UNIVERSIDADE FEDERAL DE MINAS GERAIS
ALGORITMOS E ESTRUTURAS DE DADOS III

TP6
PARALELIZAÇÃO

Aluna: Thalia de Almeida Magalhães Campolina
e-mail: thaliacampolina@gmail.com
Matrícula: 2010054827

Belo Horizonte, Minas Gerais. Julho de 2012.

STABLE MARRIAGE PROBLEM

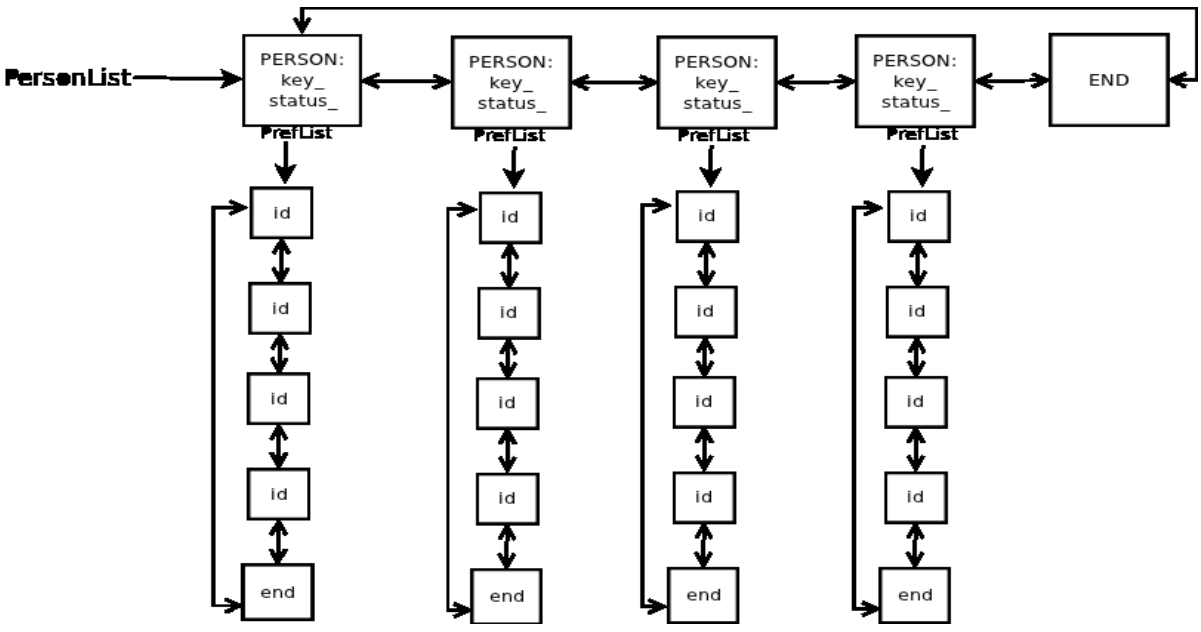
1. INTRODUÇÃO

O problema apresentado a seguir é conhecido como Problema do Casamento Estável, para o qual será apresentado o algoritmo para sua resolução.

O SMP (Stable Marriage Problem) consiste em encontrar a melhor combinação de casais, ou seja, de casamentos, para que esses sejam todos estáveis, dado um conjunto de homens e mulheres, e suas respectivas escolhas de parceiros. Essas escolhas de parceiros são apresentadas, respectivamente, pela primeira opção de casamento, seguida das outras opções em ordem de pteferência. O número de homens e mulheres precisa ser equivalente, para que o algoritmo funcione.
Um exemplo de lista de preferências pode ser dado como:
M1: H3 H1 H2
M2: H3 H2 H1
M3: H1 H3 H2
H1: M1 M2 M3
H2: M2 M1 M3
H3: M3 M2 M1
Sendo que M é representação de Mulher, e H representação de Homem.
O casamento é instável caso um dos conjuges A do casal prefera outra pessoa C para estar casada, e essa pessoa C também prefera a pessoa A no lugar do seu atual cônjuge. Se isso nunca ocorrer entre os casais listados, os casamentos serão Estáveis.

2. MODELAGEM E SOLUÇÃO

A estrutura utilizada para a resolução foi Lista Encadeada Circular. Essa lista pode ser dividida entre lista de mulheres, e lista de homens. Cada lista, chamada de PersonList*, possui pessoas (chamadas Person*) como nós, e cada Person possui um inteiro como key_, representando seu id; um inteiro como status_, que possui o id da pessoa com quem se casou; e uma PrefList* preferences_, que é uma lista de inteiros representando sua lista de preferências do sexo oposto.



2.1. Módulos

Módulo list.h
Esse arquivo pussui a estrutura da lista encadeada circular (List*), que será compostas por nós, e terá um end_ marcando o ínicio e o fim da lista. Também possui a estrutura dos nós (Node*), que possui um apontador para o próximo nó (next_) e anterior (prev_), e um void* info_ representando o conteúdo do nó. No caso do problema corrente, o info_ é comporto por pessoas (Person*) para a PersonList, e composto por inteiros para a PrefList.
Nesse módulo está a chamada das funções do módulo list.c.

Módulo list.c
Possui as funções que manipulam a lista. São as funções que criam um nó NewNode, criam uma lista vazia NewList, inserem um elemento ao final da lista InsertBack, a função que aponta para o próximo elemento das lista nextList, retorna o último elemento lastElement ou o ponteiro para o sentinela end_ backList, a função que retorna o info_ da lista getInfo, e a função clearList que é usada para liberar a memória das listas criadas.

Módulo person.h
Possui as estruturas usadas no problema do SMP, que já foram citadas anteriormente nesse documento. São elas a PersonList que possui uma List list_, e essa lista é composta de Person que possui sua key_, status_ e preference_ conforme já citados, e sua preference_ é do tipo PrefList que também possui uma List list_.
Nesse módulo está a chamada das funções do módulo person.c.

Módulo person.c
Possui as funções para manipular as listas de pessoas, as pessoas e a lista de preferência de cada uma. As funções para criar os elementos e alocar memória são a createPerson, createPrefList e createPersonList. Também tem as funções findPersonWithKey, que retorna a pessoa que possui a key (id) que foi passado como parâmetro; insertPersonInList que insere uma pessoa ao final da lista; insertPref que insere uma pessoa na PrefList; as funções que imprimem informações na tela dumpPersonList e dumpPerson; e as funções que imprimem os elementos no arquivo de saída dumpPersonStatusToOutput e dumpPersonListStatusToOutput.

Módulo marriage.h
Possui a chamada das funções do marriage.c.

Módulo marriage.c
Possui as funções que manipulam as funções auxiliares para a resolução do Stable Marriage Problem, e também as funções de Satisfabilidade. IsMarried retorna 1 se a pessoa já estiver casada ou noiva, ou seja, se seu status_ for diferente de -1 (status inicial definido para solteiros). AllMarried retorna 1 se todas as pessoas da lista estiverem casadas ou noivas. Prefers retorna 1 caso a mulher prefira o homem que a propôs ao invés daquele que ela está casada. Marry casa duas pessoas e Divorce as divorcia. Enfim vem a função SMP que realiza o algoritmo que será explicado em **2.2**. As funções de satisfabilidade são satisfactionBySex, que retorna a satisfação de todos os homens ou de todas as mulheres, satisfactionGeneral que calcula a satisfação geral, e writeOutputSatisfaction que imprime todas as satisfações calculadas no arquivo de saída.
Também estão nesse módulo as funções que foram paralelizadas, a satisfactionByPrefList e a satisfactionPersonParalel que executa a satisfactionByPrefList. Essas são a paralelização da satisfactionBySex, que está comentada no código.

Módulo main.c
Possui o tratamento dos parâmetros para a execução do programa no termina, as leituras do arquivo

input criando as listas de preferências dos homens e das mulheres, chama a função SMP e chama também as funções que criam o output, são elas dumpPersonListStatusToOutput e writeOutputSatisfaction. Em seguida a memória é liberada sendo chamada a função clear.

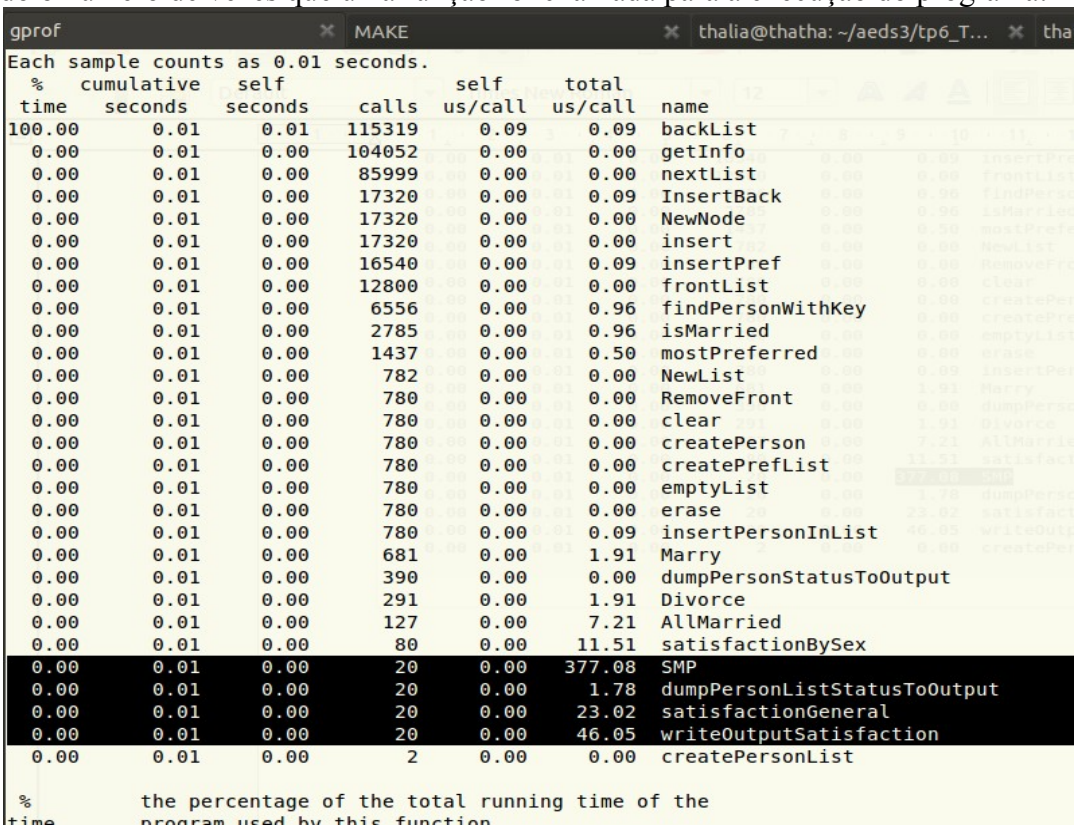
Módulo paralelo.h
Para a paralelização é necessária a criação de uma estrutura thread_data para que a função que será paralelizada e será passada de parâmetro para a pthread_create tenha apenas um parâmetro void*. Isso porque a função a ser paralelizada precisa ser do tipo void* com esse parâmetro void*. Portando a pthread_create possui todos os parâmetros que seriam passados na satisfactionBySex porém, dentro da estrutura.

2.2. Algoritmo para o Stable Marriage Problem

A solução foi dada por David Gale e Lloyd Shapley, conhecida como solução de Gale-Shapley, que provaram que sempre é possível resolver esse problema para que todos os casamentos sejam estáveis. O algoritmo consiste em iniciar a busca pelo parceiro ideal sempre pelo homem, e isso pode ser feito com um homem por vez, ou todos os homens juntos, realizando essa busca. O homem pede em casamento a primeira mulher da sua lista de preferências. Se essa estiver solteira, eles se casam, se essa estiver casada ou noiva, ela pode ficar com o parceiro que esteja na posição mais privilegiada de sua lista. Então o homem que foi rejeitado continua a busca a partir a sua segunda mulher predileta.

3. PARALELIZAÇÃO

Para encontrar oportunidades de paralelizar o algoritmo foi usado o gprof, gerando um profile e mostrando o número de vezes que uma função foi chamada para a execução do programa:



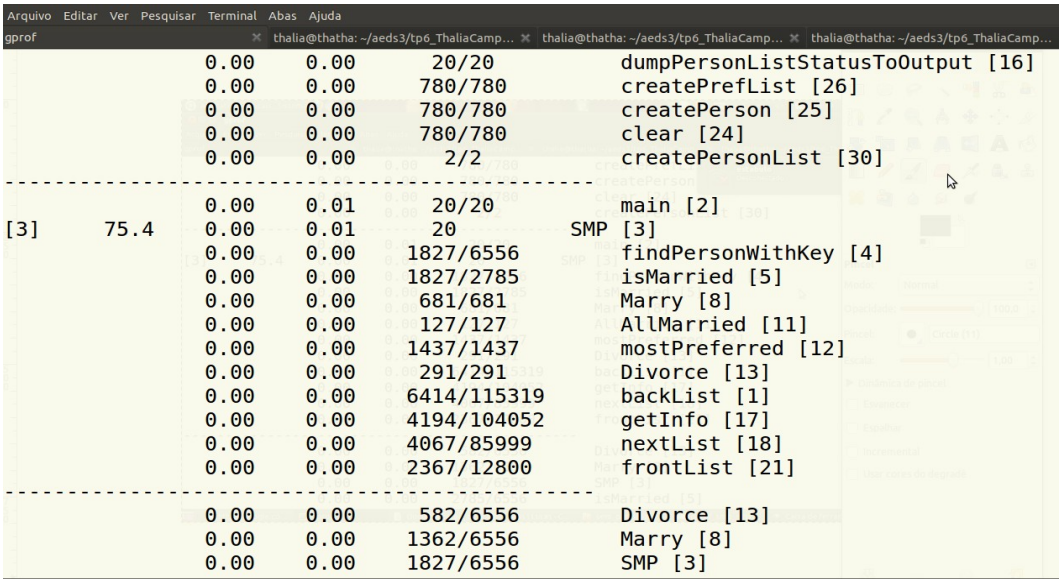

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.00      0.01     0.01   115319     0.09     0.09  backList
  0.00      0.01     0.00   104052     0.00     0.00  getInfo
  0.00      0.01     0.00    85999     0.00     0.00  nextList
  0.00      0.01     0.00    17320     0.00     0.09  InsertBack
  0.00      0.01     0.00    17320     0.00     0.00  NewNode
  0.00      0.01     0.00    17320     0.00     0.00  insert
  0.00      0.01     0.00    16540     0.00     0.09  insertPref
  0.00      0.01     0.00    12800     0.00     0.00  frontList
  0.00      0.01     0.00     6556     0.00     0.96  findPersonWithKey
  0.00      0.01     0.00     2785     0.00     0.96  isMarried
  0.00      0.01     0.00     1437     0.00     0.50  mostPreferred
  0.00      0.01     0.00      782     0.00     0.00  NewList
  0.00      0.01     0.00      780     0.00     0.00  RemoveFront
  0.00      0.01     0.00      780     0.00     0.00  clear
  0.00      0.01     0.00      780     0.00     0.00  createPerson
  0.00      0.01     0.00      780     0.00     0.00  createPrefList
  0.00      0.01     0.00      780     0.00     0.00  emptyList
  0.00      0.01     0.00      780     0.00     0.00  erase
  0.00      0.01     0.00      780     0.00     0.09  insertPersonInList
  0.00      0.01     0.00      681     0.00     1.91  Marry
  0.00      0.01     0.00      390     0.00     0.00  dumpPersonStatusToOutput
  0.00      0.01     0.00      291     0.00     1.91  Divorce
  0.00      0.01     0.00      127     0.00     7.21  AllMarried
  0.00      0.01     0.00       80     0.00    11.51  satisfactionBySex
  0.00      0.01     0.00       20     0.00   377.08  SMP
  0.00      0.01     0.00       20     0.00     1.78  dumpPersonListStatusToOutput
  0.00      0.01     0.00       20     0.00    23.02  satisfactionGeneral
  0.00      0.01     0.00       20     0.00    46.05  writeOutputSatisfaction
  0.00      0.01     0.00        2     0.00     0.00  createPersonList

 %         the percentage of the total running time of the
time       program used by this function.
```

O número de vezes que a função é chamada pode ser percebido na coluna *calls*, mas o *total us/call* mostra o tempo que a função e seus descendentes gastou. Podemos perceber que por mais que a SMP tenha sido chamada poucas vezes (foi usado um input de 20 instâncias para esse teste), quando

são considerados os descendentes, o tempo de execução da mesma fica muito maior do que o das outras funções.
Outras funções muito chamadas são as satisfactionGeneral, writeOutputSatisfaction e satisfactionBySex. Essas três funções é que foram tratadas para serem paralelizadas.

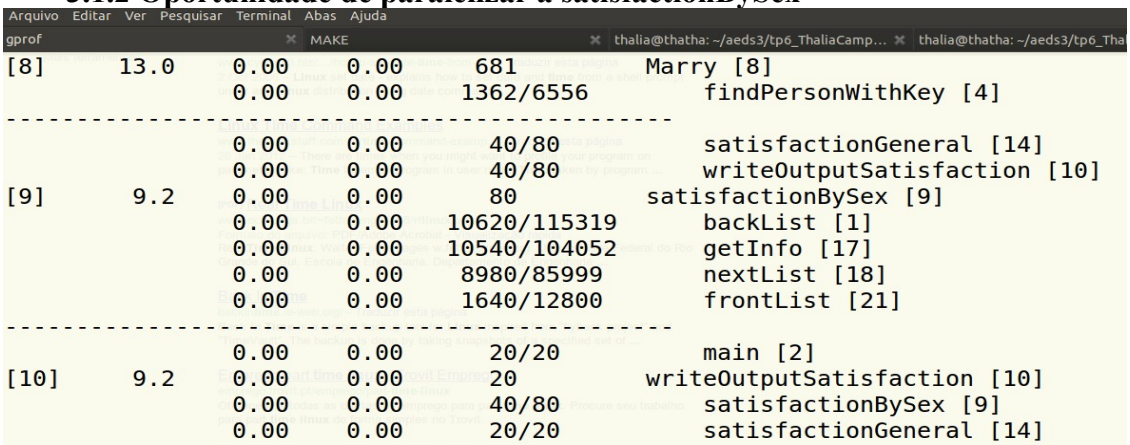
3.1 Oportunidades de paralelização

3.1.1 Oportunidade de paralelizar a SMP



Das funções chamadas pela SMP, as que são chamadas mais vezes são findPersonWithKey, isMarried, backList, getInfo, nextList e frontList. Outra oportunidade de paralelização seria paralelizar as funções internas que são muito referenciadas, por exemplo a findPersonWithKey, que percorre toda a lista de pessoas até encontrar a key_ desejada. Se fosse paralelizar a própria SMP, que possui vários loops, um deles poderia ser dividido em threads, como por exemplo o que percorre a lista dos homens para ver se esses estão casados, se todos estiverem casados a execução do algoritmo para. Esse cálculo poderia ser separado em threads para procurar paralelamente entre cada um dos homens.

3.1.2 Oportunidade de paralelizar a satisfactionBySex



A satisfactionGeneral e a writeOutput chamam a função satisfactionBySex, como podemos perceber nas informações relativas ao índice [9]. Com a paralelização da satisfactionBySex, as duas outras serão automaticamente paralelizadas. Essa foi a função escolhida para realizar a paralelização, conforme explicado no ítem **3.2**, abaixo.

3.2 Paralelizando a satisfactionBySex

Foi escolhida a paralelização dessa função, já que o cálculo da satisfação de cada sexo (homens ou mulheres) é realizado calculando a satisfação de cada pessoa desse sexo. Portanto é possível dividir esse cálculo de cada pessoa em threads, para melhorar o desempenho.
O tarefa paralelizada é o cálculo da satisfação de cada PrefList (lista de preferências do sexo oposto de cada pessoa).
Para sua paralelização, foram criadas duas funções separadas: A satisfactionByPrefList, que executa um dos loops, que será de fato paralelizado. Essa será a função passada para a pthread_create. A função satisfactionPersonParalel executa a satisfactionByPrefList com um loop externo, chamando as threads criadas.
A função writeOutputSatisfaction chama a função satisfactionPersonParalel. Portanto, indiretamente, também foi paralelizada.
A paralelização pode ser observada em *marriage.c* e em *paralelo.h*, a segunda foi citada no ítem **2.1**.

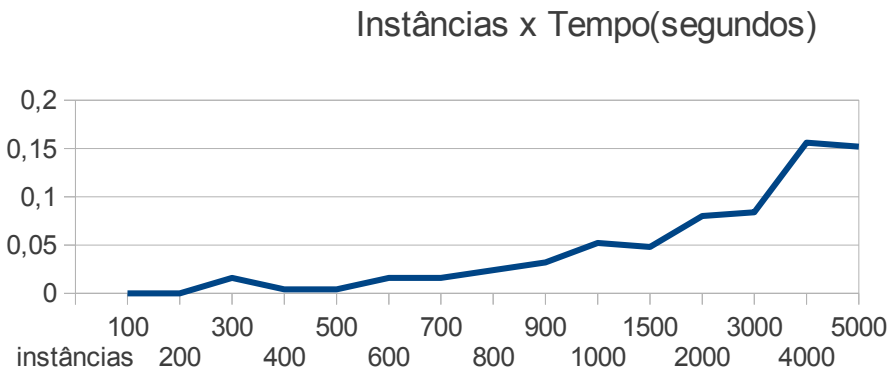
4.RESULTADOS

4.1 Estratégia para configurações de entrada

4.1.1 Entrada do programa sequencial

Para configurar a entrada, foram criados exemplos aumentando o número de instâncias, e em cada uma o número de pessoas está variado. Para as instâncias abaixo de 100 o tempo foi insignificante, sendo retornado como zero segundos a execução do sistema, de acordo com a função “time” do linux. Portanto são medidas instâncias de 100 a 1000, aumentando de 100 em 100, e de 1000 a 5000 aumentando de 1000 em 1000.

4.1.2 Tempo sem paralelização, em segundos



4.2.1 Entrada do programa paralelo

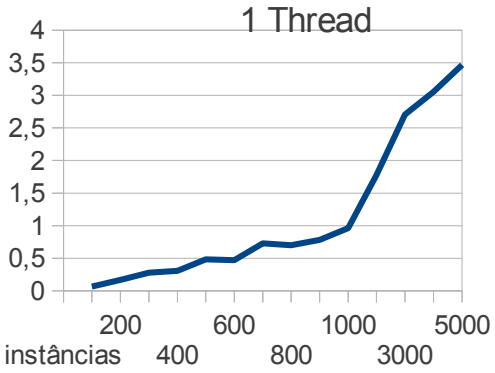
Para a entrada do programa paralelizado,foram realizadas duas estratégias. O problema só funciona quando nas instâncias o número de pessoas de cada conjunto é fixo, e o número máximo de threads aceitável é o número de pessoas no problema.

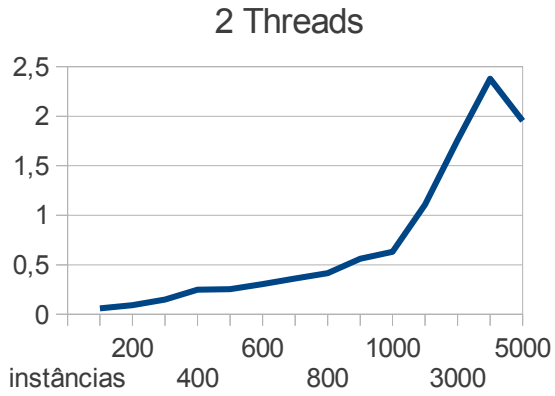
Primeiro, o número de instâncias aumenta enquanto o número de threads se mantém estável. Isso foi realizado para 1, 2 e 4 threads, que são os números de threads que funcionam para o exemplo da entrada que foi passado como exemplo no documento da especificação.Foi realizado para o mesmo número de instâncias do código sequencial.

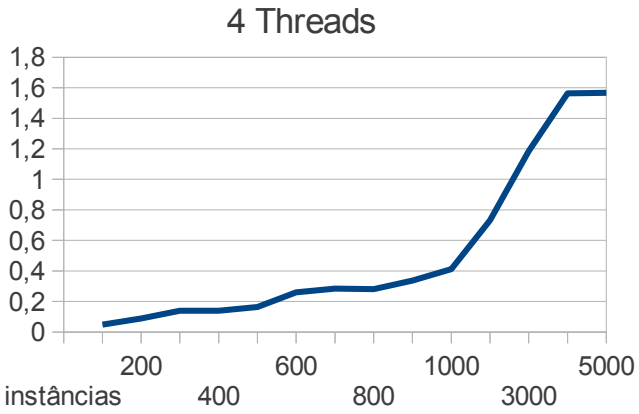
Na outra estratégia o número de threads aumenta entre os valores citados acima, mas manteve-se o número de instâncias. Foi realizado o teste para entradas de 1000, 3000 e 5000 instâncias.

4.2.2 Tempo com paralelização

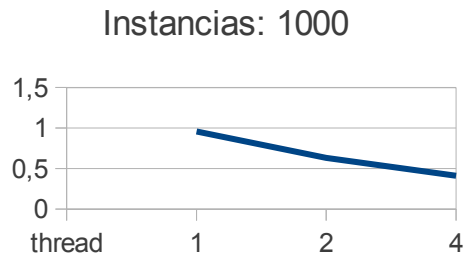
Para a estratégia em que se manteve o número de threads e aumentou o número de instâncias, foram gerados os seguintes gráficos:

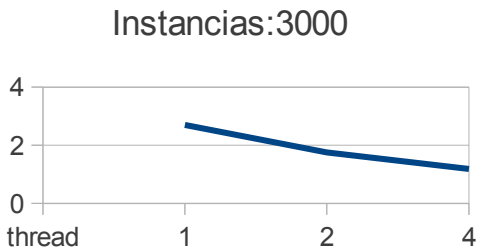


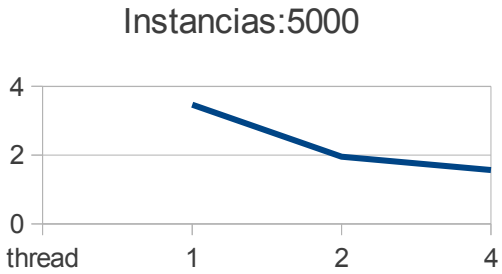




Foi possível observar que conforme o número de instâncias aumenta, o tempo também aumenta, quando mantém-se o número de threads utilizada.
Também pode-se observar que o tempo geral diminuiu quando aumentou os números de threads, pois o tempo da thread quatro sempre é o menor dentre os testes. Isso é mais perceptível com os gráficos abaixo:







Quando afixamos o número de instâncias, é possível observar mais claramente a queda de tempo com o aumento de threads.

4.2 Comparação dos testes paralelo e sequencial

A paralelização da forma como foi feita não aprimorou o algoritmo. A primeira observação a ser feita é que o teste com apenas uma thread demorou mais tempo do que o programa sequencial, porém, os testes com mais threads no geral também demoraram mais. Isso é perceptível pois os sequenciais raramente passaram de 0,15 segundos enquanto a execução do paralelo ficou na maior parte das entradas muito acima desse valor.

5. COMPLEXIDADE DE TEMPO E ESPAÇO

5.1 Complexidade de tempo

Função SMP:

A complexidade pode ser observada de acordo com a função SMP, que possui quatro laços na mesma. Um deles percorre todos os homens para checar se eles são casados, e depois o outro percorre novamente essa lista realizando os casamentos. O terceiro loop pode percorrer toda a lista de preferências dos homens até encontrar a mulher que quer casar com eles. O quarto é para encontrar a mulher que o homem escolheu na lista de mulheres, para que possa descobrir se ela é ou não casada.

Caso a mulher já esteja casada quando o homem a escolhe, é necessário percorrer sua lista de preferências também, entrando em um quinto loop na função.

Portanto a função, em seu pior caso, possui complexidade $O(n^5)$, sendo “n” o número de indivíduo de cada conjunto (número de homens e/ou número de mulheres), que é o tamanho de todas as listas usadas no problema.

Quando a função é chamada pelo main, a complexidade é multiplicada pelo número de instâncias “i”, tornando-se $O(i*n^5)$.

Função satisfactionPersonParalel:
A função percorre toda a lista de pessoas, depois o número de threads é lido para que as tarefas sejam distribuídas entre elas, e então é chamada a função satisfactionByPrefList. Portanto a complexidade é $O(t*n^2)$, sendo "n" o número de pessoas de cada conjunto e "t" o número de threads. A writeOutputSatisfaction, que é de complexidade linear, chama a função satisfactionPersonParalel, e também é chamada pelo main. No main ela também é realizada para cada instância "i", e a complexidade torna-se $O(t*n^2*i)$.

Complexidade do programa:
O programa possui a mesma complexidade do SMP, que é a função com maior complexidade. Portanto, é $O(i*n^5)$.

5.2 Complexidade de espaço

A complexidade de espaço é n^2 por se tratar de duas listas, uma dentro da outra, conforme representado no ítem **2** desse documento. "n" refere-se ao número de indivíduos de cada conjunto.

6. CONCLUSÃO

É possível concluir que com a paralelização, o aumento do número de threads é muito importante para a melhora do desempenho de um programa. Porém os testes sequenciais apresentaram um tempo menor para a execução do programa do que os testes realizados com o código paralelizado. O custo para a implementação de threads pode apresentar-se maior do que os ganhos com a divisão de tarefas, já que existe o custo de comunicação entre as mesmas, e de sincronização.

É importante ressaltar que só foi implementada de fato uma das oportunidades de paralelização observadas, relativa à função satisfactionBySex. Outras oportunidades foram citadas no trabalho, porém não implementadas. A paralelização também não funciona para todas as entradas, apenas pelo exemplo enviado pelos professores na especificação do trabalho, conforme foi observado no ítem de testes **4.2.2.** Essa entrada foi replicada, aumentando o número de instâncias para a realização dos testes.