

PSPD – Programação p Sistemas Paralelos e Distribuídos / Turma A, Prof.: Fernando W. Cruz

Projeto de pesquisa: Construindo aplicações de larga escala com frameworks de programação paralela/distribuída

1. Objetivos:

O objetivo deste projeto é construir uma aplicação que atenda aos requisitos de performance e elasticidade para que ela possa se comportar como uma aplicação de larga escala (*large-scale application*).

2. Sobre a aplicação e os requisitos

Existem diversas soluções para atender ao requisito de performance de uma aplicação, geralmente envolvendo computação paralela e distribuída. Da mesma forma, existem soluções tecnológicas para garantir o requisito da elasticidade, que é a capacidade da aplicação se adaptar às mudanças de carga sem afetar a *performance* ou a qualidade do serviço oferecido (pode crescer ou diminuir em função da demanda dos usuários).

Neste projeto, o foco é a pesquisa em tecnologias que resolvem esses dois requisitos, embutindo-os no algoritmo “jogo da vida”¹, cuja especificação está detalhada no Anexo desta especificação. Para o atendimento de tais requisitos, considera-se o seguinte:

- a) O requisito da performance deve ser resolvido pelo teste de dois tipos de paralelismo: (i) usando o Apache Hadoop/Spark, e, (ii) usando as bibliotecas e respectivos pragmas OpenMP e MPI, em conjunto, no código original do jogo da vida. Essas duas opções de paralelismo são denominadas aqui de *engines*.
- b) O requisito da elasticidade deve ser resolvido pela adoção do orquestrador kubernetes (<https://kubernetes.io>), que deve ser estudado e devidamente configurado para uso numa versão em cluster, preferencialmente.

Os serviços da aplicação devem ser providos por uma interface de acesso via rede (*Socket server*), por onde chegam as requisições dos usuários, conforme ilustrado na Figura 1. Cabem aqui algumas observações:

- Na parte esquerda da figura estão os clientes interessados em enviar suas demandas de processamento. Neste caso, cada tupla refere-se a uma requisição do cliente, indicando o intervalo do jogo da vida no qual ele quer calcular os estágios (equivale a POWMIN e POWMAX, primeiras linhas do código que está no anexo desta especificação). Essas informações podem estar em arquivo ou em um *socket* cliente onde se digitam os parâmetros de entrada que serão enviados ao *Socket server*. Este último, por sua vez, deve ser programado para receber várias conexões de rede ao mesmo tempo (via socket TCP, UDP, etc.).
- Conforme já dito, o *engine*, nesta especificação refere-se ao módulo principal da aplicação responsável por fazer o cálculo dos estágios do jogo da vida, conforme os parâmetros de entrada recebidos pelo *Socket server*. Para este projeto, devem ser geradas duas versões do *engine*: (i) uma que faça o cálculo do jogo da vida utilizando o Apache Spark e (ii) outra que faça esse mesmo cálculo usando as bibliotecas OpenMP/MPI. Em qualquer dos casos, o *engine* deve estar integrado ao *Socket server* para receber as demandas dos clientes.

¹ Em linhas gerais, o jogo da vida é um algoritmo que calcula estágios evolutivos de uma população (simulação segundo critérios definidos) considerando os limites de teste fornecidos pelo usuário. A versão testada neste projeto realiza esta simulação para todos os tamanhos de sociedade, segundo um par de valores, informados pelo usuário. Ao final, o servidor retorna um OK – se a simulação da sociedade funcionou adequadamente, ou um NOK – se a simulação das sociedades não deu certo.

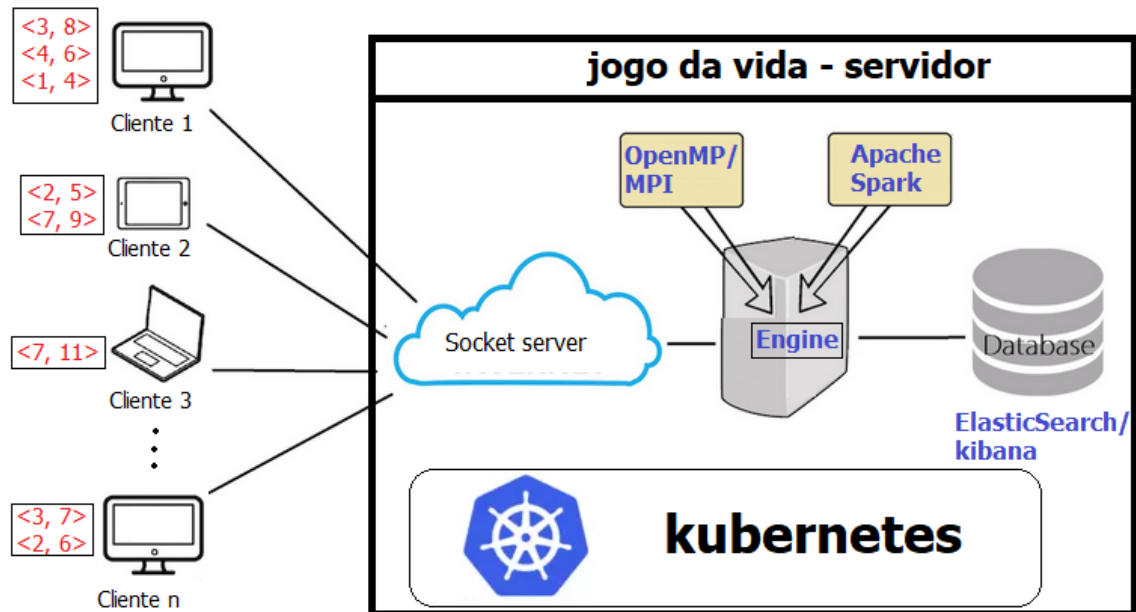


Figura 1 – Arquitetura da aplicação

- Na parte direita da figura está representada uma instância do banco de dados ElasticSearch/Kibana, que deve ser instalado e configurado para registrar e apresentar, graficamente as contabilizações de processamento dos *engines*. Algumas métricas que podem ser apresentadas são: número de clientes simultâneos, tempo médio de processamento por intervalo de tempo, tempo total gasto com cada cliente, total de requisições atendidas, média de requisições atendidas etc. (use a criatividade)
- Os módulos da aplicação servidora devem ser organizados na forma de *containers*, gerenciados pelo kubernetes (<https://kubernetes.io>) para garantir a orquestração das partes (que podem ser vistas como microserviços).

3. Dicas e restrições sobre a aplicação de larga escala

Algumas observações:

- Sobre o *Socket server*, este pode ser substituído pelo uso de um *broker* Kafka (<https://kafka.apache.org/>), de modo a atender diversos clientes, se essa alternativa for julgada mais conveniente do que a proposição original. Neste caso, essa decisão também deve ser discutida no relatório de entrega, a fim de que se possa entender porque tais mudanças foram implementadas.
- Na aplicação, é preciso definir uma estratégia de conexão entre os módulos *Socket server*, *ElasticSearch* e adotá-la em todos os testes. Por exemplo, a conexão do *engine* com ElasticSearch se dará por meio de um conector Kafka? Ou será instanciado em um *container*?
- A aplicação servidora deve ser construída de tal forma que seja possível comparar a performance dos engines OpenMP/MPI (trabalhando em conjunto, como uma aplicação mista) e Apache Hadoop/Spark (considerar aqui apenas os módulos necessários relevantes para o contexto de um “cluster” local).
- Os alunos devem decidir qual a melhor forma de testar os *engines* de processamento. Por exemplo, a aplicação vai ter dois *containers* simultâneos para os *engines* Apache Hadoop/Spark e OpenMP/MPI? Ou será melhor fazer execução em separado?
- As contabilizações registradas no banco de dados são de livre escolha dos alunos, mas devem permitir verificar o comportamento da aplicação sob diferentes demandas (quantidade de clientes, de processamento, etc.), sempre com foco na comparação dos elementos que impactam na *performance* e na elasticidade entre os *engines*. Obs.: Opcionalmente, se os alunos perceberem

outras variáveis que impactam no desempenho da aplicação, podem incluir no relatório de comparação

- A infraestrutura utilizada para instanciar essa aplicação deve ser planejada de modo a não gerar comparações errôneas de desempenho entre os *engines*. Ou seja, as infraestruturas devem ser padronizadas e os testes devem ser calibrados para evitar conclusões contaminadas por configurações diferentes. Por exemplo, se houver uso de GPUs, essa decisão deve ser adotada para os dois *engines*.
- Para teste de *stress* da aplicação, faz-se necessária a instanciação de vários clientes de modo que se tenham conexões distintas para o servidor para que se possa medir a elasticidade da aplicação em diferentes condições de carga. Talvez a construção de uma aplicação cliente que abra várias conexões de rede possa solucionar esse problema. Nesse caso, o código dessa aplicação adicional deve ser documentado e incluído na entrega do projeto.
- Todos os arquivos de configuração e limites definidos para ampliação/redução do número de instâncias da aplicação devem ser apresentados no relatório final.

4. Questões de Ordem

- Este projeto pode ser feito por grupos de até 3 alunos
- Este projeto deve ter os artefatos entregues no Moodle (arquivo zipado) e apresentado em data estabelecida pelo professor. A entrega deve ser composta por: (i) relatório do laboratório + slides, e (ii) códigos, instruções de uso e todas as informações necessárias para esclarecimento e uso dos programas entregues.
- Em qualquer das versões, se o programa estiver certo, o veleiro (configuração inicial do tabuleiro com indicação de células vivas) deve sair do canto superior esquerdo e chegar no canto inferior direito do tabuleiro. Portanto, as comparações de desempenho só serão válidas se essa condição for satisfeita durante as execuções.
- Os alunos podem realizar o experimento em qualquer plataforma, inclusive em equipamentos locais, mas devem estar preparados para demonstração da aplicação funcionando *in loco*.
- O relatório deve conter o seguinte:
 - i) Identificação da disciplina/turma, do grupo (matrícula e nome) e o nome do projeto
 - ii) Introdução – pequeno texto explicando o contexto do projeto e a metodologia adotada para alcançar os resultados solicitados
 - iii) Metodologia usada para resolver os requisitos da aplicação
 - iv) Requisito de performance – fazer um texto introdutório e abrir, no mínimo, uma subseção para cada um dos engines citados (Apache Spark e OpenMP/MPI) de modo que se possa descrever o que foi feito. Documentar todos os passos relevantes, comparações feitas dificuldades encontradas e soluções para os problemas percebidos.
 - v) Requisito de elasticidade – fazer texto introdutório e abrir no mínimo duas subseções, sendo uma para falar sobre o kubernetes e outra para falar sobre a aplicação e as configurações feitas, quais as modificações feitas na aplicação para garantir a elasticidade. Documentar todos os passos relevantes, resultados encontrados.
 - vi) Análise dos resultados de comparação, em função do que foi decidido colocar no banco de dados Elasticsearch. Incluir os desenhos das telas de eventuais gráficos gerados e informações relevantes sobre o aspecto da comparação. Mostrar comparações dos códigos acima, de modo a se identificar qual deles apresenta melhor performance (menor tempo de execução), sob as mesmas condições (dimensões da sociedade, porções do código paralelizadas, número de *threads*, núcleos, etc.).
 - vii) Conclusão – apresentação de resultados, visão geral, comentários sobre dificuldades e soluções encontradas em cada uma das versões do jogodavida.c e também sobre a facilidade de

implantação das soluções tecnológicas para performance e elasticidade. Ao final, cada membro do grupo abre uma subseção para comentários pessoais sobre a pesquisa, indicando as partes que mais trabalhou, aprendizados e uma nota de autoavaliação.

- viii) No anexo, colocar listagem dos códigos com comentários e indicações sobre qual parte foi paralelizada, instruções de execução, scripts para o container e outras informações relevantes.

5. Referências

[1] Gardner, M. "Mathematical Games – The fantastic combinations of John Conway's new solitaire game life", Scientific American 223, Oct, 1970, pp 120-123. Disponível em http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm.

ANEXO – Código jogodavida.c

O Jogo da Vida, criado por John H. Conway (GARDNER, 1970), utiliza um autômato celular para simular gerações sucessivas de uma sociedade de organismos vivos. Este jogo é composto por um tabuleiro bidimensional, infinito em qualquer direção, de células idênticas. Cada célula tem exatamente oito células vizinhas (todas as células que compartilham uma aresta ou um vértice com a célula original). Para identificar essa vizinhança, fixe os olhos em uma célula de um tabuleiro de xadrez e perceba as casas vizinhas); cada célula pode estar em um de dois estados: viva ou morta.

Uma geração da sociedade é representada pelo conjunto dos estados das células do tabuleiro. Sociedades evoluem de uma geração para a próxima aplicando simultaneamente, a todas as células do tabuleiro, regras que estabelecem o próximo estado de cada célula. As regras são:

- Células vivas com menos de 2 vizinhas vivas morrem por abandono
- Células vivas com mais de 3 vizinhas vivas morrem de superpopulação
- Células mortas com exatamente 3 vizinhas vivas tornam-se vivas
- As demais células mantêm seu estado anterior.

A simulação desse jogo pode ser feita pelo uso de um tabuleiro NxN, orlado por células eternamente mortas, no qual o "veleiro" (uma configuração de células vivas, permeadas por células mortas) sai do canto superior esquerdo e chega no canto inferior direito do tabuleiro, de modo a simular as mudanças geracionais desta sociedade de organismos.

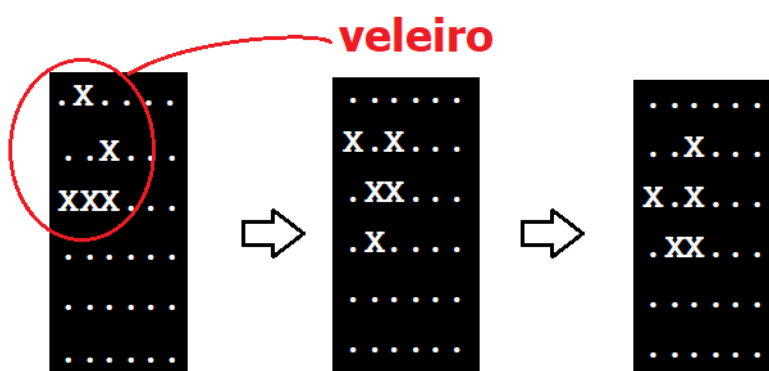


Figura 2 – Tabuleiro com matriz quadrada 6 x 6 e a configuração do "veleiro"

A Figura 2, é um exemplo de tabuleiro 6x6 com células vivas (1, expressas por um X) e mortas (0, expressa por um ponto). Neste exemplo, aparecem três gerações da sociedade de organismos, pela aplicação das regras citadas. Vale ressaltar que um tabuleiro 6x6 orlado por células mortas obriga a definição de uma matriz 8 por 8 (as células de orla não aparecem na figura).

A seguir, a listagem do código do jogo da vida a ser adaptado para resolução do projeto.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define ind2d(i,j) (i)*(tam+2)+j
#define POWMIN 3
#define POWMAX 10
double wall_time(void) {
    struct timeval tv;    struct timezone tz;
    gettimeofday(&tv, &tz);
    return(tv.tv_sec + tv.tv_usec/1000000.0);
} /* fim-wall_time */

double wall_time(void);
void UmaVida(int* tabulIn, int* tabulOut, int tam) {
    int i, j, vizviv;

    for (i=1; i<=tam; i++) {
        for (j= 1; j<=tam; j++) {
            vizviv = tabulIn[ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j )] +
            tabulIn[ind2d(i-1,j+1)] + tabulIn[ind2d(i ,j-1)] +
            tabulIn[ind2d(i ,j+1)] + tabulIn[ind2d(i+1,j-1)] +
            tabulIn[ind2d(i+1,j )] + tabulIn[ind2d(i+1,j+1)];
            if (tabulIn[ind2d(i,j)] && vizviv < 2)
                tabulOut[ind2d(i,j)] = 0;
            else if (tabulIn[ind2d(i,j)] && vizviv > 3)
                tabulOut[ind2d(i,j)] = 0;
            else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
                tabulOut[ind2d(i,j)] = 1;
            else
                tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
        } /* fim-for */
    } /* fim-for */
} /* fim-UmaVida */

void DumpTabul(int * tabul, int tam, int first, int last, char* msg){
    int i, ij;

    printf("%s; Dump posições [%d:%d, %d:%d] de tabuleiro %d x %d\n", \
    msg, first, last, first, last, tam, tam);
    for (i=first; i<=last; i++) printf("="); printf("\n");
    for (i=ind2d(first,0); i<=ind2d(last,0); i+=ind2d(1,0)) {
        for (ij=i+first; ij<=i+last; ij++)
            printf("%c", tabul[ij]? 'X' : '.');
        printf("\n");
    }
    for (i=first; i<=last; i++) printf("="); printf("\n");
} /* fim-DumpTabul */
```

```
void InitTabul(int* tabulIn, int* tabulOut, int tam){
    int ij;
    for (ij=0; ij<(tam+2)*(tam+2); ij++) {
        tabulIn[ij] = 0;
        tabulOut[ij] = 0;
    } /* fim-for */
    tabulIn[ind2d(1,2)] = 1; tabulIn[ind2d(2,3)] = 1;
    tabulIn[ind2d(3,1)] = 1; tabulIn[ind2d(3,2)] = 1;
    tabulIn[ind2d(3,3)] = 1;
} /* fim-InitTabul */

int Correto(int* tabul, int tam){
    int ij, cnt;
    cnt = 0;
    for (ij=0; ij<(tam+2)*(tam+2); ij++)
        cnt = cnt + tabul[ij];
    return (cnt == 5 && tabul[ind2d(tam-2,tam-1)] &&
            tabul[ind2d(tam-1,tam )] && tabul[ind2d(tam ,tam-2)] &&
            tabul[ind2d(tam ,tam-1)] && tabul[ind2d(tam ,tam )]);
} /* fim-Correto */

int main(void) {
    int pow, i, tam, *tabulIn, *tabulOut;
    char msg[9];
    double t0, t1, t2, t3;
    // para todos os tamanhos do tabuleiro
    for (pow=POWMIN; pow<=POWMAX; pow++) {
        tam = 1 << pow;
        // aloca e inicializa tabuleiros
        t0 = wall_time();
        tabulIn = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
        tabulOut = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
        InitTabul(tabulIn, tabulOut, tam);
        t1 = wall_time();
        for (i=0; i<2*(tam-3); i++) {
            UmaVida(tabulIn, tabulOut, tam);
            UmaVida(tabulOut, tabulIn, tam);
        } /* fim-for */
        t2 = wall_time();
        if (Correto(tabulIn, tam))
            printf("***Ok, RESULTADO CORRETO**\n");
        else
            printf("***Nok, RESULTADO ERRADO**\n");
        t3 = wall_time();
        printf("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7f, tot=%7.7f \n",
            tam, t1-t0, t2-t1, t3-t2, t3-t0);
        free(tabulIn); free(tabulOut);
    }
    return 0;
} /* fim-main */
```

