Grupo: B Disciplina: Programação Concorrente Data: 18/11/2014
Integrantes do grupo: Caique Rodrigues Marques, Fernando Jorge Mota

Introdução

Dado um problema, ele pode ter várias formas de ser resolvido e, consequentemente, ter diferentes resultados possíveis. A possibilidade de achar a única e perfeita resposta é uma utopia, mas o desenvolvimento e a ideia implementados geram um processo de aprendizado e reflexão que sempre motiva a buscar uma solução melhor, mesmo que esta não seja a "perfeita", e isto é o que mantém os projetos vivos, tais quais as ideias por trás da Torre de Babel e das descobertas científicas.

Neste Trabalho Prático, o problema consiste em encontrar todos os possíveis números mutuamente amigos de um dado intervalo, tanto de forma sequencial quanto de forma concorrente. Das diversas respostas possíveis que nós desenvolvemos, chegamos a um consenso de que, em nossas avaliações, a apresentada nesse trabalho foi a melhor que conseguimos encontrar, tanto em questão de simplicidade, quanto de performance e otimização. Portanto, boa leitura e esperamos que você goste da implementação proposta!

Os Autores

Descrição do Problema

O problema consiste em encontrar todos os números mutuamente amigos em um determinado intervalo, ambos informados como argumentos ao programa.

Um par de números mutuamente amigos é definido da seguinte forma:

> *Dados dois números A e B pertencentes ao conjunto dos naturais positivos, ambos serão mutuamente amigos se, e somente se, a razão entre a soma de todos os divisores do número e ele próprio são iguais.*

Ou seja, para encontrar um par de números mutuamente amigos, um algoritmo precisa realizar os seguintes passos para descobrir se os números A e B são mutuamente amigos:

1. Detectar e somar os divisores do numero A, obtendo assim um número C
2. Detectar e somar os divisores do numero B, obtendo assim um número D
3. Dividir C por A e salvar em uma variável E
4. Dividir D por B e salvar em uma variável F
5. Comparar E e F. Se ambos forem iguais, os números A e B são mutuamente amigos, caso contrário, os números A e B não são mutuamente amigos.

Por exemplo, considerando A=30 e B=140, temos que:

- $E = \frac{C}{A} = \frac{(1+2+3+5+6+10+15+30)}{30} = \frac{72}{30} = \frac{12}{5} = 2.4$
- $F = \frac{D}{B} = \frac{1+2+4+5+7+10+14+20+28+35+70+140}{140} = \frac{336}{140} = \frac{12}{5} = 2.4$

Assim, como E é igual a F , temos que os números A e B são mutuamente amigos.

Se considerarmos A=35 e B=130, temos que:

- $E = \frac{C}{A} = \frac{(1+5+7+35)}{35} = \frac{48}{35} = 1.37$
- $F = \frac{D}{B} = \frac{(1+2+5+10+13+26+65+130)}{130} = \frac{252}{130}$ =1.93

Logo, como E é diferente de F, temos que os números A e B, nesse caso, não são mutuamente amigos.

Outro problema que foi resolvido neste trabalho é a paralelização da solução do problema. Nesse caso específico, é normal enfrentar desafios como decidir qual etapa do algoritmo vale deixar paralela entre seções como a soma dos divisores e as diferentes combinações de números mutuamente amigos, por exemplo.

Descrição das soluções paralela e sequencial

Sobre o algoritmo

Para resolver o problema foram criadas duas versões em C, uma solução que resolve o problema de forma paralela e outra que o resolve de forma sequencial. As duas versões possuem semelhanças tanto na lógica usada quanto no código em si, entretanto, a versão paralela, como esperado, fornece os resultados de forma mais rápida que a versão sequencial, embora isso também dependa do processador e do número de threads usados (veja a seção “Resultados” para conferir o desempenho das versões em diferentes processadores e com diferentes números de threads).

Solução Sequencial

O programa em sua versão sequencial tem uma única preocupação: resolver o problema um a um, ou seja, buscando números mutuamente amigos apenas um número de cada vez em qualquer período do tempo. Em geral, o programa executa na seguinte sequência:

A execução é iniciada pela função *main* e uma instrução é executada por vez seguindo a ordem definida. Um vetor é reservado, se possível (ou seja, caso haja memória suficiente), para a alocação da soma de todos os números de um dado intervalo definido pelas variáveis *mínimo* e *máximo*, esse espaço representa um ponteiro e é armazenado na variável *cache*.

```
10  unsigned int somaDivisores(unsigned int numero) {
11      unsigned int divisores, contagem;
12      divisores = 1 + numero;
13      for (contagem = 2; contagem < numero; contagem++) {
14          if (numero % contagem == 0) {
15              divisores += contagem;
16          }
17      }
18      return divisores;
19  }
20
```

A função *somaDivisores* (veja figura acima) realiza a soma dos divisores de um número especificado pelo seu único parâmetro. Um laço irá realizar a operação mostrada na equação abaixo (linha 14), onde x é o número especificado pelo parâmetro da função. Se o resultado de (x mod i) for zero, ou seja, se a razão de x por i resultar em resto zero, x é divisível por i e, portanto, é armazenado na variável *divisores* que, durante a execução, vai sendo somado com os outros valores divisíveis por x. Ao final a variável é retornada.

$$\sum_{i=2}^{x-1} (x \bmod i)$$

```
31  double* criaCacheEntre(unsigned int minimo, unsigned int maximo) {
32      unsigned int intervalo, n1;
33      double* cache;
34      intervalo = maximo - minimo;
35      cache = (double*) malloc((intervalo+1)*sizeof(double));
36      for(n1 = 0; n1 <= intervalo; n1++) {
37              cache[n1] = (double) somaDivisores(n1 + minimo);
38      }
39      return cache;
40  }
41
```

Na função *criaCacheEntre* (veja figura acima), um espaço na memória é armazenado de acordo com o número de elementos presentes em um intervalo definido pelos parâmetros *mínimo* e *máximo* (linha 31) somado mais um (linha 35). Se o espaço for alocado com sucesso, um laço irá chamar a função *somaDivisores* e colocar em cada espaço do vetor cache a soma dos divisores (linhas 36 e 37), ao final, a variável é retornada. Se o espaço não for alocado com sucesso, a função simplesmente retorna um ponteiro NULL, indicando que houve uma falha na alocação da memória.

Fazer processamento repetido não é bom para performance, por causa disso, durante o desenvolvimento do trabalho, foi escolhido fazer uso de uma cache pré-calculada para evitar o cálculo da soma dos divisores (que é uma das tarefas computacionais mais intensivas para o programa) durante a procura por números mutuamente amigos. Assim, esse cálculo, quando possível, é executado apenas uma vez, melhorando consideravelmente a performance do programa (veja subseção “Complexidade” mais adiante).

Se a cache tiver sido devidamente alocada, a função *calculaNumerosMutuamenteAmigosUsandoCacheEntre* é chamada, levando a cache como parâmetro. Caso não for possível alocar espaço de memória para a *cache*, a função *calculaNumerosMutuamenteAmigosEntre* realiza, em dois laços, a chamada da função *somaDivisores* e armazena os valores nas variáveis temporárias *soma1* e *soma2*, detectando assim a soma dos divisores sob demanda, sem a necessidade de uma alocação de memória.

Solução Paralela

A solução paralela tem um funcionamento muito similar ao da Solução Paralela, entretanto, ocorre que, nesta implementação, mais de um número é processado por vez em qualquer período do tempo (daí o nome de “Solução Paralela”), dependendo apenas do número de threads definidas como argumento para o programa (ou seja, se apenas uma thread for inicializada, aí realmente o programa vai rodar de forma sequencial). Nesta solução, o modelo de programação OpenMP foi utilizado nas regiões especificadas, quantas threads irão trabalhar é passada como argumento para o programa. A paralelização da execução, neste caso, são realizadas nos laços - que são muito usados por todo o programa -, através de diretivas nativas do OpenMP.

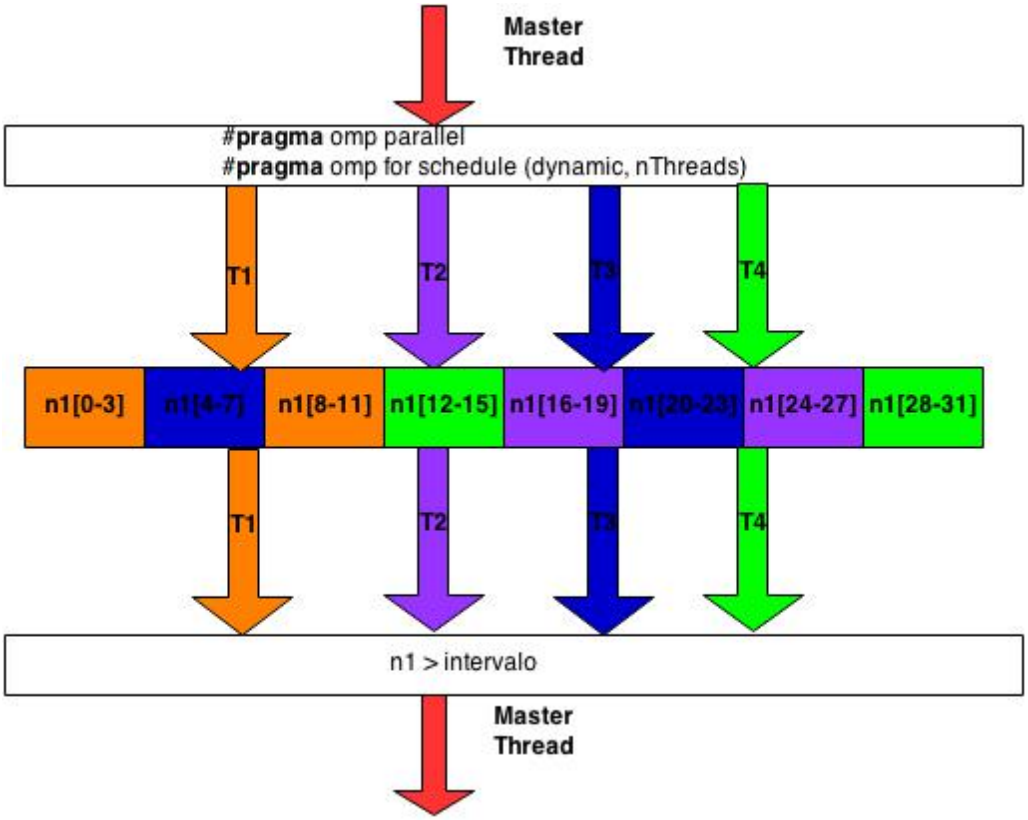
O OpenMP foi escolhido como solução primária para a criação da versão paralela por alguns motivos bem simples:

- Divisão de tarefas semiautomática entre as threads - Basta especificar um algoritmo de divisão e seus parâmetros e a divisão do trabalho é feita automaticamente pelo OpenMP;
- Compatibilidade - É compatível com os principais sistemas operacionais do mercado;
- Facilidade de uso - Devido a recursos como a divisão de tarefas semiautomática, é muito fácil de usar, não necessitando nem mesmo de bibliotecas ou mesmo executáveis para que os programas que usam OpenMP sejam compilados, por exemplo;
- Estabilidade - O OpenMP é um framework estável e testado, tendo sido, inclusive, integrado aos compiladores mais populares disponíveis atualmente;
- Velocidade - Por usar diretamente diretivas do compilador, o OpenMP consegue alcançar velocidade próxima ou até superior à soluções nativas baseadas na biblioteca **pthread**.

Na função *main* a quantidade de threads é armazenada na variável *nThreads*, que é passada como parâmetro para as outras funções para que as operações em paralelo sejam definidas em quantos blocos de tarefa.

```
102  double* criaCacheEntre(unsigned int minimo, unsigned int maximo, unsigned int nThreads) {
103      unsigned int intervalo, n1;
104      double *cache;
105      intervalo = maximo-minimo;
106      cache = (double*) malloc((intervalo+1)*sizeof(double));
107      #pragma omp parallel
108      #pragma omp for schedule(dynamic, nThreads) private(n1)
109      for(n1 = 0; n1 <= intervalo; n1++) {
110          cache[n1] = (double) somaDivisores(n1 + minimo);
111      }
112      return cache;
113  }
114
```

Em *criaCacheEntre* (veja figura acima), a paralelização ocorre na atribuição das somas dos divisores na variável *cache*, onde as diretivas do OpenMP definem como isso deve ser feito (linhas 107 e 108). Para um laço, foi definido que as threads dividissem cada iteração em blocos (diretiva *for* do OpenMP) de tamanhos iguais (parâmetro *dynamic*), o tamanho é definido pelo parâmetro após a vírgula, neste caso, pelo número de threads. Por exemplo, para um intervalo de trinta e dois números (isto é, trinta e duas iterações) e quatro threads operando no programa, a divisão dos blocos fica com cada thread tendo uma fila interna de quatro números, **embora** apenas um número seja processada por uma thread ao mesmo tempo (veja figura abaixo).



Ao entrar numa região paralela, definida pela diretiva "#pragma omp", a master thread divide as operações com o número de threads especificadas no início do programa e o parâmetro após o dynamic define quantas operações cada thread deve fazer (quatro, em ambos os casos). O vetor indica o intervalo de operações realizadas por cada thread, onde cada bloco é uma iteração do laço e a realização deste pela respectiva thread (T1[0 até 3], T3[4 até 7], etc.). Ao final os resultados são "unidos" na master thread.

A paralelização nas funções *calculaNumerosMutuamenteAmigosEntre* e *calculaNumerosMutuamenteAmigosUsandoCacheEntre* foram implementadas de forma semelhante. Assim como foi na função *criaCacheEntre*, a estratégia de paralelização utilizada foi a mesma: dividir o trabalho entre diferentes threads diretamente no laço **for** usando diretivas do OpenMP, assim, números mutuamente amigos podem ser encontrados paralelamente e assim acelerar a execução total do programa.

Complexidade

A complexidade de algoritmos correlata o número de instruções que um programa executa a medida que os dados de entrada vão aumentando, portanto, se uma aplicação levar mais tempo para executar a medida que os dados aumentam, ele não possui um desempenho desejável. A hipótese de complexidade sempre leva a considerar o pior caso possível, portanto, se considera que tudo no programa será executado (isto é, supondo que todas as condições são satisfeitas) e se leva em conta a versão sequencial, afim de facilitar o cálculo e o entendimento.

Para fins de comparação, é usada a ordem de complexidade de algoritmos, ou a notação big-O, que informa qual o limite de uma dada função para uma função grande o bastante, a resposta é considerar o de maior índice. Por exemplo, dada a função:

$$f(x) = 3x^4 + 2x + 5$$

Estipulando um limite com x tendendo a infinto, ou seja, a função vai tender a que número se os valores de x crescerem infinitamente?

$$\lim_{x \to \infty} (3x^4 + 2x + 5) = \infty$$

Como $3x^4 > 2x > 5$ (sendo x quaisquer valores maiores que 3), $3x^4$ crescerá mais rápido que 2x, enquanto 5 permanece estável, portanto, a função tem ordem de complexidade, ou notação big-O, de $O(n^4)$.

Na função *somaDivisores*, uma variável é usada para armazenar a soma dos divisores, então ela realiza uma soma e uma atribuição, levando instruções de 1 + 1 = 2, para isso uma condição precisa ser satisfeita. Uma comparação é de 1 mais uma atribuição implícita, portanto, o corpo do laço executa de 2 + 2 = 4, já que será executado repetidas vezes, ou seja, todos os n números de um intervalo menos dois valores (um e ele mesmo), logo, 4 * (n-2) = 4n - 8. O laço fará uma atribuição, n-1 comparações e n-2 incrementos, assim sendo (1) + (n-1) + (n-2) = 2n - 2. O número de instruções executadas pela função *somaDivisores* é: (2n - 2) + (4n - 8) = 6n - 10, onde n corresponde ao número de entradas.

Para a função *criaCacheEntre*, só o corpo do laço leva (6n-10) * (n+1) = $6n^2$ - 4n - 10 instruções, incluindo as operações feitas pelo laço, ou seja, um incremento, n+2 comparações e n+1 incrementos, para colocar as somas na cache leva ($6n^2$ - 4n - 10) + (1 + n+2 + n+1) = $6n^2$ - 2n - 6 instruções.

Se o cálculo de números mutuamente amigos for feito sem o uso do vetor cache, isto é, com a função *calculaNumerosMutuamenteAmigosEntre*, ele deverá chamar *somaDivisores* várias vezes, o impactará no desempenho, veja a seguir:

1- A impressão de uma mensagem, 1, antes, uma condição de comparação deve ser satisfeita, levando mais 1 para execução. Quando a função *somaDivisores* é chamada, ela gasta 6n - 10 instruções, o valor retornado é atribuído a uma variável levando mais 1; outra variável recebe uma divisão. O corpo do laço interno é de (6n - 10) + 2 + 2 + 1 = 6n - 5 e será executado n vezes e, este laço, n+1 vezes, logo, total de instruções do laço interno é de (6n-5) * (n) * (n+1) = $6n^3 + n^2 + 5n$;

2- O laço externo, assim como no interno, possui uma chamada de função, duas atribuições e uma divisão, levando tempo de (6n - 10) +(2) + (1) = 6n - 7 e será executado n vezes, logo, (6n - 7) * (n) = $6n^2$ - 7n;

3- Os laços em si realizam atribuições, comparações e incrementos. No laço interno há uma soma e uma atribuição, portanto, 1 + 1 = 2; n+1 comparações, onde a última é falsa, saindo do laço; e como incrementos só são realizados quando uma comparação é verdadeira, logo, n incrementos são feitos. O laço interno tem (2) + n+1 + n = 2n + 2 instruções e como será executado n vezes pelo laço externo: (2n + 2) * n = $2n^2$ + 2n. De

forma semelhante, o laço externo executa 1 incremento, n+2 comparações e n+1 incrementos, portanto, 1 + (n+2) + (n+1) = 2n + 4;

4- Total de instruções da função: $(6n^3 + n^2 + 5n) + (6n^2 - 7n) + (2n + 4) + (2n^2 + 2n) = 6n^3 + 9n^2 + 2n + 4$.

Com o uso do vetor cache, a execução é a mesma, porém, não é feita diversas chamadas à função *somaDivisores*.

1- No corpo do laço interno, há uma impressão, uma condição, três somas, uma divisão e uma atribuição, portanto, 1 + 1 + 3 + 1 + = 6. como será executado n vezes pelo laço interno e este n + 1 vezes pelo laço externo, terá execução de 6 * (n) * (n+1) = $6n^2 + 6n$;

2- No corpo do laço externo possui uma divisão, uma soma e uma atribuição, 1 + 1 + 1 = 3 e será executado n + 1 vezes pelo laço externo, portanto o total de instruções do corpo do laço externo é $(3*(n+1)) + 6n^2 + 6n = 6n^2 + 9n + 3$;

3- Os laços interno e externo executam, respectivamente, uma soma, uma atribuição, n+1 comparações e n incrementos, tendo resultado de (n+1) + n + 2 = 2n + 3 e será executado n+1 vezes, logo, $(2n + 3) * (n+1) = 2n^2 + 5n + 3$; uma atribuição n+2 comparações e n+1 incrementos, total de (1) + (n+2) + (n+1) = 2n + 4 instruções;

4- Total de instruções para execução: $(6n^2 + 9n + 3) + (2n + 4) + (2n + 3) = 6n^2 + 13n + 10$, somando com o que leva para a cache alocar cada espaço somado: $(6n^2 + 13n + 10) + (6n^2 - 2n - 6) = 12n^2 + 9n + 4$.

A ordem de complexidade dos cálculos usando cache é de $O(n^2)$, caso contrário, é de $O(n^3)$, ou seja, a medida que o número de entradas n aumenta, *calculaNumerosMutuamenteAmigosEntre* leva mais instruções para processar as informações que a função *calculaNumerosMutuamenteAmigosEntreUsandoCache*.

Resultados

Para comprovar que o programa funciona de forma consistente e estável, resolvemos testar ambas as versões do programa - tanto a versão sequencial, quanto a versão paralela - em dois computadores com configurações distintas:

Computador A (desktop)	**Computador B (notebook)**
● Intel Core i5 650 ○ Clock: 3.2 GHz ○ Clock *máximo*: 3.46 GHz ○ Quantidade de núcleos reais: 2 ○ Quantidade de núcleos virtuais: 4 ○ Cache: 4 MB ○ Conjunto de instruções: 64 Bit ● 4 GB de RAM DDR3 1333Mhz ● 1 TB de HD 7200 RPM ● Ubuntu 14.04 LTS	● Intel Core i7 3612QM ○ Clock: 2.1 GHz ○ Clock *máximo*: 3.1 GHz ○ Quantidade de núcleos reais: 4 ○ Quantidade de núcleos virtuais: 8 ○ Cache: 6 MB ○ Conjunto de instruções: 64 Bit ● 8 GB de RAM DDR3 1333Mhz ● 1 TB de HD 5400 RPM ● Ubuntu 12.04 LTS

No caso do Computador B, que é um notebook, os testes foram realizados com o dispositivo conectado à tomada, com o modo de economia de energia desligado, possibilitando o aproveitamento máximo do conjunto de hardware do computador.
Note também que o valor que se refere ao "Clock Máximo" *pode ou não* ser atingido durante a execução do programa, sendo de responsabilidade do algoritmo interno da Intel - fabricante do processador - a decisão de aumentar ou não a velocidade do clock caso for possível. Para todo efeito, vamos assumir, neste trabalho, que o algoritmo está executando a velocidade de clock nominal padrão.
Além disso, há algumas informações importantes a serem consideradas:

- No momento da execução de cada teste, nenhum programa não essencial está em execução. Ou seja, softwares como o navegador, IDE, editor de texto e outras coisas do gênero não estarão em execução para não afetar no desempenho do programa testado;
- Ambas as versões do programa testado foram compiladas adicionando "**-O3**" aos argumentos do compilador. Tal flag ativa todas as otimizações mais indicadas pelo compilador, de forma a gerar um código assembly mais otimizado e eficaz;
- Ambas as versões são executadas 3 vezes em cada computador, com cada conjunto de parâmetros. Isso foi feito para garantir que a execução do programa segue uma velocidade mais ou menos constante, e assim obter maior confiabilidade sobre quanto tempo cada versão demora para ser executada com um determinado conjunto de parâmetros;
- A impressão do programa (chamadas a funções **printf**) foi desativada para realizar os benchmarks. Isso acontece por dois motivos principais:
    - Quando uma função do sistema é executada, o programa se torna dependente da implementação dessa função. Isso significa que não é mais somente o algoritmo do programa que está em jogo, mas sim o do sistema;

    - Além disso, funções do sistema fazem com que o sistema operacional tome posse do processador, em termos mais claros, isso significa que enquanto a função é executada, o sistema operacional pode ter a liberdade de passar o comando do processador a outro programa, afetando a velocidade do programa sendo testado. A prova disso é que, em todas as execuções, o campo “sys” indica um valor insignificante para a execução do software. Esse valor só não é zero pois há outras formas também do sistema operacional ganhar a posse do processador, mas, na prática, tomar o controle a partir da chamada de funções do sistema é uma das formas mais simples do sistema operacional tomar posse da execução.
- Esses são os parâmetros escolhidos para a execução do programa. Note que o parâmetro “*NTHREADS*” é aplicável **somente** à versão paralela do programa:
    - Parâmetros 1
        - MIN=1
        - MAX=100000
        - NTHREADS=2
    - Parâmetros 2
        - MIN=1
        - MAX=100000
        - NTHREADS=4
    - Parâmetros 3
        - MIN=10000
        - MAX=200000
        - NTHREADS=2
    - Parâmetros 4
        - MIN=10000
        - MAX=200000
        - NTHREADS=4
- Em cada teste realizado, é mostrado a média aritmética dos tempos, assim como o desvio padrão (raiz quadrada da média) e o coeficiente de variação (desvio padrão/média), sendo que os dois últimos mostram o quanto os dados estão dispersos em relação à média. Por exemplo, se coeficiente de variação for menor que 0.25, então os dados são homogêneos, tendo pouca distinção entre si. *(Nota: alguns valores estão em suas aproximações)*;
- As comparações entre as execuções são realizadas de acordo com a razão das médias geradas.

Para facilitar o entendimento dos testes, resolvemos separar essa seção em resultados por **computador**, de forma que todos os parâmetros relatados acima são testados em um único computador por seção. Veja abaixo:

Computador A

Parâmetros 1

Versão sequencial:

real 0m20.501s
user 0m20.498s

sys 0m0.008s

real 0m20.467s
user 0m20.448s
sys 0m0.024s

real 0m20.502s
user 0m20.503s
sys 0m0.004s

Média: 20.49s
Desvio padrão: 4.526s
Coeficiente de Variação: 0.22

Versão paralela:

real 0m10.232s
user 0m20.449s
sys 0m0.020s

real 0m10.239s
user 0m20.477s
sys 0m0.004s

real 0m10.234s
user 0m20.465s
sys 0m0.008s

Média: 10.235s
Desvio padrão: 3.2s
Coeficiente de Variação: 0.31

Parâmetros 2

Versão sequencial:

real 0m20.429s
user 0m20.430s
sys 0m0.004s

real 0m20.492s
user 0m20.472s
sys 0m0.024s

real 0m20.486s
user 0m20.462s
sys 0m0.028s

Média: 20.469s
Desvio padrão: 4.524s
Coeficiente de Variação: 0.22

Versão paralela:

real 0m10.228s
user 0m40.255s
sys 0m0.020s

real 0m10.229s
user 0m40.207s
sys 0m0.012s

real 0m10.226s
user 0m40.196s
sys 0m0.040s

Média: 10.227s
Desvio padrão: 3.198s
Coeficiente de Variação: 0.32

Parâmetros 3

Versão sequencial

real 1m20.225s
user 1m20.132s
sys 0m0.116s

real 1m20.504s
user 1m20.489s
sys 0m0.036s

real 1m20.540s
user 1m20.523s
sys 0m0.036s

Média: 80.423s = 1m20.423s
Desvio padrão: 8.967s
Coeficiente de Variação: 0.12

Versão paralela

real 0m40.228s
user 1m20.451s
sys 0m0.024s

real 0m40.231s
user 1m20.413s
sys 0m0.068s

real 0m40.219s
user 1m20.455s

sys 0m0.004s

Média: 40.226s
Desvio padrão: 6.342s
Coeficiente de Variação: 0.15

Parâmetros 4

Versão sequencial

real 1m20.416s
user 1m20.421s
sys 0m0.020s

real 1m20.512s
user 1m20.493s
sys 0m0.040s

real 1m20.493s
user 1m20.490s
sys 0m0.024s

Média: 80.473s = 1m20.473s
Desvio padrão: 8.970s
Coeficiente de Variação: 0.12

Versão paralela

real 0m40.216s
user 2m38.026s
sys 0m0.120s

real 0m40.213s
user 2m37.982s
sys 0m0.167s

real 0m40.216s
user 2m37.659s
sys 0m0.175s

Média: 40.215s
Desvio padrão: 6.341s
Coeficiente de Variação: 0.15

Conclusão do desempenho do computador A: A partir dos dados, nota-se que os coeficientes de variação são bem homogêneos (pouco dispersos), confirmando que a média das versões sequencial e paralela estão nos valores definidos. A versão paralela consegue ser, aproximadamente, duas vezes mais rápida que a versão sequencial nos parâmetros 1 e 3, porque consegue dividir as tarefas entre os dois núcleos reais do processador, enquanto sequencialmente, não. Mesmo trabalhando com quatro threads nos parâmetros 2 e 4, a versão paralela ainda consegue ser apenas duas vezes mais

rápida que a sequencial, porque o processador deste computador possui apenas dois núcleos reais e quatro virtuais, portanto, em alguns períodos, a divisão de tarefas entre dois núcleos virtuais em um mesmo núcleo real pode não ser perfeita, diminuindo assim o potencial de aceleração. Evidentemente, os dois últimos parâmetros levam mais tempo para processar devido ao grande intervalo e os próprios valores pertencentes definidos pelos extremos.

Computador B

Parâmetros 1

Versão Sequencial

real 0m22.886s
user 0m22.897s
sys 0m0.000s

real 0m22.827s
user 0m22.831s
sys 0m0.004s

real 0m22.835s
user 0m22.843s
sys 0m0.000s

Média: 22.849s
Desvio padrão: 4.780098s
Coeficiente de Variação: 0.209201

Versão Paralela

real 0m11.809s
user 0m23.624s
sys 0m0.000s

real 0m11.787s
user 0m23.577s
sys 0m0.004s

real 0m11.790s
user 0m23.584s
sys 0m0.004s

Média: 11.795s
Desvio padrão: 3.434433s
Coeficiente de Variação: 0.291169

Parâmetros 2

Versão Sequencial

real 0m23.066s
user 0m23.070s

sys 0m0.004s

real 0m23.059s
user 0m23.067s
sys 0m0.000s

real 0m23.054s
user 0m23.058s
sys 0m0.004s

Média: 23.059s
Desvio padrão: 4.802048s
Coeficiente de Variação: 0.208244

Versão Paralela

real 0m6.269s
user 0m25.079s
sys 0m0.004s

real 0m6.269s
user 0m25.078s
sys 0m0.004s

real 0m6.268s
user 0m25.078s
sys 0m0.000s

Média: 6.268s
Desvio padrão: 2.503731s
Coeficiente de Variação: 0.399404

Parâmetros 3

Versão Sequencial

real 1m30.380s
user 1m30.404s
sys 0m0.012s

real 1m30.498s
user 1m30.534s
sys 0m0.000s

real 1m30.432s
user 1m30.468s
sys 0m0.000s

Média: 1m30.436s
Desvio padrão: 9.509819s
Coeficiente de Variação: 0.105154

Versão Paralela

real 0m46.559s
user 1m33.150s
sys 0m0.004s

real 0m46.594s
user 1m33.224s
sys 0m0.000s

real 0m46.610s
user 1m33.249s
sys 0m0.008s

Média: 46.587s
Desvio padrão: 6.825516s
Coeficiente de Variação: 0.146509

Parâmetros 4

Versão Sequencial

real 1m32.199s
user 1m32.203s
sys 0m0.008s

real 1m33.259s
user 1m33.238s
sys 0m0.004s

real 1m33.175s
user 1m33.119s
sys 0m0.004s

Média: 92.877s
Desvio padrão: 9.637306s
Coeficiente de Variação: 0.103763

Versão Paralela

real 0m24.874s
user 1m39.471s
sys 0m0.004s

real 0m24.824s
user 1m39.281s
sys 0m0.000s

real 0m24.828s
user 1m39.296s

sys 0m0.000s
Média: 24.842s
Desvio padrão: 4.984175s
Coeficiente de Variação: 0.200635

Conclusão do desempenho do computador B: Os dados mostram que, graças aos quatro núcleos reais do processador, nos parâmetros 2 e 4 a versão paralela é aproximadamente 3.6 vezes mais rápida que a versão sequencial, justamente pela maior divisão de tarefas realizadas pelo processador. Nos parâmetros 1 e 3 houve um aumento de desempenho perto de duas vezes (1.9) pela versão paralela sobre a sequencial.

Conclusão

Nos computadores avaliados o primeiro (A) possui resultados favoráveis quando se explora os dois núcleos reais presentes, aplicação com duas threads, conseguindo ter um desempenho superior ao do computador B, com mais núcleos. Por exemplo, nas versões paralelas do parâmetro 1 de ambos, o computador A consegue ser 1 vez mais rápido que o computador B. O segundo (B) é ótimo para soluções com intervalos definidos por números elevados, justamente pela maior divisão de tarefas que é capaz de fazer, por exemplo, comparando o parâmetro 3 do computador A com o parâmetro 4 de B nas versões paralelas, embora possuam o mesmo intervalo e exploram todos os núcleos reais, o computador B consegue ter um desempenho aproximadamente 1.6 vezes maior que o do computador A.