Pedro Cunial Campos
Luciano Soares e Igor Montagner
Engenharia de Computação – Insper
Maio 2018

# APS 02 Supercomputação: Modelos de Linguagem Distribuídos

Para a segunda APS da matéria de supercomputação, era esperado que os alunos implementassem um gerador de textos de linguagem natural baseado em um modelo probabilístico. Além disso, era esperada a implementação distribuída do mesmo modelo, visando o treinamento dos nós de um cluster. Nesta implementação, o aluno deveria utilizar a biblioteca MPI para C ou C++ para a comunicação entre estes diversos nós do cluster.

Para modelar a linguagem natural utilizou-se o seguinte modelo probabilístico:

$$P(w_1 w_2 \dots w_m) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)\dots P(w_m|w_1 \dots w_{m-1})$$

Onde *P(w)* corresponde a probabilidade de uma palavra (*w*) ocorrer, tal que a probabilidade de uma nova palavra em uma frase corresponde a probabilidade da dada palavra ocorrer dado as palavras anteriores à mesma.

No entanto, este modelo não seria escalável para textos muito grandes, uma vez que o algoritmo precisaria armazenar todas as possíveis cadeias de palavras conhecidas. Para isso, utilizamos um modelo de n-gramas, ou seja, o contexto considerado para a predição da próxima palavra resume-se apenas às ultimas *n* palavras, sendo *n* um valor definido pelo usuário. Em termos de implementação em código, foi assumido também que o valor de *n* será sempre menor do que o tamanho esperado de frase de saída, tal que caso o contrário a utilização de n-gramas não seria necessário.

Além disso, outra exigência do projeto era que o código rodasse a partir de *dumps* da Wikipedia[1], arquivos *xml* com o conteúdo e meta-dados dos artigos da mesma. Para auxiliar no *parsing* utilizei a biblioteca *pugixml*[2], facilitando o acesso a camadas mais profundas do *xml*.

Ainda no tópico de bibliotecas externas, utilizei a biblioteca do *boost* para MPI[3] para facilitar o acesso às funções do MPI em C++. Ao contrário do *pugixml*, é esperado que a máquina tenha o *boost-mpi* instalado, enquanto a *pugixml* compila junto com o código fonte.

[1] https://dumps.wikimedia.org/enwiki/20180401/

[2] https://pugixml.org/

[3] O boost-mpi não vem na instalação padrão do boost, para maior documentação:

https://www.boost.org/doc/libs/1_65_0/doc/html/mpi.html

Além disso, para maior facilidade de armazenamento dos n-gramas, criei uma estrutura de dados semelhante à *trie*, mas que armazena palavras ao invés de apenas caracteres.

Por fim, a geração destas *tries* e a sua interpretação funcionam a partir de uma estrutura de *tokenizador* e *parser*, onde o *tokenizador* abre o arquivo *xml* de entrada e devolve uma lista de *tokens*, sendo estes as palavras dos textos do *dump* (apenas textos dos artigos e comentários do mesmo), ordenados conforme a sua aparição. Com esta lista, o *parser*, gera as *"tries"* de maneira necessariamente sequencial (uma vez que a ordem na qual as palavras aparecem é de extrema importância para a lógica do algoritmo).

Com isso já seria possível a implementação sequencial da solução, mas para a implementação distribuída ainda seria necessário definir mais algumas respostas, como por exemplo qual seria o papel de cada nó nesta arquitetura.

Assim, fora decidido que cada nó realizaria o treinamento baseado em um *dump* e, no momento da geração de texto, os nós se comunicariam conforme fosse necessária a predição de uma nova palavra segundo a seguinte lógica:

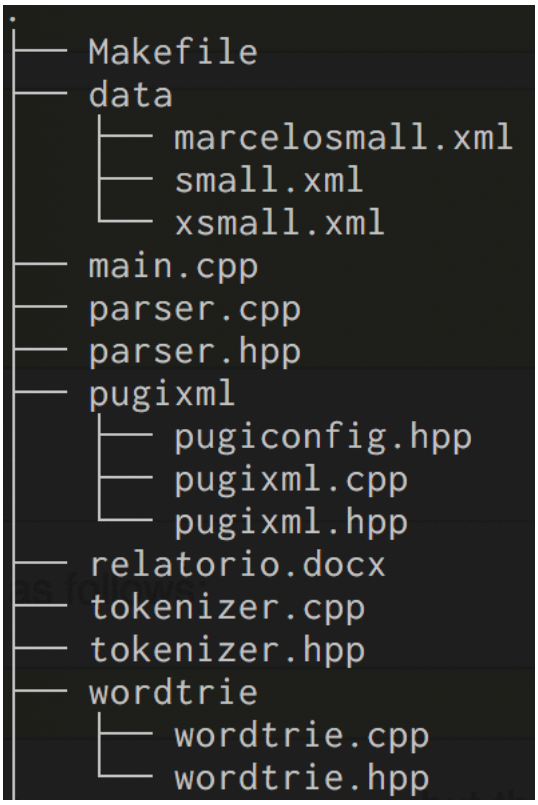
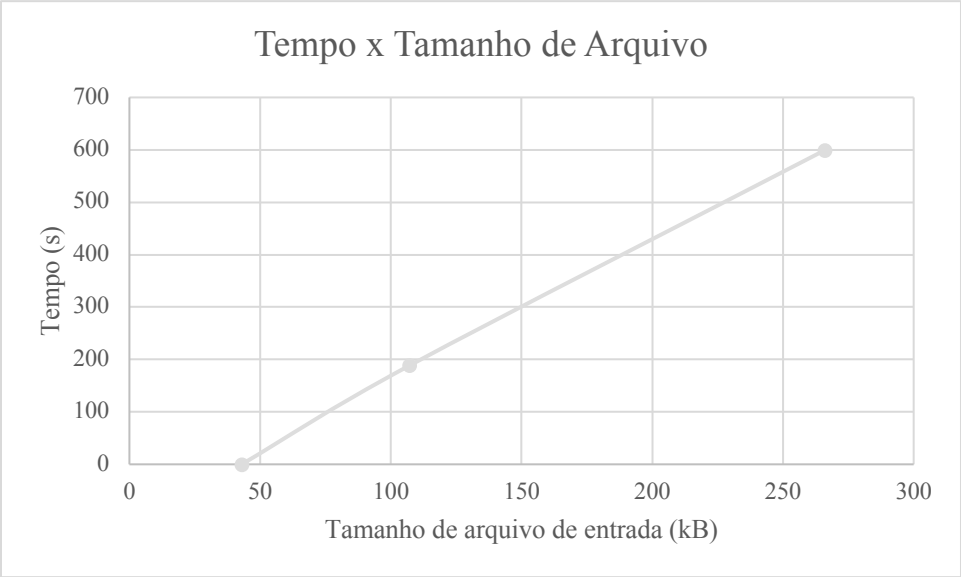



*Figura 1: Estrutura do projeto*

- Cada nó define a próxima palavra a ser adicionada na frase conforme o seu modelo já treinado;
- Os nós não-master enviam a sua palavra escolhida e a probabilidade da mesma de maneira assíncrona (*isend*) com uma barreira travando o mesmo até que o nó máster tenha recebido estas mensagens;
- O nó master recebe todas as palavras e probabilidades de maneira assíncrona (*irecv*) com uma barreira travando o mesmo até que tenha recebido o valor de todos os nós filhos;
- Com todos os valores no nó master, ele escolhe das palavras com base nas probabilidades enviadas;
- O master comunica todos os nós da palavra escolhida (*broadcast*, que é uma chamada blocante);
- Todos os nós atualizam as suas respectivas versões atuais da frase.

Com isso, para testar o desempenho do código, rodei o mesmo com dois processos (pois o meu computador possuí apenas duas threads) passando o mesmo arquivo de entrada para cada thread (tal que ambas tenham um volume equivalente de dados a serem processados). O gráfico abaixo mostra o desempenho de cada caso (note que o tamanho do arquivo de entrada corresponde ao tamanho de cada arquivo da entrada, de forma que o total a ser processado seria equivalente ao dobro deste volume em dados). É valido ressaltar, também, que o tempo de execução de um único arquivo destes na versão paralela do

código (a qual pode ser encontrada no *branch parallel* do repositório[4]) era apenas um pouco menor do que o da execução em paralelo com o dobro de dados, mesmo com o *overhead* atrelado à comunicação do MPI; Dado que esta comunicação ocorre apenas na predição do modelo (o que é muito pouco custoso se comparado ao treinamento do mesmo).




Assim, é possível concluir que a implementação paralela do problema é extremamente mais eficiente na solução do mesmo o que é devido, principalmente, a pequena magnitude do *overhead* atrelado a ela na implementação. Para maiores estudos de desempenho, seria interessante o teste do código para entradas maiores e em sistemas realmente distribuídos – como *clusteres*.

[4] https://github.com/pedrocunial/supercomp-insper