UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO PARA SIMULAÇÃO DA DIFUSÃO DO CALOR EM UM AMBIENTE DISTRIBUÍDO COM CARGA BALANCEADA

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

JÚLIO CESAR ZIMERMANN

BLUMENAU, NOVEMBRO/2002

2002/2-41

PROTÓTIPO PARA SIMULAÇÃO DA DIFUSÃO DO CALOR EM UM AMBIENTE DISTRIBUÍDO COM CARGA BALANCEADA

JÚLIO CESAR ZIMERMANN

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Dr. Paulo Cesar Rodacki Gomes — Orientador

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Dr. Paulo Cesar Rodacki Gomes

_____

Maurício Capobianco Lopes

_____

Francisco Adell Péricas

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Paulo César Rodacki Gomes, pela confiança, calma, incentivo e cumplicidade durante todo o processo de desenvolvimento do trabalho.

Ao meu co-orientador Prof. Dr. Nelson Hein pela sugestão do tema que originou o presente trabalho.

A minha família, em especial a minha mãe Maria Odete Zimermann, por todo o apoio e incentivo oferecido para a realização do sonho que me leva a condição de formando. Vencemos mais esta batalha e que este seja apenas o primeiro título do primeiro filho a se formar na universidade.

Um especial agradecimento a minha namorada, Silvana da Silva Luis, que mesmo sendo preterida em tantas oportunidades ao longo do tempo em que freqüentei a academia soube ser cúmplice e companheira na conquista desta realização.

Agradeço aos professores da academia, que tiveram paciência ao longo desta convivência crítica, inquieta, questionadora e às vezes solitária frente aos desafios da busca do conhecimento.

O agradecimento especial a todos os colegas da FURB e aos donos de bares que garantiram a cerveja gelada, que contracenaram em todos os momentos de alegrias e tristezas ao longo desta jornada.

Finalmente, o agradecimento ao Deus que me inspira e me oferece companhia nos momentos mais difíceis.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

RESUMO

O presente trabalho apresenta a implementação de um protótipo de dissipação de calor em um ambiente de processamento distribuído, onde vários computadores são utilizados para ajudar a solucionar a necessidade de cálculo do problema. Enfoca, ainda, discussões sobre a utilização de objetos distribuídos, através do estudo comparativo entre a performance da mesma aplicação desenvolvida através de um protótipo *stand-alone* e outro distribuído em ambiente de carga balanceada. Para promover o estudo comparativo, foram desenvolvidas duas aplicações, para a solução do mesmo problema e que tem por base a solução de um problema clássico do cálculo numérico, envolvendo processos de dissipação de calor em uma chapa metálica. Os supracitados protótipos foram desenvolvidos utilizando a Interface de Programação de Aplicações Java™ 2 (API– Java™ 2), versão 1.4.0_2, Standart Edition, com a utilização do pacote Java RMI na aplicação distribuída.

ABSTRACT

The present work presents the implementation of a distributed software prototype for a classical problem of heat dissipation simulation with load balancing among components of the distributed architecture. The problem involves numeric heat flow calculation, including the processes of heat dissipation in a metallic foil. Also, the present work promotes a discussion on the use of distributed objects, through a comparative study. To promote such study, two applications were developed: a distributed prototype with load balancing among the its components and a stand-alone version of the distributed prototype. The software prototypes were developed using API – Applications Programming Interface – Java™ 2, version 1.4.0_2, Standard Edition, along with Java RMI for the distributed application.

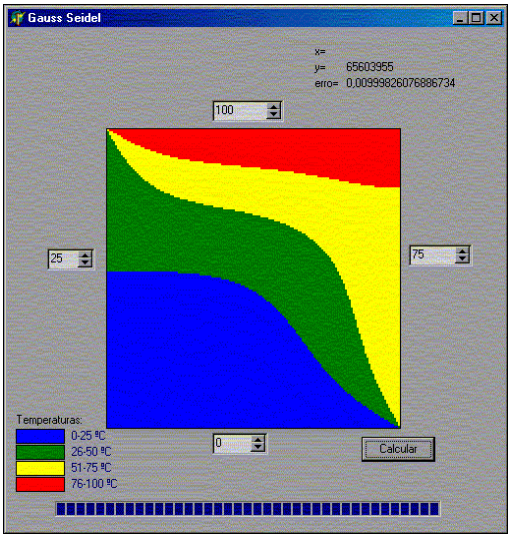
1 INTRODUÇÃO

A termodinâmica apresenta como um de seus temas o estudo de modelos de difusão térmica, onde são abordados fenômenos de transferência de calor. Dentro deste tema são discutidas as formas como o calor se propaga em diferentes matérias. Supondo-se uma chapa de um material sólido, homogêneo e que permita a difusibilidade térmica constante, de comprimento (L_1) e largura (L_2) iguais, desprezando-se a espessura, perdas de calor e constantes de difusibilidade térmica. Considerando-se que haja nas extremidades desta chapa quatro fontes de calor, com temperaturas constantes e de diferentes valores. Partindo-se deste modelo acima descrito, apresenta-se um problema de condutividade de calor, onde o calor das regiões mais quentes migra para as regiões mais frias, ao longo do tempo, até que haja equilíbrio (Wolgemuth, 1996, p.289).

Discretizando-se esta chapa em partículas de tamanhos conhecidos e através de métodos de cálculos discutidos pela álgebra linear, como o método de Gauss-Seidel, modelado por Hein (2000), pode-se obter a temperatura numérica de cada partícula.

O modelo acima proposto foi trabalhado culminando no desenvolvimento de um protótipo, para computadores, implementado em Delphi 5.0, utilizando a modelagem funcional definida por Furlan (1998), conforme ilustrado na fig. 1. Nesta aplicação, fornece-se a temperatura inicial, de quatro fontes de calor, uma em cada extremidade de uma chapa quadrada, para que após o processamento matemático seja apresentada, através de uma interface gráfica, as isofaixas de temperaturas, distintas uma das outras através de cores.

FIGURA 1 – PROTÓTIPO DESENVOLVIDO EM DELPHI 5.0



Ao utilizar-se este protótipo, implementado em Delphi e com modelagem funcional, concluiu-se que o mesmo era lento no caso de se discretizar a chapa em um número maior de elementos ou determinar uma maior precisão, também chamado de erro, do resultado. A constatação de a estratégia adota para o processamento desta primeira versão, desenvolvida em Delphi, é lenta motivou o desenvolvimento de novas estratégias, utilizando a modelagem orientada a objetos e desenvolvendo um ambiente distribuído, abandonando a linguagem Object Pascal (Delphi), buscando uma nova opção entre as linguagens disponíveis, no caso espefícico optou-se por Java, atraído pelas facilidades encontradas na literatura sobre Java RMI.

Uma proposta, para solucionar o problema do tempo de processamento, é o de dividir o cálculo algébrico entre vários computadores, que processem uma porção do problema paralelamente.

Considerando-se que um sistema de objetos distribuídos pode adotar uma estrutura cliente-servidor, a invocação de métodos remotos permite que diferentes objetos em diferentes computadores se comuniquem, através de chamadas a objetos remotos. (Deitel, 2001, p. 891).

Assim, através de um sistema de processamento paralelo distribuído pode-se discretizar o problema da condutividade do calor, particionando as unidades de processamento. Uma unidade gestora (cliente) divide o sistema de equações para cálculo da temperatura em sistemas menores e os envia para unidades de cálculo (servidores de cálculo), distribuídas ao longo de uma rede de computadores, para que processem paralelamente uma porção dos cálculos e retornem o resultado, seqüencialmente até que uma certa tolerância numérica, determinada como o equilíbrio térmico seja alcançada, chegando-se à solução numérica do problema.

Esta aplicação gestora deve promover a distribuição estática de carga entre as unidades de cálculo que apresentarem melhor desempenho ao longo do tempo de processamento. A distribuição estática de carga solicita alocação de recursos antes do processamento numérico do problema. Um modelo de distribuição dinâmica de carga, ao contrário do estático, re-aloca recursos em tempo de execução, o que aumenta o tráfego de mensagens entre os módulos (cliente/servidor) (Hull, 1994), aumentando o tempo de processamento da aplicação. Desta

forma, o modelo estático foi escolhido como base para a arquitetura a ser implementada neste trabalho.

Como um sistema distribuído apresenta como características a utilização de objetos escritos numa mesma linguagem de programação, com diferentes sistemas operacionais, diferentes máquinas e com diferente capacidade de processamento (Júnior, 2001, p. 146), através do balanceamento de carga analisa-se a carga sobre todos os recursos disponíveis para determinar qual a melhor opção de alocação destes recursos (Balen, 2000, p. 232).

O propósito do presente trabalho é desenvolver duas aplicações, uma *stand-alo*ne e outra distribuída com balanceamento de carga e comparar o seu desempenho. Ambas aplicações serão concebidas utilizando o modelo orientado a objetos. Rumbaugh (Rumbaugh apud Furlan, 1998, p. 15) define orientação a objeto como “uma nova maneira de pensar os problemas utilizando modelos organizados a partir de conceitos do mundo real. O componente fundamental é o objeto que combina estrutura e comportamento em uma única entidade”. Por opção do autor do presente trabalho a aplicação desenvolvida em Delphi 5.0 foi descartada e utiliza-se a Interface de Programação de Aplicações Java™ 2 (API – *Applications Programming Interface* – Java™ 2), versão 1.4.0_2, Standard Edition, para o desenvolvimento de ambos os protótipos, criando uma base mais sólida para a comparação dos protótipos. O protótipo distribuído utiliza os recursos de Java RMI, que é a tecnologia que permite a comunicação distribuída de um objeto Java com outro. “Uma vez que um método (ou serviço) de um objeto Java é registrado como sendo remotamente acessível, um cliente pode pesquisar este serviço e receber uma referência que permita ao cliente utilizá-lo, isto é, chamar o método” (Deitel, 2001, P. 891).

1.1 OBJETIVOS

O específico deste trabalho é o desenvolvimento de dois protótipos de simulação de difusão de calor, que demonstre através de cores a temperatura em cada região de uma chapa, sendo um *stand-alone* e outro em um ambiente distribuído com balanceamento de carga estática.

Estão relacionados os seguintes objetivos secundários:

a) implementação de uma arquitetura de balanceamento de carga estática para a aplicação distribuída, com base em serviços Java RMI para simulação da difusão de calor, baseada em arquiteturas de balanceamento de carga já existentes;
b) comparação do desempenho entre a aplicação em um sistema de processamento distribuído de carga balanceada estática de simulação de calor e um sistema stand-alone.

1.2 ESTRUTURA

O presente trabalho foi estruturado em sete capítulos, apresentando inicialmente os objetivos do trabalho e a sua estrutura.

O capítulo 2 apresenta a fundamentação teórica sobre processos de difusão de calor, a solução matemática da difusão do calor, a forma para discretização de uma chapa com uma solução de como dividir o problema de forma a permitir a distribuição dos dados e apresentação da solução algébrica para o balanceamento de carga.

O capítulo 3 apresenta estudos sobre objetos distribuídos e serviços Java RMI, que representam a base tecnológica para a implementação do sistema de processamento distribuído.

No capítulo 4 é feita a referência sobre balanceamento de carga, escalonamento e sistemas de processamento paralelo, utilizado pela comunicação entre a interface local e objetos remotos da aplicação distribuída.

No capítlo 5 é apresentado o desenvolvimento dos protótipos, utilizando a Linguagem de Modelagem Unificada (UML - *Unified Modeling Language*) com a ferramenta CASE Power Designer para modelagem e a implementação; e codificação utilizando a linguagem de programação Java, utilizando a ferramenta de desenvolvimento Forte for Java Comunnity Edition da Sun Microsystems.

No capítulo 6 é apresentada a comparação entre a performance o sistema distribuído paralelo e um sistema *stand-alone*.

No capítulo 7, são feitas as considerações finais sobre o presente trabalho, comentando as conclusões, dificuldades encontradas e sugestões para trabalhos futuros.

2 PROCESSOS DE TRANSMISSÃO OU DIFUSÃO DE CALOR

Este capítulo apresenta a fundamentação teórica sobre processos de difusão de calor, solução matemática da difusão do calor, a discretização de uma chapa com uma solução de como dividir o problema de forma a permitir a distribuição dos dados e apresentação da solução algébrica para o balanceamento de carga.

2.1 TRANSMISSÃO DE CALOR

A termodinâmica é a ciência que estuda as transições quantitativas e as transformações de energia em calor. Dentre os objetos de estudo desta ciência encontra-se a transmissão ou difusão de calor (Kern, 1982, p. 1). Segundo Cerbe (1973, p. 284) um sistema diatérmico troca calor com o ambiente quando entre os dois existe uma diferença de temperatura. Essa transmissão realiza-se sempre na direção das temperaturas decrescentes. O calor que segue da fonte quente para a fonte fria, desloca-se no tempo e no espaço, assim pode-se assumir que a temperatura (T) depende do espaço (x) e do tempo (t), ou seja, $T_{(x/t)}$.

Um corpo sólido isolado está em equilíbrio térmico se a sua temperatura for a mesma em qualquer parte do corpo. Se a temperatura no sólido não for uniforme, o calor será transmitido por atividade molecular das regiões de temperaturas elevadas paras as regiões de baixas temperaturas. O processo, chamado de condução de calor, é dependente do espaço e do tempo, e continuará até que um campo uniforme de temperatura exista em todo o corpo isolado (Wolgemuth, 1996, p. 289).

2.2 PROCESSOS DE DIFUSÃO DE CALOR

Há um relacionamento direto entre as fases, ou estado físico da matéria, de uma substância com seus respectivos conteúdos de energia. O calor pode ser transmitido por três processos diferentes, que na prática geralmente ocorrem simultaneamente entre vários estados, mas que precisam ser estudados separadamente: a condução, a convecção e a radiação (também denominada irradiação). (Cerbe, 1973, p. 284). O presente trabalho reserva-se a discutir apenas a condução do calor, que diz respeito ao estado sólido da matéria,

onde não há energia suficiente para causar o distanciamento entre as moléculas ou átomos e por estarem muito próximos, origina-se a sua rigidez.

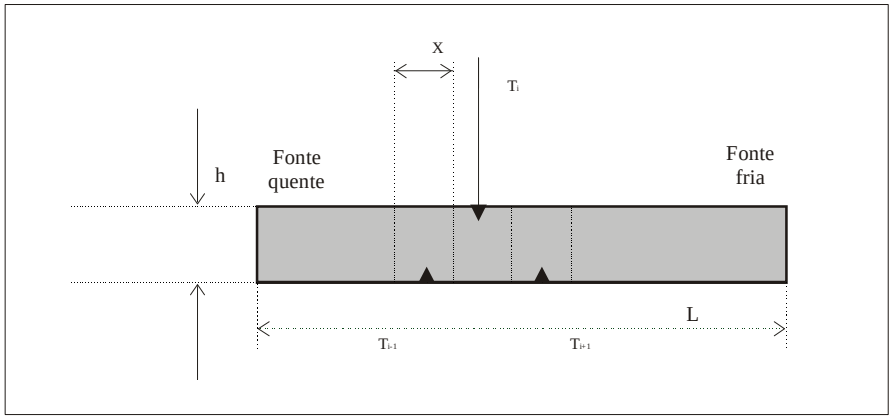
A condução de calor só se realiza entre as partículas imediatamente vizinhas de corpos sólidos ou entre gases ou líquidos em repouso. (Cerbe, 1973, p. 284) O processo ocorre, por exemplo, em chapas metálicas.

2.2.1 DIFUSÃO POR CONDUÇÃO UNIDIMENSIONAL DE CALOR

No problema considerado, a condução de calor (ou energia térmica) pode ocorrer através de um material fixo ou sólido, tal como barras ou chapas metálicas. O sentido do fluxo de calor numa chapa sólida pode ocorrer em todas as direções desde que as superfícies possuam as mesmas propriedades térmicas, apresentando portanto homogeneidade do material, e as mesmas constantes numéricas de dissipação de calor (Kern, 1982, p. 1).

Para formulação do problema de difusão por condução unidimensional de calor, considera-se uma barra de comprimento qualquer (L), e largura e altura infinitesimais, feita de material sólido, homogêneo e que permita a difusibilidade térmica[1] constante. Desprezando-se eventuais perdas de calor, pode-se supor ainda que cada uma das extremidades da barra possua uma fonte de calor a temperatura constante, conforme demonstrado na fig. 2.

FIGURA 2 – CONDUÇÃO DE CALOR EM UMA BARRA METÁLICA



Fonte: Hein (2000, p 1)

[1] Difusibilidade ou condutividade térmica é o inverso a resistência ao fluxo de calor, podendo ser definida, também, como o produto entre o fluxo de calor e potencial térmico (Kern, 1982, p. 1).

A barra é discretizada em um número n (com n=3) de partículas de comprimento Δx. Como há uma fonte quente (T_{i-1}) e outra fria (T_{i+1}), há o deslocamento de calor, no espaço Δx, da região de maior para a de menor temperatura. Durante a condução de calor na barra, considerando-se que o calor se desloca no sentido decrescente de temperatura tem-se:

QUADRO 1 – EQUAÇÃO 1

$$T_{i-1} > T_i > T_{i+1}$$

Fonte: Hein (2000, p. 26)

O calor que emana da fonte quente para a fonte fria, desloca-se no tempo e no espaço (unidimensional). Assim pode-se assumir que a temperatura (T) depende do espaço (x) e do tempo (t), ou seja, reedita-se: T(x,t) ou seja, temperatura T no espaço x, no tempo t. Ao considerar um instante t, fixo no tempo, pode-se escrever que a variação de temperatura é dada por:

QUADRO 2 – EQUAÇÃO 2

$$\frac{\Delta_i}{\Delta x} = \frac{T_{i+1} - T_i}{\Delta x}$$

Fonte: Hein (2000, p. 26)

Porém, a temperatura em Ti é influenciada pela partícula imediatamente anterior, e assim sucessivamente, logo:

QUADRO 3 – EQUAÇÃO 3

$$\frac{\Delta T_i - \Delta T_{i-1}}{\Delta x} = \frac{\frac{T_{i+1} - T_i}{\Delta x} - \frac{T_i - T_{i-1}}{\Delta x}}{\Delta x} = \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = \frac{\Delta^2 T_i}{\Delta x^2} \cong \frac{\partial^2 T_i}{\partial x^2}$$

Fonte: Hein (2000, p. 26)

Similarmente, considerando um ponto fixo x no espaço da barra:

QUADRO 4 – EQUAÇÃO 4

$$\frac{\Delta T}{\Delta t} = \frac{T_{i+1} - T_i}{\Delta t} \cong \frac{\partial T}{\partial t}$$

Fonte: Hein (2000, p. 26)

As duas variações são proporcionais entre si, assim:

QUADRO 5 – EQUAÇÃO 5

$$\frac{\Delta T}{\Delta t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

Fonte: Hein (2000, p. 26)

Onde $\alpha = \frac{k}{c\rho}$ é a constante de difusibilidade térmica. A função de condução do calor T(x,t), tem como unidade x (posição) em metros e t (temperatura) em graus centígrados. A constante k é a constante de condutividade térmica, c é o calor específico e ρ é a densidade do material. Como inicialmente o meio foi considerado homogêneo, tem-se que k, c, e ρ são independentes de x e passíveis de desconsideração quando o propósito for a simulação do calor.

A partir do processo de modelagem das expressões descrito em Hein (2000, p. 27), reduz-se as expressões, abstraindo parâmetros como a difusibilidade térmica e as perdas de calor, de forma a criar condições de simulação, abstraindo também o conhecimento da redução matemática. “O desenvolvimento de modelos matemáticos (modelagem matemática) consiste em buscar o aprendizado através da matemática para a criação de uma solução computável, utilizando um dispositivo eletrônico através do resultado da expressão alcançada” (Hein, 2000, p. 28).

Como o processo de simulação é iterativo pode-se escrever a equação em função dos índices i (partícula) e j (iteração), como demonstra quadro 6.

QUADRO 6 – EQUAÇÃO 5

$$T_i^{\,j+1} = \frac{T_{i-1}^{\,j} + T_i^{\,j} + T_{i+1}^{\,j}}{3}$$

Fonte: Hein (2000, p. 28)

Partindo-se da redução, proposta pela modelagem matemática, pode-se fazer uma simulação com uma barra de comprimento L, dividida em 5 partículas como demonstrado na tabela 1. É conveniente convencionar que a diferença da temperatura pode ser expressa em termos de graus Kelvin (K), ou graus Celsius (ºC). Uma vez que se trabalha com diferenças de temperaturas, é necessário trabalhar com uma escala consistente de temperatura na escala utilizada. A diferença de temperatura em ºC é numericamente igual a diferença de temperatura na escala K (Wolgemuth, 1996, p. 298). Na tabela 1, todas as temperaturas apresentam uma escala de temperatura consistente, ou seja, utilizando um mesmo padrão.

TABELA 1 – SIMULAÇÃO FEITA EM UMA BARRA, DIVIDIDA EM 5 PARTÍCULAS.

j	t(i)				
	t1	t2	t3	t4	t5
0	100	20	20	20	0
1	100	46,67	20	13,33	0
2	100	55,56	26,67	11,11	0
3	100	60,74	31,11	12,59	0
4	100	63,95	34,81	14,57	0
5	100	66,26	37,78	16,46	0
6	100	68,01	40,16	18,08	0
7	100	69,39	42,09	19,41	0
8	100	70,49	43,63	20,50	0
9	100	71,37	44,87	21,38	0
10	100	72,08	45,88	22,08	0

Baseado em exemplos de Hein, (2000, p.29), para promover o entendimento numérico do que está acontecendo, foi criado um modelo de estudos. Reporte-se a iteração a tabela 1, onde j = 9, na partícula t = 3. A temperatura naquela iteração foi calculada, conforme demonstrado no quadro 7.

QUADRO 7 – CÁLCULO DA PARTÍCULA 3 NA ITERAÇÃO 9

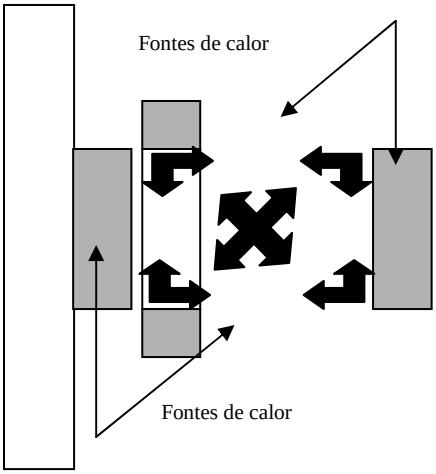
$$t_3^9 = \frac{t_2^8 + t_3^8 + t_4^8}{3} = \frac{70{,}49 + 46{,}63 + 20{,}50}{3} = 44{,}87$$

2.2.2 DIFUSÃO POR CONDUÇÃO BIDIMENSIONAL DE CALOR

Em um modelo bidimensional “a distribuição de temperaturas ocorre em duas direções” (Hein, 2000, p. 29), conforme demonstrado na fig. 3. Este modelo é adotado no presente trabalho, supondo uma chapa quadrada de material sólido e homogêneo.

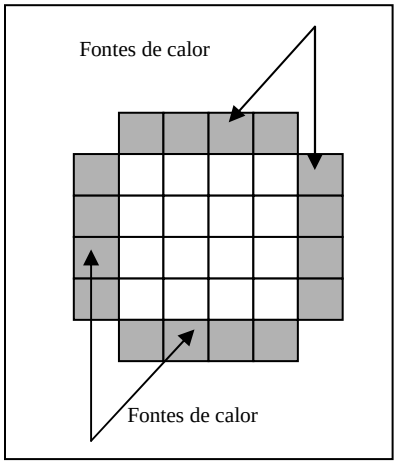
A distribuição de temperatura no estado estacionário bidimensional é dada pela equação de Laplace cujo resultado é nulo, pois se supõe que ao final do processo ocorra o equilíbrio ou alcance um valor onde as sucessivas interações não resultem em valores significativos, definidos como erro, que por hora define-se como tolerância de cálculo, sensibilidade ou precisão do cálculo.

FIGURA 3 - DIFUSÃO POR CONDUÇÃO BIDIMENSIONAL DE CALOR



Segundo Hein (2000, p. 30) “o modelo matemático de Laplace é construído fazendo-se uso das diferenças finitas”. A fig. 4 auxilia na compreensão desta construção, supondo uma placa bidimensional e transformando-a numa composição de pequenas placas, com a criação de uma malha, resultando em uma matriz.

FIGURA 4 – CONSTRUÇÃO DE LAPLACE EM UMA PLACA BIDIMENSIONAL



Como no esquema unidimensional, há um fluxo de calor, onde as partículas recebem e repassem calor paras as partículas imediatamente próximas a elas e em duas direções.

Ao modelar-se matematicamente o problema, conforme demonstrado no quadro 8, pode-se igualar os valores do Δx e Δy, ou seja, criando assim uma matriz quadrada. Ignorando-se a constante do coeficiente de dissipação e as perdas de calor, Hein (2000, p. 30) obtém-se a expressão da difusão bidimensional de calor, demonstrada no quadro 9. Esta

expressão da difusão bidimensional de calor apresenta significativa relevância para este trabalho, pois é através dela é que se configura o fluxo e a difusão de calor, aplicada no protótipo de difusão de calor.

QUADRO 8 – MODELO MATEMÁTICO DE LAPLACE

A expressão apresentada no quadro 9, aplicada a cada partícula de coordenadas (i,j) na iteração k, resulta em uma equação linear e o conjunto de todas elas forma um sistema linear. Conforme o crescimento do sistema, algo em torno de 5x5 partículas, o modelo torna-se complexo para ser solucionado algebricamente e então surge a necessidade do desenvolvimento de problemas de condutividade de calor solucionados através de algoritmos apresentados pela Álgebra Linear, como os métodos de Gauss e Gauss-Seidel. É necessária ainda a adoção de uma solução computacional, que agilize o processamento e apuração e, através dos sistemas modernos, demonstre visualmente a forma como o calor se dissipa na superfície de uma chapa.

QUADRO 9 – EXPRESSÃO DA DIFUSÃO BIDIMENSIONAL DE CALOR

$$T_{i,j}^{k} = \frac{T_{i+1,j}^{k-1} + T_{i-1,j}^{k-1} + T_{i,j-1}^{k-1} + T_{i,j+1}^{k-1}}{4}$$

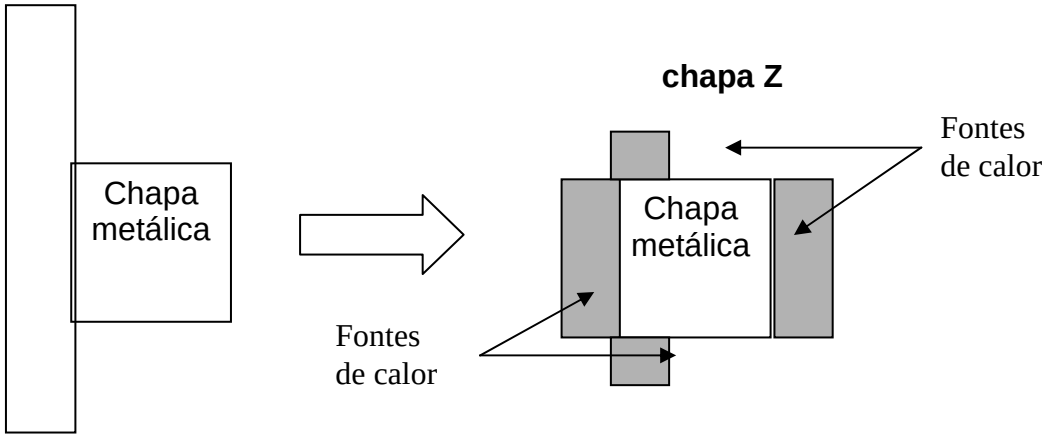
Fonte: Hein (2000, p. 26)

2.3 DISCRETIZAÇÃO DE UMA CHAPA

O presente trabalho apresenta uma solução para a dissipação de calor na superfície de uma chapa feita com um material homogêneo e de mesma difusibilidade térmica, como uma chapa de aço, por exemplo. Através da álgebra linear e de algoritmos de Gauss-Seidel, que consiste em realizar iterações sucessivas em busca do equilíbrio térmico, encontra-se a solução para o problema descrito. O ponto de partida é o entendimento de como transformar uma chapa em um modelo matemático e ainda como se deve proceder para dividir este modelo em modelos menores que representa o todo, sem que haja perda das características originais.

Considere-se uma chapa de comprimento e largura qualquer, sem levar em consideração a sua espessura. Seja ela formada por um material homogêneo que permita a difusibilidade térmica constante. Desconsidere-se ainda a dissipação de calor, ou seja, não há perdas. Supõe-se ainda a existência de quatro fontes de calor em suas extremidades, que mantém constantes suas temperaturas, conforme ilustrado na fig. 5.

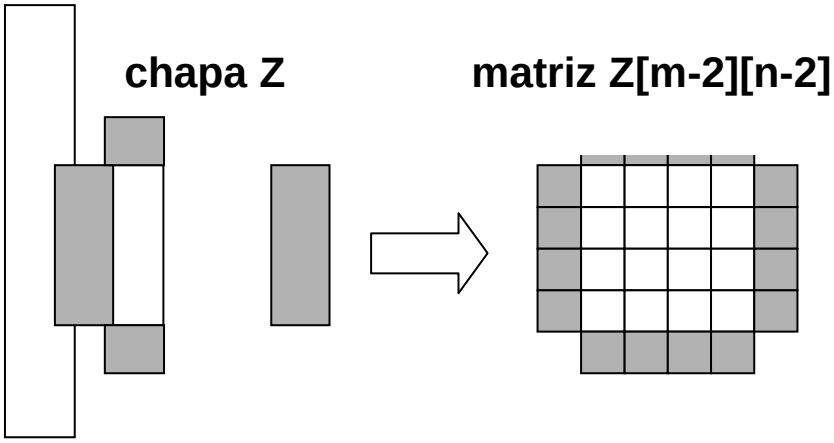
FIGURA 5 – CHAPA METÁLICA TRANSFORMADA EM CHAPA Z, AO INCLUIR AS FONTES DE CALOR NA SUA PERIFERIA.



Discretizando-se a chapa metálica em m linhas, por n colunas, temos um modelo que consiste em uma chapa Z com m-2 por n-2 partes, resultando na matriz Z[m-2][n-2], ou seja, é necessário incluir 2 linhas e 2 colunas contendo os valores das fontes de calor, conforme exemplificado na fig. 6. Esta matriz consiste em uma tabela, que contém as temperaturas, em cada espaço $x_{m,n}$ que pode ser reconhecido pelo nome de célula ou partícula de calor,

dispostas em m linhas e n colunas. Diz-se que essa matriz tem ordem m x n (lê-se: m por n), sendo m≥1 e n≥1 (Silva, 2000, p. 311).

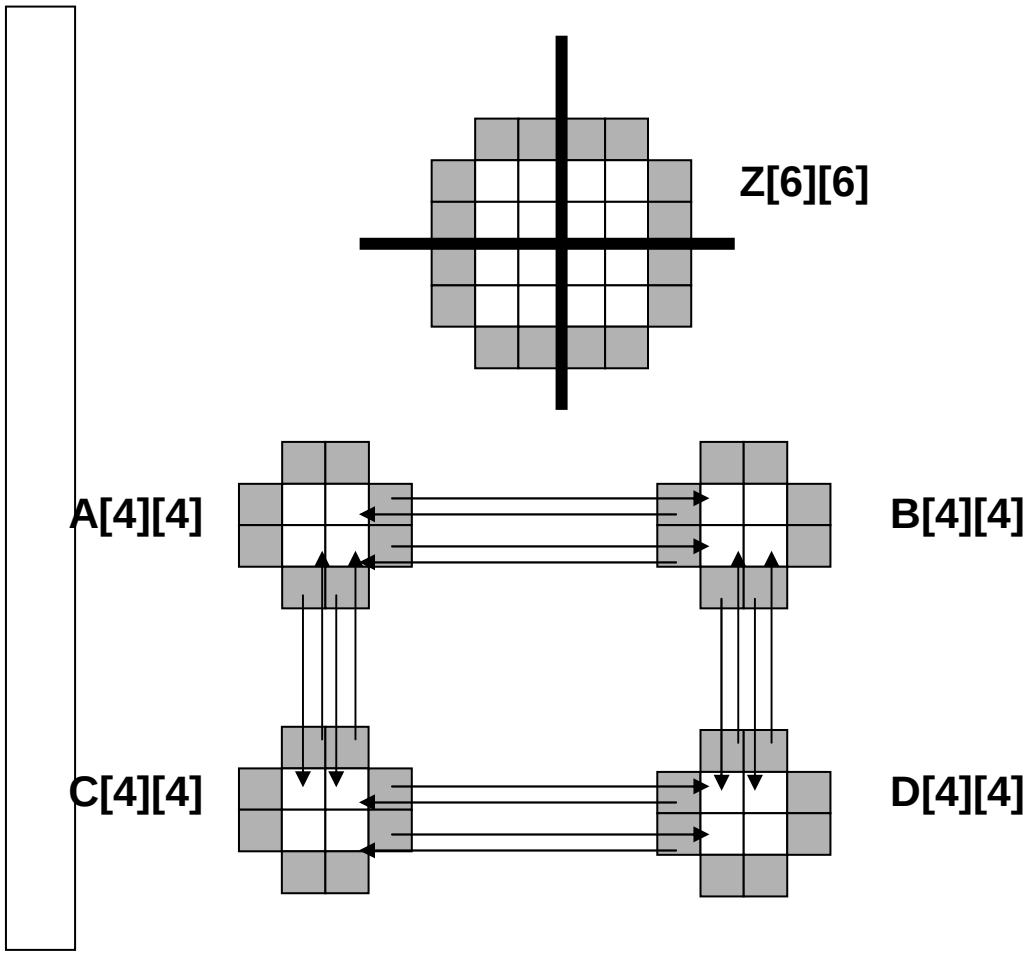
FIGURA 6 – CHAPA Z, DISCRETIZADA, RESULTA NA MATRIZ Z[m-2][n-2], QUE CORRESPONDE A CHAPA METÁLICA.



Se o objetivo do protótipo é dividir o cálculo da matriz em diversas unidades de cálculo, podendo ser vários processadores em um computador ou vários computadores diferentes distribuídos ao longo de uma rede de computadores, deve-se dividir a matriz Z[m+2][n+2] por i e j partes, obtendo as sub-matrizes de dimensões m/i+2 por n/j+2. Conforme apresentado na fig. 7, a matriz Z[m+2][n+2], foi dividida por i e j, onde i=2 e j=2, resultando nas sub-matrizes $A_{\left[\frac{m+2}{i},\frac{n+2}{j}\right]}, B_{\left[\frac{m+2}{i},\frac{n+2}{j}\right]}, C_{\left[\frac{m+2}{i},\frac{n+2}{j}\right]}$ e $D_{\left[\frac{m+2}{i},\frac{n+2}{j}\right]}$, que possuem as seguintes relações:

c) a coluna 3 das matrizes A e B, corresponde respectivamente à coluna 1 das matrizes C e D;
d) a coluna 0 das matrizes C e D, corresponde respectivamente à coluna 2 das matrizes A e B;
e) a coluna 3 das matrizes A e C, corresponde respectivamente à coluna 1 das matrizes B e D;
f) a coluna 0 das matrizes B e D, corresponde respectivamente à coluna 2 das matrizes A e C.

FIGURA 7 –MATRIZ Z[m-2][n-2], DIVIDIDA NAS SUB-MATRIZES
A, B, C e D, COM AS DEVIDAS CORRESPONDÊNCIAS DE CÉLULAS.



Através da análise da fig. 7, percebe-se a necessidade de cuidados ao realizar o particionamento de uma matriz de temperaturas, pois embora seja totalmente seguro realizar o cálculo faz-se necessário manter as relações entre matrizes, uma vez que as células adicionais, da periferia, que representam as fontes de calor, sofrem alterações nas matrizes adjacentes. Analisando, por exemplo, que a fonte de calor superior da matriz C são células de calor da matriz A, enquanto que a fonte de calor inferior da matriz A são células de calor na matriz C. Fontes de calor não sofrem alteração do valor na matriz onde estão localizadas, somente células de calor de uma matriz é que podem sofrer alteração, resultado da aplicação de um cálculo, durante uma iteração.

3 OBJETOS DISTRIBUÍDOS

Este capítulo tem por objetivo apresentar a fundamentação sobre sistemas distribuídos e serviços Java RMI, que representam a base tecnológica para a implementação do sistema de processamento paralelo distribuído.

3.1 SISTEMAS DISTRIBUÍDOS

A criação de redes de computadores teve um grande impulso a partir da redução de custos de equipamentos e a crescente necessidade de acesso a informações compartilhadas. A realidade deste novo milênio é de redes espalhadas por todos os lares, empresas, escolas e outros, partindo da concepção das facilidades surgidas com a massificação dos serviços oferecidos pela internet. A internet pode ser entendida como um sistema distribuído de grandes proporções, disperso pelo mundo, que oferece o acesso a uma variedade de serviços aos seus usuários. Projetistas para esta modalidade de sistemas enfrentam múltiplos desafios, em função de problemas de heterogeneidade, segurança e confiabilidade (Oliveira, 2002, p. 39).

Sistemas distribuídos adquiriram gradualmente grande relevância para a indústria da computação, em função da ampliação e difusão da internet. “As arquiteturas para o desenvolvimento de aplicações evoluíram para arquiteturas a objetos. Estas aplicações não são mais vistas simplesmente como blocos monolíticos de software, mas sim como coleções de componentes de softwares que cooperam para implementar a funcionalidade requerida da aplicação” (Júnior, 2001, p. 146). Programadores se adaptaram a trabalhar com eficácia e eficiência com as complexidades inerente de aplicações do mundo real. Programas passaram a ser coleções de componentes, construídos por diferentes autorias, que não precisam estar localizados em uma mesma máquina (mas talvez em outra, na rede), a linguagem de programação de cada componte não precisa mais ser a mesma e podem estar executando em diferentes sistemas operacionais com máquinas de arquiteturas diversas (PCs, estações de trabalho, Macintoches, etc.) (Júnior, 2001, p. 146).

“Pode-se definir um sistema distribuído como aquele no qual os componentes de hardware ou de software encontram-se em comunicação através de uma de rede de computadores e só coordene suas ações passagem de mensagens. Esta simples definição cobre

uma gama inteira de sistemas, às quais transmitem em rede e que possuam diferentes desdobramentos” (Coulouris, 2001, p. 2).

Segundo Coulouris (2001, p. 2), computadores que pertencem a uma rede não precisam estar no mesmo espaço geográfico. Eles podem estar em diferentes continentes, separados por salas ou prédios. Esta definição de sistemas distribuídos remete a reflexão das seguintes características e comportamentos destes sistemas:

g) concorrência: mais de um serviço pode ser feito paralelamente enquanto um trabalho é feito em um computador. “A capacidade de o sistema manipular recursos compartilhados pode aumentar com a adição de mais recursos (por exemplo, computadores) na rede” (Coulouris, 2001, p. 2).
h) nenhum relógio global: a colaboração entre objetos de diferentes computadores é feita através das trocas de mensagem. O sincronismo da troca de mensagens é feito através do compartilhamento do tempo em que a ação de uma aplicação realiza. Mas processar a sincronização dos tempos com precisão de troca de mensagens é inconsistente, pois não há noção correta do tempo.
i) falhas independentes: O projetista possui a responsabilidade de planejar as conseqüências de possíveis falhas, pois todo computador falha.

3.2 EXEMPLOS DE SISTEMAS DISTRIBUÍDOS

Estes exemplos estão baseados nas redes de computadores mais familiares e amplamente utilizados: Internet e intranet.

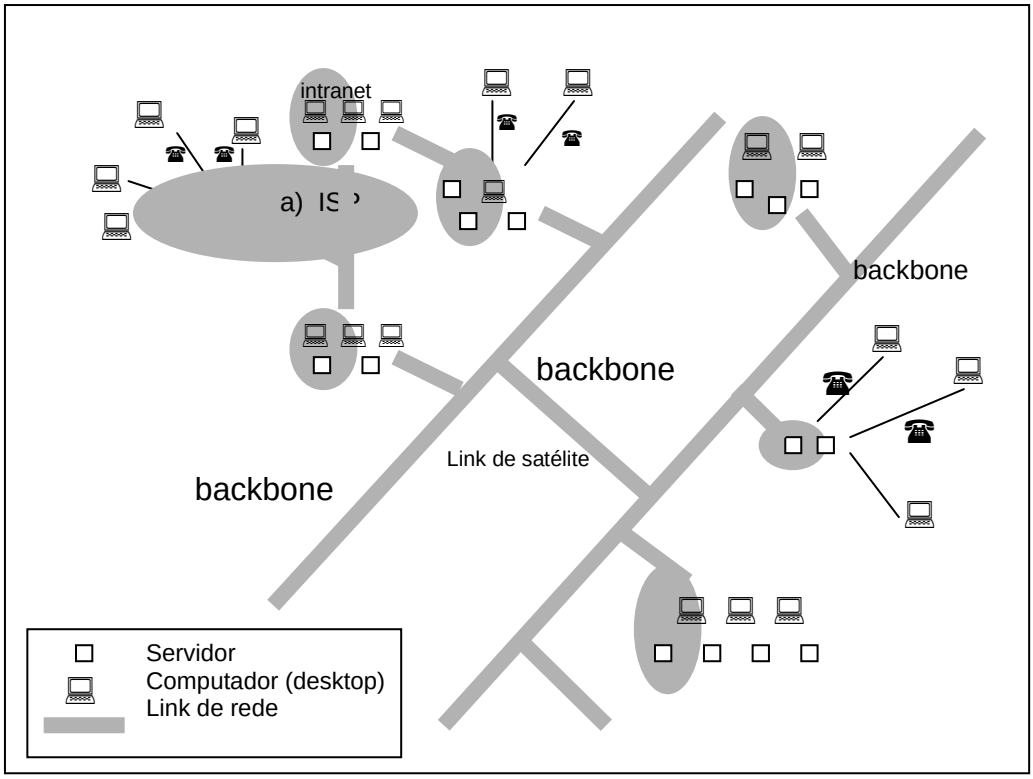
3.2.1 A INTERNET

“A internet é uma vasta coleção de diferentes tipos de redes de computadores interconectadas. A fig. 8 ilustra uma porção típica da internet. Programas executando em computadores conectados interagindo através da passagem de mensagens, empregando um meio de comunicação comum” (Coulouris, 2001, p. 3).

A internet possibilita que usuários façam uso de serviços como páginas web, email, transferência de arquivos, chat, videoconferências, caracterizando-se como um amplo sistema distribuído. A fig. 8 mostra uma coleção de intranets. Um ISP (*internet server provider*) é uma empresa responsável por prover acesso à internet a usuários empresariais ou domésticos.

As intranets são unidas à internet por *backbones*, que são ligações de rede de alta capacidade de transmissão, que empregam conexões por satélite, cabos de fibra óptica e outros circuitos de banda larga (Coulouris, 2001, p. 4).

FIGURA 8 – UMA TÍPICA PORÇÃO DA INTERNET

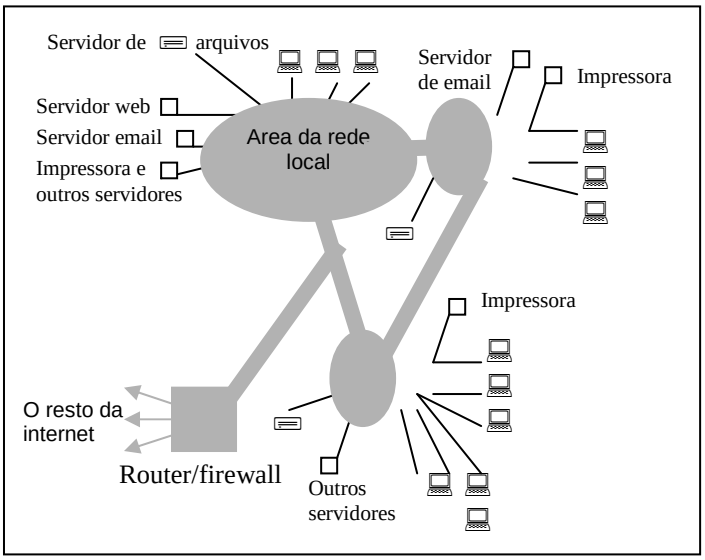


Fonte: Coulouris (2001, p. 3)

3.2.2 INTRANETS

Uma intranet é um conjunto de redes locais (LAN) ligadas à Internet, conforme ilustra a fig. 9, e que é administrada separadamente, possuindo configurações próprias, com políticas de segurança locais. Estas redes locais são ligadas à Internet por um roteador que a conecta com backbone, que por sua vez, é a infra-estrutura de telecomunicação que o liga à Internet (Coulouris, 2001, p. 3).

FIGURA 9 – UMA INTRANET TÍPICA



Fonte: Coulouris (2001, p. 3)

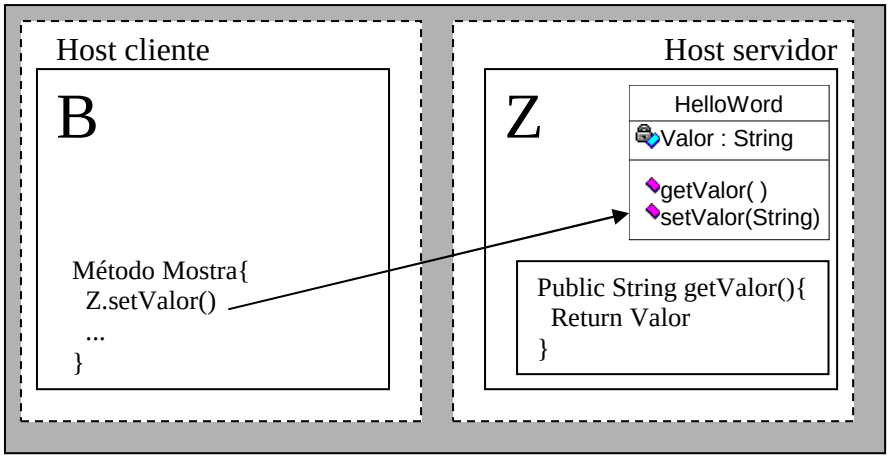
3.3 JAVA RMI (*REMOTE METHOD INVOCATION*)

"Java RMI extende o modelo de objeto Java para promover suporte a objetos distribuídos na linguagem Java. Permite invocar métodos de objetos remotos usando a mesma sintaxe da invocação local. Além disso, verificação de tipo aplica igualmente as invocações remotas sobre objetos locais. Porém, ao desenvolver um objeto com invocação remota deve-se estar atento que o objeto é remoto, porque é necessário controlar as exceções remotas (*RemoteExceptions*). Embora o modelo de objeto distribuído ser integrado com Java por um caminho natural, a semântica da passagem de parâmetro é diferente, porque invoca métodos localizados remotamente em outro objeto" (Coulouris, 2001, 194).

A tecnologia de Invocação de Métodos Remotos (RMI) tem origem na Chamada Remota de Procedimentos (RPC – *Remote Procedure Call*). Através da invocação remota de métodos, objetos Java disponíveis em um computador remoto ficam transparentes como se fossem objetos locais. A princípio qualquer linguagem orientada a objetos pode utilizar dos mecanismos do RMI. "No caso de Java, a grande vantagem está na homogeneidade. Por exemplo, como todas as máquinas virtuais Java representa os dados da mesma maneira, não existe a necessidade de conversão de dados. Além disso, o mecanismo de serialização de

objetos em Java permite que os objetos completos sejam passados como parâmetros de invocações remotas". O propósito do RMI é o mesmo do RPC: a programação de uma aplicação distribuída é facilitada na medida que abstrai a maioria dos detalhes relacionados com a comunicação das camadas de rede, em um ambiente distribuído. "Uma vantagem do RMI-Java sobre outras soluções, é a construção de interfaces usando a própria linguagem Java, sem a necessidade de uma linguagem separada para a descrição de interfaces" (Oliveira, 2002, p. 52).

FIGURA 10 – TRANSPARÊNCIA DE LOCALIZAÇÃO



Fonte: Hübner (2000, p. 2)

3.3.1 FUNCIONAMENTO DE JAVA RMI

Em Hübner (2000, p. 3) é apresentado um roteiro de como é o funcionamento de Java RMI, ilustrado pela Fig. 12:

a) o servidor de registro atribui um nome a um servidor, que oferece um determinado serviço;
b) o servidor de registro recebe do cliente um pedido de referência para um objeto servidor;
c) o cliente recebe do servidor de registro uma referência do objeto servidor (e o RMI carrega e cria o *stub* na máquina do cliente). Um *stub* possui as definições de como os clientes invocam métodos nos servidores, importando os métodos do objeto servidor através do *stub*. Para o cliente um *stub* funciona como uma

chamada local (*proxy*). Um servidor *proxy* contém as referências e os métodos de um servidor;

d) O cliente manda uma mensagem para o servidor (no caso, manda para o *stub* que serve de *proxy* para o servidor, mascarando seus métodos);

e) o *stub* abre uma conexão com o *skeleton* do servidor, e serializa o parâmetro nesta conexão (podem ser quaisquer objetos Java). Um *skeleton* tem por função exportar os métodos de um objeto.

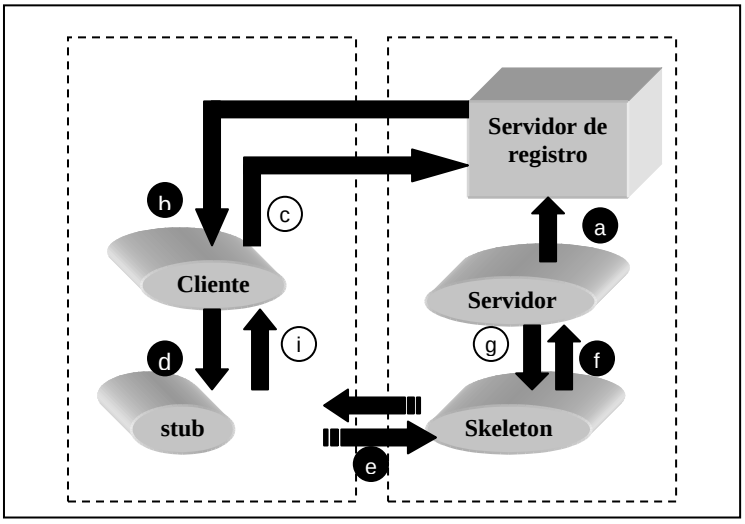
f) o *skeleton* recebe o pedido, desserializa os parâmetros e chama o método solicitado no servidor;

g) recebe a resposta do servidor e serializa a resposta no canal de comunicação;

h) o *stub* recebe a resposta e desserializa a resposta;

i) o *stub* retorna a resposta para o cliente.

FIGURA 11 – FUNCIONAMENTO DO JAVA RMI



Fonte: Hübner (2000, p. 3)

3.3.2 IMPLEMENTAÇÃO DE UM MODELO EM JAVA RMI

Em Rocha (2002, p. 6) é apresentado um roteiro em 11 passos de como implementar um aplicação RMI:

a) Definição da interface: para cada objeto que será acessado na rede são declarados todos os métodos que serão acessíveis remotamente em uma interface Java que estende java.rmi.Remote. Todos os métodos devem declarar *throws*[2] Java java.rmi.RemoteExeception. O código é exibido no quadro 9.

QUADRO 9 – CÓDIGO DE REMOTEHELLO.JAVA

```
import java.rmi.*;
public interface RemoteHello extends Remote{
  public String getMensagem()
                        throws RemoteException;
  public void setMensagem( String msg )
                        throws RemoteException; }
```

b) Implementar os objetos remotos: cada objeto remoto é uma classe que estende a classe Java.rmi.server.UnicastRemoteObject e que implementa a interface criada no passo 1. Todos os métodos declaram causar java.rmi.RemoteException, inclusive o construtor, mesmo que seja vazio. O código é exemplificado no quadro 10.

QUADRO 10 – CÓDIGO HELLOIMPL.JAVA

```
import java.rmi.server.*;
import java.rmi.*;

public class HelloImpl extends UnicastRemoteObject
                                              implements RemoteHello{
  private String mensagem = "Inicial";
  public HelloImpl() throws RemoteException{}
  public String getMensagem() throws RemoteException{ return ensagem; ) }
  public void setMensagem(String msg) throws RemoteException{
       mensagem = msg;   }   }
```

[2] "Uma cláusula throws lista as exceções que podem ser disparadas por um método. Os tipos de exceções que são disparadas por um método são especificadas na definição do método com uma cláusula throws" (Deitel, 2001, p. 664).

c) Estabelecer um servidor: criar uma classe que obtenha uma instância do objeto a ser servido. O código é exemplificado no quadro 11.

QUADRO 11 – CÓDIGO HELLOSERVER.JAVA

```
import java.rmi.*;
public class HelloServer {
  public static void main(String[] args)
              throws RemoteException {
    RemoteHello hello = new HelloImpl();
    try {
          Naming.rebind("//localhost/HelloServer", hello);
    }catch ( java.net.MalformedURLException e ){
          System.err.println("nome incorreto");
    }
       System.out.println("Servidor no ar. Nome do Objeto”+
                          “servidor:\mensagens\”);
       System.out.println("Hello World!");
    }
      } /* Este cliente foi simplificado
         * (não tem SecurityManager),
         * facilitando a deonstracao */
```

d) Compilar os objetos remotos: compilar todas as interfaces e classes utilizadas para implementar as interfaces *Remote*. A linha de comando é mostrada no quadro 12.

QUADRO 12 – COMPILAÇÃO DAS INTERFACES E CLASSES

```
javac HelloRemote.Java HelloImpl.java
```

e) Gerar *stubs* e *skeletons*: usar rmic para gerar um arquivo HelloImpl_stub.class e um arquivo skeleton HelloImpl_Skel.class. A linha de comando é mostrada no quadro 13.

QUADRO 13 – GERAÇÃO DOS STUBS E SKELETONS

```
rmic HelloImpl
```

f) Compilar e instalar o(s) cliente(s): escrever uma classe cliente que localize o(s) objeto(s), obtenha uma instância remota para cada objeto e chame os objetos remotos de cada objeto. O código é exemplificado no quadro 14.

QUADRO 14 – CÓDIGO DE HELLOCLIENT.JAVA

```
import java.rmi.*;
public class HelloClient {
       public static void main(String args[])
                 throws Exception {
              //String hostname = args[0];
              //String objeto = args[0];
              Object obj =
                     Naming.lookup("//localhost/HelloServer");
              RemoteHello hello = (RemoteHello) obj;
              System.out.println("Mensagem recebida: "
                                  + hello.getMensagem() );
              hello.setMensagem("Julio esteve aqui!");
       }
}
```

g) Instalar o *stub* no(s) cliente(s): compilar e distribuir o cliente para as máquinas cliente. A(s) classe(s) que implementa(m) o cliente, os *stubs* e as interfaces *Remote*. Geralmente os *stubs* são mantidos no servidor. O cliente pode fazer *download* do *stub* e começar a usá-lo. É preciso definir propriedades adicionais (omitidas neste exemplo simples). Aplicações RMI típicas usam *codebase*: “CLASSPATH distribuído”, security manager e arquivos com políticas de segurança. O conteúdo do arquivo POLICE é mostrado no quadro 15 e descreve as políticas de segurança do exemplo descrito nesta seção. Observa-se que os usuários deste objeto possuem permissões para todos os métodos deste objeto.

QUADRO 15 – CONTEÚDO DO ARQUIVO POLICE

```
grant {
 java.security.AllPermission;
};
```

h) Iniciar o RMI *registry* no servidor: no windows `start rmiregistry` ou no Unix `rmiregistry&`. Neste exemplo é necessário iniciar o RMIRegistry no diretório onde estão os stubs e a interface Remote. Desta forma o RMI Registry pode usar o mesmo CLASSPATH que o resto da aplicação.

i) Iniciar o servidor de objetos: O servidor é uma aplicação executável que registra os objetos. A linha de comando para executar o servidor é mostrada no quadro 16.

QUADRO 16 – INICIANDO UM SERVIDOR

```
c:\> Java HelloServer
  Servidor no ar. Nome do objeto servido: mensagens
```

j) É preciso definir a propriedade Java.rmi.server.codebase contendo os caminhos onde se pode localizar o código. Para facilitar a instalação, recomenda-se a criação de um arquivo em lote (batch - .bat), que contenha os comandos para instalação do servidor, conforme mostrado no quadro 17.

QUADRO 17 – SERVIDOR.BAT

```
start rmiregistry
java -Djava.rmi.server.codebase=file:/hello/ -
Djava.security.policy=\hello\policy HelloServer
```

k) Iniciar os clientes: O quadro 18 mostra a linha de comando que inicializa um cliente.

QUADRO 18 – INICIANDO UM CLIENTE

```
c:\> Java HelloClient
```

De forma conclusiva, Java RMI permite que objetos Java sejam executados em computadores separados, ou no mesmo computador, comunicando-se via chamadas de métodos remotos. Java RMI trata de todos os detalhes de rede, suportando a comunicação entre clientes e servidores e a transferência de argumentos entre objetos. Em um ambiente distribuído, o cliente utiliza o *host* e o número da porta[3] (porta padrão é 1099) para localizar o *registry* do servidor. Basicamente para que o objeto servidor seja localizado pelo cliente, em cada máquina que for hospedado este objeto servidor e o *registry* precisam ser inicializados e estarem rodando no momento da conexão e invocação dos métodos.

[3] “Portas neste caso não são portas físicas de hardware, são números inteiros que permitem aos clientes solicitar diferentes serviços em uma máquina remota (Deitel, 2001, p. 860)”.

4 MULTIPROGRAMAÇÃO E BALANCEAMENTO DE CARGA

Buscando melhorar o desempenho do cálculo das iterações das matrizes, propõe-se que o protótipo distribuído seja composto de várias unidades de cálculo que realizam o processamento das matrizes paralelamente. Neste capítulo apresentam-se conceitos fundamentais sobre multiprogramação e sobre balanceamento de carga.

4.1 MULTIPROGRAMAÇÃO

"O objetivo da multiprogramação é ter sempre algum processo em execução para maximizar a utilização da CPU. Para um sistema uniprocessador, nunca haverá mais de um processo em execução para minimizar, se houver mais processos, o restante terá de esperar até que a CPU esteja livre e possa ser reescalonada". (Tanenbaum, 2000, p. 29)

A idéia de multiprogramação envolve o conceito de um processo que é executado até ter de esperar geralmente pela conclusão de um pedido de entrada/saída (I/O). Em um sistema de computação simples, a CPU ficaria ociosa. Todo esse tempo de espera é perdido; nenhum trabalho útil é realizado. Com a multiprogramação, esse tempo é usado de forma produtiva. Vários processos são mantidos na memória no mesmo tempo. Quando um processo precisa esperar, o sistema operacional tira a CPU do processo e a passa para outro processo. Esse padrão continua: toda vez que um processo precisa esperar, outro pode presumir o uso da CPU (Tanenbaum, 2001, p. 95). Aplicando a noção de multiprocessamento ao protótipo de dissipação de calor, supõe-se vários servidores de cálculo sendo executados ao mesmo tempo, de forma remota. A aplicação servidora através de um processo envia dados a um servidor, para que seja calculado. Enquanto aguarda que o cálculo seja concluído, o processo adormece. Um outro processo pode assumir o controle do processador, para enviar os seus dados e depois adormecer. Quando as unidades servidoras concluírem o cálculo, enviam a resposta para o cliente, acordando o processo que o desencadeou, sucessivamente a cada iteração, até a conclusão do processamento.

4.1.1 *THREADS* EM JAVA

Em aplicações tradicionais há apenas um processo sendo executado de cada vez, utilizando uma única linha de controle e um único contador de programa para cada aplicação.

Mas a evolução dos sistemas operacionais modernos implementou a capacidade de executar vários processos ao mesmo tempo, mantendo várias linhas de controle e vários contadores de programas em um mesmo equipamento. Estes processos que ocorrem paralelamente são chamados de *threads* ou, processos leves (Tanenbaum, 2000, p. 70).

O protótipo de simulação de calor distribuído adota o uso de *threads*, visto que o balanceamento de carga realizado pelo cliente e as atividades de cálculo em uma iteração realizadas pelos servidores, apesar de sincronizadas, podem acontecer paralelamente.

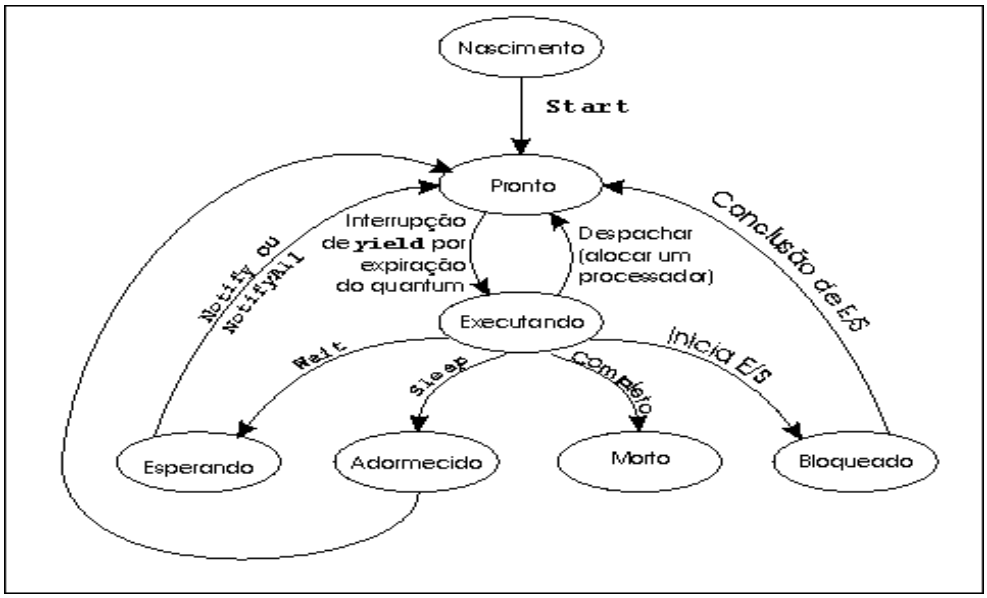
4.1.1.1 ESTADOS DE *THREAD*: CICLO DE VIDA DE UM THREAD

"Diz-se que um *thread*, a qualquer momento, está em um de vários estados de *thread* (ilustrados na fig. 14). Um *thread* que acaba de ser criado está no estado de nascimento. O *thread* permanece neste estado até o método *start* do *thread* ser chamado; isso faz com que o *thread* passe para o estado pronto (também conhecido como estado executável). O *thread* pronto de prioridade mais alta entra no estado executando quando o sistema aloca um processador para o *thread* (isto é, o *thread* inicia a execução). Um *thread* entre no estado morto quando um seu método *run* completa ou termina por alguma razão – um *thread* morto acabará sendo descartado pelo sistema mais cedo ou mais tarde" (Deitel, 2001, p. 684).

Segundo Deitel (2001, p. 684), "uma maneira comum de um *thread* executando entrar no estado bloqueado é quando o *thread* emite uma solicitação de entrada/saída (E/S). Neste caso, um *thread* bloqueado torna-se disponível (ou em estado pronto) quando a E/S pela qual está esperando é concluída. Um *thread* bloqueado não pode utilizar um processador, mesmo se algum estiver disponível".

"Quando um método *sleep* é chamado em um *thread* executando, esse *thread* entra no estado adormecido. Um *thread* adormecido torna-se disponível depois que o tempo designado para dormir expire. Um *thread* adormecido não pode utilizar um processador mesmo se algum estiver disponível" (Deitel, 2001, p. 684). A fig. 14 ilustra o ciclo de vida de um thread.

FIGURA 12 – CICLO DE VIDA DE UM *THREAD*



Fonte: Deitel (2001, p. 684)

Quando um thread executando chama *wait*, ele entra em estado de espera pelo objeto particular para o qual *wait* foi chamado. Um thread no estado de espera por um objeto particular torna-se pronto quando uma chamada *notify* é emitida por outro *thread* associado com aquele objeto. Todos os *threads* em estado de espera por um objeto dado tornam-se prontos quando uma chamada para *notifyAll* é feita por outro *thread* associado com aquele objeto. Um *thread* entra no estado morto quando seu método *run* completa ou dispara uma exceção não interceptada (Deitel, 2001, p. 684).

4.1.1.2 PRODUTORES E CONSUMIDORES

O protótipo paralelo e distribuído implementado no presente trabalho apresenta como requisito o balanceamento de carga. Há um processo do tipo balanceamento de carga e vários processos do tipo calcular, que se comunicam com servidores de cálculo. O processo do tipo balanceamento de carga distribui uma porção de células de calor para cada processo do tipo calcular. O processo calcular envia sua porção de dados ao objeto remoto e adormece, aguardando a resposta. Quando cada o processo do tipo calcular recebe o resultado do cálculo adormece; e quando o último processo do tipo calcular está prestes a adormecer, desperta o processo do tipo balanceamento de carga. Este sistema gerente/cliente(s) é implementado seguindo o modelo do problema dos produtores e consumidores. “Dois processos

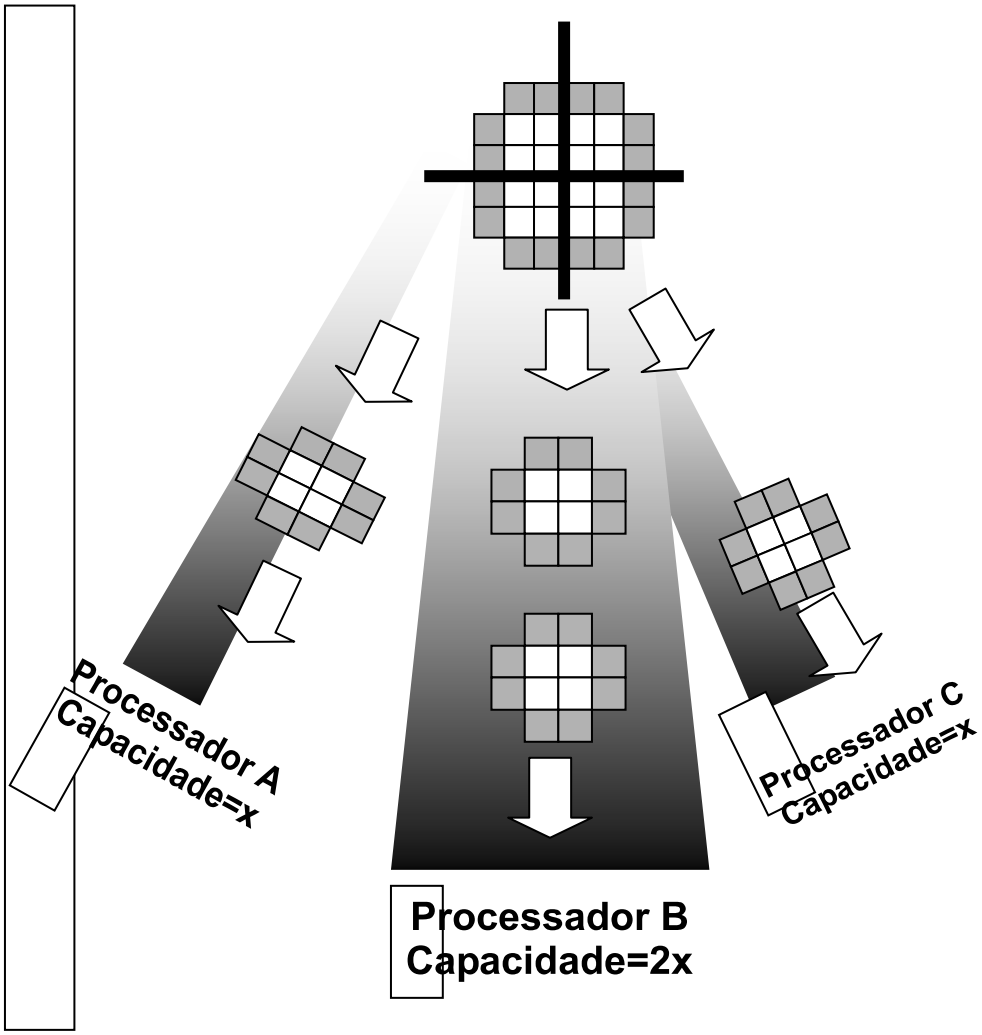
compartilham um buffer ou área comum. Um deles (o produtor) coloca as informações no buffer e o outro (consumidor) retira" (Tanenbaum, 2000, p. 61). O problema caracteriza que o consumidor precisa esperar o fruto da atividade do produtor estar disponível, para iniciar suas atividades. E o produtor precisa parar de produzir quando o buffer estiver cheio. No caso da aplicação distribuída de dissipação do calor, o gerente é o produtor, realizando a tarefa de escalonamento e balanceamento de carga para os consumidores, que são os processos que se comunicam com os servidores remotos. Não se trata de problema de área crítica de memória, pois nenhum dos processos, inclusive o gerente, acessam uma mesma área de memória ao mesmo tempo. Os processos de cálculo aguardam, porque o módulo de gerenciamento deve fazer o balanceamento de carga. Enquanto o gerente promove o balanceamento de carga, os processos de comunicação com os servidores de cálculo adormecem. O controle do problema dos produtores e consumidores nesta implementação trabalha com um monitor, garantindo que os dois diferentes tipos de processos ocorram de forma sincronizada. "O monitor permite que seja executado um processo de cada vez, bloqueando um tipo de processo quando o outro está sendo executado" (Deitel, 2001, p. 688).

4.2 BALANCEAMENTO DE CARGA

Distribuição da carga de trabalho de um programa por um conjunto de processadores assegura que todos os processadores tenham cargas de trabalho aproximadamente iguais. Dependendo da natureza da aplicação, a estratégia de distribuição pode ser estática simples (como apresentado na imagem da fig. 13), ou pode precisar de balanceamento dinâmico. Este modelo surge por causa de sistemas com multiprocessadores (Hull, 1994, P. 149).

"Um sistema de balanceamento de carga analisa a carga sobre todos os recursos disponíveis para determinar qual a opção de alocação destes recursos" (Balen, 2000, p. 232).

FIGURA 13 – DISTRIBUIÇÃO DE CARGA ESTÁTICA SIMPLES



4.2.1 EQUAÇÃO DE BALANCEAMENTO DE CARGA

A obtenção do equilíbrio de carga para cada unidade servidora consiste basicamente no controle do número de pacotes enviados e o tempo decorrido entre o envio, cálculo e a devolução dos pacotes. Através da equação do quadro 19 mostra-se a expressão do cálculo do rendimento de cada servidor de cálculo.

QUADRO 19 – EXPRESSÃO DO CÁLCULO DO RENDIMENTO

$R^n = \left(\frac{t}{t'}\right).p$	onde, R^n: Rendimento do processo n t: tempo total de todos os processos t´: tempo do processo n p: número de pacotes do processo n

Após obter o rendimento de cada processo, é necessário calcular o índice de escalabilidade dos processos. Este índice determina a quantidade de pacotes que são escalados para o processo de cálculo na próxima iteração. O quadro 20, apresenta a expressão do índice de escalabilidade dos processos. Uma vez obtido este índice, deve-se multiplicá-lo pela quantidade total de pacotes que aguardam o processamento.

QUADRO 20 – EXPRESSÃO DO CÁLCULO DO ÍNDICE DE ESCALABILIDADE

$E^n = \frac{R^n}{\sum R}$	E^n = Índice de escalibilidade do processo n R^n = Rendimento do processo n $\sum$R = Soma do rendimento de todos os processos

4.2.1.1 ERROS DE ARREDONDAMENTO

Como no presente trabalho, deve-se escalonar sub-matrizes, entende-se que o número de matrizes escalonadas para cada servidor deve ser um valor inteiro. O arredondamento explícito dos valores pode causar um erro, em alguns casos de mais uma matriz e em outros casos menos uma matriz, ao ser totalizado o número de matrizes como demonstrado no ANEXO A. É fundamental promover a correção deste desvio. Na implementação do protótipo adotou-se o critério de que se o erro de arredondamento for igual a +1, subtrai-se do servidor com maior carga uma matriz e se o erro de arredondamento for de –1, acrescenta-se uma matriz ao servidor com maior carga.

A tabela 2 apresenta uma situação hipotética, com números e tempos de processos definidos aleatóiamento para efeito de cálculo, na coluna rendimento aplica-se a expressão definida no quadro 19 e na coluna índice de escalabilidade aplica-se a expressão definida no uqadro 20. Supõe-se um cenário hipotético, representa-se um sistema com 5 servidores, utilizando valores de simulação matemática aleatórios, que processaram hipoteticamente o

total de 45 processos, conforme totalizado na coluna número de processos, a qual discrimina quanto cada servidor processou. A coluna tempo de processos demonstra o tempo demandado em cada processo, totalizado ao final da coluna.

TABELA 2 – MODELO DE BALANCEAMENTO DE CARGA

Servidor	Número de Processos	Tempo Processos	Rendimento	Índice de escalabilidade
1	13	20	195	0,5361
2	11	40	82,5	0,2268
3	9	60	45	0,1237
4	7	80	26,25	0,0722
5	5	100	15	0,0412
total	45	300	363,75	1

As informações hipotéticas, demonstradas na tabela 2, servem como base para balancear uma nova iteração. Na tabela 3 propõe-se quatro situações diferentes, supondo-se que fossem diferentes valores de iterações, prevendo respectivamente carga de 100, 99, 96, e 95 novos pacotes a serem escalonados. Nas colunas com carga de 100 e 96 pacotes não há qualquer tipo de alteração, pois não houve erros de arredondamento. Na coluna com carga de 99 pacotes houve um erro de arredondamento, pois após aplicação do índice de escalonamento para todos os servidores e totalização do número de pacotes há um a menos, conforme demonstra a coluna onde os valores NÃO foram ajustados. Como solução adotada neste trabalho aumentou-se um pacote no servidor com maior carga, pois entende-se que o mesmo possui o maior desempenho, conforme indica coluna onde os valores ajustados são iguais a SIM. Já na coluna com carga de 95 pacotes, após aplicação dos índices de escalonamento, arredondamento e totalizando a coluna onde os valores NÃO foram ajustados, há um pacote a menos e adotando o critério definido no presente trabalho diminui-se um pacote do servidor com maior desempenho, conforme indica a coluna onde os valores ajustados são iguais a SIM.

TABELA 3 – ESCALONAMENTO COM E SEM AJUSTE

	Próxima carga							
Carga(Pacotes)	100		**99**		96		**95**	
Ajustados	NÃO	SIM	NÃO	SIM	NÃO	SIM	NÃO	SIM
Servidor								
1	54	54	53	54	51	51	51	50
2	23	23	22	22	22	22	22	22
3	12	12	12	12	12	12	12	12
4	7	7	7	7	7	7	7	7
5	4	4	4	4	4	4	4	4
Total	100	100	**98**	99	96	96	**96**	95

Para concluir esta etapa apresenta-se a prova real, que é a estimativa de tempo que cada servidor levaria para processar os pacotes enviados por cada cliente. Observa-se que os tempos estimados para realização do cálculo não são iguais, mas representa uma eficiência de 0,89% entre a diferença entre o maior e o menor tempo de processamento dos pacotes, conforme pode-se visualizar na tabela 4.

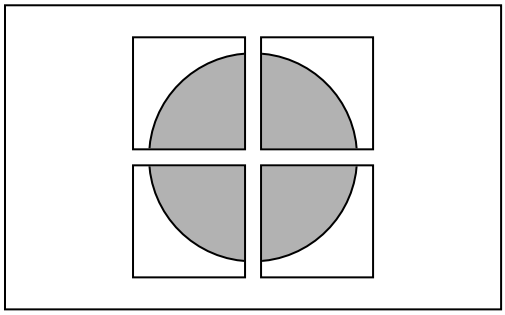
TABELA 4 – PROVA REAL E EFICIÊNCIA DO PROCESSO DE BALANCEAMENTO

Servidores	Tempo por processo	Número de processos	Tempo total	Diferença	%
1	1,5385	54	83,08	3,08	0,76%
2	3,6364	23	83,64	3,64	0,89%
3	6,6667	12	80,00	0,00	0,00%
4	11,4286	7	80,00	0,00	0,00%
5	20,0000	4	80,00	0,00	0,00%
			406,71		

4.2.2 DISTRIBUIÇÃO ESTÁTICA DE DADOS

No modelo de balanceamento de carga com distribuição estática de dados, os dados são particionados por um número apropriado de segmentos e distribuído por segmentos de processamento. Na fig. 14 é ilustrado que diferentes processadores atuam em apenas um segmento diferente da imagem simultaneamente.

FIGURA 14 – DISTRIBUIÇÃO ESTÁTICA DE DADOS



O modelo estático de balanceamento de carga, sub-divide a matriz em sub-matrizes menores. Uma porção de células de calor é um conjunto de sub-matrizes. O tamanho de cada porção é proporcional à capacidade de processamento de cada unidade servidora. O módulo gestor deve, portanto, solicitar alocação de recursos antes do processamento numérico do problema.

O modelo dinâmico, ao contrário, re-aloca recursos em tempo de execução (Hull, 1994).

As características do tipo de dado e a forma como acontece o processamento desses dados, no presente trabalho, indica a utilização de distribuição pelo modelo estático como base para a arquitetura a ser implementada.

No modelo adotado para o protótipo de difusão de calor distribuído o balanceamento de carga é estático, porque as matrizes de cálculo possuem um tamanho fixo. A variação de carga se dá através da quantidade de sub-matrizes que serão enviadas para cada computador remoto. Os servidores de cálculo que possuem maior poder de processamento, ou seja, a combinação de fatores que incluem memória, velocidade do processador, tráfego de rede, outros processos realizados paralelamente, recebem um número maior de pacotes, enquanto que os de menor poder de processamento recebem um menor número de pacotes. A avaliação do poder de processamento de um servidor é feita através da tomada de tempo antes de enviar os pacotes a serem processados e após de receber o resultado do seu processamento.

5 DESENVOVIMENTO DO PROTÓTIPO

Neste capítulo é apresentado o desenvolvimento dos protótipos *Stand-Alone* e distribuído, utilizando a *Unified Modeling Language* (UML), com a ferramenta CASE Power Designer para modelagem e a implementação e codificação utilizando a linguagem de programação Java, utilizando a ferramenta de desenvolvimento *Forte for Java Comunnity Edition* da *Sun Microsystems*.

5.1 PROTÓTIPO PARA SIMULAÇÃO DA DIFUSÃO DO CALOR *STAND-ALONE*

A concepção das funcionalidades do protótipo *stand-alone* partiu do protótipo, desenvolvido pelo autor deste trabalho, utilizando o modelo funcional, desenvolvido no Delphi 5.0.

Basicamente, o protótipo é formado por três etapas principais:

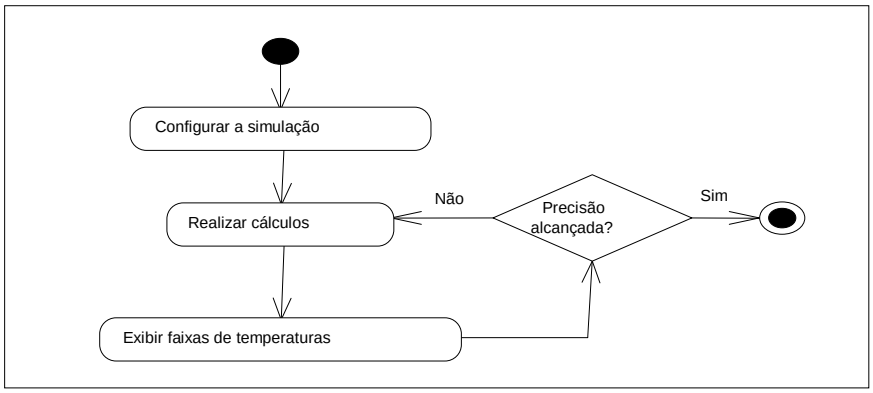
a) configuração das informações necessárias para que o protótipo realize a difusão do calor. Nesta etapa um usuário informa a temperatura ambiente, as temperaturas das fontes de calor: superior, inferior, esquerda e direita e a precisão do cálculo;

b) após a realização da configuração o usuário determina o início do processamento, que consiste na aplicação dos cálculos sobre cada partícula de calor da chapa. A quantidade de células de calor implementadas neste protótipo foi fixada em 10.000, divididas em 100 matrizes contendo 100 células de calor cada;

c) ao findar cada iteração, ou seja, ao calcular a dissipação de calor nas 10.000 células de calor, são apresentadas ao usuário, através de uma interface gráfica, as isofaixas de temperatura onde a temperatura em cada célula de calor é representada através de cores. Neste modelo foram fixadas dez isofaixas de temperaturas, representadas pelo mesmo número de cores, partindo da menor até a maior temperatura informada durante a etapa de configuração da simulação, descrito no item (a) desta seção. A cada nova iteração é mostrada ao usuário a precisão alcançada naquele instante. A imagem apresentada ao usuário sofre mudanças ao longo do tempo de processamento, pois a cada iteração as

temperaturas estão sujeitas a mudanças de valores, até que a precisão determinada pelo usuário seja alcançada. O diagrama de atividade apresentado na fig. 15 demonstra a ação dos processos.

FIGURA 15 – DIAGRAMA DE ATIVIDADE DO PROTÓTIPO STAND-ALONE



5.1.1 ESPECIFICAÇÃO DO PROTÓTIPO *STAND-ALONE*

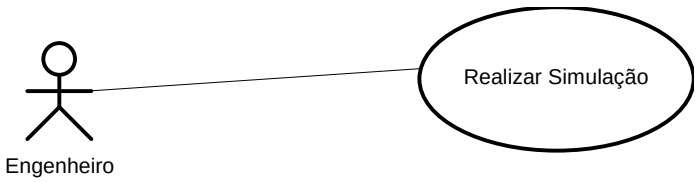
Como este protótipo partiu das funcionalidades de um protótipo que adotava um modelo funcional e o novo protótipo é concebido a partir do modelo orientado a objetos, foi criada uma nova modelagem, adequada as novas necessidades.

Partindo do conceito de que “a Linguagem Unificada de Modelagem (UML) é uma linguagem padrão para especificar, visualizar, documentar e construir artefatos e não uma metodologia” (Furlan, 1998, p.33), utiliza-se os diagramas de casos de uso, classes e seqüência para especificar a metodologia de modelagem orientada a objetos, através da ferramenta case Power Designer (Sybase, 2001).

5.1.1.1 CASOS DE USO DO PROTÓTIPO *STAND-ALONE*

Apresenta-se na fig. 16 o caso de uso especificados para o protótipo *stand-alone*.

FIGURA 16 – DIAGRAMA DE CASOS DE USO



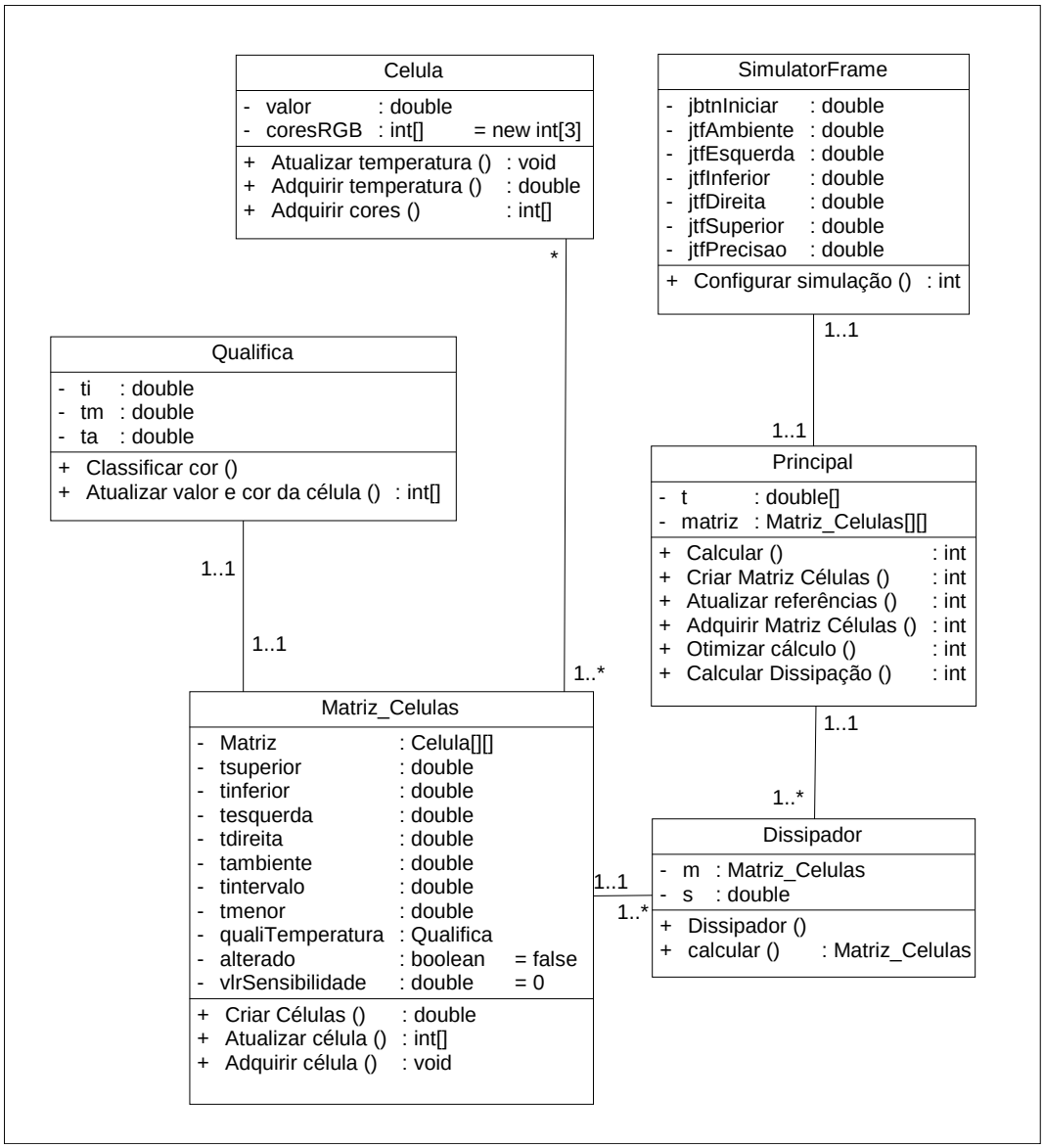
Este caso de uso realiza basicamente três processos relacionados e dependentes, conforme relacionado:

a) configurar simulação: um engenheiro, responsável pela entrada dos dados, através de uma interface do usuário informa a temperatura ambiente, as temperaturas das fontes de calor: superior, inferior, esquerda e direita e a precisão do cálculo. Os valores informados para as temperaturas devem apresentar consistência, informando uma mesma escala de temperatura;
b) realizar cálculos: o sistema é composto por uma chapa quadrada cercada em suas periferias por quatro fontes de calor (superior, inferior, esquerda, direita), informados no item anterior desta seção. Havendo uma matriz Z, dividida em 10 linhas e 10 colunas, totalizando 100 matrizes Y. As matrizes Y estão divididas em 12 linhas e 12 colunas, onde as 44 células da parte periféria são fontes de calor, e as demais 100 células centrais representam as partículas ou células de calor, resultado da discretização da chapa. Uma iteração consiste em varrer toda a matriz aplicando a expressão definida no quadro 9. Uma forma de otimização do processo é garantir durante a implementação do protótipo que se todos os valores de uma matriz de células forem iguais a matriz não seja calculada. O caso de uso “realizar cálculos” é intercalado com o caso de uso descrito no item “c” desta seção e encerra quando a precisão determinada pelo usuário for alcançada;
c) exibir graficamente os resultados: Todas as iterações são exibidas ao usuário através de uma interface gráfica, demonstrando as isofaixas de temperaturas através de ocres. São definidas em dez as faixas de cores, resultantes da divisão do intervalo entre a menor e a maior temperatura dividida por 10. Este caso de uso acontece de forma intercalada ao processo, descrito no caso de uso do item “b”, desta mesma seção até o fim do processamento, permitindo que um engenheiro possa estudar o comportamento do fluxo e da difusão do calor ao longo do processo.

5.1.1.2 DIAGRAMA DE CLASSES DO PROTÓTIPO *STAND-ALONE*

A fig. 17 demonstra o diagrama de classes utilizados neste protótipo e suas relações.

FIGURA 17 – DIAGRAMA DE CLASSES



São identificadas seis principais classes no protótipo, abstraindo as classes derivadas da Interface de Programação de Aplicações Java (API Java – *Applications Programming Interface* – Java):

a) SimulatorFrame: implementa a interface onde o usuário configura a simulação, determinando os parâmetros de temperatura e precisão;

b) Principal: nesta classe é implementada a otimização dos cálculos e a apresentação ao usuário, através da interface gráfica da evolução da dissipação de calor.

c) Dissipador: é a classe que realiza o cálculo da dissipação do calor, através da aplicação da expressão definida no quadro 9.

d) Matriz_Células: através desta classe é implementa a estrutura de dados composta por uma matriz de partículas ou células de calor. Seus métodos fornecem acesso aos métodos da classe célula.
e) Célula: esta é uma classe que encapsula a estrutura de dados de uma partícula ou célula de calor. Possui a temperatura e um vetor com as cores RGB desta temperatura.
f) Qualifica: esta classes está associada à classe Matriz_Células e determina o vetor cor RGB de uma célula, gerado a partir da sua temperatura.

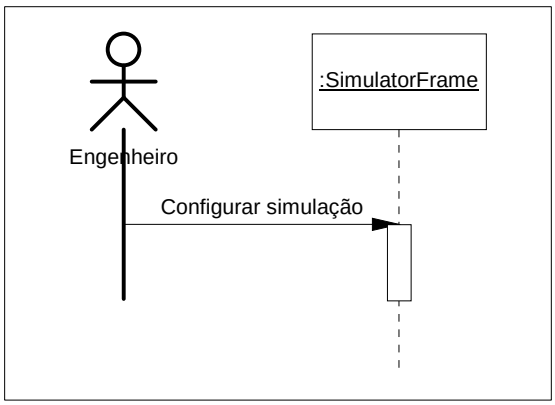
5.1.1.3 DIAGRAMAS DE SEQÜÊNCIA DO PROTÓTIPO *STAND-ALONE*

Estes diagramas representam a seqüência em que as ações ocorrem dentro do sistema, através da troca de mensagens entre os objetos das classes. Cada caso de uso resultou em um diagrama de seqüência, conforme apresentado nas fig. 18, 19 E 20.

5.1.1.3.1 CONFIGURAR SIMULAÇÃO

A fig. 18 demostra através do diagrama de seqüência configurar simulação. O ator do tipo Técnico informa os dados da configuração da Simulação, ativando a classe SimulatorFrame.

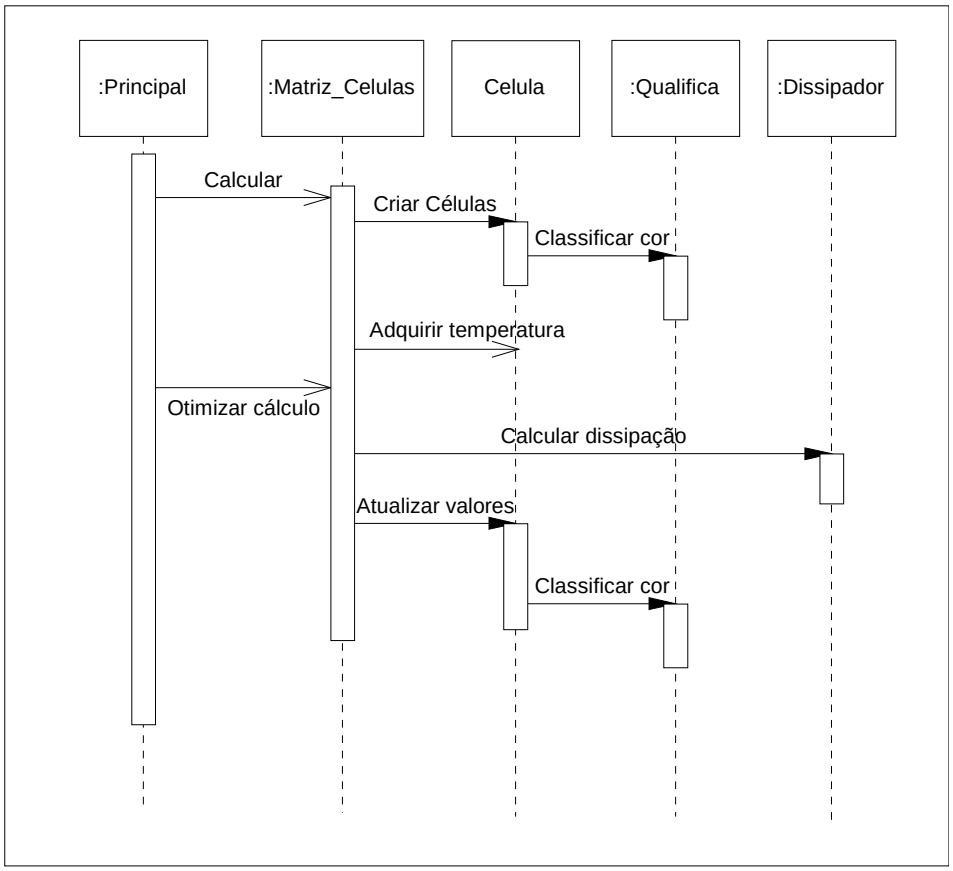
FIGURA 18 – DIAGRAMA DE SEQUÊNCIA CONFIGURAR SIMULAÇÃO



5.1.1.3.2 REALIZAR CÁLCULO

Um objeto da classe Principal cria uma matriz com 10 linhas e 10 colunas de instâncias de objetos da classe Matriz_Células. Cada Matriz_Célula cria uma matriz de 12 linhas e 12 colunas de objetos da classe Célula, passando como parâmetro a temperatura e a cor RGB da célula, adquirida através dos métodos da instância do objeto Qualifica. Após criar todas as instâncias de Célula o objeto da classe principal relaciona as Células de calor com as de fonte de calor. Ativa-se o cálculo da matriz, realizando a otimização das matrizes, descartando aquelas que possuem todos os valores iguais. Os objetos da classe Matriz_Células remanescentes são passados como parâmetro para a instância do objeto Dissipador que calcula a dissipação de calor, atualiza os objetos da classe Celula com a temperatura e a cor, adquirida através dos métodos do objeto da classe Qualifica. O diagrama de seqüência Realizar Cálculo é apresentado na fig. 19.

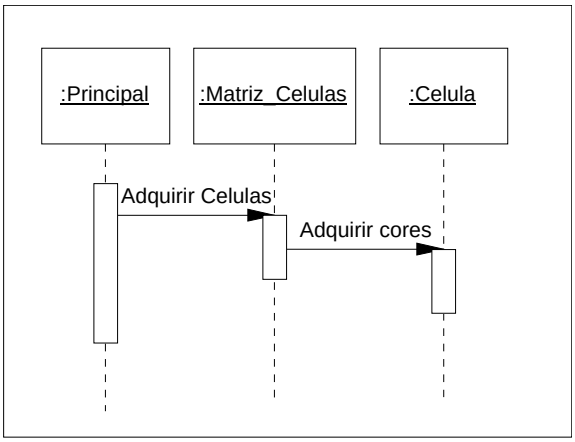
FIGURA 19 – DIAGRAMA DE SEQUÊNCIA REALIZAR CÁLCULO



5.1.1.3.3 EXIBIR DISSIPAÇÃO

O instância do objeto da classe Principal varre a matriz de Matriz_Células, que busca em todas as células a cor que representa a faixa de temperatura da célula. Adquirindo a cor que é desenhada no formulário principal, formando a interface de saído com o usuário. A fig. 20 especifica o diagrama de seqüência exibir dissipação.

FIGURA 20 – EXIBIR DISSIPAÇÃO



5.1.2 DETALHES DA IMPLEMENTAÇÃO DO PROTÓTIPO *STAND-ALONE*

Nesta seção apresentam-se detalhes da implementação do Protótipo de Dissipação de Calor *Stand-Alone.*

5.1.2.1 INTERFACE DE ENTRADA PARA CONFIGURAÇÃO DO PROTÓTIPO DE DISSIPAÇÃO DE CALOR *STAND-ALONE*

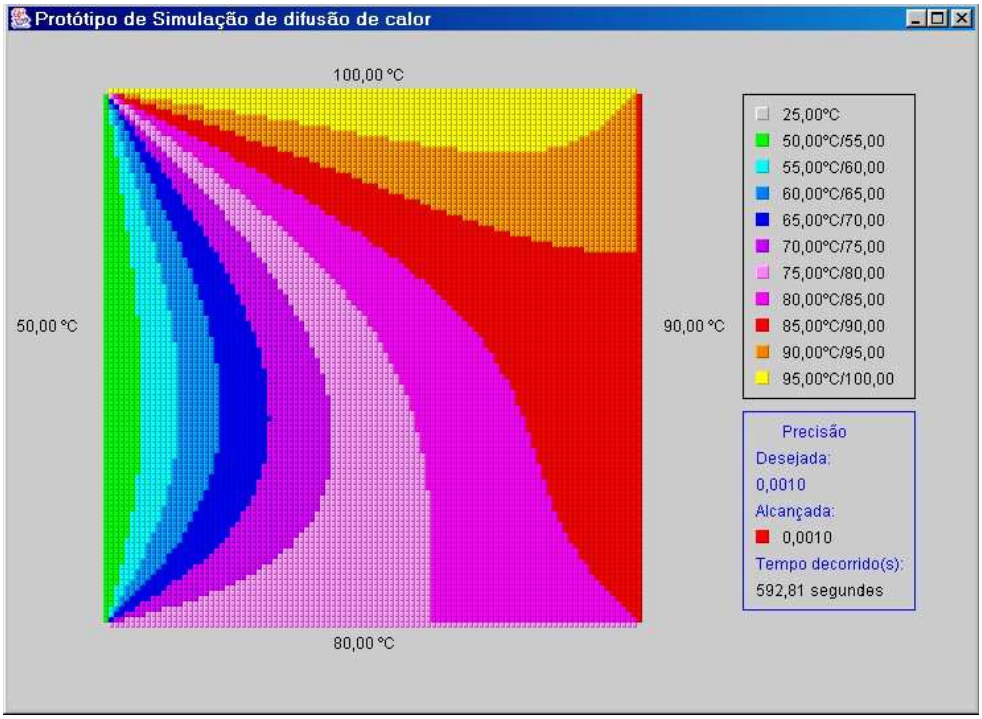
A fig. 21 apresenta a interface gráfica para configuração da simulação de dissipação de calor. Consiste basicamente em cinco campos onde são informadas as temperaturas das fontes de calor: superior, inferior, esquerda, direita e a temperatura ambiente, além da precisão desejada. O botão “iniciar” determina o processamento da difusão de calor.

FIGURA 21 –INTERFACE DE CONFIGURAÇÃO DO PROTÓTIPO DE DISSIPAÇÃO DE CALOR *STAND-ALONE*

5.1.2.2 INTERFACE DE SAÍDA DO PROTÓTIPO DE DISSIPAÇÃO DE CALOR *STAND-ALONE*

A fig. 22 demonstra a interface de saída, onde o protótipo exibe as sucessivas iterações, demostrando ao usuário a condução do calor. As temperaturas das fontes de calor são indicadas nas regiões periféricas da representação da chapa. As faixas de cores são reconhecidas através da legenda, posicionada do lado esquerdo da tela de saída. São mostradas ainda, informações sobre a precisão desejada e após cada iteração, além do tempo de processamento.

FIGURA 22 – INTERFACE DE SAÍDA DO PROTÓTIPO DE DISSIPAÇÃO DE CALOR



5.1.2.3 RELACIONAMENTO ENTRE MATRIZES

Na seção 2.3 é apresentada a necessidade de haver o relacionamento entre fontes de calor e as células de calor entre matrizes. O quadro 21 apresenta a forma como foi implementado este relacionamento. Depois de instanciada as matrizes de células foram criadas referências entre as linhas e colunas que ligam uma matriz na outra.

QUADRO 21 – PRINCIPAL. JAVA – RELACIONAMENTO ENTRE MATRIZES

```
for ( int i=0 ; i<10 ; i++ ){
  for ( int j=0 ; j < 10 ; j++ ) {
    m = new Matriz_Celulas( t, i, j);
    matriz[i][j] = m;

    if ( i > 0 ) {
//    inter-relacionamento das células das linhas entre das matrizes
      for ( int x=0 ; x<12 ; x++){
        matriz[i][j].setRelaciona(0,x,matriz[i-1][j].getRelaciona(10,x));
        matriz[i-1][j].setRelaciona(11,x,matriz[i][j].getRelaciona(1,x));
      }
    }
    if ( j > 0 ) {
//    inter-relacionamento das células entre colunas das matrizes
      for ( int x=0 ; x<12 ; x++){
        matriz[i][j].setRelaciona( x,0,matriz[i][j-1].getRelaciona(x,10));
        matriz[i][j-1].setRelaciona( x,11,matriz[i][j].getRelaciona(x,1));
      }
    }
  }// fim do for j
} //fim do for i
```

5.1.2.4 OTIMIÇÃO DA ROTINA DE CÁLCULO

Uma Matriz_Celulas, composta por 100 células de calor só sofre dissipação de calor se uma de suas células tiver valor de temperatura diferente das demais. A matriz que possuir todas as temperaturas iguais e estabilizadas não sofre o processo de cálculo. O código demonstrado no quadro 22 demonstra a implementação desta funcionalidade, utilizando uma estrutura composta rotulada. A rotina não precisa varrer todas as 144 células para detectar que ela precisa ser calculada. Isso acontece no momento que se encontra um valor diferente na matriz. Neste momento o laço que varre toda a matriz é encerrado e a matriz é calculada.

QUADRO 22 – PRINCIPAL.JAVA – OTIMIZAÇÃO DO CÁLCULO DE DISSIPAÇÃO

```
//varre a matriz de matrz células para efetuar o cálculo de dissipação
    for (int x=0 ; x<10 ; x++){
      for (int y=0 ; y<10 ; y++){
  /* esta instrução rotulada "break" realiza a otimização do cálculo,
   * varrendo a matriz_celulas em busca de células que necessitam ser
   * calculadas, ao encontrar a primeira incidência, habilita a necessidade
        * do cálculo e sai do bloco */

        otimiza : { //instrução composta rotulada
          for (int i=1 ; i<11 ; i++){
            for (int j=1 ; j<11 ; j++){
              cel[0] = matriz[x][y].getTemp(i-1,j);
              cel[1] = matriz[x][y].getTemp(i+1,j);
              cel[2] = matriz[x][y].getTemp(i,j-1);
              cel[3] = matriz[x][y].getTemp(i,j+1);
              for (int a=0 ; a<4 ; a++ ){
                if ( matriz[x][y].getTemp(i,j) != cel[a] ){
                   calc_ok = true;
                   break otimiza;
                }
              }
            }
          }
        }
        if ( calc_ok ){
          m = matriz[x][y];
 //         d = new Dissipador(m, t[7]);
          matriz[x][y] = d.calcular(m);
          calc_ok = false;
          if ( matriz[x][y].getSensibilidade() ){
            processar = true;
            matriz[x][y].setSensibilidade(false);
          }
          if (matriz[x][y].getVlrSen() > maior_sensibilidade)
            maior_sensibilidade = matriz[x][y].getVlrSen();
          //processar = true;
        }
      }
    }
```

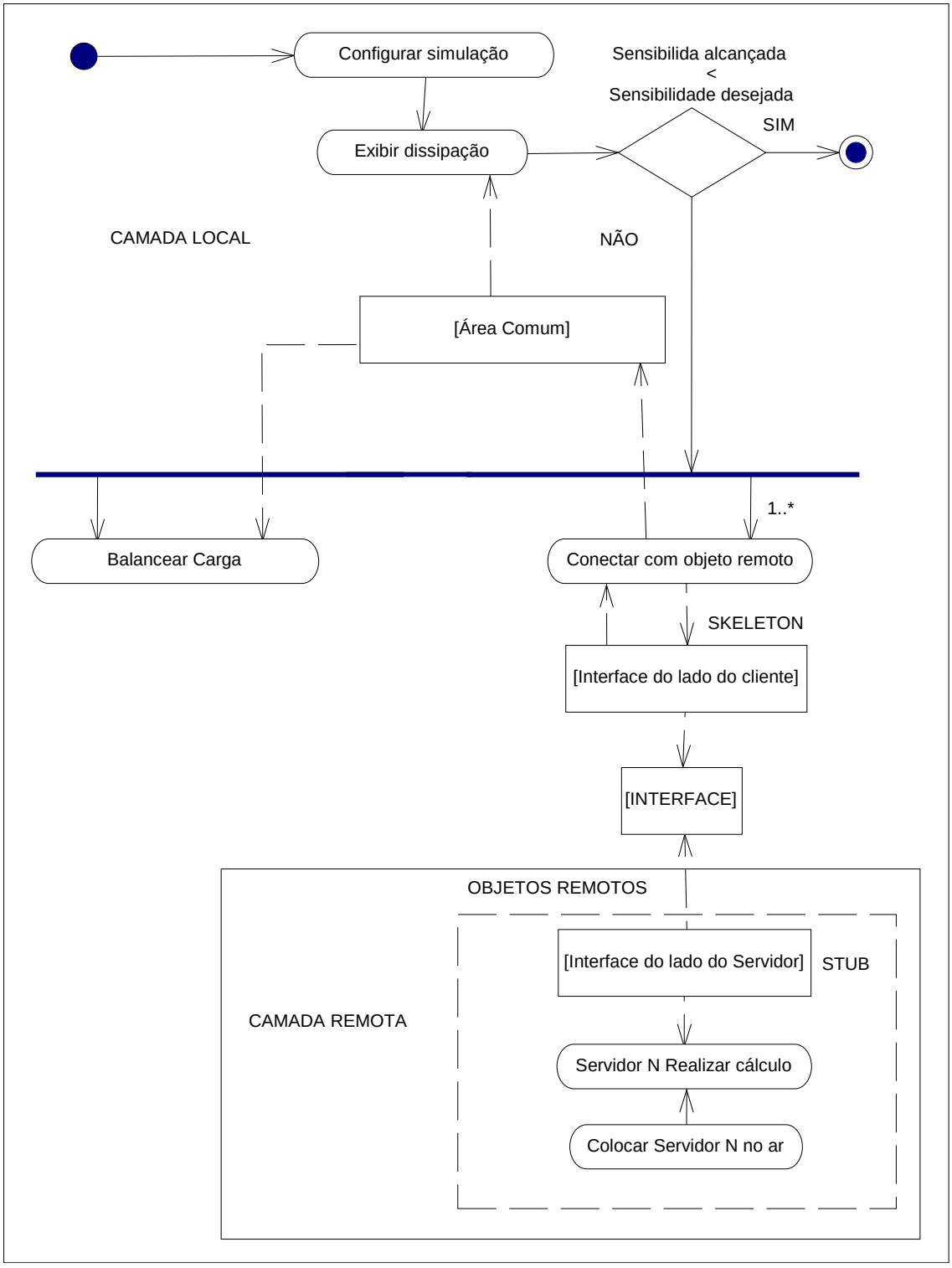
5.2 PROTÓTIPO DISTRIBUÍDO

A concepção das funcionalidades do protótipo distribuído tem como base a aplicação *stand-alone*. Muitas funcionalidades descritas são idênticas às do protótipo que o originou, mas ao longo desta seção há uma ênfase maior nos requisitos do protótipo distribuído.

Basicamente, o Protótipo para simulação da difusão do calor em um ambiente distribuído com carga balanceada é formado por cinco etapas principais:

a) configuração das informações necessárias para que o protótipo realize a difusão do calor. Nesta etapa um usuário informa a temperatura ambiente, as temperaturas das fontes de calor: superior, inferior, esquerda e direita e a precisão do cálculo;
b) os servidores de cálculo são inicializados para que se conectem com os objetos remotos e se comuniquem remotamente.
c) o módulo gerente realiza o balanceamento de carga, utilizando as informações sobre o poder de processamento, ou desempenho, de cada servidor.
d) o objeto cliente, localizado na máquina local, envia os dados para o objeto remoto para que sejam calculados, aguardando os resultados.
e) ao findar cada iteração, ou seja, ao calcular a dissipação de calor nas 10.000 células são apresentadas ao usuário, através de uma interface gráfica, as isofaixas de temperatura em que a temperatura de cada célula de calor é representada através de cores. De forma análoga ao protótipo *stand-alone*, neste modelo foram fixadas dez faixas de temperaturas, representadas pelo mesmo número de cores, partindo da menor até a maior temperatura informada durante a etapa de configuração da simulação, descrito no item (a) desta seção. A cada nova iteração é mostrada ao usuário a precisão alcançada naquele instante. A imagem apresentada ao usuário sofre mudanças ao longo do tempo de processamento, conforme mudam as temperaturas. O diagrama de atividade apresentada na fig. 23 demonstra a ação dos processos.

FIGURA 23– DIAGRAMA DE ATIVIDADE DO PROTÓTIPO DISTRIBUÍDO



Inicialmente o usuário interage com a interface do protótipo, fornecendo as configurações iniciais. O protótipo estabelece as conexões com os servidores e transmite os dados a serem calculados, realiza o balanceamento de carga para cada servidor, iniciando o

processo de cálculo. Ao terminar cada iteração o protótipo demonstra através de cores as faixas de temperaturas alcançadas.

5.2.1 ESPECIFICAÇÃO DO PROTÓTIPO DISTRIBUÍDO

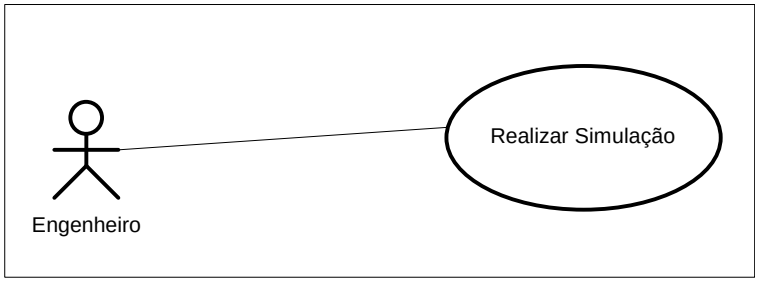
Como este protótipo partiu das funcionalidades de um protótipo *stand-alone* e o novo protótipo é distribuído, foi adaptada a modelagem com os requisitos das novas necessidades desta implementação.

Utiliza-se a Linguagem Unificada de Modelagem (UML), através dos diagramas de casos de uso, classes e seqüência para especificar a metodologia de modelagem orientada a objetos, através da ferramenta case Sybase Power Designer (Sybase, 2001).

5.2.1.1 CASOS DE USO DO PROTÓTIPO DISTRIBUÍDO

Apresenta-se na fig. 24 o caso de uso especificado para o protótipo distribuído. Este caso de uso é igual ao caso de uso do protótipo *stand-alone*, apresentado na seção 5.1.1, mas apresenta cinco processos, no lugar de três, em decorrência da natureza deste modelo possuir objetos distribuídos e paralelos. Optou-se por descrever os cinco processos em itens para aumentar o nível de detalhamento e entendimento, embora sejam relacionados e dependentes, conforme é mostrado a seguir:

FIGURA 24 – DIAGRAMA DE CASOS DE USO



a) configurar simulação: descrito no item 5.1.1.1 ítem a.
b) conectar com o servidor: o número de processos iniciados será num número igual ao de servidores disponíveis para o processamento de cálculo. Para cada servidor será criado um processo de conexão (thread), que é controlado por um processo cliente, permitindo a transferência dos dados a serem tratados.
c) balancear carga: um módulo gerente é responsável pela distribuição de carga entre os servidores, procurando equalizar a quantidade de pacotes enviados a

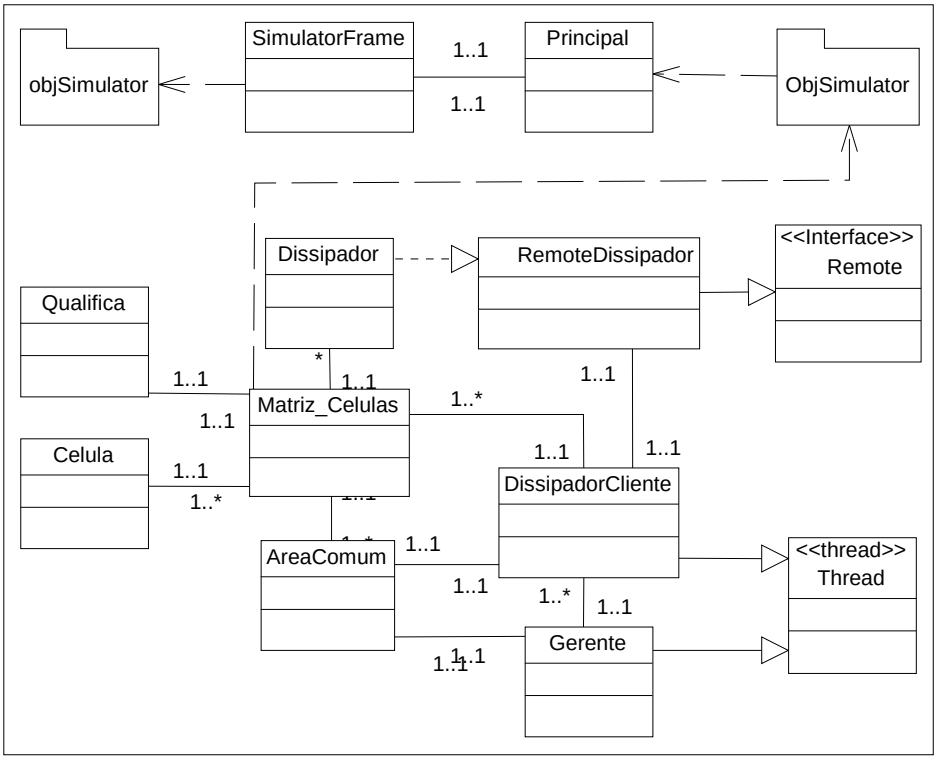
cada servidor, igualando o tempo de execução processo (thread), para que termine num tempo aproximado. Na primeira iteração não há informações sobre o poder de processamento de cada servidor, o módulo gerente então envia o mesmo número de pacotes para todos os servidores. A partir da segunda iteração os tempos e a quantidade de pacotes enviados para cada servidor são registrados e através da expressão de escalabilidade, definida na seção 4.2.1 são definidas as novas quantidades de pacotes a serem enviadas para cada servidor. O princípio da expressão de escalabilidade é enviar mais pacotes para o servidor que tenha o maior poder de processamento.

d) realizar cálculos: descrito na seção 5.1.1.1 ítem, b.
e) Exibir graficamente os resultados: descrito na seção 5.1.1.1 ítem, c.

5.2.1.2 DIAGRAMAS DE CLASSES DO PROTÓTIPO DISTRIBUÍDO

São utilizados três diagramas de classes para especificação do protótipo de dissipação de calor distribuído. Na fig. 25 as classes são apresentadas de forma consolidada, oferecendo uma visão de conjunto do protótipo, pois as principais classes encontram-se divididas em dois pacotes hierárquicos. A fig 26 e fig. 26 detalham os métodos e atributos das classes em cada pacote. A fig. 26 demonstra o diagrama de classes do pacote SimulatorRemote, que importa as classes do pacote objSimulator, apresentado na fig. 27. O pacote SimulatorRemote possui duas classes principais, enquanto que o pacote objSimulator implementa oito classes principais e utiliza 2 importantes classes da Interface de Programação de Aplicações Java (SUNc, 2002), que são demonstradas. Até o presente momento todas as classes da API Java foram ocultadas, por representar funcionalidades com pouca importância para o projeto, mas no momento se faz necessário especificar no diagrama de classes duas delas, *Threads* e *Remote*, objetivando a maior compreensão do modelo.

FIGURA 25 – DIAGRAMA DE CLASSES CONSOLIDADO



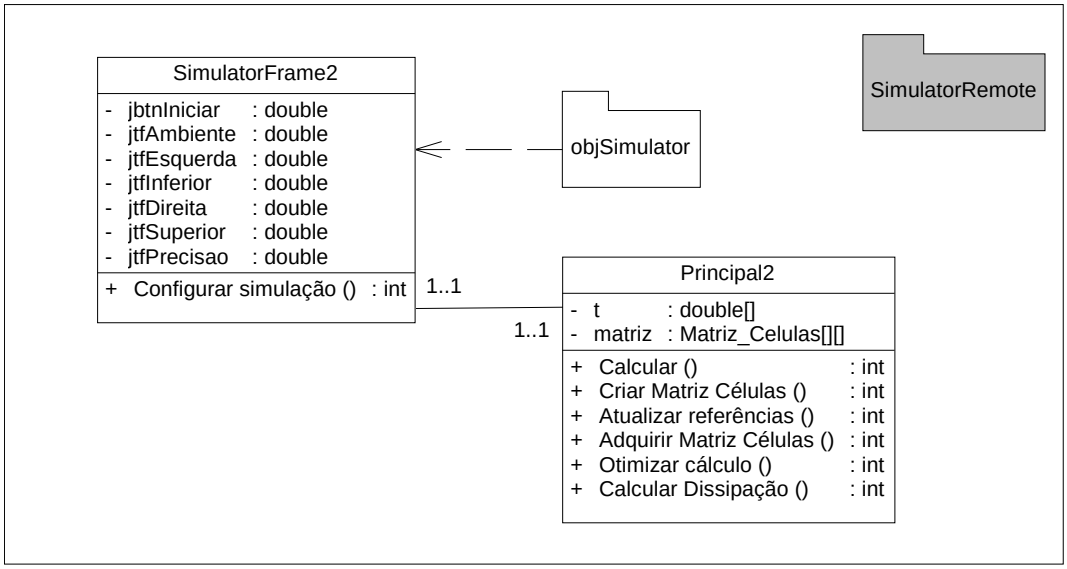
5.2.1.2.1 DIAGRAMA DE CLASSES DO PACOTE SIMULATORREMOTE

Implementa as duas principais classes do pacote SimulatorRemote, conforme demonstrado na fig. 26, importando as classes do pacote objSimulator, conforme relacionado.

a) SimulatorFrame: implementa a interface onde o usuário configura a simulação, determinando os parâmetros de temperatura e precisão

b) Principal: nesta classe são instanciadas os objetos das classes AreaComum, Gerente e DissipadorCliente, descritas na seção 5.2.1.2.2 e apresentação dos resultados ao usuário, através da interface gráfica da evolução da dissipação de calor. Na classe principal são configurados os servidores remotos, conforme o número de servidores existentes são criadas instâncias o mesmo número de objetos da classe DissipadorCliente.

FIGURA 26 – DIAGRAMA DE CLASSES: PACOTE SIMULATORREMOTE



5.2.1.2.2 DIAGRAMA DE CLASSES DO PACOTE OBJSIMULATOR

Diagrama de classes do pacote objSimulator implementa a modelagem das classes do pacote objSimulator, conforme apresentado na fig. 27. Possui oito classes principais e duas secundárias, da API Java (SUNc, 2002), promovendo o entendimento do modelo, relacionado a seguir.

a) `Matriz_Células`: através desta classe é implementa a estrutura de dados composta por uma matriz de partículas ou células de calor. Seus métodos fornecem acesso aos métodos da classe `Célula`;

b) `Célula`: esta é uma classe que encapsula a estrutura de dados de uma partícula ou célula de calor. Possui a temperatura e um vetor com as cores RGB desta temperatura;

c) `Qualifica`: esta classes está associada à classe `Matriz_`Células e determina o vetor cor RGB de uma `célula`, gerado a partir da sua temperatura;

d) `ÁreaComum`: esta classe implementa as variáveis de instância que compartilham os dados da aplicação. Seus métodos fornecem acesso aos dados e implementam o monitor que sincroniza os processos paralelos;

e) `Gerente`: implementa o balanceamento da carga destinado a cada servidor de cálculo. Com base na tomada de tempo do e o número de pacotes processados por um servidor aplica a expressão de escalonamento e determina o número de

pacotes destinados para cada servidor em uma nova iteração. A classe `Gerente` herda os métodos da classe *Thread* da API Java (Sun-c, 2002), permitindo o seu processamento paralelo a outros processos;

f) `DissipadorCliente`: Implementa a conexão e comunicação com os objetos remotos da classe `Dissipador`. Cada servidor instancia um objeto remoto Dissipador, que possui um objeto correspondente `DissipadorCliente` que é executado na camada local do protótipo. A comunicação entre estes objetos é feita através de uma interface, instanciada por um objeto correspondente da classe `RemoteDissipador`. `ClienteDissipador` recebe uma quantidade de pacotes do Gerente, envia para uma instância remota de um objeto Dissipador. Ao receber os resultados atualiza a `AreaComum`, com o resultado dos dados, e o tempo demandado para o processamento dos pacotes. Instâncias de objetos da classe `DissipadorCliente` são herdeiros da classe *Thread* da API Java (Sun-c, 2002), que implementam através dos seus métodos o processamento paralelo ou multi-processamento.

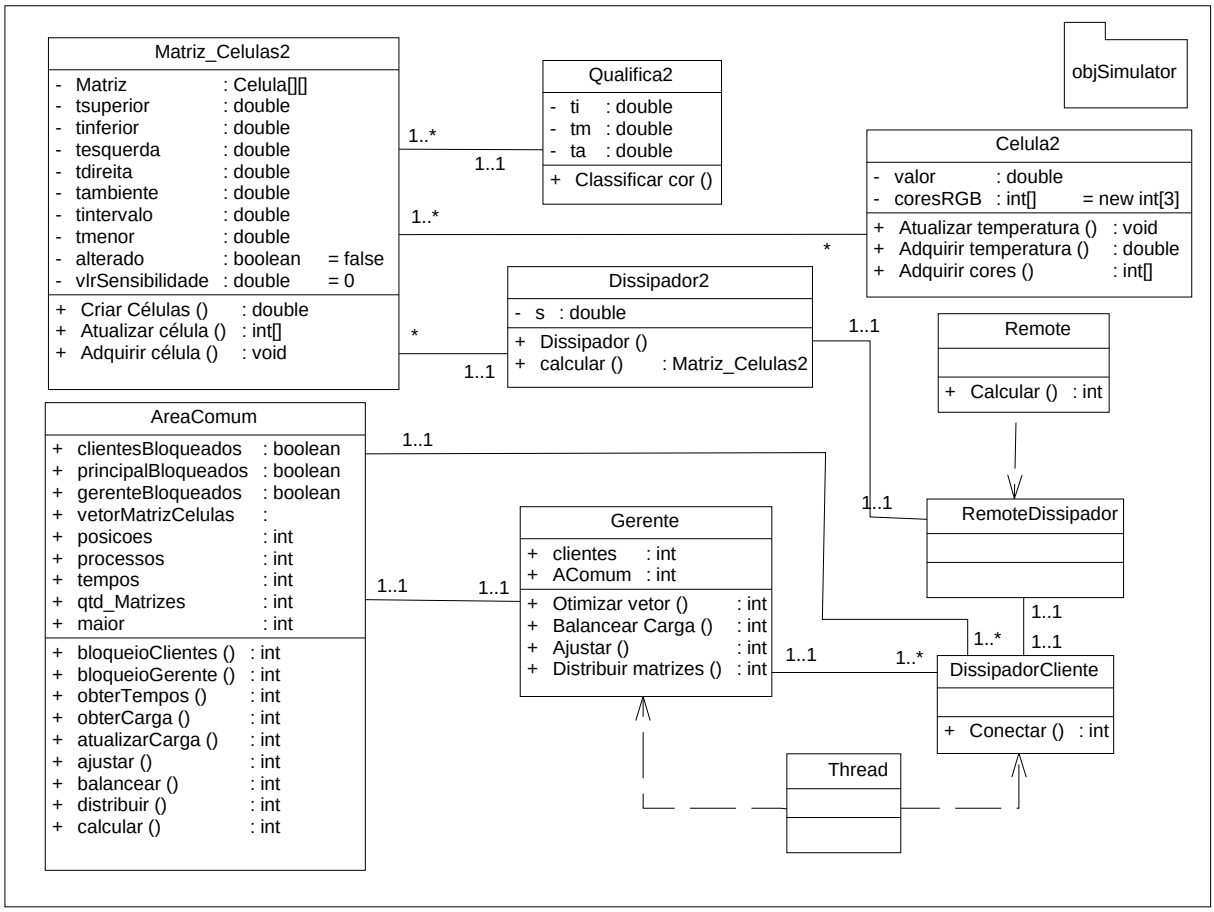
g) `Dissipador`: é o servidor de cálculo. Recebe os valores das temperaturas, calcula e devolve a instância do objeto `ClienteDissipador`, através do s*keleton* e do *Stub*, interfaceado por `RemoteDissipador`. A partir do `Dissipador` são gerados os s*keleton* e o s*tub* que permitem a comunicação entre os objetos remotos.

h) `RemoteDissipador`: herda os métodos da classe da API Java (Sun-c, 2002) *Remote*, possibilitando a interface entre os Skeletons e os Stubs, que realiza a comunicação entre os objetos remotos.

i) *Thread*: uma instância de objeto da classe *Thread* é um processo de execução em um programa. A Máquina Virtual Java permite que uma aplicação tenha múltiplos processos em execução concorrentemente.

j) Remote: A interface *Remote* serve para identificar interfaces que podem invocar métodos de uma máquina virtual não-local. Qualquer objeto que é um objeto remoto deve diretamente ou indiretamente implementar esta interface. Somente objetos que especificam uma interface *remote* estão remotamente disponíveis.

FIGURA 27 – DIAGRAMA DE CLASSES: PACOTE OBJSIMULATOR



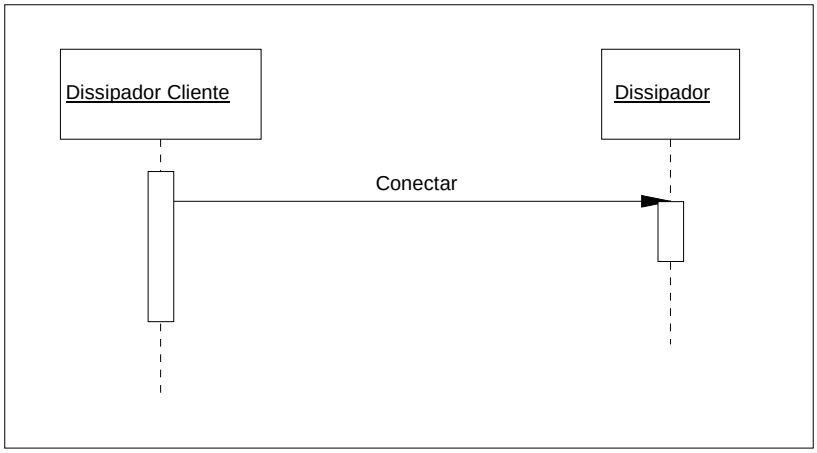
5.2.1.3 DIAGRAMAS DE SEQÜÊNCIA DO PROTÓTIPO DISTRIBUÍDO

Estes diagramas representam a seqüência em que as ações ocorrem dentro do sistema, através da troca de mensagens entre os objetos das classes. Cada caso de uso resultou em um diagrama de seqüência, conforme apresentado nas figuras 28, 29, 30, e 31.

5.2.1.3.1 CONECTAR SERVIDORES

A fig. 28 demonstra o diagrama de seqüência Conectar Servidores. Objetos da classe `DissipadorCliente` conectam os objetos remotos `Dissipador`.

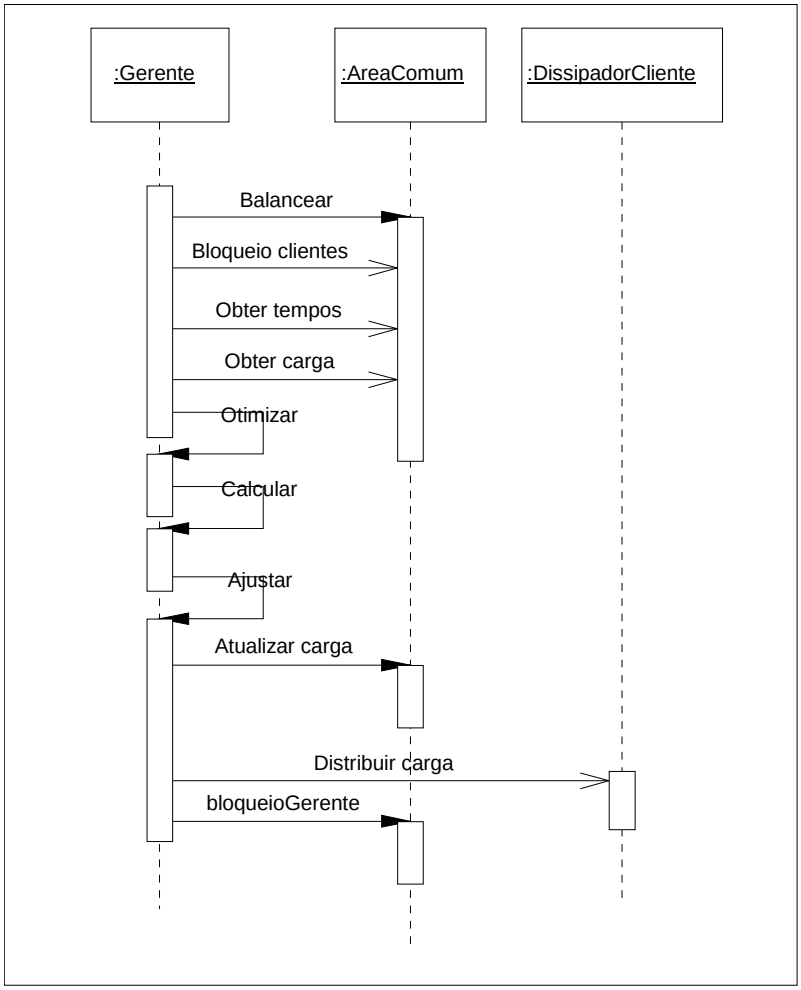
FIGURA 28 – DIAGRAMA DE SEQUÊNCIA CONECTAR SERVIDORES



5.2.1.3.2 BALANCEAR CARGA

A fig. 29 demonstra o diagrama de seqüência Balancear Carga.

FIGURA 29 – DIAGRAMA DE SEQUÊNCIA BALANCEAR CARGA

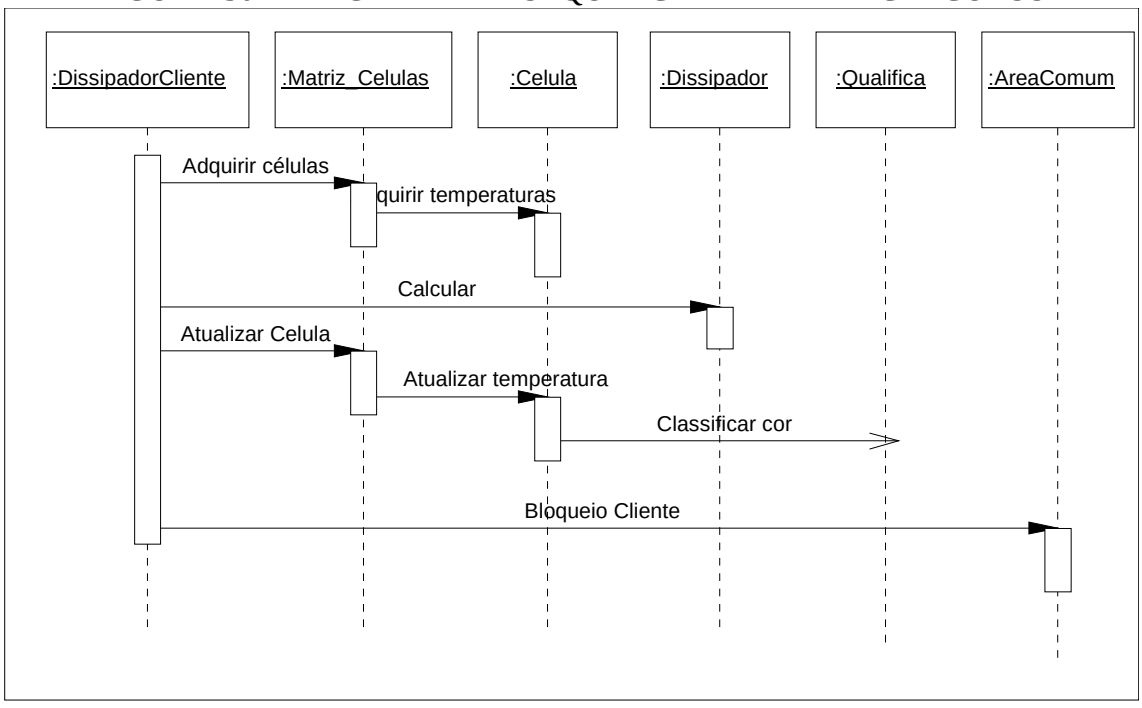


Uma instância de objeto da classe `Gerente`, bloqueia a execução de objetos da classe `DissipadorCliente`, obtém na área comum os tempos de processamento e o número de pacotes processados na iteração anterior. Na primeira vez que acontece o balanceamento de carga não há um histórico da iteração anterior, mas como estes valores de tempo e pacotes foram inicializados com um valor padrão, o módulo `Gerente` enviará um número igual de pacotes para cada servidor. Após a segunda iteração, acontece efetivamente o balanceamento de carga, baseado na iteração anterior real. O `gerente` aplica a equação de balanceamento de carga, descrita na seção 4.2.1, obtendo o número do pacotes ideais para cada servidor, obtém a matriz `Matriz_Células`, converte em um vetor `Matriz_Células`, contendo apenas os pacotes a serem processados por cada módulo e envia para cada unidade de processamento, para finizalizar o `Gerente` se bloqueia e libera os objetos `DissipadorCliente` para realizarem os cálculos.

5.2.1.3.3 REALIZAR CÁLCULOS

A fig. 30 demonstra o diagrama de seqüência Realizar Cálculos.

FIGURA 30 – DIAGRAMA DE SEQUÊNCIA REALIZAR CÁLCULOS



O objeto da classe `Cliente` obtém do vetor `Matriz_Células`, buscando os valores das células superior, inferior, esquerda e direita, localizadas em torno da célula que objeto do calculo. Para evitar erros de cálculo, é importante perceber que a célula que sofre o

cálculo não contribui com o cálculo. O módulo cliente envia os valores das células para o dissipador, que recebe, calcula e devolve o valor calculado. O módulo cliente atualiza a célula. Após terminar de calcular todas as células o Módulo cliente atualiza a `AreaComum`.

5.2.2 DETALHES DA IMPLEMENTAÇÃO DO PROTÓTIPO DISTRIBUÍDO

Nesta seção são apresentados detalhes da implementação do Protótipo de Dissipação de Calor Distribuída.

5.2.2.1 MONITOR DE PROCESSOS

O quadro 23 descreve parte do código utilizado para realizar o bloqueio dos três diferentes tipos de processos: exibir dissipação, balancear carga e realizar cálculos.

QUADRO 23 – AREACOMUM.JAVA – DESCREVE PARTE DO MECANISMO DE MONITOR DOS PROCESSOS CONCORRENTES

```
public synchronized void setClientesBloqueados( int prc,
boolean aux){
   // bloqueou cliente:true
   clientesBloqueados[prc] = aux;
   // desbloqueou gerente
   gerenteBloqueado = false;
   for (int i=0 ; i<clientesBloqueados.length ; i++ ){
     if (!clientesBloqueados[i] ){
         //se ainda há clientes desbloqueados
       // bloquear novamente o gerente de novo
       gerenteBloqueado = true;
     }
   }
   if (!gerenteBloqueado)
     System.err.println("desbloqueou principal");
     principalBloqueado = false;
   notifyAll();
 }
```

Neste trecho de código supõe-se que um ou mais processos de cálculos estão acontecendo. Quando ele termina, chama o método setClientesBloqueados, para bloquear o processo que terminou, verificando se ainda há clientes desbloqueados, ou seja, processando. Se não há mais processos clientes (responsáveis pelos cálculos) bloqueados, então gerente é desbloqueado, e se o gerente está desbloqueado (classe gerente realiza o balanceamento de carga), e o processo da classe Principal (responsável pela exibição do resultado) é

desbloqueado também, ambos passam a disputar o tempo de processamento da CPU. Um mecanismo semelhante bloqueia estes dois últimos processos quando forem concluídos, desbloqueando os clientes responsáveis pelo cálculo.

5.2.2.2 BALANCEAMENTO DE CARGA

O quadro 24 demonstra, com comentários nas linhas, o método da classe Gerente balancearCarga(), descrevendo a forma de implementação o processo de balanceamento de carga. É complementado pelo quadro 24, que apresenta o método ajustar, da mesma classe, responsável pelo ajuste do balanceamento em caso de ocorrer erro de arredondamento, conforme descrito na seção 4.2.1.1.

QUADRO 25 – GERENTE.JAVA – MÉTODO BALANCERAR CARGA

```
protected void balancearCarga(){
    //inicializando as variáveis
    //double aux[] = new double[aComum.getProcessos()];
    // criar uma variável p/ rendimento e o equilibrio para cada processo
    double rendimento[] = new double[aComum.getProcessos()];
    double escalabilidade[] = new double[aComum.getProcessos()];
    // adquirir o vetor que possui as matrizes a serem calculadas
    Vector vMCelulas = aComum.getVetorCelulas();
    /* inicializa o número de matrizes q seraum processadas
     * tempo total de todos os processos, total de matrizes adquiridas
     * e total de matrizes distribuídas após o balanceamento*/
    System.err.println("T="+vMCelulas.size());
    int nro_matrizes=vMCelulas.size();
    int total_tempo=0;
    int nv_nro_matrizes=0;
    //total do rendimento adquirido
    double total_rend=0;
    //vetor auxiliar com os tempos de cada processo
    double[] tempos = aComum.getTempos();
    /*vetor auxiliar para armazenar o número de
     matrizes q cada processador receberá após o balanceamento*/
    int qtMatrizes[] = aComum.getQtd_matrizes();

    //totaliza o tempos
    for (int i=0; i<aComum.getProcessos(); i++){
      total_tempo += tempos[i];
    }

    /* Calcula e totaliza o rendimento
     * esta é a primeira expressão chave do BALANCEAMENTO DE CARGA*/
    for (int i=0; i<aComum.getProcessos(); i++){
      total_tempo += tempos[i];
      rendimento[i] = (total_tempo/tempos[i])*qtMatrizes[i];
      total_rend += rendimento[i];
    }

    /* Calcula e totaliza o índice de escalabilidade
     * esta é a segunda expressão chave do BALANCEAMENTO DE CARGA*/
    for (int i=0; i<aComum.getProcessos(); i++){
      escalabilidade[i] = rendimento[i]/total_rend;
    }

    int maior = 0;

    for (int i=0; i<aComum.getProcessos(); i++){
      /* distribui proporcionalmente as matrizes conforme o seu tempo
       * de processamento e quantidade de matrizes processadas
       * no último processamento*/
      qtMatrizes[i] = (int) Math.floor(escalabilidade[i]*nro_matrizes);
      nv_nro_matrizes += qtMatrizes[i];
      // busca o índice do processo que recebeu mais processos
      if (qtMatrizes[i] > maior)
        maior = i;
    }
    qtMatrizes = ajustar(nv_nro_matrizes, nro_matrizes,
maior,qtMatrizes);
    aComum.setQtd_matrizes( qtMatrizes );
 }

      return qt_mat;
}
```

QUADRO 26 – GERENTE.JAVA – MÉTODO AJUSTAR

```
protected int[] ajustar(int nv_n_m, int n_m, int aux_maior, int[]
qt_mat){ /* recebe o novo total de matrizes distribuídas, o numero
       *efetivo de matrizes, a matriz com maior carga e um
       * vetor contendo a carga em cada matriz.
       * Este método corrige possíveis distorções de arredondamento
       * causados pelo arredondamento dos valores e garante
       * que todos os processos recebam ao menos
       * uma matriz para cálcular, para
       * garantir a tomada de tempo do processamento*/
       //Se a carga atual for maior que a efetiva
      if (nv_n_m > n_m)
              //o processo de maior carga perde uma
        qt_mat[aux_maior] = qt_mat[aux_maior]--;
       //se a carga atual for menor que a efetiva
      if (nv_n_m < n_m)
       //o processo de menor carga recebe uma carga
        qt_mat[aux_maior] = qt_mat[aux_maior]++;
       //Se algum processo naum receber nenhuma carga
      for ( int i=0 ; i<qt_mat.length ; i++ ){
        if (qt_mat[i] < 1){
          //processo recebe uma carga
          qt_mat[i]=qt_mat[i]++;
         //processo com maior carga perde uma carga
          qt_mat[aux_maior] = qt_mat[aux_maior]--;
        }
      }
```

6 COMPARAÇÃO ENTRE O SISTEMA STAND-ALONE E DISTRIBUÍDO

Este capítulo apresenta o resultado dos testes comparativos entre o simulador Stand-alone e distribuído.

6.1 DESCRIÇÃO DOS TESTES

Estabeleceram-se 5 configurações de testes em diferentes cenários. As configurações de simulação estão definidas na tabela 5. Definindo-se 5 cenários de teste, descritos nas tabelas 6 e 7. Foram realizando 5 baterias de teste em cada cenário, utilizando-se as 5 configurações da tabela 5 nos 5 cenários da tabela 6. Em todos os cenários somente as aplicações protótipos estavam sendo executadas, não havia outras aplicações sendo executas simultaneamente.

TABELA 5 – CONFIGURAÇÕES DA SIMULAÇÃO

Configuração	Temperaturas					Precisão
	superior	inferior	esquerda	direita	ambiente	
1	100	100	10	30	25	0.001
2	100	100	10	30	25	0.01
3	30	150	150	30	25	0.001
4	30	40	35	45	25	0.001
5	-25	30	0	20	10	0.001

TABELA 6 – CENÁRIOS DE TESTE DO PROTÓTIPO DISTRIBUÍDO

Cenário	Máquinas	Descrição	Memória	Rede
1	Cliente	AMD Athlon XP 1.7 GHz	256	-
	Servidor 1			
2	Cliente	AMD Athlon XP 1.7 GHz	256	10 Mbit´s
	Servidor 1	Pentium IV 1.7 GHz	512	
3	Cliente	AMD Athlon XP 1.7 GHz	256	10 Mbit´s
	Servidor 1	Pentium IV 1.7 GHz	512	
	Servidor 2	AMD 400 MHz	128	
4	Cliente	AMD Athlon XP 1.7 GHz	256	10 Mbit´s
	Servidor 1	Pentium IV 1.7 GHz	512	
	Servidor 2	AMD 400 MHz	128	
	Servidor 3	Pentium MMX 233	64	
5	Cliente	AMD Athlon XP 1.7 GHz	256	10 Mbit´s
	Servidor 1	Pentium IV 1.7 GHz	512	
	Servidor 2	AMD 1 GHz	128	
	Servidor 3	Pentium MMX 233	64	

TABELA 7 – CENÁRIOS DE TESTE DO PROTÓTIPO STAND-ALONE

Cenário	Descrição da máquina	Memória	Rede
6	AMD Athlon XP 1.7 GHz	256	-

6.2 RESULTADOS DOS TESTES

A tabela 8 apresenta o resultado obtido durante os testes, expressos em segundos(s).

TABELA 8 – RESULTADO DA APLICAÇÃO DOS TESTES

Configurações	Cenários (tempos/cenário 1)					
	1	2	3	4	5	6
1	1.719,49	1.908,63	1.914,36	2.315,17	1.542,55	537,34
2	841,78	942,79	1.055,92	1.198,28	821,66	221,52
3	1.855,24	2.430,36	2.892,13	2.943,16	1.664,33	587,10
4	1.230,94	1.427,89	1.642,08	1.607,81	967,64	379,92
5	974,40	1.110,82	1.299,65	1.234,12	853,67	243,60
Média (s)	1324,37	1564,10	1760,83	1859,71	1169,67	393,90

6.3 ANÁLISE DOS RESULTADOS

Ao analisar os resultados obtidos nos teste, percebe-se uma distorção entre a fundamentação teórica esperada e a efetiva conclusão dos testes. Inicialmente percebe-se que a diferença entre o cenário 1 e o cenário 6, onde temos a aplicação stand-alone comparada com a aplicação distribuída, com o servidor rodando localmente há uma diferença de 339%, valor que surpreende, pela efetiva elevação dos valores. Entre os cenários 1 e 5 tem-se aumento do poder processamento, pela inclusão de um número maior de máquinas. Conforme vai aumentando o poder de processamento, aumenta também o tempo total do processo. A aplicação *stand-alone* é incontestavelmente mais rápida que as distribuídas.

No gráfico demonstrado na fig. 32 pode-se visualizar a diferença dos tempos entre os diversos cenários em cada simulação. O gráfico, exibido na fig, 31, compara a média dos resultados do processamento *stand-alone* e a média de todos os outros cenários distribuídos, para perceber a diferença entre cada cenário.

O gráfico apresentado na fig. 31 agrupa os seis cenários de teste. Cada cenário de teste possui 5 simulações, com diferentes configurações, com os resultados dos tempos representados em segundos (s).

O gráfico apresentado na fig. 32 apresenta nas colunas claras a média do tempo das simulações em cenários que utilizam o protótipo distribuído e nas colunas escuras o tempo das simulações com o protótipo *stand-alone*.

FIGURA 31 – GRÁFICO COMPARATIVO DOS TEMPOS OBTIDOS

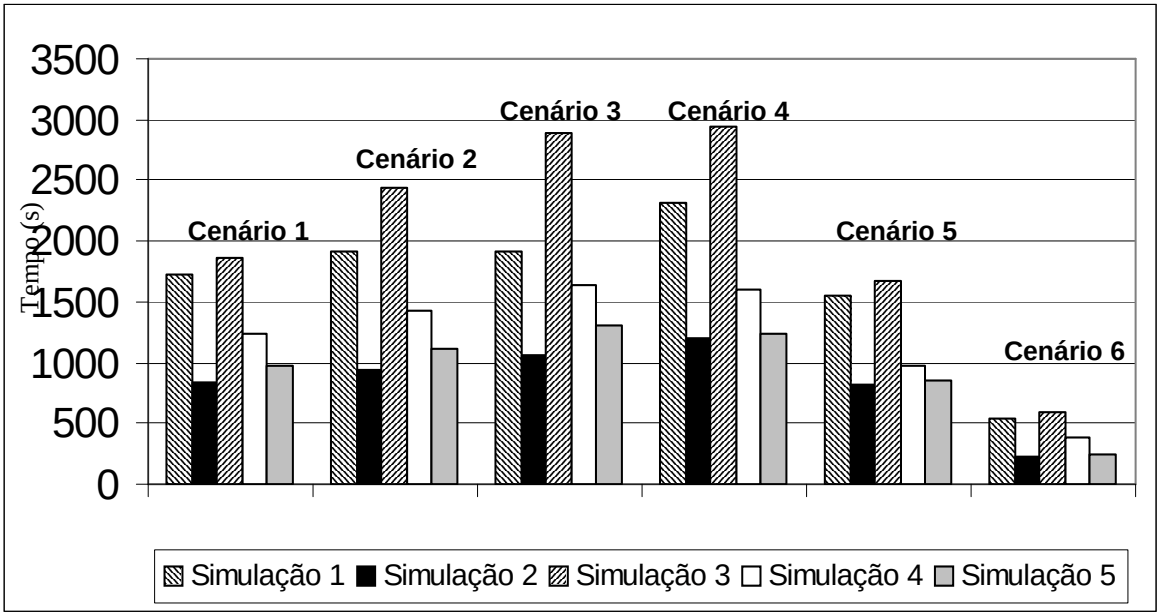
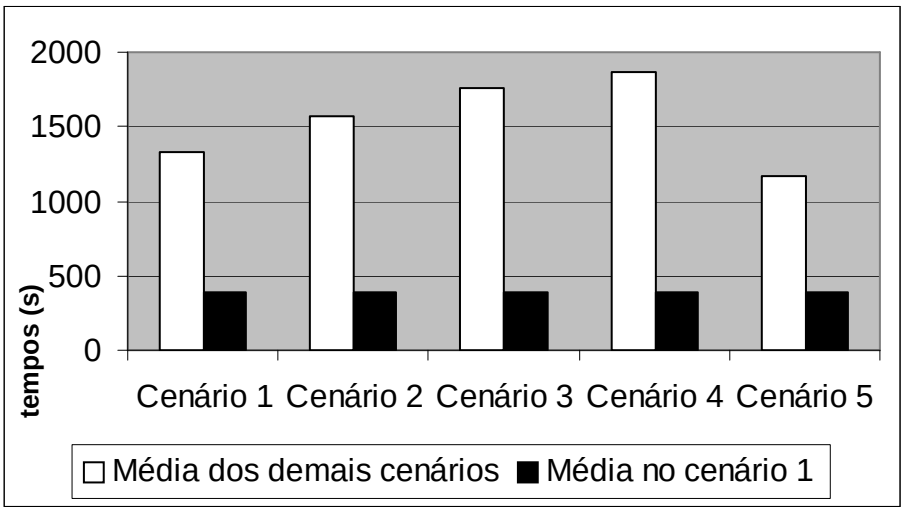


FIGURA 32 - GRÁFICO DO TEMPO MÉDIO COM O PROTÓTIPO *STAND-ALONE VERSUS* DISTRIBUÍDO



Os resultados finais apresentam-se muito distantes da realidade esperada, levando a conclusão de que o protótipo distribuído não foi implementado da forma mais eficiente. Há erros conceituais na fase de modelagem do protótipo distribuído, pois um dos requisitos necessários para o perfeito funcionamento não está implementado que é o uso correto dos processos concorrentes dos servidores, embora tenha sido feito o balanceamento de carga, os pacotes não são serializados e enviados por inteiro para que o servidor possa calculá-los de forma independente dos clientes. Quando um processo pára de enviar dados o servidor pára de calcular o problema. Embora o protótipo tenha obedecido quase todos os requisitos que um sistema distribuído de processamento paralelo, ele exige a transferência de dados por inteiro e não de forma síncrona, como foi implementado, onde o excesso de troca de mensagens torna-se um gargalo do tempo de processamento.

A ferramenta inicialmente proposta, Borland® Jbuilder™ 3 University Edition, não oferecia suporte necessário para desenvolvimento desta aplicação. Ao decorrer dos estudos também houve testes com o ambiente de desenvolvimento NetBeans™ DeveloperX2 2.1, que apresenta uma interface de recursos limitados. Estas dificuldades levaram a utilização do ambiente de desenvolvimento Forte (Sun-c, 2002). O Forte for Java 4 (Sun-c, 2002) é gratuito, didático, ótimas funções para *"debugar"* as aplicações e é muito semelhante aos melhores ambientes de programação do mercado. No entanto é importante salientar que este ambiente de desenvolvimento necessita de hardware de bom desempenho, no caso do desenvolvimento deste trabalho usou-se AMD XP 1.700 MHz e 256 Mb de memória RAM, um equipamento que já é popular, se levarmos em consideração que em dezembro de 2002 já contamos com processadores de 2.200 Mhz e 256 Mb de RAM são pré-requisitos para alguns sistemas operacionais de uso doméstico.

7 CONSIDERAÇÕES FINAIS

7.1 CONCLUSÕES

No decorrer do presente trabalho conceitos desenvolvidos ao longo da academia foram discutidos, formando uma sólida base de conhecimento sobre o assunto.

A percepção de que a falta de cuidados durante implementação resulta na conclusão que sistemas distribuídos utilizando processos paralelos são mais lentos que sistemas similares *stand-alone*, em função do excesso de troca de mensagens agir em detrimento ao desempenho da aplicação.

Outro problema detectado no protótipo distribuído é que, após a realização dos cálculos, a interface com o usuário não se atualizava, ficando o mesmo estático até o fim do processamento. Para cumprir o cronograma do trabalho e realização de testes, foi desativada a interface gráfica e levou-se em consideração somente o resultado dos processamentos.

A análise dos benefícios do balanceamento de carga tornou-se dúbia, levando em consideração que o protótipo não funcionou como esperado.

A implementação do modelo distribuído, utilizando a API Java com o pacote RMI é simples e como a própria bibliografia declara é de fácil utilização, assim como a própria API Java. Mesmo desenvolvedores com pouca experiência em Java sentem-se confortável para utilização do pacote RMI, levando em consideração que a documentação é detalhada, esquemática, conceitual e com exemplos oferecidos por inúmeros autores. Embora simples, o processo de serialização não é assim tão transparente, pouco comentado na bibliografia e exige uma atenção adicional para que não se perca tempo desnecessário durante a fase de desenvolvimento.

O presente trabalho pode ser utilizado por pessoas que queiram fundamentação básica sobre sistemas de balanceamento de carga em objetos distribuídos de processamento paralelo, mas o protótipo de processamento remoto não deve ser utilizado como referência, antes de serem revistas as falhas de modelagem.

Apesar de ter apresentado problemas, deve-se considerar que parte significativa dos objetivos propostos foram alcançados: os dois protótipos foram implementados, sendo que o *stand-alone* cumpre os requisitos previstos para o sistema, o sistema distribuído foi implementado, mas com erros de concepção na fase implementação. Entende-se também que um modelo de balanceamento de carga foi proposto e implementado, e que o protótipo caracteriza-se como uma aplicação distribuída utilizando serviços Java RMI.

7.2 SUGESTÕES

Como sugestão para trabalhos futuros recomenda-se que seja revista a modelagem e estudos de formas de serialização dos dados a serem transferidos entre cliente e servidor. Há muitas perguntas a serem respondidas neste detalhe que compromete o alcance do objetivo final.

Uma nova estratégia de processamento pode ser proposta, objetivando tornar as unidades cálculo autônomas e que troquem mensagens apenas quando uma informação tenha relação com outro objeto distribuído e que enviem apenas os resultados para o módulo cliente, quebrando a estrutura cliente-servidor implementado no presente trabalho.

Há um extenso campo na área de desenvolvimento do algoritmo ideal para mais de 2 tipos de processos concorrentes diferentes em Java, envolvendo processos e interface gráfica, implementando modelos de exclusão mútua ou monitores.

Para finalizar, sugere-se também a utilização do modelo do simulador para dissipação de calor para identificar a partir de quantos processos paralelos é mais eficiente utilizar um sistema distribuído do que um sistema *stand-alone*.

7.3 DIFICULDADES ENCONTRADAS

Não há um consenso sobre a forma como sistemas distribuídos devem ser modeladas, apesar de haver muitas literaturas não há um senso comum sobre a metodologia correta de representação.

REFERÊNCIAS BIBLIOGRÁFICAS

BALEN, Henry. **Distributed object architectures with CORBA**. Madrid: Cambridge, 2000. 285 p.

CERBE, Günter; HOFFMANN, Hans-Joachim. **Introdução à termodinâmica**. Tradução João Câmara Neiva com a colaboração de Augusto Câmara Neiva. São Paulo, Polígono, 1973.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed systems : concepts and design**. 3. ed. Harlow: Addison-Wesley, 2001. 772 p.

DEITEL. H. M.; DEITEL. P. J. **Java**: como programar. 3. ed. Tradução Edson Furmankiewicz. Porto Alegre: Bookmann, 2001. 1201 p.

FURLAN, J. D. **Modelagem de objetos através da UML-The Unifield Modeling Language**. Sao Paulo: Makron Books do Brasil, 1998. 329 p.

HEIN, Nelson; BIEMBENGUT, Maria Salett. Modelagem matemática e simulação. In Simpósio de Educação Matemática, 2, 2000, Chivilcoy. **Anais**… Chivilcoy: EDUMAT, 2000. p. 26-33.

HÜBNER, Jomi Fred. **Comunicação entre objetos distribuídos**. Blumenau: Fundação Universidade Regional de Blumenau, 2000. Disponível em <www.inf.furb.br/~jomi/java/pdf/rmi.pdf >. Acesso em 30 out 2002.

HULL, M. E. C.; CROOKES, Danny; SWEENEY, Patrick J.; **Parallel processing** : the transputer and its applications. Wokingham: Addison-Wesley, 1994. 328 p.

JÚNIOR, Luiz Augusto de Paula Lima. Objetos distribuídos. In Escola de Informática da SBC Sul, 9, 2001, Maringá, PR, Passo Fundo, RS, São José, SC. **Anais**... Porto Alegre: UFRS, 2001. 232 p. ref. 143-173.

KERN, Donald Q. **Processos de transmissão de calor**. Tradução de Adir M. Luiz. Rio de Janeiro: Guanabara Dois, 1982. Título original: Process heat transfer.

OLIVEIRA, Rômulo de; MONTEZ, Carlos; FRAGA, Joni. Programação em sistemas distribuídos. In Escola de Informática da SBC Sul, 10, 2002, Caxias do Sul, RS, Criciúma, Cascavel, PR. **Anais**...Porto Alegre: UFRGS, 2002. 165 p. ref. 39-84.

ROCHA, Helder da. **Fundamentos de objetos remotos**. 2002: Argonavis. Disponível em <www.argonavis.com.br/palestras/java/jav433/>. Acesso em 30 out 2002.

SILVA, Cláudio Xavier da; FILHO, Benigno Barreto. **Matemática**: aula por aula. São Paulo: FTD, 2000.

SUNa. **Java 2™ SDK**: Standard Edition Documentation, versão 1.4.0. Sun® Microsystem Inc, 2002. 1 CD-ROM Ferramentas Java v.1.

SUNb. **The Java™ Tutorial**: A practical guide for programmers, atualizado em 4 mar 2002. Sun® Microsystem Inc, 2002. 1 CD-ROM Ferramentas Java v.1.

SUNc. **Forte™ for Java™ 4 Community Edition**, build 020521, Java™ 1.4.0_02. Sun® Microsystem Inc, 2002. 1 CD-ROM Ferramentas Java v.1.

SYBASE®. **Sybase Power Designer®**, versão 9.0.0.438: avaliação. SYBASE. 1 CD-ROM de instalação.

TANENBAUM, Andrew S.; WOODHULL, Albert. **Sistemas Operacionais**: Projeto e implementação. Tradução Edson Furmankiewicz. Porto Alegre: Bookmann, 2000. 759 p.

WOLGEMUTH, Carl H.; SCHMIDT, Frank. W.; HENDERSON, Robert. **Introdução às ciências térmicas**: termodinâmica, mecânica dos fluídos e transferência de calor. Tradução coordenada por José Roberto Simões Moreira. São Paulo: Edgard Blücher, 1996. Título original: Introduction to thremal sciences: thermodynamics fluid dynamics heat transfer.

ANEXO A – RESULTADO DO ESTUDO DO ERRO DE ARREDONDAMENTO

Tot.	Índice de Escalabilidade						Matrizes recebidas										Total de	Err
							Antes do arredondamento					Após arredondam.						
Mat.	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5	P1	P2	P3	P4	P5	matrizes	!=
1	0,01	0,21	0,20	0,40	0,18	1	0,01	0,21	0,2	0,4	0,18	0	0	0	0	0	0	1
2	0,02	0,22	0,19	0,39	0,18	1	0,04	0,44	0,38	0,78	0,36	0	0	0	1	0	1	1
3	0,03	0,23	0,18	0,38	0,18	1	0,09	0,69	0,54	1,14	0,54	0	1	1	1	1	4	-1
4	0,04	0,24	0,17	0,37	0,18	1	0,16	0,96	0,68	1,48	0,72	0	1	1	1	1	4	0
5	0,05	0,25	0,16	0,36	0,18	1	0,25	1,25	0,8	1,8	0,9	0	1	1	2	1	5	0
6	0,06	0,26	0,15	0,35	0,18	1	0,36	1,56	0,9	2,1	1,08	0	2	1	2	1	6	0
7	0,07	0,27	0,14	0,34	0,18	1	0,49	1,89	0,98	2,38	1,26	0	2	1	2	1	6	1
8	0,08	0,28	0,13	0,33	0,18	1	0,64	2,24	1,04	2,64	1,44	1	2	1	3	1	8	0
9	0,09	0,29	0,12	0,32	0,18	1	0,81	2,61	1,08	2,88	1,62	1	3	1	3	2	10	-1
10	0,10	0,30	0,11	0,31	0,18	1	1	3	1,1	3,1	1,8	1	3	1	3	2	10	0
11	0,11	0,31	0,10	0,30	0,18	1	1,21	3,41	1,1	3,3	1,98	1	3	1	3	2	10	1
12	0,12	0,32	0,09	0,29	0,18	1	1,44	3,84	1,08	3,48	2,16	1	4	1	3	2	11	1
13	0,13	0,33	0,08	0,28	0,18	1	1,69	4,29	1,04	3,64	2,34	2	4	1	4	2	13	0
14	0,14	0,34	0,07	0,27	0,18	1	1,96	4,76	0,98	3,78	2,52	2	5	1	4	3	15	-1
15	0,15	0,35	0,06	0,26	0,18	1	2,25	5,25	0,9	3,9	2,7	2	5	1	4	3	15	0
16	0,16	0,36	0,05	0,25	0,18	1	2,56	5,76	0,8	4	2,88	3	6	1	4	3	17	-1
17	0,17	0,37	0,04	0,24	0,18	1	2,89	6,29	0,68	4,08	3,06	3	6	1	4	3	17	0
18	0,18	0,38	0,03	0,23	0,18	1	3,24	6,84	0,54	4,14	3,24	3	7	1	4	3	18	0
19	0,19	0,39	0,02	0,22	0,18	1	3,61	7,41	0,38	4,18	3,42	4	7	0	4	3	18	1
20	0,20	0,40	0,01	0,21	0,18	1	4	8	0,2	4,2	3,6	4	8	0	4	4	20	0
21	0,01	0,21	0,20	0,40	0,18	1	0,21	4,41	4,2	8,4	3,78	0	4	4	8	4	20	1
22	0,02	0,22	0,19	0,39	0,18	1	0,44	4,84	4,18	8,58	3,96	0	5	4	9	4	22	0
23	0,03	0,23	0,18	0,38	0,18	1	0,69	5,29	4,14	8,74	4,14	1	5	4	9	4	23	0
24	0,04	0,24	0,17	0,37	0,18	1	0,96	5,76	4,08	8,88	4,32	1	6	4	9	4	24	0
25	0,05	0,25	0,16	0,36	0,18	1	1,25	6,25	4	9	4,5	1	6	4	9	5	25	0
26	0,06	0,26	0,15	0,35	0,18	1	1,56	6,76	3,9	9,1	4,68	2	7	4	9	5	27	-1
27	0,07	0,27	0,14	0,34	0,18	1	1,89	7,29	3,78	9,18	4,86	2	7	4	9	5	27	0
28	0,08	0,28	0,13	0,33	0,18	1	2,24	7,84	3,64	9,24	5,04	2	8	4	9	5	28	0
29	0,09	0,29	0,12	0,32	0,18	1	2,61	8,41	3,48	9,28	5,22	3	8	3	9	5	28	1
30	0,10	0,30	0,11	0,31	0,18	1	3	9	3,3	9,3	5,4	3	9	3	9	5	29	1
31	0,11	0,31	0,10	0,30	0,18	1	3,41	9,61	3,1	9,3	5,58	3	10	3	9	6	31	0
32	0,12	0,32	0,09	0,29	0,18	1	3,84	10,24	2,88	9,28	5,76	4	10	3	9	6	32	0
33	0,13	0,33	0,08	0,28	0,18	1	4,29	10,89	2,64	9,24	5,94	4	11	3	9	6	33	0
34	0,14	0,34	0,07	0,27	0,18	1	4,76	11,56	2,38	9,18	6,12	5	12	2	9	6	34	0
35	0,15	0,35	0,06	0,26	0,18	1	5,25	12,25	2,1	9,1	6,3	5	12	2	9	6	34	1
36	0,16	0,36	0,05	0,25	0,18	1	5,76	12,96	1,8	9	6,48	6	13	2	9	6	36	0
37	0,17	0,37	0,04	0,24	0,18	1	6,29	13,69	1,48	8,88	6,66	6	14	1	9	7	37	0
38	0,18	0,38	0,03	0,23	0,18	1	6,84	14,44	1,14	8,74	6,84	7	14	1	9	7	38	0
39	0,19	0,39	0,02	0,22	0,18	1	7,41	15,21	0,78	8,58	7,02	7	15	1	9	7	39	0
40	0,20	0,40	0,01	0,21	0,18	1	8	16	0,4	8,4	7,2	8	16	0	8	7	39	1
41	0,01	0,21	0,20	0,40	0,18	1	0,41	8,61	8,2	16,4	7,38	0	9	8	16	7	40	1

42	0,02	0,22	0,19	0,39	0,18	1	0,84	9,24	7,98	16,38	7,56	1	9	8	16	8	42	0
43	0,03	0,23	0,18	0,38	0,18	1	1,29	9,89	7,74	16,34	7,74	1	10	8	16	8	43	0
44	0,04	0,24	0,17	0,37	0,18	1	1,76	10,56	7,48	16,28	7,92	2	11	7	16	8	44	0
45	0,05	0,25	0,16	0,36	0,18	1	2,25	11,25	7,2	16,2	8,1	2	11	7	16	8	44	1
46	0,06	0,26	0,15	0,35	0,18	1	2,76	11,96	6,9	16,1	8,28	3	12	7	16	8	46	0
47	0,07	0,27	0,14	0,34	0,18	1	3,29	12,69	6,58	15,98	8,46	3	13	7	16	8	47	0
48	0,08	0,28	0,13	0,33	0,18	1	3,84	13,44	6,24	15,84	8,64	4	13	6	16	9	48	0
49	0,09	0,29	0,12	0,32	0,18	1	4,41	14,21	5,88	15,68	8,82	4	14	6	16	9	49	0
50	0,10	0,30	0,11	0,31	0,18	1	5	15	5,5	15,5	9	5	15	6	16	9	51	-1
51	0,11	0,31	0,10	0,30	0,18	1	5,61	15,81	5,1	15,3	9,18	6	16	5	15	9	51	0
52	0,12	0,32	0,09	0,29	0,18	1	6,24	16,64	4,68	15,08	9,36	6	17	5	15	9	52	0
53	0,13	0,33	0,08	0,28	0,18	1	6,89	17,49	4,24	14,84	9,54	7	17	4	15	10	53	0
54	0,14	0,34	0,07	0,27	0,18	1	7,56	18,36	3,78	14,58	9,72	8	18	4	15	10	55	-1
55	0,15	0,35	0,06	0,26	0,18	1	8,25	19,25	3,3	14,3	9,9	8	19	3	14	10	54	1
56	0,16	0,36	0,05	0,25	0,18	1	8,96	20,16	2,8	14	10,08	9	20	3	14	10	56	0
57	0,17	0,37	0,04	0,24	0,18	1	9,69	21,09	2,28	13,68	10,26	10	21	2	14	10	57	0
58	0,18	0,38	0,03	0,23	0,18	1	10,44	22,04	1,74	13,34	10,44	10	22	2	13	10	57	1
59	0,19	0,39	0,02	0,22	0,18	1	11,21	23,01	1,18	12,98	10,62	11	23	1	13	11	59	0
60	0,20	0,40	0,01	0,21	0,18	1	12	24	0,6	12,6	10,8	12	24	1	13	11	61	-1
61	0,01	0,21	0,20	0,40	0,18	1	0,61	12,81	12,2	24,4	10,98	1	13	12	24	11	61	0
62	0,02	0,22	0,19	0,39	0,18	1	1,24	13,64	11,78	24,18	11,16	1	14	12	24	11	62	0
63	0,03	0,23	0,18	0,38	0,18	1	1,89	14,49	11,34	23,94	11,34	2	14	11	24	11	62	1
64	0,04	0,24	0,17	0,37	0,18	1	2,56	15,36	10,88	23,68	11,52	3	15	11	24	12	65	-1
65	0,05	0,25	0,16	0,36	0,18	1	3,25	16,25	10,4	23,4	11,7	3	16	10	23	12	64	1
66	0,06	0,26	0,15	0,35	0,18	1	3,96	17,16	9,9	23,1	11,88	4	17	10	23	12	66	0
67	0,07	0,27	0,14	0,34	0,18	1	4,69	18,09	9,38	22,78	12,06	5	18	9	23	12	67	0
68	0,08	0,28	0,13	0,33	0,18	1	5,44	19,04	8,84	22,44	12,24	5	19	9	22	12	67	1
69	0,09	0,29	0,12	0,32	0,18	1	6,21	20,01	8,28	22,08	12,42	6	20	8	22	12	68	1
70	0,10	0,30	0,11	0,31	0,18	1	7	21	7,7	21,7	12,6	7	21	8	22	13	71	-1
71	0,11	0,31	0,10	0,30	0,18	1	7,81	22,01	7,1	21,3	12,78	8	22	7	21	13	71	0
72	0,12	0,32	0,09	0,29	0,18	1	8,64	23,04	6,48	20,88	12,96	9	23	6	21	13	72	0
73	0,13	0,33	0,08	0,28	0,18	1	9,49	24,09	5,84	20,44	13,14	9	24	6	20	13	72	1
74	0,14	0,34	0,07	0,27	0,18	1	10,36	25,16	5,18	19,98	13,32	10	25	5	20	13	73	1
75	0,15	0,35	0,06	0,26	0,18	1	11,25	26,25	4,5	19,5	13,5	11	26	5	20	14	76	-1
76	0,16	0,36	0,05	0,25	0,18	1	12,16	27,36	3,8	19	13,68	12	27	4	19	14	76	0
77	0,17	0,37	0,04	0,24	0,18	1	13,09	28,49	3,08	18,48	13,86	13	28	3	18	14	76	1
78	0,18	0,38	0,03	0,23	0,18	1	14,04	29,64	2,34	17,94	14,04	14	30	2	18	14	78	0
79	0,19	0,39	0,02	0,22	0,18	1	15,01	30,81	1,58	17,38	14,22	15	31	2	17	14	79	0
80	0,20	0,40	0,01	0,21	0,18	1	16	32	0,8	16,8	14,4	16	32	1	17	14	80	0
81	0,01	0,21	0,20	0,40	0,18	1	0,81	17,01	16,2	32,4	14,58	1	17	16	32	15	81	0
82	0,02	0,22	0,19	0,39	0,18	1	1,64	18,04	15,58	31,98	14,76	2	18	16	32	15	83	-1
83	0,03	0,23	0,18	0,38	0,18	1	2,49	19,09	14,94	31,54	14,94	2	19	15	32	15	83	0
84	0,04	0,24	0,17	0,37	0,18	1	3,36	20,16	14,28	31,08	15,12	3	20	14	31	15	83	1
85	0,05	0,25	0,16	0,36	0,18	1	4,25	21,25	13,6	30,6	15,3	4	21	14	31	15	85	0
86	0,06	0,26	0,15	0,35	0,18	1	5,16	22,36	12,9	30,1	15,48	5	22	13	30	15	85	1
87	0,07	0,27	0,14	0,34	0,18	1	6,09	23,49	12,18	29,58	15,66	6	23	12	30	16	87	0
88	0,08	0,28	0,13	0,33	0,18	1	7,04	24,64	11,44	29,04	15,84	7	25	11	29	16	88	0
89	0,09	0,29	0,12	0,32	0,18	1	8,01	25,81	10,68	28,48	16,02	8	26	11	28	16	89	0
90	0,10	0,30	0,11	0,31	0,18	1	9	27	9,9	27,9	16,2	9	27	10	28	16	90	0

91	0,11	0,31	0,10	0,30	0,18	1	10,01	28,21	9,1	27,3	16,38	10	28	9	27	16	90	1
92	0,12	0,32	0,09	0,29	0,18	1	11,04	29,44	8,28	26,68	16,56	11	29	8	27	17	92	0
93	0,13	0,33	0,08	0,28	0,18	1	12,09	30,69	7,44	26,04	16,74	12	31	7	26	17	93	0
94	0,14	0,34	0,07	0,27	0,18	1	13,16	31,96	6,58	25,38	16,92	13	32	7	25	17	94	0
95	0,15	0,35	0,06	0,26	0,18	1	14,25	33,25	5,7	24,7	17,1	14	33	6	25	17	95	0
96	0,16	0,36	0,05	0,25	0,18	1	15,36	34,56	4,8	24	17,28	15	35	5	24	17	96	0
97	0,17	0,37	0,04	0,24	0,18	1	16,49	35,89	3,88	23,28	17,46	16	36	4	23	17	96	1
98	0,18	0,38	0,03	0,23	0,18	1	17,64	37,24	2,94	22,54	17,64	18	37	3	23	18	99	-1
99	0,19	0,39	0,02	0,22	0,18	1	18,81	38,61	1,98	21,78	17,82	19	39	2	22	18	100	-1
100	0,20	0,40	0,01	0,21	0,18	1	20	40	1	21	18	20	40	1	21	18	100	0