UNIVERSIDADE FEDERAL DE SÃO CARLOS

CAMPUS SOROCABA

COMPUTAÇÃO PARALELA

PROJETO 2 - PARALELIZAÇÃO COM OpenMP

Érico Alexandre Nielsen Matthiesen | 400556

Gabriel Stankevix Soares | 511340

Maurício Spinardi | 408174

Setembro de 2015

SUMÁRIO

1. O EXPERIMENTO

OpenMP é uma API para programação paralela como Pthreads que, no entanto, detém uma diferença fundamental quando em comparação com a irmã de baixo nível. PThreads requerem que o programador explicitamente especifique o comportamento de cada *thread*, enquanto OpenMP, por outro lado, possibilita que o programador simplesmente ateste que um bloco de código deve ser executado em paralelo. Essa diferença sugere o porque existem dois padrões de APIs para esse tipo de programação: a baixo nível (PThreads) e a determinada pelo compilador e sistema em tempo de execução (OpenMP).

Finalmente, podemos dizer que OpenMP provê o que é conhecido como programação baseada em diretivas de memória compartilhada. Em C e C++, isso significa que existem diretivas especificas (ou trechos de código e bandeiras específicas) que são convertidas diretamente em instruções de hardware. Tais bandeiras serão aqui conhecidas como *pragmas,* normalmente implementadas para dar suporte a comportamentos que não são naturais e, por isso mesmo, compiladores que não as reconheçam são livres para ignorá-las. Nesse caso, códigos cuidadosamente escritos podem ser executados independentemente do suporte à OpenMP (mas vale lembrar que programas naturalmente paralelos podem não fazer sentido sem o devido suporte do compilador, como veremos a seguir).

Nesse experimento, discutiremos as possibilidades de se utilizar OpenMP para incrementar o desempenho de uma aplicação serial que calcula um determinado número de primos. Serão exibidas as transformações efetuadas no código serial para que se comporte como um programa paralelo, bem como os cálculos de *speedup* e considerações sobre *tradeoff* com diferentes núcleos e regiões de corrida concorrente. Por fim, também apresentaremos uma solução mais elegante para o caso onde o programa paralelo depende, de fato, do suporte do compilador às diretivas do OpenMP.

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[1] e o que segue:

[1] serial.c

```
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++;
      }
      i++;
}
```

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

1.2. CORREÇÃO DE ERROS E CONTROLE DE BORDA

Antes de abordar a solução que faz uso de diretivas OpenMP, é válido observarmos que, uma vez que essa é uma API cujas bandeiras podem ser ignoradas pelo compilador e sistema, programas desenvolvidos com perfis naturalmente paralelos (como é o caso do modelo produtor-consumidor) requerem um cuidado adicional contra erros. Uma forma elegante de controlar o modelo produtor-consumidor nesse formato é a checagem de inclusão de bibliotecas, como podemos ver a seguir:

```
1. #ifdef _OPENMP    // checagem de erro para suporte à OpenMP
      a. #include <omp.h>
      b. int omp = 1;
2. #else
      a. int omp = 0;
3. #endif
4.
5. … // nesse techo, pode existir mais código
6.
7. int main(int argc, char* argv[]) {
8.
9.       if (!omp) {
```

```
10.             printf("OpenMP not supported\n");
11.             exit(0);
12.       }
13..
14..
```

Note, no pequeno fragmento acima, que ao executarmos esse programa em um sistema sem suporte à OpenMP, não teremos erros em cascata ou qualquer tipo de indeterminação: a execução é encerrada com um aviso sobre a falta de suporte à API. Esse tipo de solução agrega valor ao código-fonte e ao executável, ao final do desenvolvimento.

1.3. ALGUNS DETALHES SOBRE PRODUÇÃO E CONSUMO

Sobre o modelo produtor-consumidor, basta que saibamos que duas linhas de execução compartilham um mesmo buffer de tamanho fixo. Um produtor insere informação e um consumidor retira informação desse buffer.

Esse modelo se adequa perfeitamente ao problema proposto pelo código serial disponível. Um produtor produz números primos em potencial, enquanto um ou mais consumidores atestam se esses números são, de fato, primos. No entanto, para consumir corretamente (uma vez que podem existir múltiplos consumidores) e produzir corretamente (uma vez que o tamanho do buffer é fixo e as posições de inserção variáveis), é necessário um controle de sincronização. Esse controle pode ser dado pela inclusão de semáforos nos trechos críticos de código.

1.4. 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 chamada:
# pragma omp parallel num_threads(thread_count)
produtor_consumidor();
```

```
/*
```

```
 * Na função produtor_consumidor, considere os blocos abaixo, sincronizados
 * por semáforos, cujos detalhes podem ser verificados no .C disponível:
 */

      … // produtor
      buffer[fim] = i;
      fim = (fim + 1) % N_ITENS; // atualiza o índice
      if (count <= n) i++;    // produz um novo número
      else breaker();

      … // consumidor
      numero_a_ser_testado = buffer[inicio];
      inicio = (inicio + 1) % N_ITENS;
      if (count <= n) tester(numero_a_ser_testado);
      else breaker();

/*
 * Note a chamada para as funções tester() e breaker(), ambas parte
 * do bloco consumidor, como vimos acima.
 */

void tester(int i) {
      int c = 2;
      for (c = 2; c <= i-1; c++) {
            if ( i%c == 0 ) break;
      }
      if (c == i) {
            # pragma omp critical
            {
                  if (count <= n) {
                        printf("%d\n",i);
                        count++;
                  }
            }
      }
}

/*
```

```
 * Por fim, considere que o encerramento é sempre dado pela função
 * breaker(), responsável por interromper todas as linhas de execução,
 * uma vez satisfeita a condição imposta pelo usuário.
 */
void breaker() {
      # pragma omp critical
      {
            GET_TIME(finish);
            elapsed = finish - start;
            printf("Elapsed time: %f\n", elapsed);
            exit(0);    //caso satisfeito, encerra produção e consumo
      }
}
```

Note, nos diversos fragmentos acima, o núcleo dessa proposta. A paralelização é garantida pela diretiva de OpenMP na função principal, que inicia o trabalho de produção e consumo de número primos em potencial. O consumidor, em particular, verifica se esses números são primos, de fato, através da chamada à função tester(). Em tester, podemos notar o quão fiel o código é à base serial (não foram efetuadas maiores otimizações, de forma a evidenciar o ganho de eficiência somente com a paralelização).

Por fim, vale notar que, quando existe mais de um consumidor (ou seja, 3 ou mais linhas de execução) existe um diferença no resultado visual entre o código serial e essa proposta. Como primos em potencial são checados em paralelo, é impossível determinar a ordem em que serão impressos na tela. No entanto, o conjunto total, se armazenado e ordenado, será o mesmo para ambos os códigos. Essa ordenação, no entanto, foi aqui omitida[2].

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[3] 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.

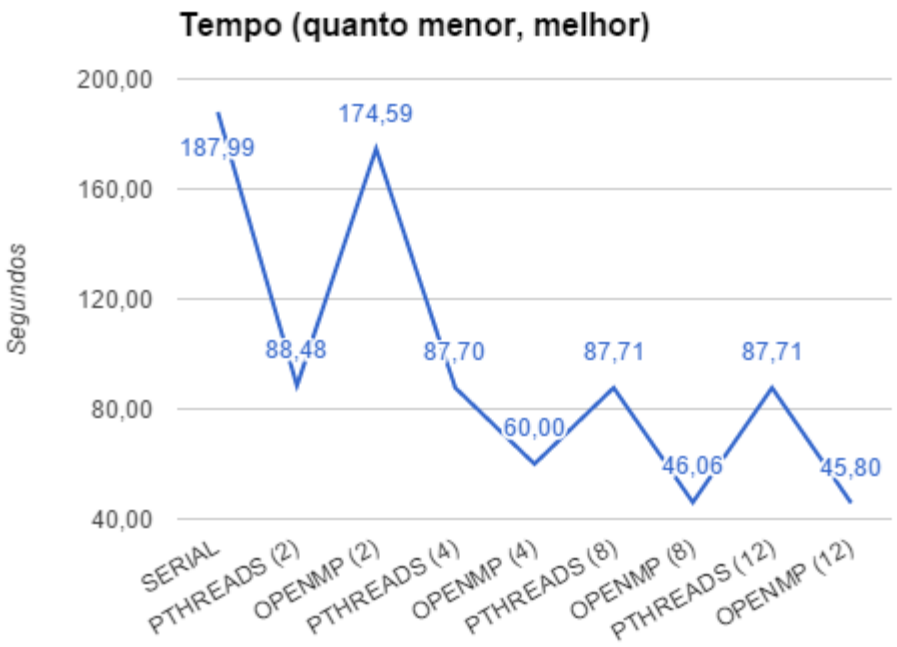
[2] implementação dispensada por orientação prévia.
[3] todos os valores são referentes à média dos resultados, dados em segundos.

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)
Disco rígido:	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

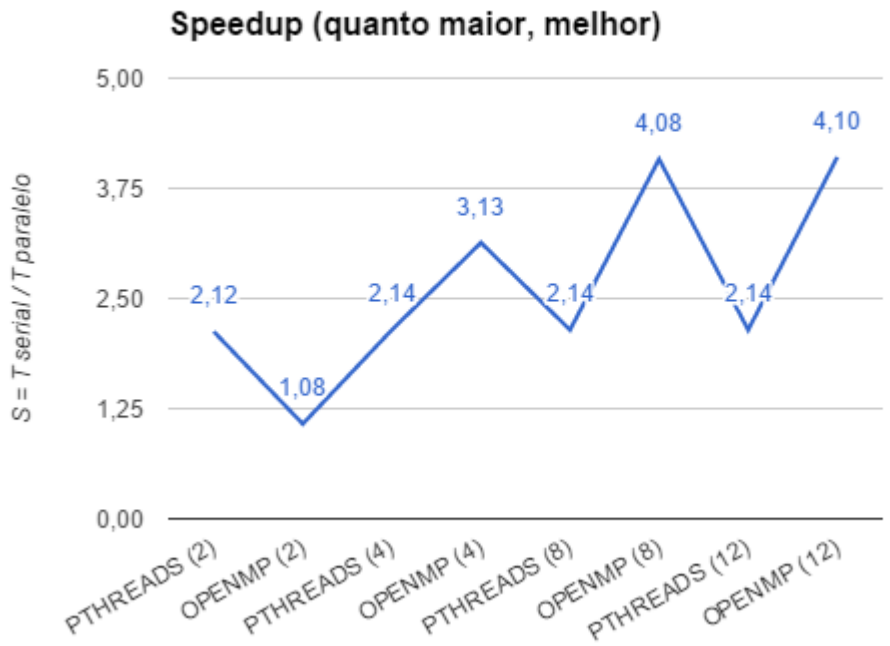


Para a realização dos cálculos de *speedup*, utilizaremos a seguinte formula:

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



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

Nos testes acima, é possível perceber que o código utilizando OpenMP, com 2 linhas de execução, não teve ganho significativo (ou sequer ganho algum) comparado ao serial (e ao mesmo tempo com aquele desenvolvido com PThreads), muito por conta do custo de

manutenção da estrutura produtor/consumidor. Por outro lado, testes com 4, 8 e 12 linhas de execução demonstraram uma clara diferença de desempenho a favor da versão OpenMP.

O que fica evidente é que a versão proposta nesse projeto é mais escalável, levando-se em consideração a estrutura da máquina em que é executada, diferentemente da versão paralela desenvolvida anteriormente. 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 E DIFICULDADES

A realização deste projeto provou a diferença teórica das APIs de desenvolvimento PThreads e OpenMP, sendo a última, sem dúvida, mais simples. Optar por aplicar um novo modelo de solução (produtor/consumidor), no entanto, trouxe dificuldades que antes não haviam sido enfrentadas, como a condição de parada, uma vez satisfeita a requisição do usuário. As implementações aplicadas, no tocante à correções de erros e controle de borda, permitiram melhorar a escalabilidade e adaptabilidade de código, considerando a máquina em que é executado. Por fim, um único desapontamento foi identificado: a solução proposta não tem sua execução encerrada através da função principal. Uma função específica foi desenvolvida para que todas as linhas de execução sejam interrompidas uma vez satisfeita a condição de parada. Em outros cenários, esse pode se provar um limitante.

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	20,83	20,76	20,72	88,36	88,59	**88,48**
OPENMP (2)	3,79	3,71	3,75	42,08	41,88	41,98	174,43	174,74	**174,59**
PTHREADS (4)	1,65	1,67	1,66	20,52	20,84	20,73	87,69	87,71	**87,70**
OPENMP (4)	1,39	1,40	1,40	14,57	14,56	14,57	60,01	59,99	**60,00**
PTHREADS (8)	1,66	1,66	1,66	20,64	20,66	20,65	87,70	87,72	**87,71**
OPENMP (8)	1,13	1,11	1,12	11,22	11,26	11,24	46,11	46,00	**46,06**
PTHREADS (12	1,71	1,70	1,71	20,86	20,89	20,88	87,71	87,71	**87,71**
OPENMP (12)	1,13	1,12	1,13	11,26	11,19	11,23	45,85	45,75	**45,80**
SPEEDUP									
PTHREADS (2)	2,16	2,14	2,15	2,15	2,14	2,15	2,14	2,11	**2,12**
OPENMP (2)	0,95	0,96	0,96	1,06	1,06	1,06	1,08	1,07	**1,08**
PTHREADS (4)	2,18	2,14	2,16	2,16	2,13	2,15	2,16	2,13	**2,14**
OPENMP (4)	2,58	2,56	2,57	3,05	3,05	3,05	3,15	3,12	**3,13**
PTHREADS (8)	2,16	2,16	2,16	2,16	2,15	2,15	2,16	2,13	**2,14**
OPENMP (8)	3,18	3,23	3,20	3,97	3,95	3,96	4,10	4,06	**4,08**
PTHREADS (12	2,10	2,11	2,10	2,15	2,15	2,15	2,15	2,13	**2,14**
OPENMP (12)	3,18	3,20	3,19	3,95	3,97	3,96	4,12	4,09	**4,10**
EFICIÊNCIA									
PTHREADS (2)	1,08	1,07	1,08	1,08	1,07	1,07	1,07	1,06	**1,06**
OPENMP (2)	0,47	0,48	0,48	0,53	0,53	0,53	0,54	0,53	**0,54**
PTHREADS (4)	0,54	0,54	0,54	0,54	0,53	0,54	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**
PTHREADS (8)	0,27	0,27	0,27	0,27	0,27	0,27	0,27	0,27	**0,27**
OPENMP (8)	0,40	0,40	0,40	0,50	0,49	0,49	0,51	0,51	**0,51**
PTHREADS (12	0,17	0,18	0,18	0,18	0,18	0,18	0,18	0,18	**0,18**
OPENMP (12)	0,28	0,27	0,27	0,33	0,33	0,33	0,34	0,34	**0,34**

TESTE 1	TESTE 2	TESTE 3
15000	50000	100000