TRABALHO PRÁTICO 4:
Problemas NP-Completo e Programação paralela

Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

20 de dezembro de 2012

Resumo. *Este relatório descreve como foi modelado e implementado um problema de alto custo computacional. Será descrito como foi modelado o problema, as estruturas utilizadas e porquê não é possível implementar uma solução que consiga resolver entradas não triviais de maneira ótima. Finalmente será detalhado a análise de complexidade dos algoritmos e, por último, uma breve conclusão do trabalho implementado.*

1. INTRODUÇÃO

O trabalho propõe resolver um problema de uma rede social exclusiva para cozinheiros, o Facecook; para isso foi utilizado a teoria dos grafos para modelar o problema. Neste grafo, cada vértice representa um membro da rede, e as arestas representam uma conexão de amizade entre dois membros.

O que devemos encontrar nessa lista de membros, é o maior grupo de amigos registrados na rede social. Podemos perceber que isso se trata de um problema conhecido na teoria dos grafos, o problema do clique. Esse problema refere-se a qualquer problema que possui como objetivo encontrar sub-grafos completos (”cliques”) em um grafo. Como exemplo, o problema de encontrar conjuntos de nós em que todos os elementos estão conectados entre si. [Wikipedia]

Esse problema é **NP-Completo**, pois não pode ser resolvido de forma ótima para entradas grandes. Ou seja, não existe um algorimo de tempo polinomial para resolver esse problema, e isso que iremos mostrar neste trabalho.

Além disso, para entradas muito grandes o algoritmo sequencial não resolveria em tempo hábil, então foi proposto que construíssimos uma versão paralelizada do algoritmo.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve como foi feita a modelagem do problema e o armazenamento das conexões do grafo. A Seção 3 descreve como resolvido o problema do clique. A Seção 4 trata de detalhes específicos da implementação do trabalho: quais os arquivos utilizados; como é feita a compilação e execução; além de detalhar o formato dos arquivos de entrada e saída. A Seção 5 contém a análise de desempenho do problema para entradas triviais e não-triviais. A Seção 6 conclui o trabalho.

2. MODELAGEM

Para simular a memória virtual neste trabalho, foi utilizado um tipo abstrato de dados de lista duplamente encadeada. Foi escolhida esta estrutura pela facilidade de manipulação e pesquisa dos elementos contidos nela.

A abstração da estrutura é a seguinte: a lista representa a memória física e cada uma das suas células representam as páginas que foram carregadas na memória. Assim, utilizando as funções de inserção e remoção da lista encadeada, podemos implementar as políticas de reposição propostas pro trabalho.

3. SOLUÇÃO PROPOSTA

É fornecido na entrada do trabalho o tamanho em bytes da memória física, o tamanho de cada página e N acessos à memória. Cada um desses acesso é referente à posição da memória virtual acessada sequencialmente (para mais detalhes sobre o formato de entrada, consultar o item 4.3.1 deste trabalho). Porém, para saber em qual página cada acesso se encontra, dividimos a posição de acesso pela quantidade de bytes de cada página. Por exemplo, dada a sequência de acesso **0 2 4 2 10 1 6 8**, podemos simplificar isso para **0 0 1 0 2 0 1 2**. Assim sabemos qual página deverá estar na memória a cada acesso.

Na situação descrita acima, chegamos às páginas **0 0 1 0 2 0 1 2**; porém, com apenas 8 bytes na memória principal, podemos carregar simultaneamente apenas duas páginas, pois cada uma possui 4 bytes de tamanho. Por esse motivo precisamos adotar políticas que removam uma certa página e a troquem por outra da memória principal. Para solucionar esse problema, foi implementado um algoritmo para cada política de reposição (FIFO, LRU e LFU).

Para o **FIFO**, a escolha da página a ser removida é a que entrou primeiro, ou seja, a primeira da fila. No **LRU**, também removemos as páginas que estão no início da fila, pois toda vez que a página é acessada ela vai pro final da fila, assim, seu início contém as páginas que foram acessadas menos recentemente. No **LFU**, para cada acesso à memória, cada página possui um contador para indicar a quantidade de acessos à ela. Assim, a cada acesso à memória, ordenamos as páginas de acordo com esse contador, então as primeiras páginas da fila serão as que foram menos frequentemente usadas.

3.1. Algoritmos implementados

- *void FIFO(TipoLista * memoria, TipoCelula pagina)*

  Descrição: "First In, First Out"; a página que entrou primeiro na memória é a primeira a ser removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

- *void LRU(TipoLista * memoria, TipoCelula pagina)*

  Descrição: "Least Recently Used"; a página que foi acessada menos recentemente que será a escolha pra ser removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

- *void LFU(TipoLista * memoria, TipoCelula pagina)*

  Descrição: "Least Frequently Used"; a página que é utilizada menos frequentemente que será removida.
  Parâmetros: Memória principal e página atual que deveria estar na memória
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

- *void removeMenosAcessada(TipoLista * memoria)*

  Descrição: Procura pela página com menor número de acessos e a remove da memória.
  Parâmetros: Memória principal
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

- *TipoApontador resideEmMemoria(TipoLista * memoria, int pagina)*

  Descrição: Percorre toda a memória para verificar se a página se encontra na memória. Retorna um apontador pra essa página.
  Parâmetros: Memória principal e um inteiro que representa a página a ser encontrada
  Complexidade: $O(n)$, onde n é o número de páginas que cabe na memória.

3.2. COMPLEXIDADE GERAL

Cada complexidade separada dos algoritmos será $O(n)$, onde n é o número de páginas na memória, igual para cada um deles. Porém, a complexidade temporal do programa deve levar em consideração a quantidade instâncias, de páginas e de acessos.

Assim, nosso cálculo de complexidade poderia ser definido como *número de páginas * número de acessos * 3*. Nesses três parâmetros, o que tem um maior peso seria os acessos, pois ele que irá dominar todos os outros. Por isso, podemos dizer então, que a complexidade temporal final do programa será de $O(n)$, porém n aqui será a quantidade de acessos feitos.

4. IMPLEMENTAÇÃO

4.1. Código

4.1.1. Arquivos .c

- **tp3.c** Arquivo principal do programa. Lê os arquivos de entrada, calcula os *page faults* pra cada política e escreve o resultado em um arquivo de saída.
- **lista.c** TAD da uma lista duplamente encadeada. Contém funções de manipulação (criação, inserção, remoção e liberação de memória), e também de impressão e cópia.
- **smv.c** Contém a implementação das políticas de reposição FIFO, LRU, LFU.
- **arquivos.c** Um tipo abstrato de dados de manipulação de arquivos, contendo funções de abertura, leitura, escrita e fechamento.

4.1.2. Arquivos .h

- **lista.h** TAD da uma lista duplamente encadeada. Contém a definição da estrutura e das funções.
- **smv.h** Contém a definição das políticas de reposição FIFO, LRU, LFU.
- **arquivos.h** Definição da das funções utilizadas para ler, escrever e fechar corretamente um arquivo.

4.2. Compilação

O programa deve ser compilado através do compilador GCC através dos seguintes comandos

Para programação dinâmica:

```
gcc -Wall -Lsrc src/tp3.c src/lista.c src/arquivos.c src/smv.c -o tp3
```

Ou através do comando $make$.

4.3. Execução

A execução do programa tem como parâmetros:

- Um arquivo de entrada contendo várias instâncias a serem simuladas.
- Um arquivo de saída que irá receber a quantidade de *page faults* pra cada política de reposição

O comando para a execução do programa é da forma:

```
./tp3 <arquivo_de_entrada> <arquivo_de_saída>
```

4.3.1. Formato da entrada

A primeira linha do arquivo de entrada contém o valor *k* de instâncias que o arquivo contém. As k instâncias são definidas em duas linhas cada uma. A primeira linha contem três inteiros: o tamanho em bytes da memória física, o tamanho em bytes de cada página, e o número N de acessos. A linha seguinte contém N inteiros representando as N posições da memória virtual acessadas sequencialmente.

A seguir um arquivo de entrada de exemplo:

```
1
8 4 10
0 2 4 2 10 1 0 0 6 8
```

4.3.2. Formato da saída

O arquivo de saída consiste em k linhas, cada uma representando o resultado de uma instância. Cada linha contém um inteiro que representa o número de falhas utilizando FIFO, LRU e LFU, necessariamente nessa ordem. Um exemplo é mostrado abaixo:

```
6 5 5
```

5. AVALIAÇÃO EXPERIMENTAL

As análises feitas para para este trabalho foram todas detalhadas na especificação. O que foi pedido foi o seguinte:

- Calcular a localidade de referência temporal.
- Calcular a localidade de referência espacial.
- Gerar o histograma das distâncias de acessos.
- Gerar o histograma das distâncias de pilha.
- Gerar um gráfico ”Tamanho da página”x ”Bytes movimentados”.
- Gerar um gráfico ”Tamanho da memória”x ”Falhas”.

Para realizar essas análise foi disponibilizado um arquivo com uma configuração diferente no fórum da disciplina. Esse arquivo possuía apenas acessos à memória, sem informações de tamanho da memória ou da página. Foi criado dois scripts que auxiliaram na execução destas análises. O primeiro de localidade espacial e outro de localidade temporal.

Nas próximas seções iremos descrever a máquina utilizada para os testes e o resultado de cada item descrito acima.

5.1. Máquina utilizada

Segue especificação da máquina utilizada para os testes:

```
model name:     Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
cpu MHz:        933.000
cache size:     3072 KB
MemTotal:       3980124 kB
```

5.2. Localidade de referência temporal

A Tabela 1 mostra os resultados que obtivemos para o cálculo da localidade de referência temporal.

Instância	Temporal
1	19.67
2	14.10
3	21.08
4	10.67

Tabela 1. Localidade de referência temporal

5.3. Localidade de referência espacial

A Tabela 2 mostra os resultados que obtivemos para o cálculo da localidade de referência espacial. É gritante a diferença entre o resultado da instância 2 pras outras, mas isso ocorreu pois ela tem um padrão bem peculiar. A grande maioria dos acessos são sequenciais na memória (posição 8, depois 7, depois 6, depois 5), isso caracteriza uma boa localidade de referência espacial.

Instância	Espacial
1	16.69
2	2.55
3	10.46
4	19.71

Tabela 2. Localidade de referência temporal

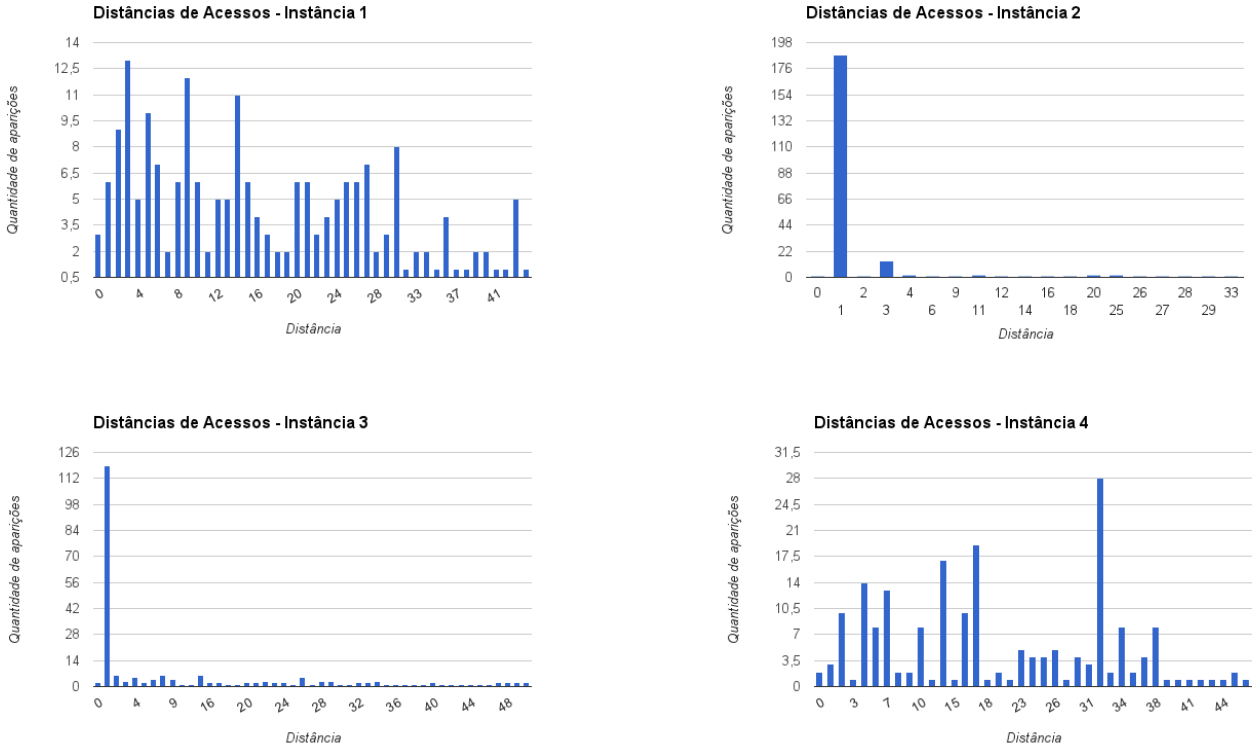

Figura 1. Distâncias de Acessos de todas as instâncias

5.4. Histogramas de Distância de Acessos

Na Figura 1 agrupamos os quatro histogramas que foram pedidos. Logo abaixo é possível ver o resultado para cada uma das instâncias.

Na instância 1 e 4 ocorre uma distribuição maior dos acessos. Na primeira os ápices se encontram nas menores distâncias, já na outra o ápice ocorre na distância 32. As distanciais de acesso da instância 1 se encontram muito melhor distribuídas que a instância 4.

Já as instâncias 2 e 3 possuem uma similaridade: quase a totalidade dos acessos se encontram à distância 1. Isso indica que a localidade espacial deles é melhor que a das outras instâncias, por não terem que caminhar tanto na memória para acessar o próximo valor.

5.5. Histogramas de Distância de Pilha

Na Figura 2 foi agrupado os histogramas referentes ao calculo de distância de pilha de cada instância.

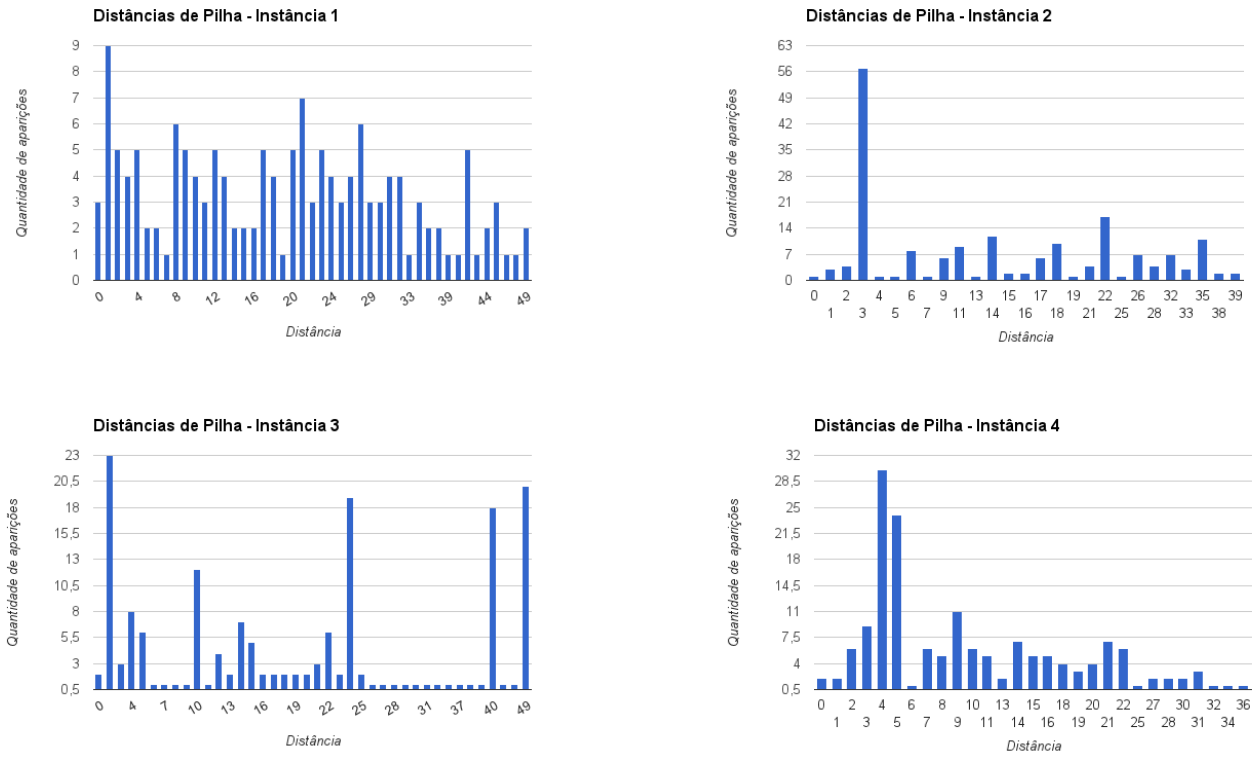

Figura 2. Distâncias de Pilha de todas as instâncias

Nessa situação a instância que apresentou o melhor resultado foi a instância 2, pois a distância na pilha que mais correu foi o valor 3. Todas as outras distâncias ficaram bem distribuídas no histograma.

A segunda melhor instância nesse quesito seria a instância 4, pois muitos acessos se concentram em distâncias menores que 5. O resto dos acessos também se distribuíram bem pelo resto das distâncias.

As piores instâncias nessa análise foram as instâncias 1 e 3. A instância 1 possui muitos acessos de distâncias variadas, desde as menores até as maiores. Já a instância 3, apesar de possuir o ápice de acessos com distância de pilha igual a 1, possui dezenas de acessos com distâncias maiores que 40, o que prejudica muito seu resultado final.

5.6. Tamanho da página x Bytes Movimentados

Na Figura 3 temos a relação de ”Tamanho da página x Bytes Movimentados”. Com essas informações podemos analisar o *trade-off* entre o tamanho da página e o volume transferido entre disco e memória. Como foi dito na especificação do trabalho ”Páginas maiores irão certamente diminuir o número de falhas, porém vão causar o carregamento desnecessário de muitos dados”, podemos verificar isso ocorrendo de fato em nossos testes.

A política adotada foi a seguinte: foi fixado o tamanho da memória em 256 bytes, e então variamos o tamanho de cada página (sempre na potência de 2), de 1 até 256. Para páginas pequenas (1, 2, 4, 8 bytes), podemos ver que poucas falhas ocorreram, então consequentemente, poucos dados foram movimentados. Já no caso de páginas maiores

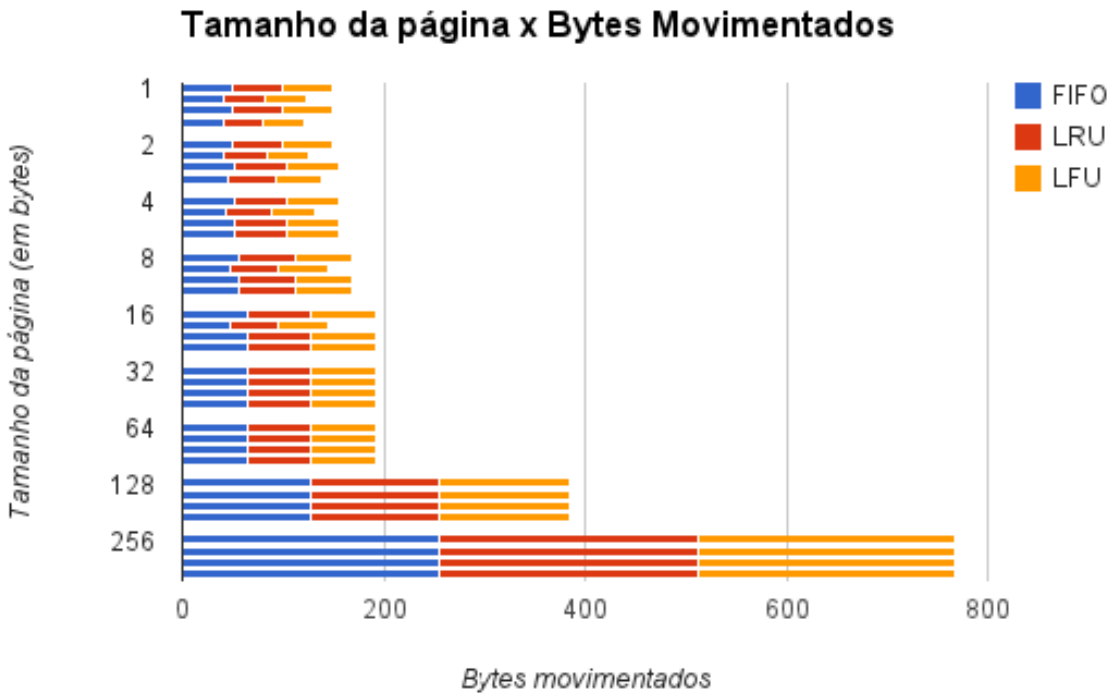

Figura 3. Tamanho da página x Bytes Movimentados (Memória fixa em 256 bytes)

(128, 256 bytes), a quantidade de falhas reduziu drasticamente, porém a quantidade de bytes movimentados também cresceu vertiginosamente.

A performance de cada uma das políticas de reposição nessa situação foi igual para todas. Todas obtiveram o mesmo número de falhas e, consequentemente, a mesma quantidade de bytes movimentados.

O tamanho da página, para memória fixa em 256 bytes, só começou a ser significativo quando chegou em 128 bytes, metade do tamanho da memória. Até então, a quantidade de bytes movimentados tinha crescido, porém de forma lenta. Então podemos dizer que o tamanho ideal da página pode ser definido como no máximo $1/4$ do tamanho da memória física disponível.

5.7. Tamanho da memória x Page Faults

Na Figura 4 temos a relação de ”Tamanho da memória x Page Faults”. O objetivo desta análise seria estimar o tamanho ideal de memória física disponível. Obviamente, quanto mais memória menos falhas ocorrerão, porém este teste deixou visível a partir de qual momento já é interessante determinar qual a melhor quantidade de memória necessária para obter menos falhas.

A política adotada neste teste foi o seguinte: foi fixado o tamanho da página em 4 bytes, após isso, variamos o tamanho da memória (sempre na potência de 2), de 4 até 256. E podemos perceber, como já era esperado, que quanto maior a memória disponível, menor a quantidade de *page faults*.

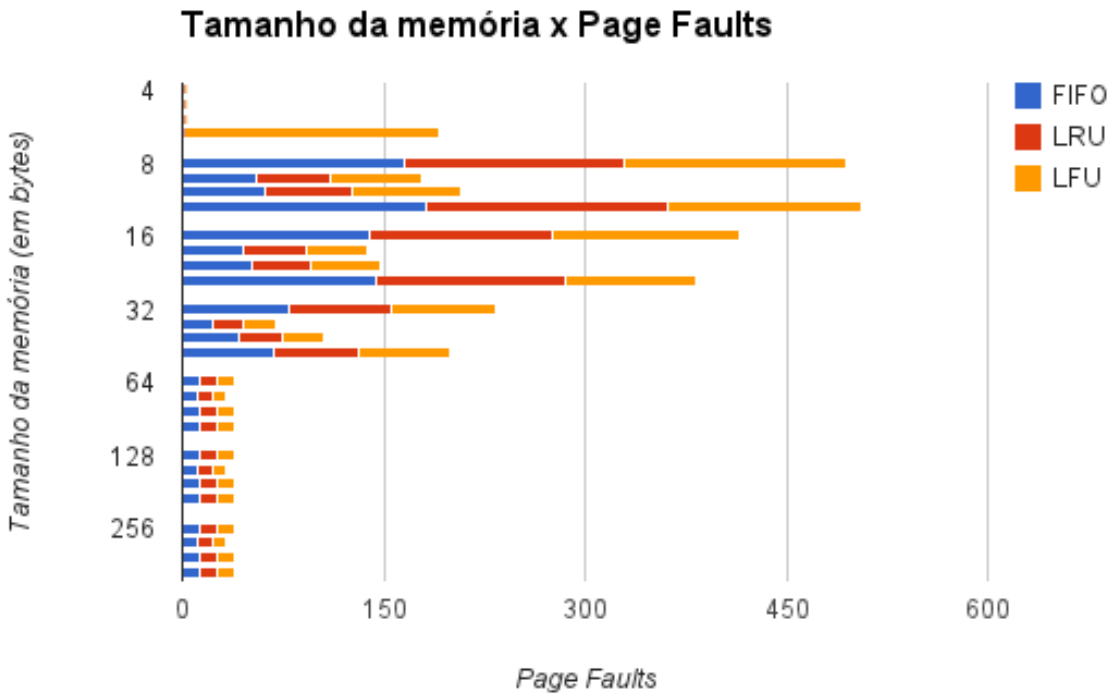

Figura 4. Tamanho da memória x Page Faults (Página fixa em 4 bytes)

Nessa situação, com páginas fixas em 4 bytes, um tamanho de memória ideal poderia ser 32 bytes, porém para sequências de acesso com uma localidade de referência espacial boa. Se não for possível saber o comportamento da sequência de acessos, o tamanho ideal da memória seria 64 bytes, uma vez que todas as políticas e todas as instâncias obtiveram um número pequeno de *page faults*.

5.8. Resultado

De acordo com as nossas análises feitas, podemos estimar os valores ideias para o tamanho da memória física e o tamanho das páginas, para obter o menor número de falhas possível para cada uma das quatro sequências de acessos disponibilizadas para os testes.

Na Figura 3 o desempenho de cada uma das políticas é igual para as quatro instâncias, obtendo a mesma quantidade de *page faults* e, consequentemente, de bytes movimentados. Isso ocorreu pois com a quantidade de memória estabelecida, independente da política escolhida, a mesma quantidade de *page faults* ocorreria.

Na Figura 4 é possível ver que a Instância 1 e 4 obtiveram os piores resultados. Isso ocorreu justamente pela caraterística de cada uma. A sequência de acessos nessas instâncias ocorreram de uma forma mais caótica, quando posições muito distantes umas das outras foram acessadas em sequência. Já no caso das Instâncias 2 e 3 ocorre o contrário, as sequências de acessos ocorrem de uma maneira mais sequencial, com uma localidade de referência espacial melhor, pois dados próximos são acessados em um curto espaço de tempo.

6. CONCLUSÃO

No geral, a relação entre cada uma das políticas se manteve praticamente constante, nos gráficos pode não estar totalmente perceptível, mas é possível perceber que a política de reposição de remover as páginas com a menor quantidade de acessos (LFU) obteve uma quantidade de page faults melhor do que as outras políticas.

Foi possível perceber também que sequências de acessos com perfis mais sequenciais, ou seja, com localidade de referência espacial melhores, possuem um resultado melhor, independente do tamanho da memória disponibilizado.

Para perfis mais desordenados, com localidade de referência espacial pior, a quantidade de *page faults* para qualquer uma das políticas aumenta consideravelmente.

Acredito que os objetivos do trabalho foram concluídos com sucesso, uma vez que o sistema de memória virtual (SMV) foi implementado com sucesso e foi possível exercitar, mais uma vez, os conceitos de gerenciamento de memória. Além disso, na parte experimental, foi possível ver, na prática, a localidade de referência de diferentes tipos de sequências de acessos e como cada política trabalha com cada um desses perfis.

Referências

Wikipedia. Problema do clique. `http://pt.wikipedia.org/wiki/Problema_do_clique`.