UNIVERSIDADE FEDERAL DE SÃO CARLOS

CAMPUS SOROCABA

COMPUTAÇÃO PARALELA

PROJETO 2 - PARALELIZAÇÃO COM MPI

Erico Alexandre Nielsen Matthiesen | 400556

Gabriel Stankevix Soares | 511340

Maurício Spinardi | 408174

Setembro de 2015

SUMÁRIO

1. O EXPERIMENTO

Note a diferença entre memória distribuída e memória compartilhada: em ambos os casos, um conjunto de tasks compartilhará dados, mas, quando trabalhamos com memória distribuída, cada uma acessa sua área exclusiva de memória, enquanto no caso compartilhado (como o nome sugere), o conjunto todo divide uma única área global. Tendo isso em mente, imagine a troca de dados entre tasks no modelo de programação de memória distribuída. É para isso que temos o recurso Message-Passing Interface (MPI), que será abordado nesse documento.

MPI não se trata de uma linguagem de programação, e sim de uma biblioteca que pode ser carregada por linguagens como C, C++ e Fortran. Essa biblioteca fornece um conjunto de funções de comunicação entre processos chamada, justamente, de comunicação coletiva[1], que permite potencializar as competências dos programas desenvolvidos nessas linguagens, com o modelo de memória distribuída em mente.

Diferentemente das APIs de memória compartilhada (PThreads e OpenMP), veremos que o custo de manutenção e mensagens para o modelo de memória distribuída é consideravelmente mais substancial. Esse experimento vai parear as três versões e considerar quais as expectativas em relação aos resultados obtidos, e os resultados em si.

1.1. CÓDIGO SERIAL

Como base do experimento de paralelização, tomemos o trecho mais relevante do código fonte (no modelo serial) que soluciona o problema de encontrar *n* números entre os primeiros primos. Considere o arquivo de extensão .C disponível[2] e o que segue:

```
for ( count = 2 ; count <= n ;  ) {
      for ( c = 2 ; c <= i - 1 ; c++ ) {
         if ( i%c == 0 ) break;
      }
      if ( c == i ) {
         printf("%d\n",i);
         count++;
      }
```

[1] Tradução livre de 'collective communication'.
[2] serial.c

```
        i++;
}
```

É possível observar porque o fragmento de código acima consiste do trecho de maior custo computacional. É nele onde os *n* primeiros números primos são, de fato, identificados.

1.2. CONTROLE DE BORDA

Antes de abordar a solução que faz uso da estrutura de memória distribuída, é válido observarmos a correção de borda que visa impedir falhas de execução, como a interrupção completa do processo. Da mesma forma como trata a usabilidade, tal correção é um cuidado para tornar o código, proposto na solução, mais elegante:

```
1. /* Uma vez que a solução proposta é baseada na versão OpenMP,
   anteriormente disponível, deverá haver sempre um 'produtor' e ao
   menos um 'consumidor'. */
2.
3.     if (comm_sz >= 2) {
4.         // Do what has to be done...
5.     }
6.     else
7.        printf("Minimum number of process should be 2\n");
8. ...
```

Note, no pequeno fragmento acima, que ao executarmos esse programa determinando uma única linha de execução, o processo será elegantemente encerrado com um aviso sobre o requerimento mínimo para correto funcionamento. Não teremos erros em cascata ou qualquer tipo de indeterminação.

1.3. PROPOSTA DE SOLUÇÃO E CÓDIGO PARALELO

Considere a estrutura de código original e assuma que existe a possibilidade de checarmos se potenciais números são primos ou não paralelamente. Essa é a base da solução proposta nesta seção:

```
/* Na função principal, considere a inicialização das estruturas MPI: */

MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Devido às características do MPI, considere a bifurcação entre o
processo 0 e os demais, como a seguir: */

switch (my_rank) {
      case 0:
            /* Sets the total amount of processes */
            sz = comm_sz;

            /* […] - esse símbolo indica que existe mais código no arquivo
fonte, mas esse código não é relevante aqui */

            start = MPI_Wtime();

            /* […] */
            /* Produce and send an interval to be tested */
            for (q = 1; q < sz; q++) {
                  for (i = 0; i < RANGE; i++) {
                        interval[i] = n;
                        n++;
                  }
                  MPI_Send(interval, RANGE, MPI_INT, q, 0, MPI_COMM_WORLD);
            }

            /* Check the return from the other processes */
            for (q = 1; q < sz; q++) {
                  MPI_Recv(result, RANGE, MPI_INT, q, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                  for (i = 0; i < RANGE; i++) {
                        if (result[i] && (counter < rnof)) {
                              printf("%d\n", r);
                              counter++;
                        }
```

```
                    r++;
                }
        }

/* […] */
default:
        while (TRUE) {
         /* Receive an interval to be tested or the signal to exit*/
              MPI_Recv(interval, RANGE, MPI_INT, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

        /* […] */

        /* Send an result about the number tested */
        MPI_Send(result, RANGE, MPI_INT, 0, 0, MPI_COMM_WORLD);

/* […] */
```

Note, nos diversos fragmentos acima, o núcleo dessa proposta. A paralelização é garantida pela bifurcação de responsabilidades, atribuídas ao processo 0 e os demais, respectivamente no papel de produtor e consumidor(es). Para minimizar o impacto da comunicação entre processos, cada consumidor recebe um intervalo (de tamanho considerável) de números primos em potencial para testar, ao invés de apenas um.

Por fim, vale ressaltar que, para controlar a ordem de apresentação dos números primos, a estrutura proposta aguarda um conjunto de mensagens de retorno em ordem (ou seja, cada processo consumidor responde após o anterior, em fila circular, até o encerramento da aplicação), garantindo que, visualmente, o resultado seja o mesmo entregue pelo código serial. Essa garantia de ordem, no entanto, tem um impacto sobre o desempenho, ainda que pequeno.

1.4. VALGRIND's MEMCHECK

Para verificar possíveis problemas de memória, a ferramenta disponibilizada pelo pacote Valgrind foi utilizada. Nos testes executados, os resultados[3] foram limpos.

[3] valgrind.txt

1.5. SPEEDUPS E CONSIDERAÇÕES

Para verificar as diferenças de desempenho entre as versões paralelas e serial, foram realizados diversos testes, incluindo encontrar 15000, 50000 e 100000 primos. Todos esses testes[4] foram realizados mais de uma vez, a fim de garantir a acurácia dos resultados e, no caso dos códigos paralelos, esse processo se repete para 2, 4, 8 e 12 linhas de execução.

Vale lembrar que o cenário utilizado para essa verificação foi o de uma máquina configurada em modo de alta performance, com as seguintes características:

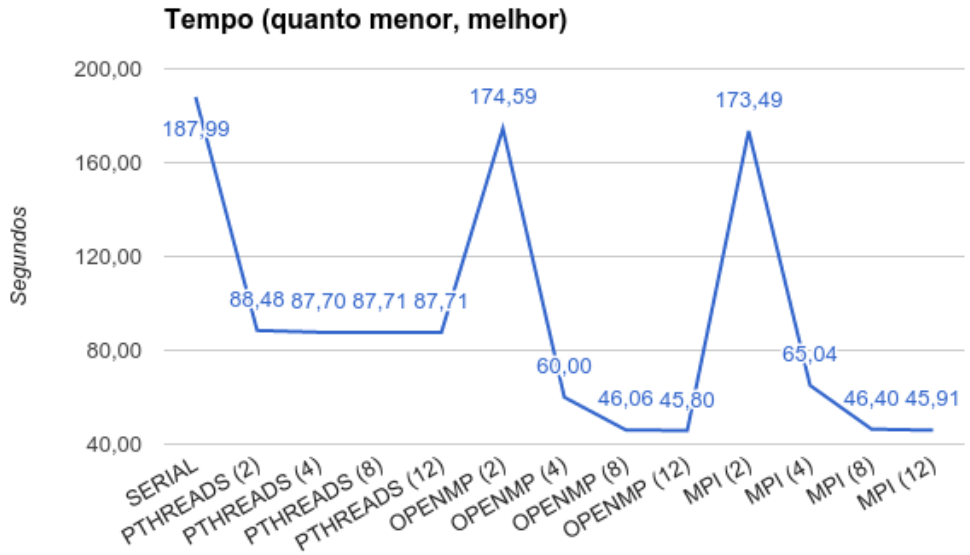
Sistema operacional: Microsoft Windows 10 Pro
Tipo de processador: QuadCore Intel Core i5-3570K, 3700 MHz (37 x 100)
Memória do Sistema: 8074 MB (DDR3-1600 DDR3 SDRAM)
Adaptador gráfico: Intel(R) HD Graphics 4000 (2112 MB)
Adaptador gráfico: NVIDIA GeForce GTX 970 (4 GB)
Armazenamento: Corsair Force 3 SSD (SATA-III)

Como os testes de maior carga são os testes de maior interesse, serão omitidos nessa seção os casos 15000 e 50000 *[ver anexos]*.

SERIAL	PTH (2)	OMP (2)	PTH (4)	OMP (4)	PTH (8)	OMP (8)	PTH (12)	OMP (12)
187,99	88,48	174,59	87,70	60,00	87,71	46,06	87,71	45,80

MPI (2)	MPI (4)	MPI (8)	MPI (12)
173,49	65,04	46,40	45,91

[4] todos os valores são referentes à média dos resultados, dados em segundos.



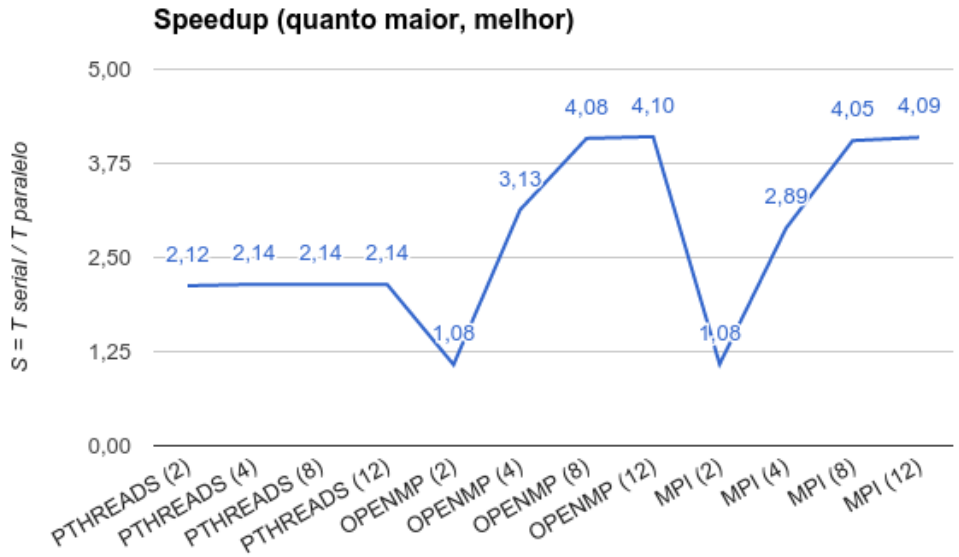
Para a realização dos cálculos de *speedup*, utilizaremos a seguinte formula:

S = T serial / T paralelo

Assuma que “T serial” é o tempo de execução do programa serial e “T paralelo” é o tempo de execução do programa paralelo.

PTH (2)	OMP (2)	PTH (4)	OMP (4)	PTH (8)	OMP (8)	PTH (12)	OMP (12)
2,12	1,08	2,14	3,13	2,14	4,08	2,14	4,10

MPI (2)	MPI (4)	MPI (8)	MPI (12)
1,08	2,89	4,05	4,09



Agora, munidos do *speedup* dos testes acima, calcularemos a eficiência do programa paralelo utilizando a seguinte fórmula:

*E = S / p = (T seria / T paralelo) / p = T serial / p*T paralelo*

Assuma que “p” é o numero de linhas de execução, “T serial” é o tempo de execução do programa serial e “T paralelo” é o tempo de execução do programa paralelo.

PTH (2)	OMP (2)	PTH (4)	OMP (4)	PTH (8)	OMP (8)	PTH (12)	OMP (12)
1,06	0,54	0,54	0,78	0,27	0,51	0,18	0,34

MPI (2)	MPI (4)	MPI (8)	MPI (12)
0,54	0,72	0,51	0,34

Nos testes acima, é possível perceber que o código utilizando MPI, com 2 linhas de execução, não teve ganho significativo comparado ao serial. No entanto, surpreende que tenha sido, ainda que de forma insignificante, melhor do que a versão OpenMP, por exemplo. Por outro lado, testes com 4, 8 e 12 linhas de execução demonstraram a vantagem de uma solução paralela, mais uma vez, em relação ao código original e, mais do que isso, do bom desempenho da versão MPI em relação às soluções de memória compartilhada.

Esse código só teve sua curva de eficiência limitada pelo hardware da máquina, como podemos notar a partir do teste com 8 e 12 linhas de execução. Especulamos que, se executado em um hardware com mais núcleos físicos disponíveis, a eficiência seria maior (ou teria um decréscimo mais lento, ao menos).

2. CONCLUSÃO

A realização deste projeto proporcionou uma nova visão sobre a diferença entre as estruturas de memória distribuída e compartilhada. Mais precisamente, os resultados demonstraram um amadurecimento nas habilidades de programação paralela, uma vez que era de se esperar que a solução desenvolvida com MPI fosse mais lenta do que as soluções de memória compartilhada e o que podemos notar é um desempenho que vence a versão PThreads e se equipara à versão OpenMP.

Vale lembrar que a implementação, no tocante à correções de erros e controle de borda, permitiu melhorar a escalabilidade e adaptabilidade de código, considerando a máquina em que é executado.

ANEXOS

TEMPOS	TESTE 1			TESTE 2			TESTE 3		
	ROUND 1	ROUND 2	AVERAGE	ROUND 1	ROUND 2	AVERAGE	ROUND 1	ROUND 2	AVERAGE
SERIAL	3.59	3.58	3.59	44.51	44.47	44.49	189.00	186.97	187.99
PTHREADS (2)	1.66	1.67	1.67	20.68	20.76	20.72	88.36	89.59	88.48
PTHREADS (4)	1.65	1.67	1.66	20.62	20.84	20.73	87.69	87.71	87.70
PTHREADS (8)	1.66	1.66	1.66	20.64	20.66	20.65	87.70	87.72	87.71
PTHREADS (12	1.71	1.70	1.71	20.66	20.69	20.68	87.71	87.71	87.71
OPENMP (2)	3.79	3.71	3.75	42.08	41.88	41.98	174.43	174.74	174.59
OPENMP (4)	1.39	1.40	1.40	14.57	14.56	14.57	60.01	59.99	60.00
OPENMP (8)	1.13	1.11	1.12	11.22	11.26	11.24	46.11	46.10	46.06
OPENMP (12)	1.13	1.12	1.13	11.26	11.19	11.23	45.85	45.75	45.80
MPI (2)	3.32	3.30	3.31	40.77	40.76	40.77	173.53	173.45	173.49
MPI (4)	1.30	1.32	1.31	15.36	15.43	15.39	65.75	64.33	65.04
MPI (8)	1.03	1.04	1.04	11.55	11.60	11.58	46.43	46.36	46.40
MPI (12)	0.95	0.96	0.96	11.33	11.22	11.28	45.85	45.97	45.91
SPEED UP									
PTHREADS (2)	2.16	2.14	2.15	2.15	2.14	2.15	2.14	2.11	2.12
PTHREADS (4)	2.18	2.14	2.16	2.16	2.13	2.15	2.16	2.13	2.14
PTHREADS (8)	2.16	2.16	2.16	2.16	2.15	2.15	2.16	2.13	2.14
PTHREADS (12	2.10	2.11	2.10	2.15	2.15	2.15	2.15	2.13	2.14
OPENMP (2)	0.95	0.96	0.96	1.06	1.06	1.06	1.08	1.07	1.08
OPENMP (4)	2.58	2.56	2.57	3.05	3.05	3.05	3.15	3.12	3.13
OPENMP (8)	3.18	3.23	3.20	3.97	3.95	3.96	4.10	4.06	4.08
OPENMP (12)	3.19	3.20	3.19	3.95	3.97	3.96	4.12	4.09	4.10
MPI (2)	1.08	1.08	1.08	1.09	1.09	1.09	1.09	1.08	1.08
MPI (4)	2.76	2.71	2.74	2.90	2.88	2.89	2.87	2.91	2.89
MPI (8)	3.49	3.44	3.46	3.85	3.83	3.84	4.07	4.03	4.05
MPI (12)	3.78	3.73	3.75	3.93	3.98	3.95	4.12	4.07	4.09
EFICIÊNCIA									
PTHREADS (2)	1.08	1.07	1.08	1.08	1.07	1.07	1.07	1.06	1.06
PTHREADS (4)	0.54	0.54	0.54	0.54	0.53	0.54	0.54	0.53	0.54
PTHREADS (8)	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27
PTHREADS (12	0.17	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18
OPENMP (2)	0.47	0.48	0.48	0.53	0.53	0.53	0.54	0.53	0.54
OPENMP (4)	0.65	0.64	0.64	0.76	0.76	0.76	0.79	0.78	0.78
OPENMP (8)	0.40	0.40	0.40	0.50	0.49	0.49	0.51	0.51	0.51
OPENMP (12)	0.26	0.27	0.27	0.33	0.33	0.33	0.34	0.34	0.34
MPI (2)	0.54	0.55	0.55	0.55	0.55	0.55	0.54	0.54	0.54
MPI (4)	0.69	0.68	0.68	0.72	0.72	0.72	0.72	0.73	0.72
MPI (8)	0.44	0.43	0.43	0.48	0.48	0.48	0.51	0.50	0.51
MPI (12)	0.31	0.31	0.31	0.33	0.33	0.33	0.34	0.34	0.34

TESTE 1	TESTE 2	TESTE 3
150.00	50000	100000