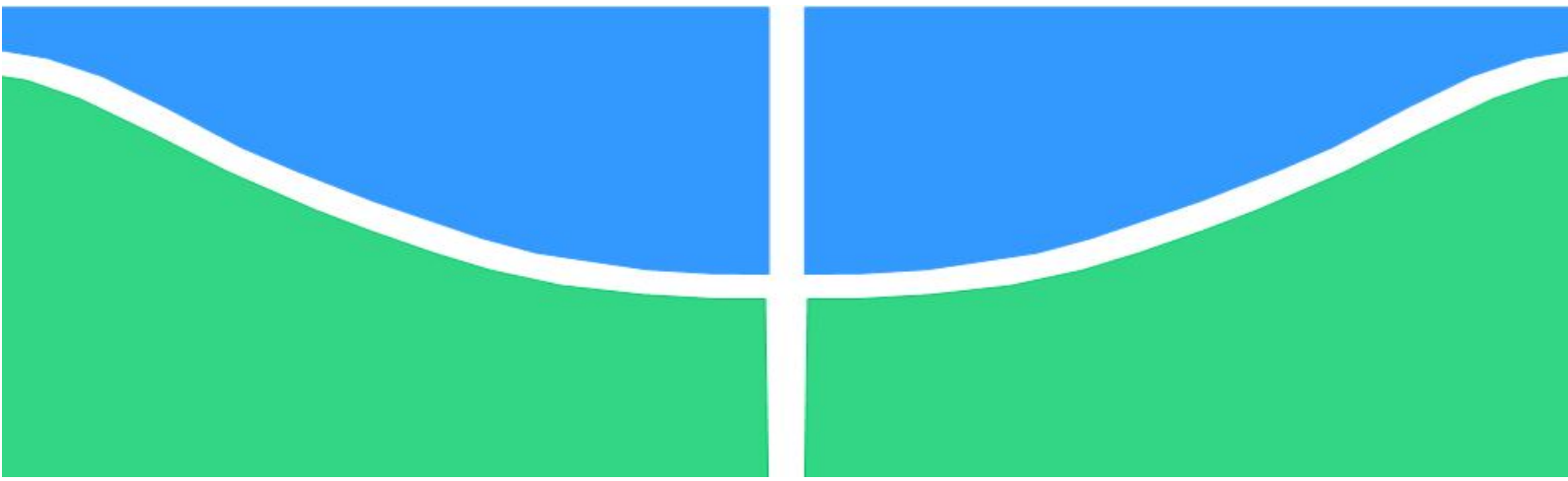

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Fundamentos de Sistemas Distribuídos

Relatório de Experimento com MPI

Phelipe Wener (12/0132893)

Professor:
Fernando W. Cruz

Brasília, DF
27 de Abril de 2017

1. Introdução

O algoritmo de GIMPS(Great Internet Mersenne Search), é um teste de primalidade lançado no início de 1996 como um buscador de grandes números primos; usa computação distribuída em rede pela Internet, comandada por um computador central [2]. Bem como na matemática, são muitas as aplicações que demandam processamento elevado, cujo um único equipamento teria dificuldade em processar. Dessa demanda nasce o Message Passing Interface (MPI), que surge como uma ferramenta de paralelismo, para resolução de problemas recorrentes, como o algoritmo acima citado acima.

Antes da década de 1990, os programadores não tinham tanta sorte quanto nós. Escrever aplicações paralelas para diferentes arquiteturas de computação era uma tarefa difícil e tediosa. Naquela época, muitas bibliotecas poderiam facilitar a criação de aplicativos paralelos, mas não havia uma maneira padrão aceita de fazê-lo [3]. Basicamente, o MPI é um padrão/interface definido para lidar com problemas comuns enfrentados por pessoas que lidam com paralelismo. Dentro do presente relatório, será explorado um problema cuja programação convencional tem gargalos, ou mesmo limitações.

2. Objetivo

O objetivo desse experimento é que o aluno compreenda as características inerentes à construção de aplicações paralelas, envolvendo comunicação por passagem de mensagens, via padrão MPI.

3. Ambiente e configuração

Todos os experimentos abaixo foram executados numa distribuição linux chamada kubuntu, que difere fundamentalmente em termos de interface com o ubuntu. Portanto, deve ser possivel executar os mesmos passos em qualquer versão do Ubuntu 14.04, bem como em um Debian 7.

A implementação MPI escolhida foi o MPICH2. No tópico de desenvolvimento tem um subtópico onde isso será discutido com mais detalhes.

4. Desenvolvimento

4.1. Instalação

Dentre as implementações do MPI, foi escolhido o MPICH, tal que segundo a página de download a API tem distribuição compatível com o sistema operacional utilizado nesse experimento. Para instação foi utilizado o comando:

```
$ sudo apt-get instal mpich
```

Depois foi utilizado o comando a seguir para verificação da instalação:

```
$ mpiexec --version
HYDRA build details:
    Version:                                 3.0.4
    Release Date:                            Wed Apr 24 10:08:10 CDT 2013
...
```

Várias linhas de configuração foram exibidas também.

4.2. Testando MPI

Para familiarização com o MPI, decidiu-se testar com o código de um tutorial no github, o código é um típico "Hello World".

O exemplo contava com um makefile que basicamente usava um compilador especial para gerar o executável, chama-se *mpicc*. Basicamente foi obtido os seguintes resultados:

```
$ make
mpicc -o hello_world hello_world.c
$ ./hello_world
Hello world from processor wenerianus, rank 0 out of 1 processors
```

No tutorial seguido para esse teste ainda havia documentação sobre execução paralela em vários hosts, mais informações: http://mpitutorial.com/tutorials/mpi-hello-world/

4.3 Desenvolvimento do problema

- O roteiro pede que o programa manipule um vetor de 1.000.000.000 de posições do tipo float, iniciado com a seguinte formula: `V[i] = (i - tamanho_vetor/2) ** 2 .`
- Em seguida o programa calcula cada posição do vetor de acordo com a formula: `V[i] = raiz_quadrada(v[i])`
- Primeiramente o programa foi escrito na versão sequencial/convencional:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define VECTOR_LEN 1000000000

// V[i] = (i – tamanho_do_vetor/2) ** 2;
int main() {
        float v[VECTOR_LEN];

        int x;

        for(x = 0; x < VECTOR_LEN; x++) {
                v[x] = pow((x - (VECTOR_LEN/2)), 2);
        }

        return 0;
}
```

- Em seguida o programa foi compilado com o comando `$ gcc simple_cases/pure_exp.c -Wall -o pure -lm`
- Então ao executar o programa usando `$ ./pure` obtemos o erro de Falha de segmentação, comum quando um programa tenta acessar região de memória inválida. É de se esperar que esse erro seja oriundo da grande alocação de memória utilizada na definição do vetor.
- Após, foi criado o mesmo código usando MPI:

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>

#define VECTOR_LEN 1000000000

int main(int argc, char** argv) {
  // Initialize the MPI environment. The two arguments to MPI Init are not
  // currently used by MPI implementations, but are there in case future
  // implementations might need the arguments.
  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size = 1;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  float v[VECTOR_LEN];

  int x;

  for(x = 0; x < VECTOR_LEN; x++) {
    v[x] = pow((x - (VECTOR_LEN/2)), 2);
  }

  // Finalize the MPI environment. No more MPI calls can be made after this
```

```
  MPI_Finalize();
}
```

- Compilado utilizando o comando: `$ mpicc -o mpie simple_cases/mpi_exp.c -lm`
- O programa foi executando sem erros:

```
$ mpirun ./mpie
```

- Tal como explica [1], o MPI utiliza de memória sem restrições, apesar de ser possivel criar essas restrições, a API provê um mecanismo de alocação especial de memória, por esse motivo a alocação do vetor não foi um problema. Porém o código implementado não contempla a segunda parte exigida no roteiro, o que iremos resolver utilizando paralelismo.
- Portanto, foi desenvolvido um programa utilizando MPI que lidasse com múltiplas tarefas, tal como sugere o roteiro. Segue abaixo trechos do código, identificado na pasta do experimento como `mpi_exp.c` :

```
#define VECTOR_LEN 10000

#define MAIN_TASK_ID 0
#define FIRST_NOMAIN_TASK_ID 1

#define TAG1 7

/**
 * Funções utilizadas na main foram ocultadas aqui,
 * se encontram no arquivo mpi_exp
 */

int main(int argc, char* argv[]) {
        // declare vector
        float v[VECTOR_LEN];

        MPI_Init(&argc, &argv);

        // Get the number of processes
        int task_numbers;
        MPI_Comm_size(MPI_COMM_WORLD, &task_numbers);

        // Needed for MPI_Recv bellow
        MPI_Status status;

        /** Commom variables **/
        float task_bigger_value, real_bigger_value;
        float task_smaller_value, real_smaller_value;
        int task_id;

        // Lenght of chunk of work
        int chunk_size = VECTOR_LEN / task_numbers;

        // Working with paralel programming, each task would filled into task_id
        MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

        printf("Runs task %d\n", task_id);

        // Main task
        if (task_id == MAIN_TASK_ID) {
                printf("Into main task\n");

                fill_vector(v);

                assign_chunks(v, chunk_size, task_numbers);

                // Run main task to run work too.
                task_bigger_value = get_bigger(v, 0, chunk_size);
                task_smaller_value = get_smaller(v, 0, chunk_size);
                printf("Bigger found in %d is %e\n", task_id, task_bigger_value);
                printf("Smaller found in %d is %e\n", task_id, task_smaller_value);

                block_and_receive(v, chunk_size, task_numbers, status);

                MPI_Reduce(&task_bigger_value, &real_bigger_value, 1, MPI_INT, MPI_MAX, MAIN_TASK_ID, MPI_COMM_WORLD);
                MPI_Reduce(&task_smaller_value, &real_smaller_value, 1, MPI_INT, MPI_MIN, MAIN_TASK_ID, MPI_COMM_WORLD);
```

```
                printf("---------Results---------\n");
                printf("Considering all tasks, the bigger found is %.2f\n", task_bigger_value);
                printf("Considering all tasks, the smaller found is %.2f\n", task_smaller_value);

        } else if (task_id > MAIN_TASK_ID) {
                // offset of task
                int offset = task_id * chunk_size;

                // Non main tasks
                MPI_Recv(&v[offset], chunk_size, MPI_FLOAT, MAIN_TASK_ID, TAG1, MPI_COMM_WORLD, &status);

                // Do work, getting bigger and smaller
                task_bigger_value = get_bigger(v, offset, chunk_size);
                task_smaller_value = get_smaller(v, offset, chunk_size);
                printf("Bigger found in %d is %e\n", task_id, task_bigger_value);
                printf("Smaller found in %d is %e\n", task_id, task_smaller_value);

                MPI_Send(&v[offset], chunk_size, MPI_FLOAT, MAIN_TASK_ID, TAG1, MPI_COMM_WORLD);

                MPI_Reduce(&task_bigger_value, &real_bigger_value, 1, MPI_INT, MPI_MAX, MAIN_TASK_ID, MPI_COMM_WORLD);
                MPI_Reduce(&task_smaller_value, &real_smaller_value, 1, MPI_INT, MPI_MIN, MAIN_TASK_ID, MPI_COMM_WORLD);
        } else {
                // invalid condition, finish program
                printf("Error when run tasks\n");
                MPI_Finalize();
                return;
        }
        // Finalize the MPI environment. No more MPI calls can be made after this
        MPI_Finalize();
}
```

- Note que `VECTOR_LEN` está definido com um número bem menor do que exigido no relatório(10^9), isso porque em `fill_vector(v)` preenche-se todos os espaços do vetor, por mais que o MPI permita declarar um vetor com essa quantidade de espaços, ele não conseguiria executar essa alocaçao em apenas um processo. Portanto é necessário criar a alocação do vetor separadamente em cada processo.
- Para isso, o array `float v[VECTOR_LEN];` foi colocado em escopo global e inicializado em cada processo, da seguinte maneira:

```
int x;
for (x = offset; x < offset+chunk_size; x++) {
  v[x] = pow((x - (VECTOR_LEN/2)), 2);
}
```

- Dessa forma o gargalo se torna o acesso a variável, o que poderia ser solucionado por alocação dinâmica dentro de cada subtarefa, porém o objetivo foi alcançado.
- A seguir o resultado obtido da utima versão:

```
$ mpicc -o mpie mpi_exp_final.c -lm
kuwener@wenerianus:~/workspaces/fsd/mpi$ time mpirun -np 4 ./mpie
Runs task 0
Into main task
Runs task 1
Runs task 2
Runs task 3
Sent 25000000 elements to task 1
Sent 25000000 elements to task 2
Sent 25000000 elements to task 3
Bigger found in 1 is 2.500000e+07
Smaller found in 1 is 0.000000e+00
Bigger found in 2 is 2.500000e+07
Smaller found in 2 is 0.000000e+00
Bigger found in 3 is 5.000000e+07
Smaller found in 3 is 0.000000e+00
Bigger found in 0 is 5.000000e+07
Smaller found in 0 is 0.000000e+00
---------Results---------
Considering all tasks, the bigger found is 50000000.00
Considering all tasks, the smaller found is 0.00

real    0m0.890s
user    0m2.996s
sys     0m0.120s
```

```
kuwener@wenerianus:~/workspaces/fsd/mpi$ time mpirun -np 1 ./mpie
Runs task 0
Into main task
Bigger found in 0 is 5.000000e+07
Smaller found in 0 is 0.000000e+00
---------Results---------
Considering all tasks, the bigger found is 50000000.00
Considering all tasks, the smaller found is 0.00

real    0m1.714s
user    0m1.656s
sys     0m0.056s
kuwener@wenerianus:~/workspaces/fsd/mpi$ time mpirun -np 8 ./mpie
Runs task 0
Into main task
Runs task 1
Runs task 4
Runs task 5
Runs task 7
Runs task 6
Runs task 2
Runs task 3
Sent 12500000 elements to task 1
Sent 12500000 elements to task 2
Sent 12500000 elements to task 3
Sent 12500000 elements to task 4
Sent 12500000 elements to task 5
Bigger found in 1 is 3.750000e+07
Smaller found in 1 is 0.000000e+00
Sent 12500000 elements to task 6
Sent 12500000 elements to task 7
Bigger found in 2 is 2.500000e+07
Smaller found in 2 is 0.000000e+00
Bigger found in 3 is 1.250000e+07
Smaller found in 3 is 0.000000e+00
Bigger found in 4 is 1.250000e+07
Smaller found in 4 is 0.000000e+00
Bigger found in 5 is 2.500000e+07
Smaller found in 5 is 0.000000e+00
Bigger found in 6 is 3.750000e+07
Smaller found in 6 is 0.000000e+00
Bigger found in 0 is 5.000000e+07
Smaller found in 0 is 0.000000e+00
Bigger found in 7 is 5.000000e+07
Smaller found in 7 is 0.000000e+00
---------Results---------
Considering all tasks, the bigger found is 50000000.00
Considering all tasks, the smaller found is 0.00

real    0m5.976s
user    0m22.860s
sys     0m0.376s
```

- Percebe-se que a execução mais otimizada foi utilizando 4 tarefas, isso porque em certo momento se torna custoso gerenciar a divisão de tarefas.
- O MPI ainda possibilitaria trabalhar com outros hosts via rede, o que possibilitaria dividir em mais tarefas sem que isso se torne um gargalo.

5. Dificuldades

Uma das grandes dificuldades do experimento foi pensar de forma paralela. O compartilhamento de dados e fluxos de execução não são uma tarefa trivial. Infelizmente o experimento não abordou as sucessivas falhas, até porque este relatório teria incontáveis páginas. O MPI fornece extensa documentação e um suporte relativamente bom da comunidade, apesar da API ser bem complexa para se trabalhar.

Uma falha da API é a exibição dos erros, que se dá de uma forma nada intuitiva através de um código de saída. A depuração de uma aplicação paralela tem naturalmente também suas dificuldades.

De modo geral, o que mais foi dificil nesse experimento foi lidar com o array, uma vez que existem diversas possibilidades para sua manipulação.

6. Limitações

Primeira limitação é quanto o espaço do vetor, apesar de rodar com 10^9 ele provavelmente não conseguiria executar pra mais do que isso, ou teria sérios problemas de desempenho. Para resolver isso, bem como explicado no desenvolvimento do experimento, seria possivel trabalhar com um array próprio dentro de cada tarefa.

Outra limitação seria uso indiscriminado de tarefas, provavelmente para um número muito alto(na casa das centenas), o computador não conseguiria operar de forma que esse paralelismo se tornasse na verdade um problema.

7. Conclusão

Portanto, abre-se novos horizontes para computação paralela utilizando MPI, muitas funções que antes eram recorrentemente implementadas para problemas de comunicação de dados agora encontram solução sofisticada nesse padrão. Dessa forma o programador tem um auxilio maior pelo suporte da comunidade, bem como em outros interessados através de um código que utilize MPI.

Apesar de não ser contemplado no experimento por fatores externos, a comunicação em múltiplas máquinas é com certeza mais eficiente, visto que múltiplos processos executando em múltiplas máquinas conseguem um desempenho elevado, desde que a distribuição de tarefas seja ponderada devidamente.

8. Referências

[1] http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node54.htm

[2] https://www.ucb.br/sites/100/103/TCC/12008/JoseAluizioFerreiraLima.pdf

[3] http://mpitutorial.com/tutorials/mpi-introduction/