CENTRO UNIVERSITARIO DE ARARAQUARA

Departamento das Ciências da Administração e Tecnologia

Curso de Engenharia de Computação

Relatório Final – Iniciação Cientifica

Implementação de algoritmo paralelo para simulação de condução de calor em objeto bidimensional utilizando o método de diferenças finitas

Aluno: Flavio Luiz dos Santos de Souza

Orientador: Prof. MSc. Rodrigo Daniel Malara

Araraquara, Novembro de 2010

SUMÁRIO

1. RESUMO

A computação de alto desempenho tem sido utilizada como recurso para resolver problemas de grande complexidade em que na maioria das vezes não se tem um poder computacional disponível. Apesar de se tornarem áreas distintas de um mesmo seguimento, são difíceis de separar no conceito ou na análise de computação paralela hardware e software. Contudo, as super máquinas apresentam dificuldade para resolver vários quebra-cabeças da física e da matemática, como a previsão climática em uma escala inferior a 50 km^2, em um tempo hábil. Com o objetivo de utilizar programação concorrente, este trabalho propõe estudar os paradigmas de sistemas distribuídos, que permite utilizar toda gama de potencial da maquina disponível, exigindo assim o domínio das diferentes configurações de hardware existente no mercado e o conceito que envolve a distribuição de tarefas e a comunicação interprocessos a fim de simular um problema físico. Realizando uma simulação de condução de calor em uma chapa metálica, através de cálculos matemáticos, deve-se gerar um speedup linear e positivo com programação seqüencial e outra concorrente. De forma que o resultado apresente um aumento da desempenho que seja diretamente proporcional ao aumento do recurso computacional empregado na execução.

***Palavras chaves**: alto desempenho, concorrente, distribuída*

2. INTRODUÇÃO

Técnicas de computação de alto desempenho e programação concorrente são utilizadas para resolver problemas de grande complexidade e conseqüentemente alto custo computacional. Os computadores atuais não são capazes de produzir resultados em tempo hábil para vários problemas da física e da matemática, por exemplo, a previsão climática em escalas menores que 50 km2.

As utilizações de princípios de programação concorrente envolvem conhecimentos adicionais sobre programação que envolve a distribuição de tarefas e a comunicação interprocessos e este trabalho se propõe a estudar as diferentes configurações de hardware existentes no mercado para computação de alto desempenho, estudar os paradigmas de programação concorrente que permitam ao desenvolvedor utilizar o poder computacional disponibilizado por tais equipamentos e produzir uma prova de conceito utilizando o conhecimento adquirido.

Tal protótipo precisa obter speedup positivo e linear de forma que o aumento de desempenho seja diretamente proporcional ao aumento de recursos computacionais empregados para a sua execução.

3. JUSTIFICATIVA

A computação de alto desempenho possibilita ao aluno a aprofundar seus conhecimentos a respeito de arquiteturas de computadores e a desenvolver seu intelecto em direção a uma lógica de programação mais complexa e apropriada para os próximos anos na área da Computação, uma vez que até mesmo computadores pessoais já apresentam componentes de hardware que anteriormente só existiam em computadores de centros de pesquisa, como mais do que um processador em um mesmo computador.

4. OBJETIVO

- Produzir uma revisão bibliográfica acerca do hardware e técnicas de programação e frameworks disponíveis para elaborar programas concorrentes adequados para computadores de alto desempenho.
- Programar um programa paralelo usando um framework de paralelização para simular o fenômeno físico “Condução de calor” em uma chapa metálica utilizando o método numérico de diferenças finitas.
- Criar representações gráficas da chapa através de imagens antes e após a aplicação do algoritmo paralelo.
- Calcular o *speedup* da solução paralela.
- Documentar os resultados obtidos e publicar artigos.

5. MATERIAIS E MÉTODOS

5.1. SOFTWARES

- Sistema operacional GNU/Linux *OpenSuse 11.2*
- Compilador Intel, *Intel Compiler 11.1*
- IDE (*Integrated Development Environment* ou *Ambiente Integrado de Desenvolvimento*) Eclipse 3.5 com CDT (*C/C++ Development Tooling ou Ferramenta de Desenvolvimento C/C++*)

5.2. HARDWARE

- Processador Intel Core 2 Duo T650
- 3 Gb de memória

6. REVISÃO BIBLIOGRÁFICA

6.1. ARQUITETURA PARALELA CONTEXTO HISTÓRICO

Em 1958, John Cocke e Daniel Slotnick discutir o uso de paralelismo em cálculos numéricos em uma nota de pesquisa IBM. Slotnick depois propõe SOLOMON, uma máquina SIMD com PES 1024 1-bit, cada um com memória para 128 valores de 32 bits. A máquina nunca foi construída, mas o design é o ponto de partida para um trabalho muito mais tarde.

A IBM oferece STRETCH primeiro computador em 1959. Um total de oito é construído, grande parte da tecnologia é reutilizada no IBM 7090, que foi entregue no mesmo ano. Junto ao IBM 7090 a Honeywell introduz Honeywell 800, com suporte de hardware para compartilhamento de tempo entre os oito programas.

Estamos em 1964, nesse ano Daniel Slotnick propõe a construção de uma máquina massivamente paralela para o Laboratório Nacional Lawrence Livermore (LLNL), a Comissão de Energia Atômica dá o contrato de CDC em vez disso, constroem o STAR-100. Slotnick do projeto financiado pela Força Aérea, e evolui para o ILLIAC-IV. A máquina é construída na Universidade de Illinois, com Burroughs e Texas Instruments como subcontratastes primários. Advanced Texas Instruments Computação Científica (ASC), também cresce a partir desta iniciativa.

Em 66, Michael Flynn publica um artigo descrevendo a taxonomia de arquitetura que leva seu nome e referencia usada até os dias atuais.

Já 1969, o trabalho começa no Compass Inc. em uma paralelização compilador Fortran para o ILLIAC-IV chamado IVTRAN. Honeywell fornece o primeiro sistema Multics (multi-processamento simétrico com até oito processadores). Um projeto multi- processador de 1970 da Universidade Carnegie Mellon, desenvolve o C.mmp com o apoio de DEC. Ainda neste ano é desenvolvido o PDP-6/KA10 multiprocessador assimétrico conjuntamente pelo MIT e DEC.

Computadores paralelos SIMD podem ser rastreados até a década de 1970. A motivação por trás dos computadores SIMD foi antecipada para amortizar o retardo da porta da unidade do processador, o controle sobre várias instruções. Em 64, Slotnick tinha proposto a construção de um computador massivamente paralelo para o Laboratório Nacional Lawrence Livermore. Seu projeto foi financiado pela Força Aérea norte americana. Seu projeto de paralelismo bastante é elevado, com até 256 processadores, o que permitiu a máquina a trabalhar em grandes conjuntos de dados no que seria mais tarde seria conhecido como processamento vetorial. No entanto, ILLIAC IV foi chamado de "the most infamous of Supercomputers", porque um quarto de seu projeto foi completado em 11 anos e custou quase quatro vezes a estimativa inicial. Quando finalmente estava pronto para executar a sua primeira aplicação real em 1976, foi superado por supercomputadores comerciais existentes, como o Cray-1.

Supercomputadores, popular na década de 80 como o Cray X-MP foram chamados transformadores "vector". O Cray X-MP tinha até quatro processadores vetoriais que possam funcionar de forma independente ou trabalhar em conjunto, utilizando um modelo de programação chamada "autotasking". Autotasking foi semelhante ao OpenMP. Estas máquinas tinham processadores muito fácil escalar e também

processadores vetoriais para cálculos do vetor de comprimento, por exemplo, a adição de dois vetores de 100 números cada

A primeira era de máquinas SIMD foi caracterizada por supercomputadores, como o Thinking Machines CM-1 e CM-2. Estas máquinas tinham muitos processadores funcionalidade limitadas que iria trabalhar em paralelo. Por exemplo, cada um dos 64.000 processadores em um Thinking Machines CM-2 vai executar a mesma instrução ao mesmo tempo, de modo que você poderia fazer 64.000 multiplica em 64.000 pares de números em um momento.

6.2 CONTEXTO HISTÓRICO PROGRAMAÇÃO PARALELA

Começamos em 1955 quando a IBM introduz o 704. Principal arquiteto é Gene Amdahl, é a primeira máquina comercial com o hardware de ponto flutuante, é capaz de cinco KFLOPS aproximadamente. No ano seguinte a IBM inicia o projeto do 7030 (conhecido como STRETCH) para produzir supercomputador para Los Alamos National Laboratory (LANL). Seu objetivo é produzir uma máquina com 100 vezes o desempenho de qualquer disponível no momento.

Dez anos mais tarde James W. Cooley e John W. Tukey descrever o algoritmo Fast Fourier Transform, que posteriormente é um dos maiores consumidores única de ciclos de ponto flutuante.

Em 1967 Gene Amdahl e Daniel Slotnick publicam AFIPS debate na Conferência sobre a viabilidade do processamento paralelo. Argumento de Amdahl sobre os limites do paralelismo se torna conhecida como "Lei de Amdahl".

1976, Carl Hewitt, no MIT, inventa o modelo de atores, em que as estruturas de controle são os padrões de mensagens. Este modelo é a base para trabalhos posteriores tanto em alto nível, modelos de programação paralela.

Já em 1978 Brinch Hansen descreve chamada de procedimento remoto (RPC), em papel em processos distribuídos, embora ele não use esse termo. Steven Fortune e James Wyllie descrever o modelo PRAM, que se torna o modelo padrão para a análise de complexidade de algoritmos paralelos.

1979 Josh Fisher em Yale descreve rastreamento de programação, um método de compilação de programas escritos em linguagem convencional para máquinas palavra larga. No próximo ano, PFC (Parallel Fortran Compiler), desenvolvido na Universidade Rice, sob a direção de Ken Kennedy.

1981 J. Bruce Nelson, da Xerox PARC e Carnegie-Mellon University, descreve e nomes de chamada de procedimento remoto. RPC é a base para mais tarde muitos paralelos e de programação de sistemas distribuídos.

6.2 CONTEXTUALIZAÇÃO

Apesar de se tornarem áreas distintas de um mesmo seguimento, hardware e software são difíceis de separar no conceito ou na analise de computação paralela de alto desempenho.

O excelente desempenho de do paralelismo depende de ambos, não ajudará em nada ter uma boa arquitetura paralela se não tiver um bom proveito do algoritmo e vice-versa!

6.3 CONCEITUAÇÃO ARQUITETURA PARALELA

A taxonomia de Flynn abrange quatro modelos de arquitetura SISD, SIMD, MISD e MIMD o qual todos são baseados nos fluxos de instruções e dados. A proposta de Flynn é ainda a forma mais comum de classificar sistemas de processamento paralelo.

As definições de a taxonomia a seguir foram baseadas na descrição do livro Arquitetura e Organização de Computadores de Willian Stallings.

6.3.1 SISD (SINGLE INSTRUCTION SINGLE DATA)

A caracterização desse modelo são computadores que executam uma instrução por vez de um único software, utilizando dados armazenados em uma única memória. As arquiteturas SISD trabalham como uma única unidade de controle, logo tem um baixo poder de calculo. É a mesma característica encontrada em maquinas de Von Neumann, que nada mais são que computadores tradicionais (tendo processadores convencionais) ou os antigos Mainframes.

FIGURA 1 REPRESENTAÇÃO DE ARQUITETURA SISD

6.3.2 SIMD (SINGLE INSTRUCTION MULTIPLE DATA)

Esta classificação corresponde ao processamento de vários dados sob o comando de apenas uma instrução. Cada elemento de processamento tem uma memória de dados associada. Nesta classe estão os processadores vetoriais e matriciais.

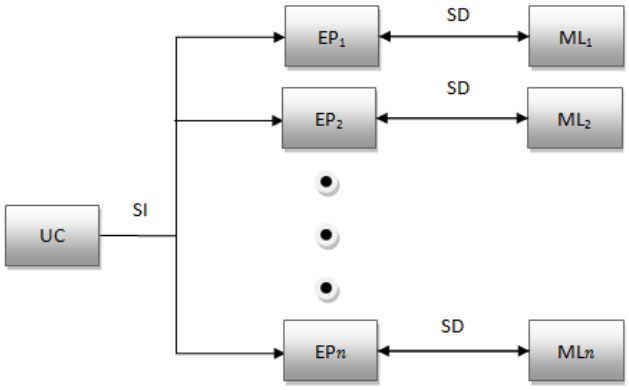


FIGURA 2 REPRESENTAÇÃO DE ARQUITETURA SIMD

6.3.3 MISD (MULTIPLE INSTRUCTION SINGLE DATA)

Uma seqüência de dados é transmitida para um conjunto de processadores, cada um dos quais executa uma seqüência de instruções diferentes. Essa estrutura nunca foi implementada.

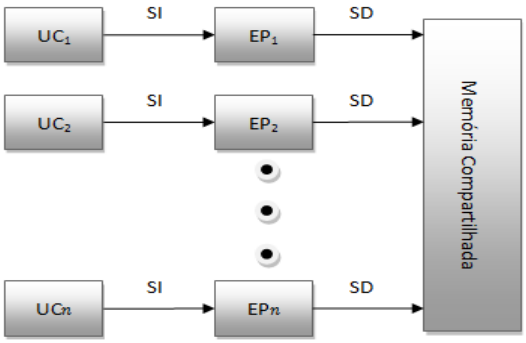


FIGURA 3 REPRESENTAÇÃO DE ARQUITETURA MISD

6.3.4 MIMD (MULTIPLE INSTRUCTION MULTIPLE DATA)

Um conjunto de processadores executa simultaneamente seqüência de instruções, sobre conjunto de dados distintos. Os SMPs, cluster e sistemas NUMA pertencem a essa categoria.

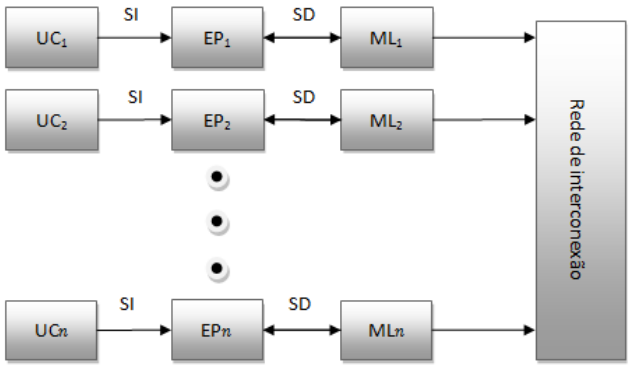


FIGURA 4 REPRESENTAÇÃO DE ARQUITETURA MIMD

6.4 CONCEITUAÇÃO PROGRAMAÇÃO PARALELA

Em um nível de abstração mais elevado, os modelos de programação paralela são divididos em duas categorias:

Sistemas de memória compartilhada, onde a memória que pode ser acessa simultaneamente por vários programas com a intenção de proporcionar uma comunicação entre eles ou evitar cópias redundante.

Sistemas de memória distribuída são múltiplos processadores, cada um com sua própria memória. Logo cada processador executa sua tarefa podendo operar somente dados locais, caso o dado for remeto exigirá outra tarefa computacional.

6.4.1 SPSD (SINGLE PROGRAM SINGLE DATA)

Todos os processos executam somente um programa, utilizando-se dos mesmos dados que os demais.

6.4.2 SPMD (SINGLE PROGRAM MULTIPLE DATA)

Cada processo roda exatamente o mesmo programa (embora algumas partes do programa sejam executadas somente pelo processo mestre), mas com um conjunto diferente de dados. Dessa forma, os

dados são divididos pelo processo mestre e distribuídos entre os processos escravos, ou seja, cada processo receberá informações diferentes.

6.4.3 MPSD (Multiple Program Single Data)

Cada processo executa um diferente programa, porém usando os mesmos dados. Os dados são passados de um processo para o outro em cada iteração. É o caso usado no processamento de áudio, onde o conjunto de dados passam por filtros distintos, sendo que cada filtro está rodando em um processo diferente.

6.4.4 MPMD (Multiple Program Multiple Data)

Um único processo (mestre) distribui as tarefas entre os processos escravos. Os processos escravos executam diferentes programas, mas também podem executar o mesmo programa. Cada processo escravo executa sua tarefa independentemente, e só se comunica com o processo mestre quando chegar ao final de sua tarefa. Cada processo escravo opera sobre dados diferentes.

7. METODOLOGIA

Uma chapa metálica em temperatura ambiente transmite o calor de uma fonte constante. Qual seria a temperatura em um ponto específico num dado tempo?

Partindo do questionamento descrito, dividiremos a placa em uma matriz 5x5, assim é possível determinar o valor da temperatura em uma dada célula dessa matriz. Quanto maior for à dimensão da matriz maior será a precisão do resultado logo exigirá um poder computacional maior.

Considerando que somente a primeira linha da matriz, que esta em contato direto com a fonte de calor, tem uma temperatura inicial de 100 graus e o restante a 0 grau. Adotando também que a cada iteração é o tempo necessário para aquecer a placa.

A nova temperatura de uma dada célula sofre influencia do calor de todas ao seu redor, portanto a somatória da temperatura dos elementos com a temperatura dela mesma, dividido pela quantidade de células participante dará a temperatura atual da célula.

100	100	100	100	100
25	5	α	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Definiremos a temperatura da célula α, todas as células sinaladas em não vermelho entra no calculo, sabendo que a temperatura atual de α é de 0°. Logo temos:

$$\alpha = \frac{100 + 5 + \ \alpha \ + 0 + 0}{5} = \frac{100 + 5 + \ 0 \ + 0 + 0}{5} = 21$$

Tanto para a codificação seqüencial quanto para paralela, o cenário de simulação é o mesmo uma matriz de 700 x 700 com 100.000 iterações. O resultado esperado é de que ambas apresentem a mesma resposta (ambas as matrizes com a mesma temperatura nas mesmas células), mas com tempo de processamento diferente (tempo paralelo < tempo seqüencial), pois as tarefas de calculo serão distribuídas em Threads na simulação paralela e na seqüencial será um único thread.

8. RESULTADOS FINAIS

Executando e simulando o mesmo cenário, uma matriz 300 x 300 com 150.000 iterações, obteve o seguinte imagem para a versão seqüencial o que representa a matriz em dois estados inicial e final.

FIGURA 5 REPRESENTAÇÃO DA MATRIZ INICIAL NO CÓDIGO SEQÜENCIAL SEQÜENCIAL

FIGURA 6 REPRESENTAÇÃO DA MATRIZ FINAL NO CÓDIGO

As imagens a seguir são referentes aos resultados iniciais e finais da versão paralela.

FIGURA 7 REPRESENTAÇÃO DA MATRIZ INICIAL NO CÓDIGO PARALELO PARALELO

FIGURA 8 REPRESENTAÇÃO DA MATRIZ FINAL NO CÓDIGO

Ambos os resultado são idêntico mesmo sendo executados em produzidos por fontes e sincronismo diferentes, produzindo a mesma matriz.

Executando 10 vezes cada uma das versões e marcando respectivo tempo de processamento, o gráfico a seguir faz um comparativo com temporal para cada versão.

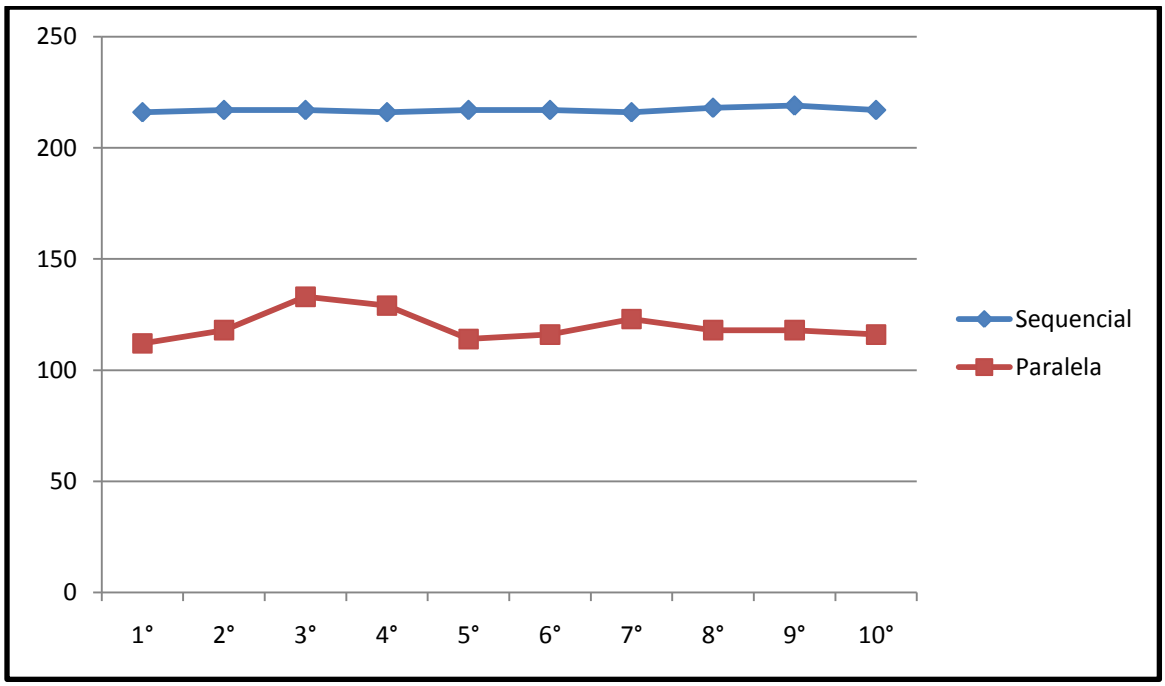


FIGURA 9 GRÁFICO COM AS DEZ MEDIÇÕES

Para obtermos o *speedup*, foi ignorando a média ponderada entre o 2° e o 9° medição e ignorando o primeira e a ultima medição, sendo assim a versão seqüencial tem um tempo mediando de 217,125 segundos já a paralela de 121,125 segundos.

9. CONCLUSÃO

Os resultados obtidos foram positivos. Entretanto deve-se enfatizar que nunca é possível obter um *speedup* perfeito em que o número de processo é diretamente proporcional ao *speedup* gerando um gráfico linear.

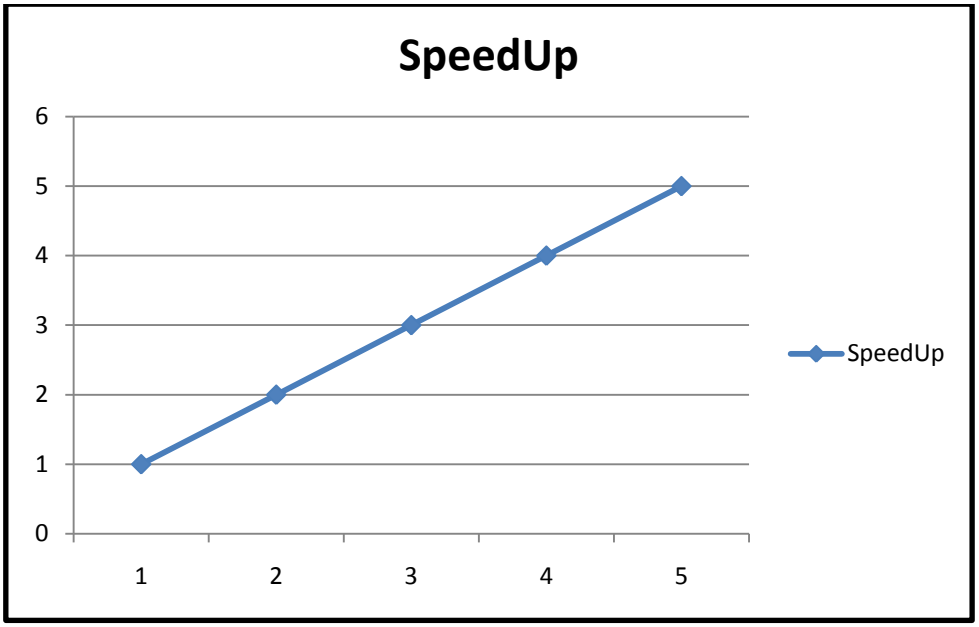


FIGURA 10 SPEEDUP PERFEITO.

Pois o mesmo depende diretamente da relação de computação e comunicação, e assim, deve-se ter um trabalho computacional de maior tempo em relação ao gasto com comunicação entre processadores. Situações muito comuns com a simulação da condução de calor com matrizes pequenas.

10. REFERÊNCIA BIBLIOGRAFICA

FOSTER, I. Designing and Building Parallel Programs. Addison-Wesley, Inc. 1995.

TANENBAUM, A.S. - Organização Estruturada de Computadores. Prentice Hall, Inc.2007

STALLINGS, W. – Arquitetura e Organização de Computadores. Prentice Hall, Inc.2002

TANENBAUM, A.S. - Sistemas Operacionais Modernos, , Prentice Hall, São Paulo , 2003, 2ª. ed.

COULOURIS, G., DOLLIMORE, J., KINBERG. Sistemas Distribuídos: Conceitos e Projetos. 1a. edição. Bookmann, 2007

TANENBAUM, A. VAN STEEN, M. Sistemas Distribuídos: Princípios e Paradigmas. 2ª. Edição. Prentice-Hall, 2007

11. PUBLICAÇÕES E EVENTOS

- V Congresso de Iniciação Cientifica da UNIARA (26 e 27 de Outubro de 2010)

12. ANEXOS

12.1 SOURCE CODE

12.1.1 CÓDIGO SEQÜENCIAL (SEQUENTIAL CODE)

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <png.h>
#include <gd.h>

#define ITERATIONS 150000
#define INIT_TEMP_HOT 100
#define INIT_TEMP_NEUTRAL 0
#define COLUMN 300
#define ROW    300
#define SIZE_IMAGE 10

struct Whole_matrix {
     double calculated[ROW][COLUMN];
     double updated[ROW][COLUMN];
};

enum TYPE_IMAGE {IMAGE_BEGIN, IMAGE_END};

void outputMatrixToPNG(double matrix[][COLUMN],enum TYPE_IMAGE type) {
          int maxvalue =100;
          int i,j, tot = 0;
   FILE *pngout;

   gdImagePtr im = gdImageCreate(COLUMN, ROW);

   gdImageColorAllocate(im, 255, 255, 255);

   for(i = 0; i < COLUMN; i++) {
      for (j =0; j < ROW; j++) {
         int valor = 100 - (int)(matrix[i][j]);
         int color = gdImageColorExact(im, valor, valor, valor) ;
         if (color == -1) {
            color = gdImageColorAllocate(im, valor, valor, valor);
         }
```

```
            gdImageSetPixel(im, j, i, color);
        }
    }

    if (IMAGE_BEGIN == type)
            pngout = fopen("imageIcSequencial_Begin.png", "wb");
    else if(IMAGE_END == type)
            pngout = fopen("imageIcSequencial_End.png", "wb");

    gdImagePng(im, pngout);

    fclose(pngout);
    gdImageDestroy(im);

    return;
}


void mountsEmptyArray(double array[][COLUMN]){
    int i, j;
    /* The initial temperature of the first row for the first
       column takes the value of 'INIT_TEMP_HOT' */
    for (j = 0; j < COLUMN; j++)
        array[0][j] = INIT_TEMP_HOT;

    /* The other cells receive zero values */
    for (i = 1; i < ROW; i++)  {
            for (j = 0; j < COLUMN; j++)
            array[i][j] = INIT_TEMP_NEUTRAL;
    }
}


void printArray(double array[][COLUMN]){
    int i, j;
    printf("\n \n");
    for (i = 0; i < ROW; i++)  {
        for (j = 0; j < COLUMN; j++)
            printf("| %f ",array[i][j]);
        printf("\n");
    }
}


/* The parameter 'i' represented the horizontal coordinate (row)
   and the parameter 'j' the vertical coordinate (column) */
double calculatesNewTemperature(double array[][COLUMN], int i, int j){
     int computes_dividers = 1;
     double accumulates_current_temp = array[i][j];
```

```
    // computes values of cells available in row
    if(i == 0){
        accumulates_current_temp += array[i+1][j];
        computes_dividers += 1;
    }else if(i == ROW-1){
        accumulates_current_temp += array[i-1][j];
        computes_dividers += 1;
    }else if(i > 0 && i < ROW-1){
        accumulates_current_temp += array[i-1][j];
        accumulates_current_temp += array[i+1][j];
        computes_dividers += 2;
    }

    // computes values of cells available in column
    if(j == 0){
        accumulates_current_temp += array[i][j+1];
        computes_dividers += 1;
    }else if(j == COLUMN-1){
        accumulates_current_temp += array[i][j-1];
        computes_dividers += 1;
    }else if(j > 0 && j < COLUMN-1){
        accumulates_current_temp += array[i][j-1];
        accumulates_current_temp += array[i][j+1];
        computes_dividers += 2;
    }

    return (accumulates_current_temp / computes_dividers);
}


void calculateHeatTransfer(double updated[][COLUMN], double calculated[][COLUMN]){
  int i,j;
  for(i = 1; i < ROW; i++){
    for(j = 0; j < COLUMN; j++)
          updated[i][j] = calculatesNewTemperature(calculated, i, j);
  }
}

int main(int argc, char **argv) {
        time_t t1,t2;
        (void) time(&t1);

        struct Whole_matrix matrix;
        mountsEmptyArray(matrix.calculated);
        mountsEmptyArray(matrix.updated);

        if (ITERATIONS%2 == 0){
                outputMatrixToPNG(matrix.calculated,IMAGE_BEGIN);
        }
        else{
```

```
		outputMatrixToPNG(matrix.updated,IMAGE_BEGIN);
	}

	int cont = 0;
	while(cont <= ITERATIONS){
		if(cont%2 == 0)
			calculateHeatTransfer(matrix.updated, matrix.calculated);
		else
			calculateHeatTransfer(matrix.calculated, matrix.updated);
		cont++;
	}

	(void) time(&t2);
	printf("\nO tempo de processamento foi de %f segundos.", (double) t2-t1);

	if (ITERATIONS%2 == 0){
		outputMatrixToPNG(matrix.calculated,IMAGE_END);
	}
	else{
		outputMatrixToPNG(matrix.updated,IMAGE_END);
	}
	return 0;
}
```

12.1.2 Código Paralelo (Parallel Code)

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <png.h>
#include <gd.h>

#define ITERATIONS 150000
#define INIT_TEMP_HOT 100
#define INIT_TEMP_NEUTRAL 0
#define COLUMN 300
#define ROW    300
#define CHUNKSIZE 5
#define SIZE_IMAGE 10

time_t t1, t2;

struct Whole_matrix {
	double calculated[ROW][COLUMN];
	double updated[ROW][COLUMN];
};
```

```
enum TYPE_IMAGE {IMAGE_BEGIN, IMAGE_END};

void outputMatrixToPNG(double matrix[][COLUMN],enum TYPE_IMAGE type) {
	int maxvalue =100;
	int i, j, tot = 0;
	FILE * pngout;

	gdImagePtr im = gdImageCreate(COLUMN, ROW);

	gdImageColorAllocate(im, 255, 255, 255);

	for (i = 0; i < COLUMN; i++) {
		for (j =0; j < ROW; j++) {
			int valor = 100 - (int)(matrix[i][j]);
			int color = gdImageColorExact(im, valor, valor, valor) ;
			if (color == -1) {
				color = gdImageColorAllocate(im, valor, valor, valor);
			}
			gdImageSetPixel(im, j, i, color);
		}
	}

	if (IMAGE_BEGIN == type)
		pngout = fopen("imageIcParallel_Begin.png", "wb");
	else if(IMAGE_END == type)
		pngout = fopen("imageIcParallel_End.png", "wb");

	gdImagePng(im, pngout);

	fclose(pngout);

	gdImageDestroy(im);

	return;
}

void mountsEmptyArray(double array[][COLUMN]) {
	//dividir em threads a criancao da matriz
	int i, j;
	/* The initial temperature of the first row for the first
	 column takes the value of 'INIT_TEMP_HOT' */
	#pragma omp parallel for shared(array) private(j) schedule(static,CHUNKSIZE)
		for (j = 0; j < COLUMN; j++)
			array[0][j] = INIT_TEMP_HOT;

	#pragma omp parallel for shared(array) private(i,j) schedule(static,CHUNKSIZE)
		for (i = 1; i < ROW; i++) {
			for (j = 0; j < COLUMN; j++)
				array[i][j] = INIT_TEMP_NEUTRAL;
		}
```

```
}

void printArray(double array[][COLUMN]) {
	int i, j;
	printf("\n \n");
	for (i = 0; i < ROW; i++) {
		for (j = 0; j < COLUMN; j++)
			printf("| %f ", array[i][j]);
		printf("\n");
	}
}

/* The parameter 'i' represented the horizontal coordinate (row)
 and the parameter 'j' the vertical coordinate (column) */
double calculatesNewTemperature(double array[][COLUMN], int i, int j) {
	int computes_dividers = 1;
	double accumulates_current_temp = array[i][j];

	// computes values of cells available in row
	if (i == 0) {
		accumulates_current_temp += array[i+1][j];
		computes_dividers += 1;
	} else if (i == ROW-1) {
		accumulates_current_temp += array[i-1][j];
		computes_dividers += 1;
	} else if (i > 0 && i < ROW-1) {
		accumulates_current_temp += array[i-1][j];
		accumulates_current_temp += array[i+1][j];
		computes_dividers += 2;
	}

	// computes values of cells available in column
	if (j == 0) {
		accumulates_current_temp += array[i][j+1];
		computes_dividers += 1;
	} else if (j == COLUMN-1) {
		accumulates_current_temp += array[i][j-1];
		computes_dividers += 1;
	} else if (j > 0 && j < COLUMN-1) {
		accumulates_current_temp += array[i][j-1];
		accumulates_current_temp += array[i][j+1];
		computes_dividers += 2;
	}

	return (accumulates_current_temp / computes_dividers);
}

void calculateHeatTransfer(double updated[][COLUMN],double calculated[][COLUMN]) {
	int i, j;
	#pragma omp parallel for shared(updated, calculated) private (i,j) schedule(static,CHUNKSIZE)
```

```
		for (i = 1; i < ROW; i++) {
	//#pragma omp parallel for shared (updated, calculated, i) private(j) schedule(dynamic)
		for (j = 0; j < COLUMN; j++)
			updated[i][j] = calculatesNewTemperature(calculated, i, j);
	}
}

int main(int argc, char *argv[]) {
	//Thread master
	(void) time(&t1);

	struct Whole_matrix matrix;
	#pragma omp master
	{
		mountsEmptyArray(matrix.calculated);
		mountsEmptyArray(matrix.updated);

		if (ITERATIONS%2 == 0){
			outputMatrixToPNG(matrix.calculated,IMAGE_BEGIN);
		}
		else{
			outputMatrixToPNG(matrix.updated,IMAGE_BEGIN);
		}

		int cont = 0;
		while (cont <= ITERATIONS) {
			if (cont%2 == 0)
				calculateHeatTransfer(matrix.updated, matrix.calculated);
			else
				calculateHeatTransfer(matrix.calculated, matrix.updated);
			cont++;
		}

		(void) time(&t2);
		printf("\nO tempo de processamento foi de %f segundos.\n", (double)t2-t1);

		if (ITERATIONS%2 == 0){
			outputMatrixToPNG(matrix.calculated,IMAGE_END);
		}
		else{
			outputMatrixToPNG(matrix.updated,IMAGE_END);
		}
	}
	return 0;
}
```

12.2 TUTORIAL

*Tutorial publicado no portal **viva o linux**, o mesmo pode ser conferido no seguinte link:*

Instalação e integração do Intel C++ Compiler com Eclipse C/C++ na plataforma Linux.

Pré requisitos:

- JDK: Bibliotecas de dependência para ambos os softwares.
  http://java.sun.com/javase/downloads/index.jsp

- Intel Compiler: Framework de multi-processamento, utilizaremos a API na versão 11.1, existe a duas distribuições da Intel, Comercial e não Comercial.
  http://software.intel.com/en-us/intel-compilers/

- Eclipse: IDE de desenvolvimento, utilizaremos com suporte para C/C++
  http://www.eclipse.org/downloads/

É possível fazer o download e as instalações desses softwares atrevem do seu *gerenciador de pacotes*, vale ressaltar que cada distribuição Linux tem um gerenciador de pacotes (ex: OpenSuse -> Zypper; Ubuntu -> Apt-get;).

Obs: Existem algumas distribuições Linux que tem uma interface gráfica para a manipulação dos softwares no SO, aqui faremos todas as operações pelo Terminal, pois este é comum a todos.

Utilizo a plataforma OpenSuse na versão 11.0, é possível que os pacotes de distribuição seja diferente conforme a sua distribuição.

Instalando Intel Compiler C/C++ 11.1:

Para evitarmos futuras falhas de dependências nas instalações dos softwares, devemos fazer um update em nosso gerenciador de pacotes. Para isso usaremos o comando #zypper update, é sempre recomendado reiniciar o SO após todas as instalações e update.

Agora instalaremos a JDK, pois o Intel Compiler e o Eclipse necessitam dele para serem executados. Com o comando #zypper install Java o download será iniciado e em seguida a instalação (caso já tenha JDK execute #zypper update Java).

Reinicie o computar para continuar.

Os arquivos do download da Intel certamente terá extensão tar.gz, logo será necessário descompactá-lo. Pelo terminal dirija-se ate o diretório onde estam os arquivos compactados. O seguinte comando descompactará o arquivo tar –xzvf name-of-downloaded-file Existem duas formas de executar a instalação da API Intel, com ou sem o arquivo Silent.

Instalando com arquivo Silent

Para fazer essa instalação você terá que ter um arquivo de licença da Intel, caso você não tenha acesse o link a seguir https://registrationcenter.intel.com e registre seu serial, você recebera um email com o arquivo de licença.

Será necessário criar um arquivo de configuração (<name_file>.ini), nele faremos a seguinte configuração que são necessária para a instalação.

ACTIVATION=<nome do seu arquivo de licença Intel>
CONTINUE_WITH_INSTALLDIR_OVERWRITE=yes
CONTINUE_WITH_OPTIONAL_ERROR=yes
PSET_INSTALL_DIR=<diretório de sua preferência, local onde será instalado a API>
INSTALL_MODE=RPM (caso não queria usar RPM, mude para NONRPM)
ACCEPT_EULA=accept

Para iniciar a instalação do Intel Compiler execute o seguinte comando, certifique-se de que

esta dentro da pasta descompactada: ./install.sh: --silent <seu_diretorio>/my_install.ini

Instalando sem arquivo Silent

Algumas das configurações feitas no com o arquivo Silent serão solicitado durante a instalação. Neste modo é importante ter conexão com a internet. Para iniciar a instalação execute o comando: ./install.sh

Informações sobre a instalação serão apresentadas, para continuar pressione [Enter].

Como de costume um contrato será apresentado, após a leitura do mesmo digite *"accept"*, para aceitar o acordo e prosseguir com a instalação.

Cinco opções de instalação são apresentadas, a opção de nosso enteresse é a segunda. Logo digite *"2"* e pressione [Enter] para confirmar a opção escolhida. O Setup da Intel solicitará um serial com o seguinte formato *xxxx-xxx*, informe-o e novamente pressione [Enter]. Nesse momento a instalação verificará a existência do serial em sua base de dados, por esse motivo é necessário ter conexão com a internet.

Após a validação de sucesso do serial, a instalação continuará e finalizará automaticamente.

Obs: Caso acuse falta de dependências com do g++, execute o comando (fora da instalação) # zypper install gcc-c++

Instalando Eclipse C/C++

Assim como os pacotes da Intel o arquivo do download do eclipse terá extensão *tar.gz*, para descompactar o arquivo execute o comando tar –xzvf name-of-downloaded-file

Para executar o eclipse pressione [Alt+F2], uma janela se abrirá, digite *"eclipse"* e pressione [Enter], o software iniciará!

Integração do IDE com a API

Click em "Help -> Softwares Update" que esta no menu principal do eclipse, em seguida click em "find and instalation". Na janela que abrica click em Obs: De acordo com a versões do eclipse(Galileo, Ganymede e

Europa) isso poderá mudar, mas para efetuar a integração de ambos o caminho inicial é o mesmo “Help -> Software Update->Find and Install”.Uma janela abrirá, click em “add”; Indicando pelo browser, aponte para o diretório onde o Intel Compiler C++ foi instalado. Precionando “ok” o browser fechará, agora o botão “install” esta disponível. Pronto, seu eclipse já tem suporte para programação paralela utilizando a API da Intel.

Na tentativa de integrá-lo o eclipse poderá acusa falta de dependência, os plugins. Muitos deles serão encontrados no link. http://eclipseplugincentral.com Flavio Luiz S. Souza autor desse guia é aluno do Centro Universitário de Araraquara em Engenharia de Computação. Artigo desenvolvido em projeto de iniciação cientifica orientado pelo Prof.Msc. Rodrigo Daniel Malara.

Referencia

[Documentação do Fabricante, 2009] Intel C++ Compiler Professional Edition 11.1 for Linux* Installation Guide and Release Notes

12.3 ARTIGO

Implementação de algoritmo paralelo para simulação de condução de calor em objeto bidimensional utilizando o método de diferenças finitas.

FLÁVIO LUIZ DOS SANTOS DE SOUZA

Prof. MSc. Rodrigo Daniel Malara

CENTRO UNIVERSITÁRIO DE ARARAQUARA – UNIARA

Departamento das Ciências da Administração e Tecnologia

Curso Engenharia de Computação

RESUMO

A computação de alto desempenho tem sido utilizada como recurso para resolver problemas de grande complexidade em que na maioria das vezes não se tem um poder computacional disponível. Apesar de se tornarem áreas distintas de um mesmo seguimento, são difíceis de separar no conceito ou na analise de computação paralela hardware e software. Contudo, as super maquinas apresentam dificuldade para resolver vários quebra-cabeças da física e da matemática, como a previsão climática em uma escala inferior a 50 km², em um tempo hábil. Com o objetivo de utilizar programação concorrente, este trabalho propõe estudar os paradigmas de sistemas distribuídos, que permite utilizar toda gama de potencial da maquina disponível, exigindo assim o domínio das diferentes configurações de hardware existente no mercado e o conceito que envolve a distribuição de tarefas e a comunicação interprocessos a fim de simular um problema físico. Realizando uma simulação de condução de calor em uma chapa metálica, através de cálculos matemáticos, deve-se gerar um speedup linear e positivo com programação seqüencial e outra concorrente. De forma que o resultado apresente um aumento da performance que seja diretamente proporcional ao aumento do recurso computacional empregado na execução.

***Palavras chaves**: alto desempenho, concorrente, distribuída*

ABSTRACT

Techniques for high performance computing and concurrent programming are used to solve problems of great complexity and therefore high computational cost. Existing computers are not able to produce results in time to various problems of physics and mathematics, for example, climate prediction on scales smaller than 50 km^2. Uses the principles of concurrent programming involves knowledge about programming that involves the distribution of tasks and interprocess communication and this paper is to study the different hardware configurations on the market for high performance computing, to study the paradigms of concurrent programming allowing the developer to use the computational power available for such equipment and produce a proof of concept using the knowledge acquired. This prototype needs to get linear speedup positive and so the performance increase is directly proportional to the increase of computational resources used for its implementation.

Keywords: High performance; Concurrent.

1. INTRODUÇÃO

Técnicas de computação de alto desempenho e programação concorrente são utilizadas para resolver problemas de grande complexidade e conseqüentemente alto custo computacional. Os computadores atuais não são capazes de produzir resultados em tempo hábil para vários problemas da física e da matemática, por exemplo, a previsão climática em escalas menores que 50 km^2.

As utilizações de princípios de programação concorrente envolvem conhecimentos adicionais sobre programação que envolve a distribuição de tarefas e a comunicação interprocessos e este trabalho se propõe a estudar as diferentes configurações de hardware existentes no mercado para computação de alto desempenho, estudar os paradigmas de programação concorrente que permitam ao desenvolvedor utilizar o poder computacional disponibilizado por tais equipamentos e produzir uma prova de conceito utilizando o conhecimento adquirido.

Tal protótipo precisa obter speedup positivo e linear de forma que o aumento de performance seja diretamente proporcional ao aumento de recursos computacionais empregados para a sua execução.

2. OBJETIVO

- Produzir uma revisão bibliográfica acerca do hardware e técnicas de programação e frameworks disponíveis para elaborar programas concorrentes adequados para computadores de alto desempenho.
- Programar um programa paralelo usando um framework de paralelização para simular o fenômeno físico "Condução de calor" em uma chapa metálica utilizando o método numérico de diferenças finitas.
- Criar representações gráficas da chapa através de imagens antes e após a aplicação do algoritmo paralelo.
- Calcular o *speedup* da solução paralela.
- Documentar os resultados obtidos e publicar artigos.

3. METODOLOGIA

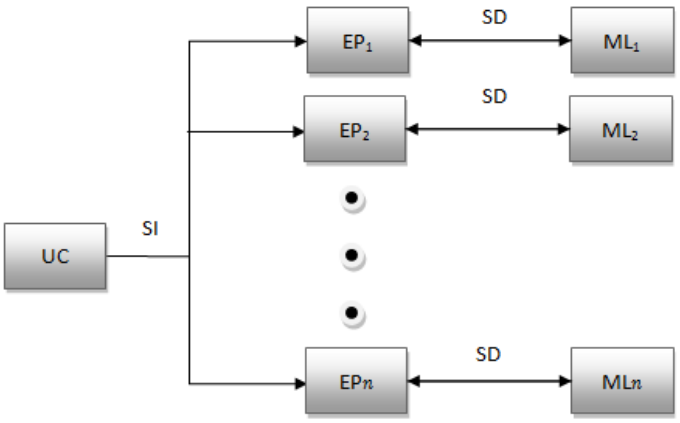
A metodologia desenvolvida tem como objetivo o estudo de computação de alto desempenho através da simulação de transferência de calor por diferenças finitas. Para a simulação é importante conhecer a *Lei de Fourier*, arquitetura SIMD (Single Instruction Mutiple Data) e em um nível de abstração maior SPMD (Single Program Multiple Data).

A taxonomia de Flynn abrange quatro modelos de arquitetura SISD, SIMD, MISD e MIMD o qual todos são baseados nos fluxos de instruções e dados. A proposta de Flynn é ainda a forma mais comum de classificar sistemas de processamento paralelo.

Flynn's Taxonomy	**Single Instruction**	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SIMD *(SINGLE INSTRUCTION MULTIPLE DATA)*

Esta classificação corresponde ao processamento de vários dados sob o comando de apenas uma instrução. Cada elemento de processamento tem uma memória de dados associada. Nesta classe estão os processadores vetoriais e matriciais.



Outros modelos de programação

Os principais modelos de parametrização são o *Message passing*, *DataParallelism* e *Shared Memory*, cada um desse paradigma tem sua flexibilidade e mecanismos de integração de tarefa.

- *Message passaing*: certamente é o modelo mais utilizado hoje. Cria-se varias tarefas com nome único, cada uma encapsula seus dados locais. As tarefas enviam e recebem mensagens de outras tarefas, fazendo assim a comunicação entre si.

- *DataParallelism*: o modelo de paralelismo é atingido na execução em cada processador de uma parte de um único processo distribuído, definido assim uma arquitetura SIMD.

- *Shared Memory:* compartilhando de um mesmo espaço de memória onde é possível ler e escrever de forma assíncrona. A grande vantagem deste modelo é a agilidade de manipulação de dados, entretanto o seu gerenciamento se tornar algo custoso e difícil.

Neste trabalho utiliza o paralelismo de dados e para isso usamos o OpenMP.

Parametrização com OpenMP

Apesar de se tornarem áreas distintas de um mesmo seguimento, hardware e software são difíceis de separar no conceito ou na analise de computação paralela de alto desempenho. O excelente desempenho do paralelismo depende de ambos, não ajudará em nada ter uma boa arquitetura paralela se não tiver um bom proveito do algoritmo e vice-versa!

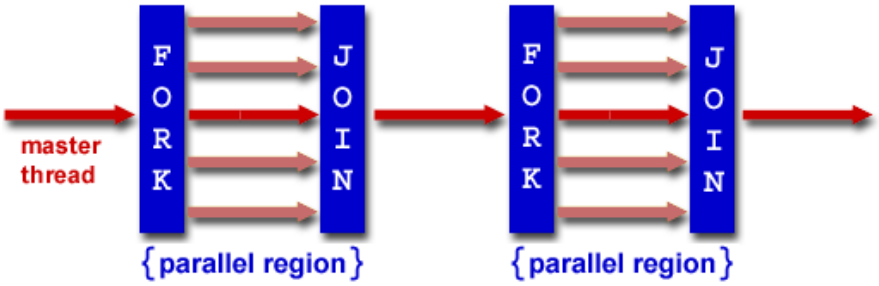
Neste projeto utilizaremos *OpenMP*, que é uma API de parametrização de memória compartilhada (todas tarefas dividem o mesmo endereçamento de memória, sendo executa de forma assíncrona).

A arquitetura do OpenMP em SPMD, cada processo roda exatamente o mesmo programa (embora algumas partes do programa sejam executadas somente pelo processo mestre), mas com um conjunto diferente de dados. Dessa forma, os dados são divididos pelo processo mestre e distribuídos entre os processos escravos, cada processo receberá informações diferentes.

Formada por um conjunto de diretivas compiláveis interpretadas em C/C++ e Fortran, permitindo total controle da parametrização, o framework utiliza o modelo de execução fork-join:

FORK: O thread mestre cria um time de segmentos paralelos. As instruções que estão dentro da construção da região paralela são executadas em paralelo pelas diversos threads do time

JOIN*:* No termino do processo paralelo é sincronizado todas as informações compartilhadas deixando-as somente ativa o *master thread*.



Toda a programação OpenMP é interpretado por diretivas de compilação, o prefixo das diretivos em Fortran são diferentes das C/C++ e para este caso deve-se conter *#pragma omp*, que indica ao compilador que o troche de código seguinte deve ser gerenciado pela API.

- **#pragma omp master**: identifica uma seção que deve ser executado apenas por um thread mestre.

- **#pragma omp parallel**: instrui explicitamente para o compilador o código que deverá ser parametrizado.

- **#pragma omp parallel for**: informa ao compilador que deve-se distribuir as iterações por diferentes threads

- **shared**: todos os dados , assinalados como shared, serão compartilhado. Deixando-o visível e acessível por todos os threads.

- **private**: cada dado, assinalado com private, terá uma copia local e será usado temporariamente em cada região paralel, ate o fim da thread em que a pertence.

- **schedule**: Muito utilizado em loops, são atribuídos de acordo com um dos 3 tipos agendamento (static, dynamic e guided)

    - Static: As iterações serão divididas igualmente entre os threads, mas é necessário definir um padrão de de loop agendados.
    - Dynamic: Uma vez que o thread termina uma iteração uma nova é atribuída a ele.
    - Guided: Assim como a Dynamic, uma nota iteração é atribuída a um thread, mas uma quantidade mínima deve ser especificada.

O principal objetivo da parametrização é diminuir o tempo de processamento comparado com uma versão serial do algoritmo. A métrica mais utilizada é o fator *speed-up*, representa o ganho de velocidade de processamento que uma aplicação apresenta diante a *n* processadores. A equação que define a métrica é:

$$Sn = \frac{Ts}{Tn} \qquad \begin{matrix} Ts - Temp\ Serial \\ Tn - Tempo\ Paralelo \end{matrix}$$

Porem nem todo o código são parametrizados, é possível de se encontrar trecho que são executados por um único thread para este caso é utilizado a *Lei de Amdahl*.

4. DESENVOLVIMENTO

Uma chapa metálica em temperatura ambiente transmite o calor de uma fonte constante. Qual seria a temperatura em um ponto específico num dado tempo?

Partindo do questionamento descrito, dividiremos a placa em uma matriz 5x5, assim é possível determinar o valor da temperatura em uma dada célula dessa matriz. Quanto maior for à dimensão da matriz maior será a precisão do resultado logo exigirá um poder computacional maior.

Considerando que somente a primeira linha da matriz, que esta em contato direto com a fonte de calor, tem uma temperatura inicial de 100 graus e o restante a 0 grau. Adotando também que a cada iteração é o tempo necessário para aquecer a placa.

A nova temperatura de uma dada célula sofre influencia do calor de todas ao seu redor, portanto a somatória da temperatura dos elementos com a temperatura dela mesma, dividido pela quantidade de células participante dará a temperatura atual da célula.

100	100	100	100	100
25	5	α	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

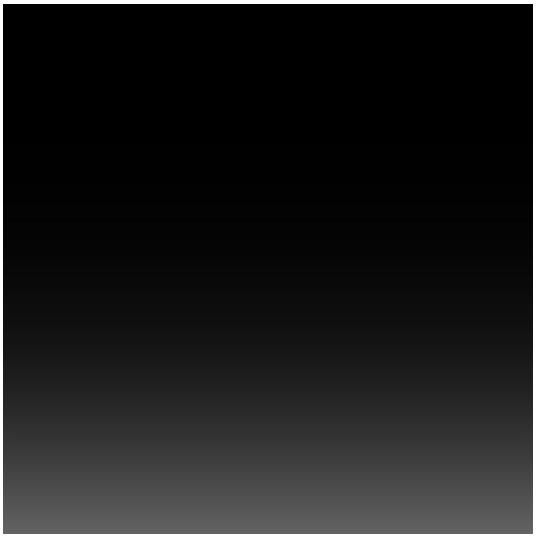
Definiremos a temperatura da célula α, todas as células sinaladas em não vermelho entra no calculo, sabendo que a temperatura atual de α é de 0°. Logo temos:

$$\alpha = \frac{100 + 5 + \alpha + 0 + 0}{5} = \frac{100 + 5 + 0 + 0 + 0}{5} = 21$$

Tanto para a codificação seqüencial quanto para paralela, o cenário de simulação é o mesmo uma matriz de 700 x 700 com iterações 100.000. O resultado esperado é de que ambas apresentem a mesma resposta (ambas as matrizes com a mesma temperatura nas mesmas células), mas com tempo de processamento diferente (parallel timer < seqüencial timer), pois as tarefas de calculo serão distribuídas em Threads na simulação paralela e na seqüencial será um único thread.

5. RESULTADOS

Para comprovar o resultado são criadas imagens PNG que representaram as matrizes geradas em cada cenário. O valor numérico de cada célula da matriz representa uma tonalidade entre as cores pretas e brancas. Quanto menor a temperatura mais próxima do branco ela estará e quanto maior a temperatura mais próxima do preto ela ficará.



Se executar a simulação em processadores mais atuais como o Core i3, Core i5 ou Core i7, o *speedup* apresentado será menor, pois o número de threads é maior do que usado na simulação concorrente, a configuração utilizada é equivalente a quantidade de núcleos existentes no processador em que a simulação ocorre (processador do ambiente Core Duo).

6. CONSIDERAÇÔES FINAIS

Apesar de se tornarem áreas distintas de um mesmo seguimento, hardware e software são difíceis de separar no conceito ou na analise de computação paralela de alto desempenho.

O excelente desempenho de do paralelismo depende de ambos, não ajudará em nada ter uma boa arquitetura paralela se não tiver um bom proveito do algoritmo e vice-versa. Contudo, os resultados obtidos foram positivos e agraveis, entretanto deve-se enfatizar que nem sempre é possível obter um *speed-up* perfeito. Pois o mesmo depende diretamente da relação de computação e comunicação, deve-se ter um trabalho computacional de maior tempo em relação ao gasto com comunicação entre processadores.

7. BIBLIOGRAFIA

FOSTER, I. *Designing and Building Parallel Programs*. Addison-Wesley, Inc. 1995.

TANENBAUM, A.S. - *Organização Estruturada de Computadores*. Prentice Hall, Inc.2007

STALLINGS, W. – *Arquitetura e Organização de Computadores. Prentice Hall*, Inc.2002

TANENBAUM, A.S. - *Sistemas Operacionais Modernos, Prentice Hall*, São Paulo , 2003, 2ª. ed.

COULOURIS, G., DOLLIMORE, J., KINBERG. *Sistemas Distribuídos: Conceitos e Projetos*. 1a. edição. Bookmann, 2007

TANENBAUM, A. VAN STEEN, *M. Sistemas Distribuídos: Princípios e Paradigmas*. 2ª. Edição. Prentice-Hall, 2007