UFRN, Instituto Metrópole Digital
Bacharelado em Tecnologia da Informação
DIM501 – Boas Práticas de Programação, 2016.1
Prof. Handerson Bezerra Medeiros

Relatório:

Aplicação de Boas Práticas de Programação no projeto *Sequencer*

Aluno:
Pitágoras de Azevedo Alves Sobrinho

Natal-RN, 22 de maio de 2016.

Sumário

Lista de figuras

Introdução

1. Remoção da paralelização

Os algoritmos do projeto S*equencer* utilizavam OpenMP, uma API para programação paralela multiplataforma.

A utilização desta API se baseia na aplicação de clausulas “#pragma” para definir formas de paralelização para determinadas regiões do código. Tais clausulas podem ser de difícil interpretação para um leitor do código que não conheça bem a API em questão.

Sequencer foi desenvolvido originalmente para o supercomputador do NPAD (Núcleo de Processamento de Alto Desempenho), onde ele teria que processar uma grande quantidade de dados. Porém foi verificado durante o desenvolvimento que, numa maquina com poucos núcleos e com a entrada sendo constituída por poucos fragmentos de texto, o desempenho não é melhorado pelo OpenMP. Na verdade, muitas vezes, ele é piorado. Isso ocorre porque o Se*quencer* acaba passando muito tempo criando *threads* desnecessariamente.

A especificação do OpenMP só é adequada oficialmente ao padrão ISO/IEC 14882:1998, ou seja, até o C++98 e também não é recomendado o uso de templates. Assim, é inseguro e uma péssima pratica utilizar capacidades implementadas na *Standard Template Library* após 1998. Em decorrência dessa e das outras desvantagens citadas anteriormente nesta sessão, foi decidido que (no contexto deste trabalho e da disciplina “Boas Praticas de Programação”) é melhor abandonar o uso do OpenMP e fazer uso do (não tão novo) padrão C++11.

O OpenMP foi removido facilmente retirando do código as suas clausulas “#pragma”.

2. Aplicação da STL do C++11 para simplificação dos algoritmos

A STL do C++ tem diversas classes e rotinas que podem melhorar a legibilidade do algoritmo, tornando tudo mais simples. Nesta sessão será descrito como foi feita a aplicação deste padrão da linguagem no projeto *Sequencer.*

2.1 Modificação do processo de I/O

A entrada de dados vem diretamente da entrada padrão dos sistema e a saída é feita diretamente na saída padrão.

Para possibilitar a inserção de uma quantidade indefinida de segmentos de texto, os segmentos são armazenados inicialmente numa estrutura de dados “std::vector”, um vetor genérico no qual espaço é alocado dinamicamente conforme novos elementos são inseridos. Porém, para evitar incompatibilidades com o OpenMP, antes de entrar no processo de paralelização, estas estruturas de dados são convertidas para vetores comuns. Esse processo confuso pode ser visto no seguinte fragmento de código:

```
int segmentsPerBucket = 9;
int totalSegments = 0;
string line;
vector<string> tempSegments;
vector<vector<string> > tempBuckets;
cout << "vars created\n";
while(getline(cin, line)){
    //Caso o texto use quebras de linha do tipo "\n\r", retira as \r"
    if(line[line.size()-1] == '\r'){
        line = line.substr(0, line.size()-1);

        //cout << "quebra de linha barra r retirada  \n";
    }
    tempSegments.push_back(line);
    totalSegments++;
    if(tempSegments.size() == segmentsPerBucket){
        tempBuckets.push_back(tempSegments);
        tempSegments.clear();
    }
}
if(!tempSegments.empty()){
    tempBuckets.push_back(tempSegments);
    tempSegments.clear();
}
cout << "temp buckets instantiated\n";
int nBuckets = tempBuckets.size();
Bucket* buckets = new Bucket[nBuckets];

for(int i = 0; i < nBuckets; i++){
    buckets[i].nSegments = tempBuckets[i].size();
    buckets[i].segments = new string[buckets[i].nSegments];
    for(int j = 0; j < buckets[i].nSegments; j++){
        buckets[i].segments[j] = tempBuckets[i][j];
    }
    tempBuckets[i].clear();
}
tempBuckets.clear();
cout << "buckets instantiated\n";
```

Um “Bucket” (balde, nome de classe nada significativo) é uma classe que, basicamente, armazena um vetor de segmentos de texto e o número destes segmentos. Seu único método é a rotina “process()”, que “funde” todos os segmentos presentes numa instancia do “Bucket” em um único texto final. Ele não possui qualquer tipo de modularização e ou construtor.

Como, sem o OpenMP, não há mais motivos para não fazer uso total das estruturas de dados da STL, o código foi simplificado para fazer amplo uso da estrutura “std::vector”.

2.1.1. Passagem de arquivo de segmentos de texto por argumentos para o programa:

A entrada de dados é feita, dentro da lógica do programa, através da entrada padrão do sistema, ao invés de ler os dados vindo de um arquivo. Porém na prática é diferente. Os operadores de direcionamento de entrada e saida “>” e “<” do UNIX são utilizados para ler os segmentos de texto vindo de um arquivo e escrever toda a saída em outro arquivo. Por isso o script para rodar o programa tem essa aparência:

$ ./bin/main-normal < inputs/A.input > outputs/A_normal.output

Esse comando, basicamente, executa o *Sequencer* de forma que os segmentos de texto sejam lidos do arquivo “inputs/A.input” e o resultado do sequenciamento seja escrito em “outputs/A_normal.output”. A entrada/saída foi feita dessa forma porque esse é o padrão para competições de programação, porém caso um usuário precise utilizar o programa, ele teria que seguir exatamente essa sintaxe, própria de sistemas UNIX. É desejável que o próprio *Sequencer* possa acessar os arquivos e ler/escrever neles, através dos seguintes argumentos:

$ ./sequencer -i “arquivo_de_entrada” -o “arquivo_de_saída”

Foi utilizada uma biblioteca de interpretação de argumentos de linha de comando, a AnyOption. A biblioteca não era compatível com C++11, então seu código foi “modernizado” para corrigir isso. Sem a necessidade de paralelização OpenMP, utilizando o padrão c++11 e com a saída e entrada do programa sendo definida pelos argumentos passados pelo usuário, o código inicial do programa ficou assim:

```
23
24  int main(int argCount, char* argVector[]){
25      Arguments arguments(argCount, argVector);
26      if(!arguments.helpFlag){
27          istream* inputStream;
28          ostream* outputStream;
29
30          //seleciona a entrada de dados
31          if(arguments.inputFileDefined){
32              ifstream *inputFileStream = new ifstream;
33              inputFileStream->open(arguments.inputFile.c_str());
34              inputStream = inputFileStream;
35          }else{
36              inputStream = &cin;
37          }
38          //seleciona a saida de dados
39          if(arguments.outputFileDefined){
40              ofstream* outputFileStream = new ofstream;
41              outputFileStream->open(arguments.outputFile.c_str(), ios::trunc);
42              outputStream = outputFileStream;
43          }else{
44              outputStream = &cout;
45          }
46          //loop de leitura
47          Bucket bucket;
48          string line;
49          while(getline(*inputStream, line)){
50              //Caso o texto use quebras de linha do tipo "\n\r", retira as \r"
51              if(line[line.size()-1] == '\r'){
52                  line = line.substr(0, line.size()-1);
53                  //cout << "quebra de linha barra r retirada  \n";
54              }
55              bucket.segments.push_back(line);
56          }
57
```

2.2. Trocar matriz de semelhança com cópias de vetores por armazenamento em *hashmap*

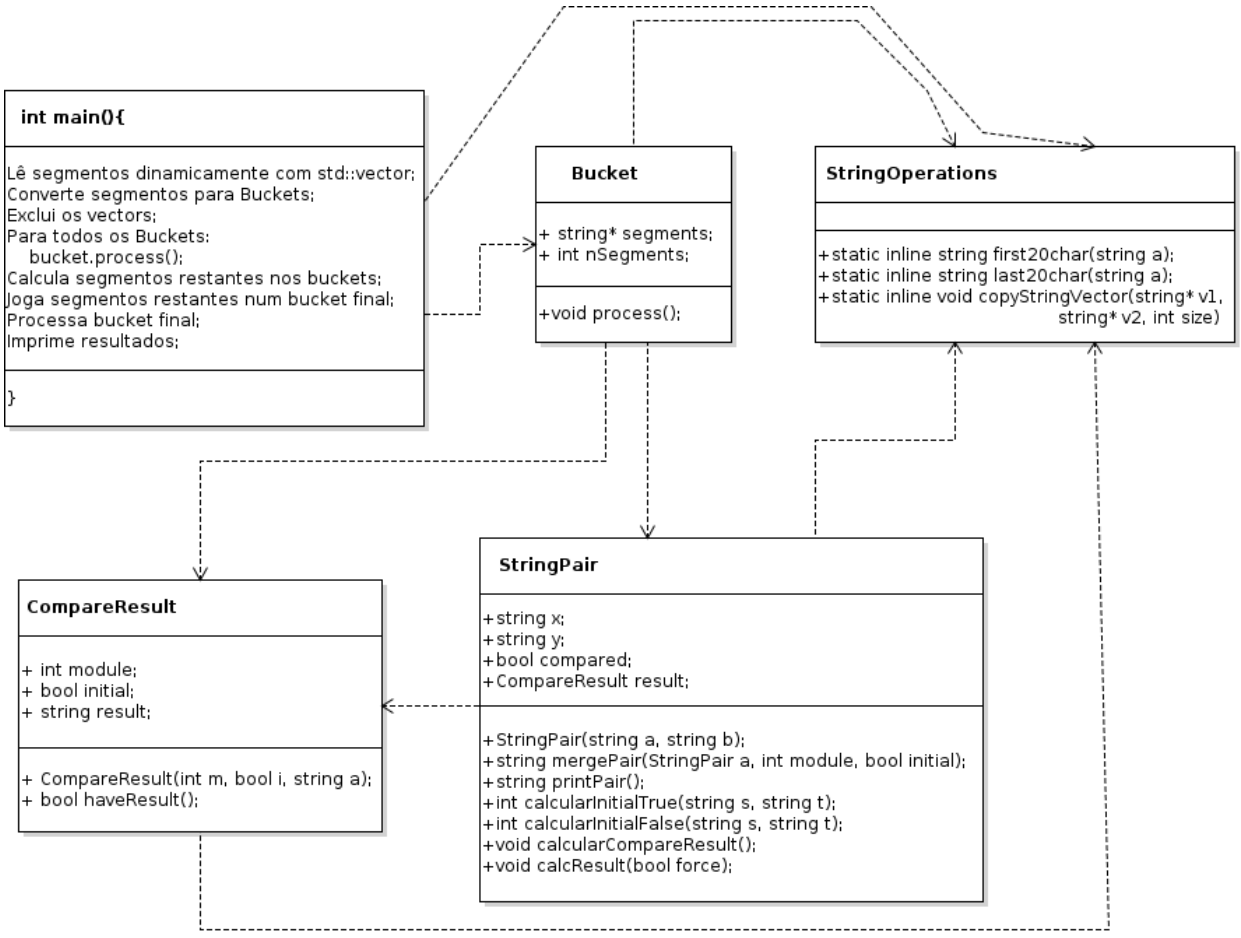
Todo o processo de transformar os diversos segmentos de texto inseridos num “Bucket” se baseia em combinar o par de segmentos que se combinam melhor e então repetir o processo, até que não haja mais o que combinar ou só reste um único segmento. Isso é feito pelo método Bucket::process(). É um enorme método com 155 linhas de código e nenhuma modularização. Esse método faz uso de uma matriz de semelhanças para decidir qual a melhor opção de combinação entre segmentos de texto. Isso tem um grande custo de memoria e o calculo da matriz, normalmente, teria complexidade quadrática ($O(n^2)$) e seria feito a cada iteração do algoritmo, resultando numa complexidade cúbica ($O(n^3)$). Porém, inspirado na programação dinâmica, foi adicionado a este método uma técnica que, através do armazenamento da versão da matriz na iteração anterior, consegue fazer com que só os elementos da matriz de semelhança que seriam diferentes sejam calculados. Isso diminuiu a complexidade do método de $O(n^3)$ para $O(n^2)$. Essa técnica apenas torna o método mais complexo.

Agora, usando o padrão c++11 e tendo toda a liberdade para usar templates, foi possível substituir o Bucket::process() antigo por uma nova versão, que armazena as semelhanças em uma estrutura de hashmap, o std::map. A complexidade *big O* ainda é a mesma (quadrática), porém o comprimento do método foi reduzido de 155 linhas para apenas 35 e agora o algoritmo é muito mais fácil de ser entendido por um leitor:

```
14  void Bucket::process(bool forceMerge){
15      map<StringPair, CompareResult> hashmap;
16      bool merged = false;
17      int iteration = 0;
18      do{
19          StringPair* keyMax = NULL;
20
21          for(int i = 1; i < segments.size(); i++){
22              for(int j = 0; j < i; j++){
23                  StringPair key = StringPair(segments[i], segments[j]);
24                  if(hashmap.find(StringPair(segments[i], segments[j])) == hashmap.end()){
25                      hashmap[key] = CompareResult(segments[i], segments[j], true);
26                  }
27                  if(hashmap[key].haveResult()){
28                      if(keyMax == NULL){
29                          keyMax = new StringPair();
30                          *keyMax = key;
31                      }else{
32                          if(hashmap[key].module > hashmap[*keyMax].module){
33                              *keyMax = key;
34                          }
35                      }
36                  }
37              }
38          }
39
40          if(keyMax != NULL){
41              segments.erase(find(segments.begin(), segments.end(), keyMax->x));
42              segments.erase(find(segments.begin(), segments.end(), keyMax->y));
43              segments.push_back(hashmap[*keyMax].result);
44              segments = segments;
45              merged = true;
46          }
47          iteration++;
48      }while(segments.size() > 1 && merged);
49  }
```

3. Modularização e revisão para nomes mais significativos

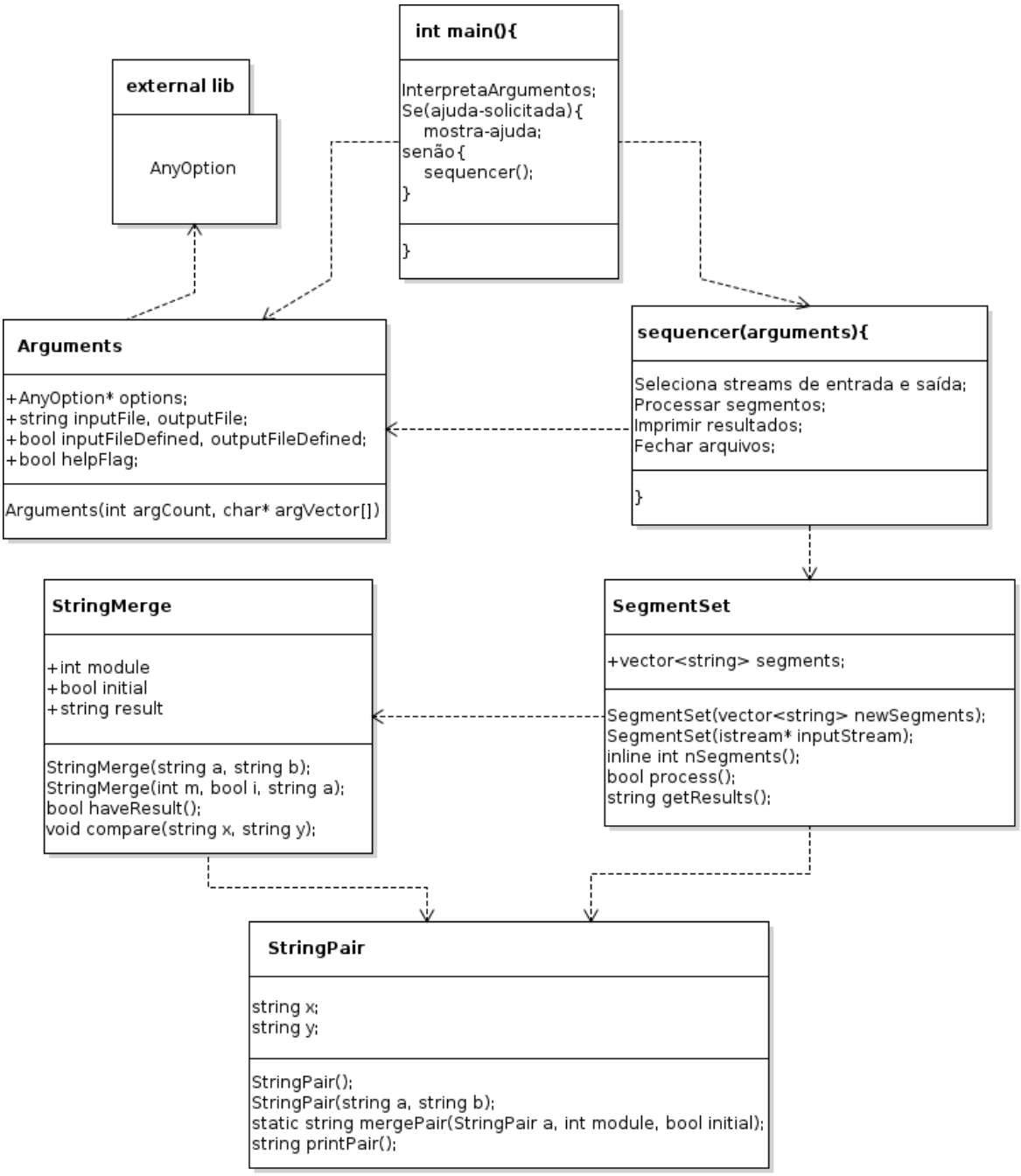
O diagrama a seguir mostra os componentes do software *Sequencer* e como eles se relacionavam antes das modificações feitas neste trabalho:



Há um evidente excesso de tarefas sendo feito unicamente na função "*int main()*" e no método Bucket.process(). Além disso os nomes em geral são pouco significativos e os próprios relacionamentos entre as funções e classes.

Serão feitas diversas mudanças, a principal delas é a mudança da classe "Bucket". Bucket (balde) não é um nome significativo, nem chega perto de ser. O nome da classe será substituído por SegmentSet (conjunto de segmentos). Essa classe também deixa de ser construída dentro da função int main(), ganhando um construtor adequado. Outra classe que teve seu nome alterado foi "CompareResult", que cuida da comparação de pares de strings e guarda o resultado dessas comparaçãoes. Ela foi renomeada para "StringMerge", para evidenciar melhor seu funcionamento. Diversas mudanças nos nomes de variáveis e separações de funções grandes em diferentes funções menores foram feitas.

O diagrama a seguir retrata o estado atual do software, após as mudanças efetivadas ao longo do trabalho:


external lib
AnyOption
int main(){
InterpretaArgumentos;
Se(ajuda-solicitada){
mostra-ajuda;
senão{
sequencer();
}
}
Arguments
+AnyOption* options;
+string inputFile, outputFile;
+bool inputFileDefined, outputFileDefined;
+bool helpFlag;
Arguments(int argCount, char* argVector[])
sequencer(arguments){
Seleciona streams de entrada e saída;
Processar segmentos;
Imprimir resultados;
Fechar arquivos;
}
StringMerge
+int module
+bool initial
+string result
StringMerge(string a, string b);
StringMerge(int m, bool i, string a);
bool haveResult();
void compare(string x, string y);
SegmentSet
+vector<string> segments;
SegmentSet(vector<string> newSegments);
SegmentSet(istream* inputStream);
inline int nSegments();
bool process();
string getResults();
StringPair
string x;
string y;
StringPair();
StringPair(string a, string b);
static string mergePair(StringPair a, int module, bool initial);
string printPair();

4. Programação defensiva

5. Testes

6. Documentação

7. Métricas

8. Referências:

- OpenMP C and C++ Application Program Interface, página 5. Disponível em: http://www.openmp.org/mp-documents/cspec20.pdf;
- http://openmp.org/wp/;
- http://npad.imd.ufrn.br/;
- https://github.com/pentalpha/AnyOption;