Nomes: Emmanuel Podestá Junior, Fernando Paladini.
Turma: 03208B (2014 / 2).
Disciplina: INE5410 - Programação Concorrente.

Relatório - Trabalho 3
Números Mutuamente Amigos

Florianópolis, 01 de Novembro de 2014.

Introdução

O trabalho de implementação três da disciplina de Programação Concorrente (INE5410) aborda o problema dos números mutuamente amigos, que está descrito na seção “Descrição do Problema” desse relatório. De acordo com a descrição, a resolução do problema deve ser feita de forma sequencial e paralelizada utilizando a tecnologia desejada pelo grupo. As soluções que encontradas estão descritas na seção “Descrição das Soluções”. Também tratamos de outras possíveis abordagens para a resolução do problema na seção “Outras Abordagens”. Ao final deste relatório apresentamos os resultados das versões sequencial e paralela e as referências utilizadas para se elaborar este trabalho.

Descrição do Problema

O problema proposto para o trabalho 3 de Programação Concorrente é o problema dos números mutuamente amigos. A imagem (1.a) mostra a especificação geral do problema.

Dois números a e b são *mutuamente amigos* se a razão entre a soma de todos os divisores do número a e o número a é igual a razão entre a soma de todos os divisores do número b e o número b.

Por exemplo, os números 30 (Equação 1) e 140 (Equação 2) são *mutualmente amigos*, pois apresentam a mesma razão entre soma dos seus divisores e eles mesmos ($\frac{12}{5}$):

$$\frac{1+2+3+5+6+10+15+30}{30} = \frac{72}{30} = \frac{12}{5} \quad (1)$$

$$\frac{1+2+4+5+7+10+14+20+28+35+70+140}{140} = \frac{336}{140} = \frac{12}{5} \quad (2)$$

O problema consiste em encontrar todos os números mutuamente amigos entre determinado intervalo de valores (por exemplo, de 10.000 à 20.000), ou seja, todos os números que tem a razão entre a soma dos seus divisores e o próprio número **igual** a razão entre a soma dos divisores de outro número da série e o próprio número.

O algoritmo para a solução do problema especificado deve ser implementado pelos grupos de forma sequencial e posteriormente de forma paralela. A tecnologia empregada na implementação do algoritmo sequencial e paralelo fica a critério de cada grupo, desde que a tecnologia escolhida tinha sido vista em aula. O usuário deve poder interagir com o programa informando como primeiro argumento o número mínimo do intervalo e como segundo argumento o número máximo do intervalo a ser verificado. No caso da implementação paralela um terceiro argumento deve estar disponível: o número de threads que se deseja executando o problema.

Ao final da implementação a eficácia de ambos os algoritmos (sequencial e paralelo) será determinada através do tempo que cada algoritmo demora para executar sob determinadas condições. Considerando-se um tempo de execução com velocidade 1x para o programa sequencial, o programa paralelo executado com duas threads deve ter uma velocidade de aproximadamente 2x, o programa paralelo executado com 4 threads deve ter uma velocidade de aproximadamente 4x - e assim por diante, mas respeitando as limitações do hardware. Além da detalhamento dos *benchmarks,* também é necessário descrever o hardware do computador em que os programas sequencial e paralelo foram executados.

Descrição das soluções

Ambas as soluções adotadas pelo nosso grupo foram implementadas na linguagem C devido à nossa experiência com esta linguagem para solução de problemas em programação paralela e concorrente. Os algoritmos sequencial e paralelo que resolvem o problema dos números mutuamente amigos foram concebidos do zero pela nossa equipe e dentre todas as soluções encontradas, estes se mostraram os algoritmos com o menor Big O. A melhor ordem de complexidade que conseguimos para os nossos algoritmos foram de O(n²).

Solução Sequencial

A solução sequencial adotada pelo nosso grupo pode ser encontrada anexada ao arquivo .zip em que está contido este relatório. O arquivo se chama “mutuamenteAmigosSeq.c” e será explicado trecho à trecho logo abaixo.

Método *main* e recuperação de valores: Na linha 41 da imagem “seq-01” temos a declaração padrão do método *main* em C. Um pouco mais abaixo (linhas 43 e 44) declaramos duas variáveis para armazenar o intervalo mínimo e máximo que o programa deve executar em busca dos números mutuamente amigos. O valor mínimo e máximo é passado para o programa através de argumentos no momento em que o programa é executado pelo usuário.

```
41  int main(int argc, char **argv) {
42
43      int minimo = atoi(argv[1]);
44      int maximo = atoi(argv[2]);
```

Imagem “seq-01”.

Variáveis úteis ao programa: A imagem “seq-02” mostra a declaração de algumas variáveis que serão úteis durante a execução do programa. Na linha 46 declaremos duas variáveis para controle de laços (*i* e *j*) e uma variável chamada *intervalo* que nos servirá para controlar alguns laços.

```
46      int i, j, intervalo = maximo-minimo;
47      double fracaoMutual;
48      double *fracoes = (double*) malloc ((intervalo + 1) * sizeof(double));
```

Imagem “seq-02”.

Na linha 47 declaramos a variável *fracaoMutual* que será usada para armazenar o valor de $\frac{(soma\ dos\ divisores\ de\ X)}{X}$, onde $minimo \geq X \leq maximo$. O vetor de variáveis do tipo “double”

denominado *fracoes*, na linha 48, armazenará todas as frações calculadas durante a execução do programa e será utilizado posteriormente para descobrir os números mutuamente amigos. O vetor não é estritamente necessário para a solução do problema, mas nesse caso a complexidade do programa teria que ser aumentada de $O(n^2)$ para $O(n^3)$. A ideia por trás dessa decisão é que trocar armazenamento por processamento é sempre vantajoso, pois o processamento de informações é mais caro do que o armazenamento destas.

Verificação de alocação de memória: após declarar o vetor *fracoes* é necessário verificar se toda a memória necessária pode ser alocada ou se há algum problema. Isso é feito através de uma comparação do vetor com NULL, como mostra a imagem "seq-03". Caso o vetor seja NULL, significa que a alocação de memória não foi bem sucedida e o programa deve encerrar com uma mensagem de erro. Caso contrário, pode prosseguir com a execução.

```
50      if(fracoes != NULL) {
```

Imagem "seq-03".

Cálculo das frações: todos os números no intervalo fornecido precisam ser calculados, para isso precisamos de um laço que percorra todos esses números e gere as suas respectivas frações. No contexto desse trabalho uma "fração" pode ser definida como: dado um número X, uma fração de um número é: $\frac{(soma\ dos\ divisores\ de\ X)}{X}$.

```
51      for (i = minimo; i <= maximo; i++) {
52          fracaoMutual = (double)calculaSomaDosDivisores(i) / i;
53          fracoes[i - minimo] = fracaoMutual;
54      }
```

Imagem "seq-04".

O laço para percorrer todos os números do intervalo está declarado na linha 51. Para calcular a fração do número "i" são necessários dois passos:

1. Cálculo da soma de todos os divisores de "i". Pode ser definido matematicamente como $\sum_{a=0}^{i} \forall(i \,\%\, a \;==\; 0)$.
2. Cálculo da fração de "i". Para tal, basta realizar $fracao \;=\; \frac{somaDosDivisores}{i}$.

Para nos auxiliar na execução deste cálculo utilizamos o método calculaSomaDosDivisores(int i), que será explicado mais abaixo. Por fim, na linha 53, a fração de "i" é armazenada no vetor *fracoes*, na posição "i - minimo" (pois queremos armazenar nos índices corretos do vetor, começando no índice zero).

Cálculo da soma dos divisores: o algoritmo para cálculo da soma dos divisores pode ser encontrado na imagem “seq-05”. Na linha 14 declaramos duas variáveis auxiliares para realizar a soma de todos os divisores do número fornecido. A variável que armazenará a soma é inicializada com o número fornecido pois ele próprio já é um divisor que precisa ser adicionado ao resultado final. A variável de controle “j” é inicializada com o valor “i / 2” pois o primeiro divisor do número dado nunca será maior do que a metade desse próprio número - no máximo será um número igual que a metade. A partir dai todos os números menores do que “i / 2” são testados para saber se são divisores do número dado. Caso o número seja um divisor, este é adicionado à variável “soma”. Ao final do cálculo é retornado a soma total dos divisores, conforme linha 20.

```
13  int calculaSomaDosDivisores(int i) {
14      int j, soma = i;
15      for(j = (int) i / 2; j > 0; j--){
16          if(i % j == 0) {
17              soma += j;
18          }
19      }
20      return soma;
21  }
```

Imagem “seq-05”.

O teste que verifica se o número fornecido é divisível por “j” é realizado entre as linhas 16 e 18. Caso o número seja um divisor, este número é adicionado à variável soma.

Checagem de números mutuamente amigos: esta é o último trecho da nossa solução e novamente é um algoritmo que possui complexidade O(n²). A implementação desse trecho pode ser visualizada na imagem “seq-06”.

```
56      for(i = 0; i <= intervalo; i++){
57          for (j = i+1; j <= intervalo; j++) {
58              if(fracoes[i] == fracoes[j]) {
59                  printf("Os numeros %d e %d são mutuamente amigos.\n", (minimo + i), (minimo + j));
60              }
61          }
62      }
```

Imagem “seq-06”.

Esse trecho de código faz uma busca no vetor procurando por frações que são iguais. Caso duas frações tenham o mesmo valor (ou seja, frações que se originaram de dois números mutuamente amigos), é impresso na tela os números que originaram essa fração (simplesmente somando o “minimo” do intervalo com o “i” e “j” atuais, o que originará os números mutuamente amigos que resultaram nessa fração).

Desalocação de memória e fim do programa: a próxima etapa é desalocar a memória alocada para o vetor de *double's* que armazenava as "frações" do programa. A desalocação de memória pode ser vista na imagem "seq-07".

```
64          free(fracoes);
65          return 0;
```

Imagem "seq-07".

Após desalocar a memória alocada , nos resta apenas sair do programa com um "return 0", indicando que a execução do programa foi realizada sem maiores problemas.

Solução Paralela

A solução paralela encontrada pelo grupo para o problema dos números mutuamente amigos utiliza a tecnologia OpenMP (biblioteca "omp.h"). Essa API foi escolhida devido a sua facilidade de implementação e manutenção, além de ser uma das tecnologias para paralelismo de mais alto nível, o que promove uma ótima abstração ao desenvolvedor. Como grande parte do código se mantém o mesmo, apenas as diferenças entre a versão paralela e a versão sequencial serão explicadas.

Variáveis e recuperação de valores: as principais diferenças estão nas linhas 55 e 61. Na linha 55 é armazenado o número de threads que vão executar o programa, que é um argumento que foi passado pelo usuário no momento em que o programa foi executado. Na linha 61 é definido para o OpenMP que as áreas paralelos desse programa devem rodar com o número de threads passado como argumento pelo usuário. Por exemplo, se o usuário passar 4 como argumento de número de threads, todas as regiões paralelas do código serão (ao menos por padrão) executadas com 4 threads.

```
53      int minimo = atoi(argv[1]);
54      int maximo = atoi(argv[2]);
55      int threads = atoi(argv[3]);
56
57      int i, j, intervalo = maximo-minimo;
58      double *fracoes = (double*) malloc ((intervalo + 1) * sizeof(double));
59
60      if (fracoes != NULL) {
61          omp_set_num_threads(threads);
```

Imagem "paral-01".

Cálculo das frações: nesse trecho de código as principais diferenças entre o programa paralelo e o programa sequencial estão nas linhas 63-65, conforme imagem "paral-02". As diretrizes "omp parallel" e "omp for schedule(static)" indicam que essa é uma área que será executada de forma paralela, onde cada thread ira executar aproximadamente $\frac{\text{total de iterações}}{\text{threads}}$

iterações do laço. Aproximadamente pois caso a divisão não seja exata, a biblioteca OpenMP cuidará da distribuição das iterações restantes.

```
63          #pragma omp parallel
64          {
65              #pragma omp for schedule(static)
66              for (i = minimo; i <= maximo; i++) {
67                  double fracaoMutual = (double)calculaSomaDosDivisores(i) / i;
68                  fracoes[i - minimo] = fracaoMutual;
69              }
70          }
```

Imagem "paral-02".

Checagem de números mutuamente amigos: a checagem acontece praticamente da mesma forma que no algoritmo sequencial, como é possível verificar na imagem "paral-03". Da mesma forma que no algoritmo anterior, as diretrizes "*omp parallel*" e "*omp for schedule(static)*" definem que esta região será executada paralelamente pelo número de threads informada pelo usuário.

```
72  for(i = 0; i <= intervalo; i++){
73      #pragma omp parallel
74      {
75          #pragma omp for schedule(static)
76          for (j = i+1; j <= intervalo; j++) {
77              if(fracoes[i] == fracoes[j]) {
78                  printf("Os numeros %d e %d são mutuamente amigos.\n", (minimo + i), (minimo + j));
79              }
80          }
81      }
82  }
```

Imagem "paral-03".

A diretriz "for schedule(static)" define que todas as threads que executarão o loop terão a mesma quantidade de iterações para realizar, ficando implícito a existência de uma ordem específica de qual thread realizará qual iteração. A diretriz "for schedule(dynamic [, chunk])" não foi utilizada pois não há um *chunk* ótimo ou bom para todos os casos, o *chunk* ideal varia de caso para caso. Por esse motivo decidimos que seria melhor utilizar o *schedule(static)*, pois seria mais adequado para casos genéricos, que é onde o programa mais será usado (intervalos com grande diferença entre si). Optamos por utilizar o paralelismo apenas no laço interno por motivos de performance, uma vez que em nossos testes o problema se mostrou alguns segundos mais rápido utilizando a região paralela no laço interno.

Outras abordagens

Durante a busca por soluções mais rápidas do que a solução descrita nesse relatório pensamos em algumas abordagens possivelmente mais rápidas para esse problema.

Otimizações na compilação

A primeira abordagem é a mais óbvia: utilizar as ferramentas do compilador gcc para melhorar a performance do programa. A redução de código morto, *loop unrolling*, escalonamento estático e outras otimizações presentes através das *flags* de compilação -O2 e -O3 otimizariam **muito** o código. Não utilizamos essas *flags* de compilação nos testes deste relatório pois esta informação não constava na descrição do problema, portanto preferimos nos manter céticos quanto ao uso das ferramentas do compilador. Entretanto, ainda assim analisamos as melhorias causadas por algumas *flags* de compilação, como a -O2 e a -O3. Esses testes de performance não foram documentados, mas pudemos analisar que ocorreram grandes melhoras de desempenho no programa ao utilizar tais recursos. Os resultados chegaram a se mostrar com a metade do tempo original (sem *flag* de otimização na compilação), o que indica um ganho gigantesco de performance.

Numerador e denominador no lugar de frações

A utilização de numeradores e denominadores no lugar das “frações” também foi uma hipótese testada nas soluções que desenvolvemos. Embora esse método aparente ser mais simples e mais eficiente por apenas utilizar valores *inteiros*, nos testes realizados não percebemos ganhos significativos de performance que pudessem ser definidos como ganhos reais de performance. A performance do programa se encontrava dentro de uma margem de erro equivalente à apresentada pela solução proposta neste relatório. Em determinado momento podemos notar até mesmo o efeito contrário: uma pequena piora no tempo de execução do programa ocorreu. A implementação que utilizou o método do **máximo divisor comum** de forma recursiva foi a que mostrou uma leve piora no tempo de execução; já a versão iterativa desse método mostrou uma leve melhora no tempo de execução do programa - mas ainda dentro da já citada margem de erro.

Método da soma dos divisores alternativo

Pensamos também em utilizar uma fórmula alternativa para realiza a soma dos divisores de um dado número (http://planetmath.org/formulaforsumofdivisors), mas após uma análise mais profunda percebemos que o desempenho dessa solução seria no máximo equivalente ao

desempenho da solução deste relatório.A performance provavelmente deterioaria bastante o tempo de execução do programa (por ter que encontrar os números primos e realizar cálculos sobre estes para só então gerar a soma dos divisores).

Uma abordagem mais avançada

A solução descrita neste relatório tem complexidade $O(n^2)$, mas podemos ser mais precisos e analisar de forma mais detalhada a complexidade desse programa. O trecho do código responsável por calcular as frações de todos os números do intervalo tem complexidade n^2. A parte do código responsável por checar os números que são mutuamente amigos também tem complexidade n^2. Desconsiderando as constantes da complexidade do algoritmo, podemos assumir que a complexidade do algoritmo é $\mathbf{n^2 + n^2}$. Dito isso, existe uma possibilidade de reduzir a complexidade do algoritmo para $\mathbf{n^2 + n}$, mas devido a sua dificuldade não conseguimos implementar na linguagem C (pelas características da própria linguagem, como por exemplo programação estruturada).

A solução consistiria em utilizar uma árvore binária balanceada (AVL) ou um hashing mapeando múltiplos valores para uma única chave. Tentarei detalhar a ideia de utilizar um hash como estrutura de dados ao invés de um vetor. A ideia é que a “key” do hash seja um *double* correspondendo à uma determinada “fração” e os “values” dessa *key* sejam os números que geraram essa “fração”. Desprezando valores reais, a representação dessa estrutura seria algo como:

{ 1.200323 => { 100, 204 }, 2.12312 => { 687, 932 }, 1.765102 => { 321, 468 }, ... }

Ao invés de adicionar às frações ao vetor de *double’s* teríamos que adicionar as “frações” como chaves do hash e o inteiro que originou essa “fração” seria adicionado como valor dessa chave recém adicionada. Vou tentar exemplificar um caso de uso dessa estrutura: digamos que a fração X do inteiro A tenha sido adicionada ao hash e que algumas iterações depois um inteiro B tenha essa mesma fração, ou seja, tenha fração X. Nesse caso o inteiro B **simplesmente seria adicionado como um novo valor da chave X**. A complexidade desse trecho do código se manteria a mesma. No trecho de código em que os números mutuamente amigos são checados poderíamos retirar os dois laços aninhados e colocar apenas um loop que iterasse por todo o hash em busca das chaves que tem mais de 1 valor no seu interior (as “frações” que tem mais de um inteiro como originário, ou seja, tem um número mutuamente amigo). As chaves que tem mais de um valor deveriam imprimir esses valores, pois ambos seriam mutuamente amigos. A complexidade desse trecho seria reduzida de $\mathbf{n^2}$ **para n**. Devido às limitações da linguagem C não pudemos implementar essa possível solução e verificar se ela era de fato mais rápida do que a solução proposta neste relatório, logo é apenas uma suposição teórica.

Resultados

Os resultados mostrados a seguir foram feitos com um computador com processador Intel® Core™ i3-3110M, com frequência de 2.40Ghz x 2 (2 núcleos, 4 threads). Sistema operacional Fedora 20, 64 bits e Kernel versão 3.16.6-200.fc20.x86_64. Os testes foram executados sem nenhum outro programa em execução, de forma que os dados foram anotados manualmente e depois passados para esta tabela.

Intervalo	Sequêncial	Paralelo(2x)	Paralelo(3x)
1 à 100000	real 0m24.184s user 0m24.177s sys 0m0.001s	real 0m13.437s user 0m22.135s sys 0m0.022s	real 0m9.567s user 0m24.067s sys 0m0.040s
1 à 200000	real 1m36.767s user 1m36.747s sys 0m0.003s	real 0m53.566s user 1m28.196s sys 0m0.052s	real 0m39.556s user 1m38.103s sys 0m0.082s
1 à 500000	real 10m9.451s user 10m9.326s sys 0m0.014s	real 5m34.041s user 9m10.059s sys 0m0.134s	real 4m22.150s user 10m26.265s sys 0m0.206s
10000 à 200000	real 1m30.653s user 1m30.634s sys 0m0.001s	real 0m50.132s user 1m23.199s sys 0m0.046s	real 0m35.477s user 1m31.594s sys 0m0.068s
10000 à 500000	real 9m54.451s user 9m54.346s sys 0m0.002s	real 5m25.468s user 8m57.548s sys 0m0.129s	real 4m19.689s user 10m21.822s sys 0m0.176s

Referências

WIKIPEDIA (Org.). **Número Amigo.** Disponível em: <http://pt.wikipedia.org/wiki/Número_amigo>. Acesso em: 01 nov. 2014.

WOLFRAM MATHWORLD (Comp.). **Friendly Number.** Disponível em: <http://mathworld.wolfram.com/FriendlyNumber.html>. Acesso em: 01 nov. 2014.

WIKIPEDIA (Org.). **Friendly Number.** Disponível em: <http://en.wikipedia.org/wiki/Friendly_number>. Acesso em: 01 nov. 2014.

WIKIPEDIA (Org.). **OpenMP.** Disponível em: <http://pt.wikipedia.org/wiki/OpenMP>. Acesso em: 01 nov. 2014.

SALIHUN, Darmawan. **OpenMP vs OpenMPI.** Disponível em: <http://darmawan-salihun.blogspot.com.br/2009/11/openmp-vs-openmpi.html>. Acesso em: 01 nov. 2014.

PLANET MATH (Org.). **Formula for sum of divisors.** Disponível em: <http://planetmath.org/formulaforsumofdivisors>. Acesso em: 04 nov. 2014.