processo. Nesse problema, por ter um custo simples não compensa distribuir a operação entre diversos processos distribuídos.

No **exercício 2**, implementamos a busca de uma string em uma lista de strings. Trechos da lista de strings foram enviadas para processos filhos. Todas as ocorrências foram retornadas ao processo pai, que mostrou suas posições na tela:

```c
#include <fcntl.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

static char *mmap_file(int fd, size_t length) {
    void *data = mmap(NULL, length, PROT_READ, MAP_PRIVATE | MAP_POPULATE, fd, 0);
    if (data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    return data;
}

static inline int same_sequence(char *a, char *b) {
    for (int i = 0; i < 10; i++) {
        if (a[i] != b[i])
            return 0;
    }
    return 1;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int comm_size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    char *data;
    struct stat st;
    int message_size;
    if (!rank) {
        if (argc != 2) {
            fprintf(stderr, "Usage: %s data_file\n", *argv);
            return 1;
        }
        int fd = open(argv[1], O_RDONLY, 0);
        if (fd == -1) {
            perror("open");
            return 1;
```

```
        }

        fstat(fd, &st);
        data = mmap_file(fd, st.st_size);

        MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
        message_size = st.st_size / comm_size;
        for (int i = 1; i < comm_size; i++) {
            int base = st.st_size * i / comm_size;
            MPI_Send(data + base, message_size, MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_CHAR, &message_size);
        data = malloc(message_size);
        if (!data) {
            perror("malloc");
            return -1;
        }
        MPI_Recv(data, message_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    char sequence[11];
    int matches[128];
    int match_count = 0;
    if (!rank) {
        printf("Sequence to look for:\n");
        scanf("%10s", sequence);
    }
    MPI_Bcast(sequence, 10, MPI_CHAR, 0, MPI_COMM_WORLD);
    for (int i = 0; i < message_size; i += 11) {
        char *base = data + i;
        if (same_sequence(base, sequence)) {
            matches[match_count++] = i / 11;
        }
    }

    if (!rank) {
        for (int i = 0; i < match_count; i++) {
            printf("%d\n", matches[i]);
        }
        for (int i = 0; i < comm_size - 1; i++) {
            // Receive data from slaves
            MPI_Status status;
            MPI_Recv(matches, 128, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
&status);
            MPI_Get_count(&status, MPI_INT, &match_count);
            for (int i = 0; i < match_count; i++) {
                printf("%d\n", matches[i] + status.MPI_SOURCE * message_size / 11);
            }
        }
        if (munmap(data, st.st_size)) {
            perror("munmap");
            return 1;
        }
    } else {
        MPI_Send(matches, match_count, MPI_INT, 0, 1, MPI_COMM_WORLD);
        free(data);
```

```
    }

    MPI_Finalize();
}
```

No programa, vemos:

- O processo 0 (usado por nós como filho) carrega o arquivo, e envia trechos (de tamanho *message_size*) para cada filho. Aqui, assume-se que o número de entradas seja divisível pelo número de processos. No processo pai, o ponteiro **data* indica o início dos dados.
- Processos filhos alocam espaço e salvam o ponteiro em **data*, depois recebem dados.
- O processo pai recebe a sequência a buscar, usando *scanf*. Ela é enviada a todos os processos por broadcast.
- Todos os processos buscam de *data* a *data* + *message_size* pela sequência e guardam as ocorrências (aqui, assume-se até 128 ocorrências por processo).
- Ao fim, o processo pai mostra suas ocorrências e recebe as dos filhos (usando *MPI_ANY_SOURCE*, ou seja, não necessariamente em ordem). Cada um dos filhos envia suas ocorrências com *MPI_Send*.

Testamos a implementação com algumas entradas (texto em azul é entrada do usuário):

```
$ mpirun -np 4 --oversubscribe ./parallel-search seq-teste.txt
Sequence to look for:
cbcbcacaab
62
303
445
2061
7261
8373
3092
4161
5161
5674
9261
11114
11956
```

Comparando com a saída do comando grep:

```
$ grep -n cbcbcacaab seq-teste.txt
```

```
63:cbcbcacaab
304:cbcbcacaab
446:cbcbcacaab
2062:cbcbcacaab
3093:cbcbcacaab
4162:cbcbcacaab
5162:cbcbcacaab
5675:cbcbcacaab
7262:cbcbcacaab
8374:cbcbcacaab
9262:cbcbcacaab
11115:cbcbcacaab
11957:cbcbcacaab
```

Vemos que os resultados do nosso programa paralelo estão corretos (exceto que nosso programa retorna índices das strings, iniciados em 0, conforme o enunciado, enquanto grep retorna números de linhas iniciados em 1).