Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

**JEFFERSON DOS SANTOS CORREA**

# ESTUDO SOBRE LINGUAGENS DE PROGRAMAÇÃO PARALELA E OS CONCEITOS CHAVE DE PARALELISMO EM NIVEL DE PROGRAMAÇÃO

**ASSIS – SP**

**2013**

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

**JEFFERSON DOS SANTOS CORREA**

# ESTUDO SOBRE LINGUAGENS DE PROGRAMAÇÃO PARALELA E OS CONCEITOS CHAVE DE PARALELISMO EM NIVEL DE PROGRAMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Municipal do Ensino Superior de Assis – IMESA e Fundação Educacional do Município de Assis – FEMA, como requisito para a obtenção do Certificado de Conclusão.

**Orientador:** Me Douglas Sanches Cunha

**Área de Concentração:** Computação Paralela e Distribuída, Arquiteturas Paralelas.

**ASSIS – SP**

2013

**DEDICATÓRIA**

Dedico esse trabalho a minha família, amigos e professores que me ajudaram direta ou indiretamente nessa conquista tão importante.

“Saber não é o bastante, é preciso aplicar. Querer não é o bastante, é preciso fazer.”

Bruce Lee

**AGRADECIMENTOS**

Aos meus pais Daniel e Cleusa e a minha irmã Jaqueline, pelo amor, conselhos e ajuda em todas as fases da minha vida.

A meus colegas de classe e amigos, pelas dicas, conselhos, ensinamentos e risadas desses anos de convivência em dias bons e ruins, porque sem eles não teria conseguido chegar até aqui.

Ao meu orientador Douglas, pela ajuda e ensinamentos, que ficarão para sempre guardados, não só profissionalmente, mas também para a vida.

## RESUMO

Na atualidade se torna cada vez mais comum a interação com arquiteturas completamente paralelas, tanto para servidores, quanto para computadores pessoais, porém existe uma grande falta de programas adequados para esse tipo de arquitetura. Este trabalho tem como foco apresentar linguagens com suporte para o paralelismo em nível de programação, assim com arquiteturas paralelas que podem suportar suas subdivisões em *threads* e processos tal como características de suas utilizações, além dos conceitos de paralelismo.



**Palavras Chave**: Paralelismo; Arquiteturas; Programação Paralela.

## ABSTRACT

These days, it has become more and more common the interaction with completely parallel architectures, both for servers and personal computers. However, there exists a lack of adequate software for this kind of architecture. This paper focuses in the introduction of programming languages that support parallelism, and parallel architectures that support threads subdivision and processes, and their characteristics such as their uses, besides the concept of parallelism.

**Keywords**: Parallelism; Architectures; Parallel Programming.

# LISTA DE ILUSTRAÇÕES

# LISTA DE CÓDIGOS

## LISTA DE ABREVIATURAS E SIGLAS

**SO:** *Sistema Operacional*

**SISD:** *Single Instruction Single Data*

**SIMD:** *Single Instruction Multiple Data*

**MISD:** *Multiple Instruction Single Data*

**MIMD:** *Multiple Instruction Multiple Data*

**PC:** *Personal Computer*

**IDE:** *Integrated  Development Environment*

**HPC:** *High-performance computing*

**HA :** *High-Availability*

**SUMÁRIO**

# 1. INTRODUÇÃO

Com a disseminação da tecnologia e o aumento da dependência humana por processos cada vez mais complexos, como nas áreas de pesquisa e industriais, torna-se indispensável o uso de computadores mais potentes. Partindo dessa necessidade surgiram os supercomputadores que possuem vários processadores físicos e lógicos, porém estes possuem também um alto custo. Como alternativa para tais, surgiu a ideia de se agregar vários computadores de médio desempenho para que, juntos, tivessem suas capacidades computacionais somadas, criando assim o conceito de *cluster*, uma solução mais barata e igualmente potente comparada aos supercomputadores. Com o surgimento de supercomputadores e *clusters* com vários processadores criou a problemática de uma programação paralela para fazer pleno uso da capacidade de processamento simultâneo dessas máquinas (SCHEPKE, 2009).

## 1.1. OBJETIVOS

Este trabalho tem por objetivo realizar um estudo sobre de programação paralela, ambientes computacionais em que se pode fazer uso de tal tecnologia e o desenvolvimento de um software para apresentar o desenvolvimento, compilação e execução de sua capacidade computacional de um ambiente de arquitetura paralela.

## 1.2. JUSTIFICATIVAS

A escolha desse tema deve-se ao grande crescimento da computação paralela e avanços de tecnologias multicore, possibilitando um estudo mais detalhado que contribua significativamente com o desempenho de ambientes computacionais que permitam o uso dessa tecnologia. O estudo sobre paralelismo também contribui para que sejam desenvolvidos outros trabalhos futuros, visando ampliar ainda mais o conhecimento sobre técnicas e comportamentos de ambientes com paralelismo.

## 1.3. MOTIVAÇÃO

A pesquisa sobre programação paralela possibilita um grande aumento no desempenho e tempo de resposta para vários tipos de *softwares*, refletindo assim em uma melhora para várias áreas de estudo, como na meteorologia, telecomunicação e etc. Ainda cria a abertura para o desenvolvimento de outros trabalhos correlativos sobre esse tema que está em constante crescimento, porém ainda pouco estudado.

## 1.4. PERSPECTIVAS DE CONTRIBUIÇÃO

Este trabalho tem como objetivo contribuir para o aumento de estudos sobre computação paralela e seus ambientes de execução, sobre os conceitos desse assunto e com uma aplicação prática, criando ainda a possibilidade da publicação de um artigo acerca do assunto tratado.

## 1.5. METODOLOGIA DE PESQUISA

Como metodologia de pesquisa para esse trabalho serão utilizados livros, artigos e monografias sobre o tema para aprofundar os conhecimentos sobre os conceitos de computação paralela e suas aplicações práticas. Para finalizar será desenvolvido um *software* como aplicação dos conhecimentos adquiridos na pesquisa.

# 2. COMPUTAÇÃO PARALELA

A programação concorrente teve origem em 1962, com o advento de canais que eram dispositivos de controle independentes e que tornavam possível para um processador executar um novo programa enquanto as operações de entrada/saída estavam sendo executadas em nome de outra aplicação suspensa. Entretanto programação concorrente foi incialmente um conceito de desenvolvedores de sistemas operacionais (SO). No fim dos anos 60 desenvolvedores de *hardware* desenvolveram os primeiros computadores com múltiplos processadores e isso não foi apenas um desafio para os desenvolvedores de SO, mas também uma oportunidade a ser explorada por programadores (ANDREWS, 1999). No fim dos anos 70 e inicio dos anos 80 surgiram as primeiras redes de computadores, tornando possível o desenvolvimento de programas distribuídos, que poderiam ter seu processamento divido entre os computadores da rede.

Para o pleno aproveitamento da capacidade computacional de uma arquitetura com múltiplas unidades de processamento é necessário se fazer uso de concorrência e programação paralela. Atualmente os computadores têm evoluído altamente sua capacidade de processamento, contudo a velocidade de acesso à memória não teve a mesma evolução, por isso a paralelização das tarefas se tornou uma forte opção para se alcançar um maior desempenho das maquinas atuais (PINHO, 2012). A ideia de se dividir tarefas em vários processadores não é nova, porém se tornou algo válido somente com os avanços recentes de *hardware* e *software*. Os computadores em geral tiveram uma grande evolução nos últimos tempos com o aparecimento de máquinas com vários processadores, também a velocidade de comunicação entre os processadores e computadores vem apresentando uma grande evolução permitindo que estes processadores troquem informações com rapidez. Essa evolução não se limita somente ao *hardware* e a arquitetura, como também se estende aos programas que possibilitam esta comunicação (SCHIOZER, 2003).

O paralelismo em potencial consiste em um programa ou instrução que pode ser dividido em várias tarefas e cada uma destas pode ser processada em qualquer ordem, sem que isso altere o resultado final do programa. O paralelismo em si ocorre quando as várias tarefas do programa possam ser executadas independentemente por vários processadores e simultaneamente, para isso é preciso primeiramente que os programas sejam divididos em processos, cada processo podendo ser executado ao mesmo tempo. Apesar de ser uma abordagem que visa o melhoramento do desempenho e diminuição do tempo gasto para a execução de uma tarefa nem todo o código pode ser paralelizado. Seguindo esse conceito foi criada a Lei de Amdahl, que calcula o ganho de desempenho por processos paralelos. O nome Amdahl é derivado do nome de seu criador Gene Myron Amdahl, essa formula calcula o *speedup* ganho pelo uso de vários processadores em programação paralela, o cálculo do *speedup* é representada pela seguinte fórmula:

$$Speedup = \frac{1}{\frac{1-s}{P} + s}$$

Na qual *Speedup* é o desempenho ganho, enquanto S é a parte serial do programa, sempre menor que 1, ou seja, a parte que não pode ser paralelizada. Sendo assim 1 – S representa a parte que pode ser paralelizada, P representa o número de processadores disponível. Seguindo essa estrutura pode-se calcular o tempo de processamento com a fórmula:

$$T_P = \frac{(1-s)}{P} + s$$

Segundo Amdahl o tempo de processamento Tp pode ser calculado com a divisão dos processos paralelizáveis (1 – S) divididos pela quantidade de processadores, somado aos processos seriais.

Os programas paralelos podem ser divididos conforme sua abordagem. Podemos ter o paralelismo por troca de mensagem ou por espaço de memória compartilhada. O método de troca de mensagem entre processadores consiste em quando os processos estão localizados em processadores distintos, cada qual sem acesso a uma mesma área de memória compartilhada, então deve haver o envio de primitivas de comunicação. Em contrapartida podemos ter o paralelismo por memória compartilhada, onde todos os processadores tem acesso à mesma área de memória. O esquema mostrado na figura 1 sugere uma abordagem de troca de mensagens.

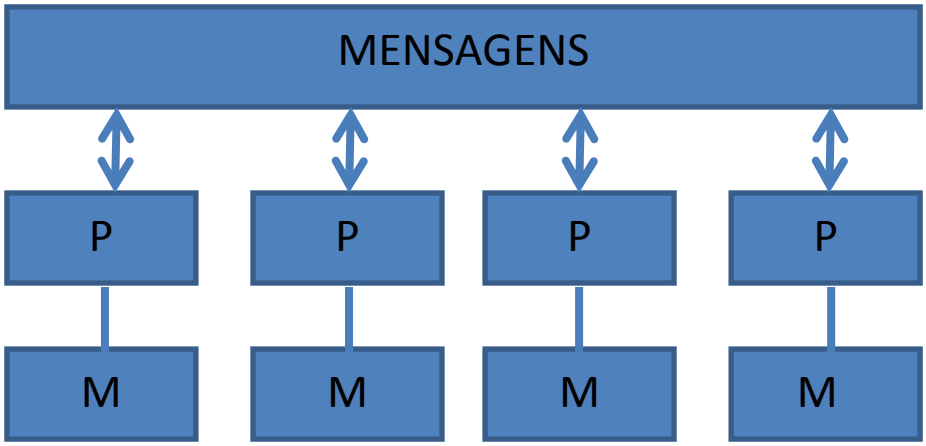


**Figura 1 - Troca de mensagens.**

Na figura 1, P representa cada processo, enquanto M a memória, módulo de memória ou mesmo um dado ao qual esse processo esteja diretamente ligado. Podemos perceber que cada processo só tem acesso ao seu módulo de memória e para que consigam interagir um com outro precisamos de troca de mensagens entre eles. Nesse modelo temos a dificuldade do tempo de comunicação entre os processos e a memória, onde os dados estão armazenados. Para resolver esse problema devemos fazer uso de *buffer* para evitar essa custosa comunicação. A figura 2 mostra o modelo de funcionamento de um sistema de memória compartilhada.

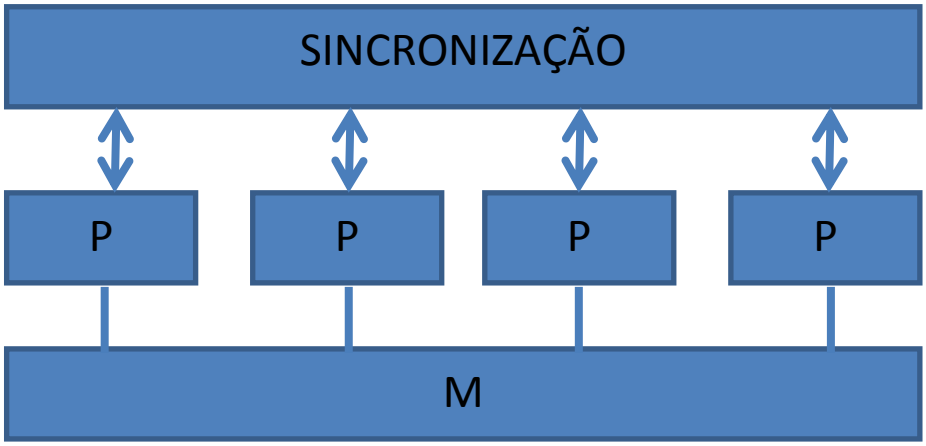


**Figura 2 - Memória compartilhada**

Na figura 2 temos o modelo de memória compartilhada, onde P representa cada processo e M os dados, ou memória compartilhada, onde todos os processos P tem acesso. Nesse modelo enfrentamos o problema dos dados acessarem a mesma memória, o que torna mais fácil e rápida a comunicação pode se tornar um problema devido a vários processos poderem alterar a mesma área da memória o que pode comprometer os resultados obtidos. Para resolver esse problema devemos sincronizar os processos e criar barreiras para que dois processos não alterem a mesma área de memória. Deve-se levar em consideração o tempo de processamento gasto para a comunicação entre os processos e de sincronização em ambos os casos para se considerar se uma aplicação terá vantagens ou não de ser paralelizada.

## 2.1. PROCESSOS E THREADS

Não existe uma definição correta do termo ‘processo’, o mais aceito é que um processo seja um programa sendo executado, possuindo informações sobre sua execução. A grande diferença entre um programa e um processo está no fato de um processo possuir seus estados e características que se alteram ao longo do tempo de sua execução. Na maioria dos casos cada programa pode gerar mais de um processo, como, por exemplo, no caso de se abrir várias planilhas ao mesmo tempo: o mesmo programa estaria criando vários processos. Mesmo através da programação podemos criar mais de um processo para se executar um único programa, através de bibliotecas de paralelismo.

Um processo pode passar por vários estados após ser criado e antes de ser destruído, ele pode estar pronto, esperando uma chamada do sistema para poder ser processado, pode estar Executando dentro do processador, ou, por fim, bloqueado. A transição de estados e o controle dos processos são feitos através do escalonador de processos. Além dessas características um processo pode gerar processos filhos, que são dependentes do ciclo de vida de um processo pai. Uma *Thread* é uma pequena parte de um processo, ou simplesmente um fluxo desse processo. Em cada um deles, nós podemos ter várias *threads*, nisso se dá o nome de *Multithreading*. (OLIVEIRA et al., 2001)

# 3. ARQUITETURAS PARALELAS

Desde o advento dos primeiros computadores a sua utilidade têm-se provado valorosa para os mais diversos ramos da humanidade, desde científicos até industriais, porém cada vez mais são exigidos computadores mais rápidos e potentes. Como forma de se manter o crescimento e atender às necessidades de processamento cada vez mais altas, atualmente as arquiteturas computacionais fazem uso de diversos níveis de paralelismo. Esse paralelismo pode ser encontrado tanto em computadores pessoais com diversos núcleos de processamento quanto em grandes *clusters* compostos de centenas de computadores trabalhando paralelamente. Este capítulo procura apresentar a classificação das principais formas de arquiteturas paralelas utilizadas atualmente.

## 3.1. MULTIPROCESSADORES

Na arquitetura de multiprocessadores, conforme mostrado na figura 3, temos *N* processadores P que se comunica com uma mesma memória principal M, compartilhada entre todos os processadores, que pode estar dividida em vários módulos de memória, porém acessível para todos os processadores. Ela é acessada através de uma rede de interconexão. Essa arquitetura pode ser dividida conforme a distância entre os processadores e a memória principal, caso todos os processadores estejam na mesma distância da memória principal, ou seja, levem o mesmo tempo para poder acessar a memória. Podemos classifica-la como uma arquitetura de acesso uniforme à memória, caso cada processador possua seu próprio módulo de memória associado a ele, todos os processadores tenham acesso a esse módulo de memória e o tempo de acesso a essa memória seja inferior para o

processador ao qual ela esteja associada, então temos uma arquitetura de acesso não uniforme a memória.

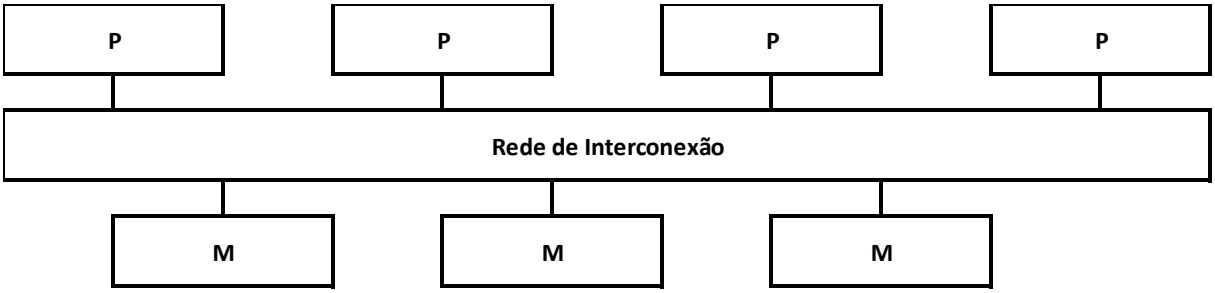


**Figura 3 – Arquitetura Multiprocessadores**

## 3.2. MULTICOMPUTADORES

A arquitetura multicomputadores se difere da arquitetura multiprocessadores pelo fato de que nela cada processador possui sua própria unidade de memória principal ligada diretamente a ele. Conforme mostra a Figura 4, o processador $P_1$ está ligado diretamente à memória $M_1$, porém para que os processadores tenham acesso a uma memória não ligada diretamente a ele se faz uso de passagem de mensagens através da rede de interconexão. Esse processo é conhecido como *Message Passing System.* Nessa arquitetura cada grupo de processador, mais memória, forma um componente individual que se comunica com os outros através da passagem de mensagens, esse processo é necessário pelo fato de que cada processador só tem o acesso a sua própria memória e para que ocorra o paralelismo no processamento das informações.

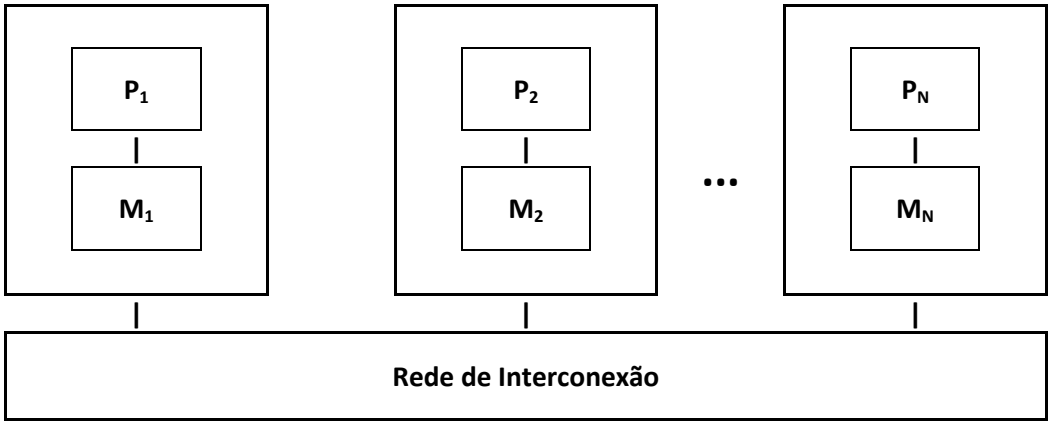


**Figura 4 - Arquiteturas Multicomputadores**

## 3.3. MULTI-CORE

As arquiteturas *multi-core* consistem em um computador que possui diversos núcleos de processamento em um mesmo chip. Em uma arquitetura *multi-core* os recursos de um processador são replicados, como se vários processadores estivessem integrados em um único chip (SCHEPKE, 2009).

As arquiteturas possuem um único núcleo de processamento, formado por vários componentes, como unidade de controle, unidade lógica aritmética, entre outros. Em uma arquitetura *multi-core* esses componentes são replicados para os diversos núcleos desse processador. Atualmente a maioria de computadores pessoais, ou PC's (*Personal computer*), que são produzidos já fazem uso desse tipo de arquitetura e possuem de dois a quatro núcleos lógicos. A grande vantagem desse tipo de arquitetura está no fato dos núcleos do processador compartilhar a memória cache, tornando assim menos suscetível a erros e mais rápido o acesso às informações pelos núcleos.

## 3.4. CLASSIFICAÇÃO FLYNN

Criada por Michael J. Flynn, em meados dos anos 70, ainda hoje é uma forma de classificação inicial válida para as arquiteturas paralelas. A classificação de Flynn subdivide as arquiteturas paralelas em quatro grandes grupos conforme seu fluxo de instruções (*instruction stream*) e seu fluxo de dados (*data stream*) (ROSE;NAVAUX, 2008). A classificação de Flynn pode ser representada da seguinte forma:

| | **SD (SINGLE DATA)** | **MD (MULTIPLE DATA)** |
|---|---|---|
| **SI (SINGLE INSTRUCTION)** | SISD | SIMD |
| **MI (MULTIPLE INSTRUCTION)** | MISD | MIMD |

**Figura 5 - Classificação Flynn**

Conforme representado na Figura 5, segundo Flynn, podemos dividir as arquiteturas de computadores em: *Single Instruction Single Data, Single Instruction Multiple Data, Multiple Instruction Single Data* e *Multiple Instruction Multiple Data. E*ssa divisão se faz em relação aos fluxos de dados e instruções que a arquitetura é capaz de suportar.

### 3.4.1. Single Instruction Single Data

O SISD (*Single Instruction Single Data*) é representado pelos computadores pessoais e estações de trabalho que possuem somente um processador com um único núcleo que executa instruções em sequencial. Nesse tipo de arquitetura não existe paralelismo em nenhum nível, o processador executa uma operação por vez, uma instrução é recebida pelo processador que a executa. Como pode ser visto na figura 6 onde um único dado passa pela unidade de controle C que encaminha para o processador P que se comunica com a memória M.

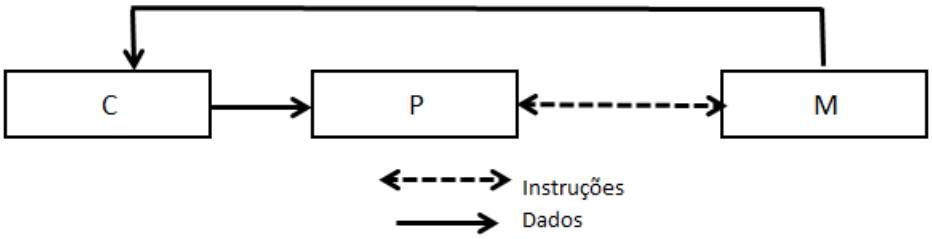


**Figura 6 – SISD (ROSE;NAVAUX, 2008)**

### 3.4.2. Single Instruction Multiple Data

Inseridas nas arquiteturas SIMD (*Single Instruction Multiple Data*) podemos citar as máquinas Array, onde uma única instrução é executada por vários processadores. Nesse tipo de arquitetura temos uma única unidade de controle que é alimentada por uma única instrução e divide o processamento entre vários processadores, passando a instrução para todos os processadores que utilizam seus próprios recursos para executar a operação solicitada, para que isso ocorra, a memória

precisa ser dividida em vários módulos, para que cada processador possa ter acesso a um desses módulos para receber e retornar os dados. Conforme mostra a figura 7 nesse tipo de arquitetura possuímos somente uma unidade de controle que será responsável por distribuir as tarefas entre todos os processadores P.

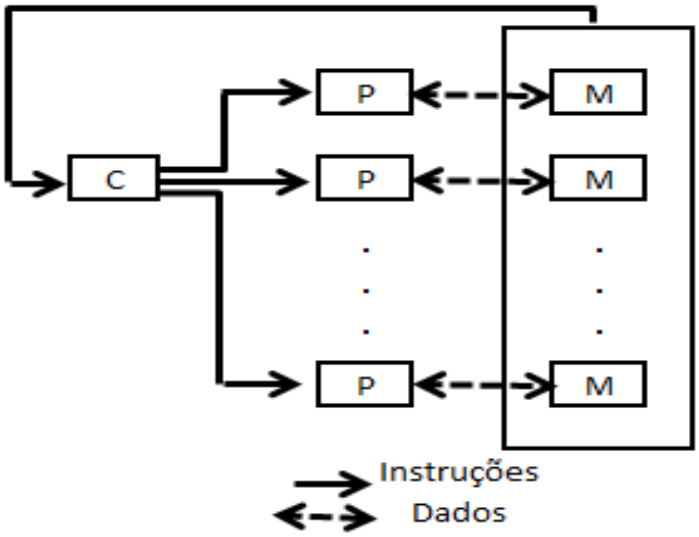


**Figura 7 - SIMD (ROSE;NAVAUX, 2008)**

### 3.4.3. Multiple Instruction Single Data

Nesse tipo de arquitetura várias instruções seriam aplicadas ao mesmo dado e endereçamento de memória. Essa classe de computadores pode ser considerada vazia, porém podemos considerar arquiteturas tolerantes a falhas que como redundância vários computadores executariam a mesma informação e apresentariam o mesmo resultado. As arquiteturas pipeline também poderiam se enquadrar nessa classificação, caso desconsiderarmos o fato de o dado não ser o mesmo após cada instrução em cada fase do pipeline. Na figura 8 podemos ver o funcionamento desse tipo de arquitetura, onde uma única informação é processada simultaneamente por vários processadores.

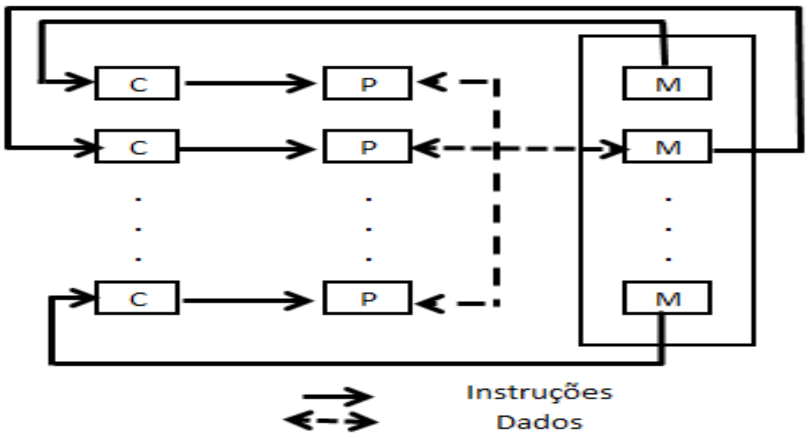


**Figura 8 –MISD (ROSE;NAVAUX, 2008)**

### 3.4.4. Multiple Instruction Multiple Data

A arquitetura MIMD (*Multiple Instruction Multiple Data)* consiste em várias unidades de controle, cada uma podendo executar uma instrução. Na arquitetura MIMD temos várias unidades de controle, cada qual com sua unidade de processamento e podendo executar cada uma paralelamente sua instrução, ou seja, podemos ter vários programas que são executados ao mesmo tempo, um para cada unidade de controle. Para que isso aconteça precisamos que a memória principal não seja implementada com um único módulo. Podemos subdividir a arquitetura MIMD quanto ao seu acesso à memória. Se a arquitetura possuir apenas uma memória compartilhada entre elas se tem um multiprocessador, como no caso dos servidores com vários processadores internos. Caso cada unidade de controle/processamento tenha sua própria memória e recursos, se tem um multi computador. Esse tipo de arquitetura é apresentada na figura 9 onde possuímos paralelismo completo com várias instruções sendo aplicadas em vários dados.

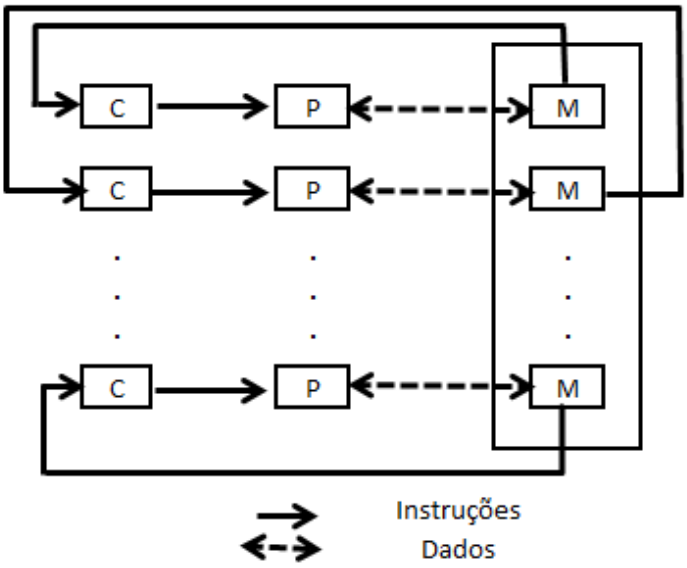


**Figura 9 - MIMD (ROSE;NAVAUX, 2008)**

## 3.5. CLUSTER

A palavra *cluster* é de origem da língua inglesa e significa um grupo de coisas do mesmo tipo. Na computação *cluster* se refere a um conjunto de computadores ligados por uma rede que, juntos, formam um grande computador com poder de processamento equivalente ao de todas as suas unidades somadas. (FAUSTINO JÚNIOR, FREITAS, 2005). Um cluster se enquadra em uma arquitetura multicomputadores, pois é constituído pela junção de vários conjuntos de memórias e processadores interligados por uma rede de interconexão, no caso uma rede *ethernet*. Na classificação de Fllyn, um *cluster* se enquadraria em uma máquina MIMD, pois ela é totalmente paralelizável, aplicando várias instruções em várias informações simultaneamente e em cada um dos nós que a compõe.

O uso de *clusters* é altamente recomentado para empresas e organizações que façam uso de sistemas pesados, grandes bancos de dados, grandes cálculos científicos ou meteorológicos, ou qualquer aplicação que necessite de algo que um único servidor não seria capaz de suportar. Para o pleno uso de todas as capacidades computacionais desse tipo de arquitetura é necessário o uso de programação paralela, que, apesar de aumentar muito a complexidade do desenvolvimento, auxilia no melhor desempenho e divisão de processamento desse tipo de aplicação (FAUSTINO JÚNIOR, FREITAS, 2005).

Em um *cluster* teremos sempre um nó centralizador cuidando de dividir o processamento entre todos os seus nós, que quase sempre serão apenas conjuntos de memória e processador, sempre enviando os dados já processados por meio da rede. Podemos dividir um *cluster* em categorias conforme sua função.

### 3.5.1. Cluster De Alta Performance

No *cluster* de Alta Performance, também conhecido como HPC (*High-performance computing*) temos como foco o processamento e o alto desempenho visando a utilização de grandes programas e processos paralelos. Um *cluster* de alta performance surge como uma opção economicamente superior ao investimento que seria usado para um servidor único com o mesmo poder computacional. Essa redução de custo se deve ao fato de que um *cluster* pode ser montado com várias estações de trabalho simples e podem ser controlados por sistema operacionais e *softwares* gratuitos. A montagem desse tipo de *cluster* é relativamente simples e barata, o que a torna uma das principais alternativas para universidades e empresas que necessitam de qualquer processamento pesado e custoso para o *hardware*. Em um *cluster* de alta performance temos a presença de vários computadores convencionais, chamados de nós, interconectados por uma rede convencional *ethernet* e um nó centralizador para controlar os envios de requisições para os outros nós.

### 3.5.2. Cluster De Alta Disponibilidade

Também conhecido como *cluster* HA (High-Availability), tem como principal objetivo manter o sistema o mais próximo possível de estar 100% do tempo ligado e operante. Nesse tipo de equipamento há vários computadores ligados entre si por uma rede de interconexão, se replicando para que a possível parada ou danificação de algum de seus nós não acarrete a paralização do serviço que o *cluster* oferece. Sendo assim quando um nó cai outro irá assumir suas funções após a queda. Existem serviços vitais para uma organização que não poderiam ficar fora do ar por tempo algum. Para esses serviços deve-se estudar o uso desse tipo de solução.

Mesmo com *backups* e servidores reservas não podemos prever uma falha de equipamento, por isso, em um *cluster* de alta disponibilidade, possuímos esse tipo de replicação. Parar um sistema vital para uma organização mesmo que por poucos minutos pode ocasionar prejuízos enormes e até abalar a saúde financeira de uma organização. A desvantagem de se utilizar um *cluster* de alta disponibilidade está no fato de que os computadores estarão focados na replicação e não em melhorar o desempenho, então é impossível nesse tipo de solução não ter um desempenho pior ao se comparar a um *cluster* de alta performance.

# 4. LINGUAGENS DE PROGRAMAÇÃO PARALELA

Este capítulo irá demonstrar algumas linguagens de programação que utilizam paralelismo em nível de programação, bem como seus conceitos chave e exemplos de suas utilizações. Atualmente existem inúmeras linguagens de programação no mercado. Várias delas possuem suporte para paralelismo de alguma forma: ou nativa da própria linguagem ou proveniente de bibliotecas desenvolvidas especialmente para esse objetivo. A escolha de uma linguagem de programação é um passo importante para o desenvolvimento de qualquer sistema. Este capítulo busca mostrar, de forma introdutória, paradigmas e características de algumas das principais linguagens de programação utilizadas atualmente e a forma de se paralelizar seus códigos.

## 4.1. JAVA PARALELO

Em 1991, a Sun Microsystems financiou um projeto de pesquisa interna que resultou na criação da linguagem de programação que foi batizada de *oak* em homenagem a um carvalho que podia ser visto da janela de seu principal criador: James Gosling. Mais tarde foi descoberto que já existia uma linguagem de computadores com esse nome e então essa nova linguagem foi rebatizada como *java*, em homenagem a cidade de origem de um café importado que eles consumiam. Java é uma linguagem de programação orientada a objeto que independe do sistema operacional, pois é interpretada por sua máquina virtual (JVM). Esta linguagem disponibiliza concorrência através de APIs próprias, pode-se desenvolver um aplicativo com *threads* em que cada *thread* terá sua pilha de execução, permitindo assim que cada uma seja executada paralelamente. Java possui primitivos de *multithreading* como parte de sua linguagem e biblioteca padrão, o que facilita sua utilização inclusive nos

diversos SO com os quais seja compatível (DEITEL, 2010). Para se criar uma classe que possa ser executada como *thread* em Java é preciso que ela implemente a classe *Runnable*. No Código 1 é demonstrado um exemplo de código de uma classe que irá implementar o *Runnable*.

```
1   package tcc;
2
3   public class ContThread implements Runnable {
4           private int cont,inicio, fim;
5
6       public ContThread(int inicio, int fim) {
7           this.inicio = inicio;
8           this.fim = fim;
9       }
10
11
12      public void run() {
13          while(inicio<=fim){
14              System.out.print(inicio + "-");
15              inicio ++ ;
16          }
17      }
18  }
19
```

**Código 1 - Classe ContThread**

No código 1, por implementar a classe *Runnable*, obrigatoriamente deverá implementar o método run() que será executado quando essa *thread* estiver pronta. Nesse exemplo a *thread* irá imprimir todos os números dentro de um intervalo de números inteiros informados no construtor da classe. O código 2 exemplifica a classe principal que irá instanciar a *thread* do tipo *ContThread*.

```
1  package tcc;
2  public class ExecutaThread {
3
4      public static void main(String[] args) {
5
6          Thread thread1 = new Thread(new ContThread(1,100));
7          Thread thread2 = new Thread(new ContThread(101,200));
8          Thread thread3 = new Thread(new ContThread(201,300));
9
10         thread1.setPriority(Thread.MIN_PRIORITY);
11         thread2.setPriority(Thread.NORM_PRIORITY);
12         thread3.setPriority(Thread.MAX_PRIORITY);
13         System.out.println("-------Inicio Main---------");
14         try{
15         thread1.start();
16         thread2.start();
17         thread3.start();
18         }catch(Exception e){
19
20         }
21
22         System.out.println("-------Fim Main---------");
23     }
24
25 }
```

**Código 2 - Classe ExecutaThread**

Neste código temos a classe *ExecutaThread*, nas linhas 6,7 e 8 são instanciados 3 objetos do tipo *thread*, passando como parâmetros Objetos do tipo *ContThread*. Depois de criados os objetos nas linhas 10,11 e 12 são alteradas as prioridades das *threads* criadas, usando as constantes presentes na classe *thread*. Essa prioridade irá alterar a forma como que o SO irá tratar as *threads*, isso irá variar de acordo com o SO utilizado para a execução do programa java. Com o método start() as *threads* passarão para o estado de pronto e serão executadas pelo processador. Note que o método *main* poderá terminar antes de todas as *threads* terem sido executadas.

## 4.2. C# PARALELO

C# é uma linguagem orientada a objetos, fortemente *tipada* e desenvolvida para a plataforma *.NET Framework.* Ela é uma linguagem de fácil aprendizagem que roda em ambientes com sistema operacional Windows. Por ser uma linguagem robusta com um Ambiente Integrado de Desenvolvimento (*Integrated Development Environment* - **IDE**) completo e de fácil utilização, a linguagem se tornou amplamente conhecida na atualidade, sendo uma das linguagens mais utilizadas. Como recursos para paralelização C# possui uma grande quantidade de alternativas: como filas de mensagens e *threads*. A criação de uma *thread* em C# é um processo simples, conforme demonstra o Código 3:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading;
6
7
8  namespace ConsoleApplication1
9  {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14
15             Thread t1 = new Thread(contThread1);
16             Thread t2 = new Thread(contThread2);
17             t1.Start();
18             t2.Start();
19             Console.Read();
20         }
21         static void contThread1()
22         {
23             for (int i = 0; i < 1000; i++)
24             {
25                 Console.Write(i+"-");
26             }
27         }
28         static void contThread2()
29         {
30             for (int i = 1001; i < 2000; i++)
31             {
32                 Console.Write(i+"-");
33             }
34         }
35
36     }
37 }
```

**Código 3 - Thread C#**

Aqui temos um exemplo de criação de *threads*. Para se criar uma *thread* em C# devemos criar um método que será passado por parâmetro na instanciação de um objeto da Classe *Thread* pertencente ao *namespace System.Threading* que é responsável pelo gerenciamento das *threads* em C#. A partir da versão *Framework* 4.0 foi adicionado o *namespace System.Threading.Tasks* para facilitar e tornar ainda mais amigável o uso da programação paralela e principalmente loops paralelos dentro da linguagem. O código 4 mostra dois métodos para a população de 2 vetores com valores aleatórios:

```
37          public void processaSeq()
38          {
39              int[] a = new int[9999999];
40              int[] b = new int[9999999];
41
42              for (int i = 0; i < 9999999; i++)
43              {
44                  Random aleatorio = new Random();
45                  a[i] = aleatorio.Next();
46                  b[i] = aleatorio.Next();
47
48
49              }
50          }
51
52          public void processaPar() {
53              int[] a = new int[9999999];
54              int[] b = new int[9999999];
55
56              Parallel.For(0, 9999999, i=>
57              {
58                  Random aleatorio = new Random();
59                  a[i] = aleatorio.Next();
60                  b[i] = aleatorio.Next();
61
62
63          });
64
65          }
```

**Código 4 - métodos para popular Vetor**

O Código 4 mostra um código com dois métodos: o método processaSeq() utiliza de um for convencional para gerar valores aleatórios para dois vetores, enquanto o segundo método utiliza do método For() disponível na classe Parallel. Esse For da classe utilizado no método processaPar() é um método desenvolvido para facilitar a programação paralela, pois ele identifica a quantidade de processadores do computador que estiver executando e automaticamente divide os processos para que sejam executados paralelamente. A Figura 6 mostra a comparação do processamento desses métodos em um mesmo computador utilizando o gerenciador de tarefas do SO *Windows*.

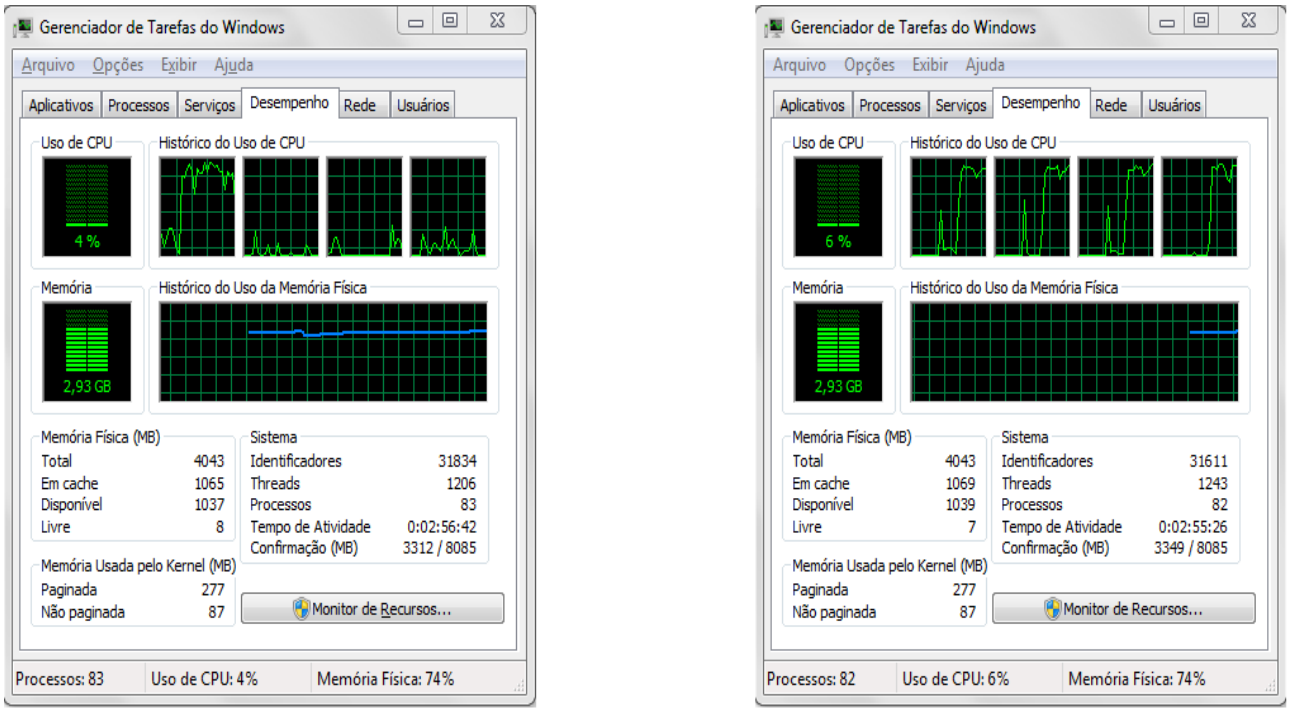


**Figura 10 - comparação utilização processador**

Na Figura 10 o gerenciador à esquerda foi registrado enquanto se executava o método, utilizando o for convencional, enquanto o gerenciador à direita se refere à execução do método Parallel.For, mesmo para processos simples como os laços do Código 4 pode-se notar uma grande diferença no tempo de execução de cada um deles, sendo que o Parallel.For se mostra muito mais rápido.

## 4.3. C / C++

A linguagem C foi originalmente criada nos anos 70 pelo físico Dennis Ritchie, voltada para o sistema operacional UNIX. C. É uma linguagem estruturada e tipada que foi originalmente baseada nas linguagens de programação BCPL e B que não possuíam tipos.

Grande parte do sistema operacional UNIX e seus programas são baseados na linguagem C (KERNIGHAN, RITCHIE, 1988). Esta linguagem é amplamente utilizada por ser uma linguagem leve e proporcionar vários recursos, como, por exemplo, acesso a memória. Contudo ela possuía algumas limitações de tamanho do código, a partir disso foi então desenvolvida a linguagem de programação C++. Originalmente chamada de *C with class,* posteriormente C++ foi adaptada e evoluiu para ser uma linguagem orientada a objetos, possuindo todos os paradigmas desse tipo de programação. Nativamente C não possui nenhum recurso *multi-thread*, operações paralelas, sincronização ou concorrência, porém para esse fim foram criadas bibliotecas como pthread.h e, para *Windows*, juntamente com a biblioteca Windows.h, foram incorporados componentes de controle de *threads* possibilitando o paralelismo para essa linguagem.

No código 5 podemos ver um exemplo de criação de *threads* utilizando a linguagem C para a plataforma *Windows* com o auxilio da biblioteca Windows.h.

```
1  #include <windows.h>
2  #include <iostream>
3  using namespace std;
4  DWORD WINAPI Thread(LPVOID p)
5  {
6
7     for(int i =0; i<1000; i++){
8                 cout << i << "-";
9      }
10     return 0;
11 }
12 int main(){
13     DWORD id;
14     HANDLE th = CreateThread(0,0,Thread,0,0,&id);
15     for(int i =1000; i<2000; i++){
16                cout << i << "-";
17     }
18     WaitForSingleObject(th,INFINITE);
19     TerminateThread(th,0);
20     CloseHandle(th);
21     getchar();
22     return 0;
23 }
24
```

**Código 5 - Thread em C**

Podemos observar a criação de uma *thread* declarada na linha 4 e instanciada na linha 14 com o comando CreateThread, essa *thread* irá ocorrer juntamente com o laço criado na linha 15, imprimindo os dados de forma misturada.

# 5. DESENVOLVIMENTO

Neste capitulo será apresentado o desenvolvimento de um software utilizando conceitos chaves de paralelismo utilizando a linguagem de programação Java. A realização do presente trabalho consiste no desenvolvimento de um *software* utilizando os conceitos de paralelismo em nível de programação para ser fazer uso das tecnologias de *multithreading*. Para a utilização e desenvolvimentos desse *software* foi utilizada a linguagem de programação Java, que possui biblioteca nativa para a criação e controle de *threads*. Utilizou-se o conceito de Semáforos para sincronizar as *threads*, impedindo que elas utilizem recursos restritos. Foi escolhido o problema do Jantar dos filósofos para empregar esses conceitos.

## 5.1. JANTAR DOS FILÓSOFOS

Essa problemática foi sugerida e resolvida por Edsger Dijkstra, por volta de 1965. Ele propõe que em uma grande mesa redonda tenhamos cinco filósofos prontos para jantar, na mesa temos cinco pratos de arroz e apenas cinco *hashi* (talheres em forma de varetas muito utilizados em países asiáticos como Japão e China), onde cada um precisa de exatamente dois *hashi* para poder se alimentar. Cada filósofo tem de comer e pensar, quando ele ficar com fome ele tende a pegar o *hashi* do seu lado esquerdo e direito e então comer, caso consiga pegar os dois. O problema dessa situação está no fato de que existe apenas um talher para cada filósofo sentado na mesa, obrigando que os filósofos disputem o tempo de uso dos utensílios. Os filósofos alternam entre ficar com fome, pegar *hashi*, comer, devolver os *hashi* e pensar. A figura 11 ilustra a situação:

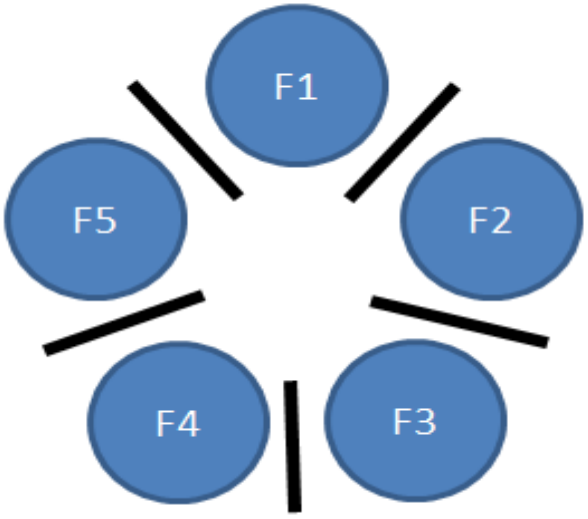


**FIgura 11 – Mesa dos filósofos**

Na figura 11 podemos ver os cinco filósofos representados pelas elipses com ‘F’, numerados de um a cinco, cada qual possuí um *hashi* na sua esquerda e um na sua direita, ambos divididos pelos filósofos dos respectivos lados. Notamos que se o filósofo F1 pegar os talheres da esquerda e da direita os filósofos F2 e F5 ficarão impossibilitados de comer. Para a solução do problema podemos resolver definindo arbitrariamente os grupos de filósofos que irão comer, notamos que na hora de se alimentar apenas é possível que no máximo dois filósofos se alimentem por vez, no caso do F1 pegar os talheres possibilitamos apenas os filósofos F3 ou F4 a se alimentarem simultaneamente. Pensando assim podemos criar três grupos para se alimentarem, por exemplo, F1+F4, F2+F5 e F3. Cada um dos grupos poderá se alimentar alternadamente. Porém a forma escolhida para se resolver esse problema neste trabalho foi a metodologia de fila, onde o primeiro que sentir fome irá pegar os talheres que estiverem desocupados e esperará que o outro fique vago, demorando um tempo aleatório para comer e para pensar.

Foi criada uma classe em java para representar os filósofos como é mostrado no código 5. Essa classe herdou características de *thread* para que cada um dos filósofos pudesse agir independente um do outro, criando assim um total de cinco *threads* representando respectivamente cada um dos filósofos sentados à mesa.

```
1  import java.util.concurrent.Semaphore;
2
3  public class Filosofo extends Thread {
4
5      private int id;
6      private String nome;
7      private static Semaphore hashis[] = new Semaphore[6];
8
9      public Filosofo(String nome, int id) {
10         for (int i = 1; i <= 5; i++) {
11             hashis[i] = new Semaphore(1);
12
13         }
14         this.setNome(nome);
15         this.setId(id);
16
17     }
```

**Código 5 – Jantar dos filósofos**

Nesse trecho de código podemos ver que a classe “Filosofo” herda dados de *thread*, e que possui três atributos: um id do tipo inteiro, o nome do filósofo e um vetor de *hashi*, do tipo *Semaphore*. A classe *Semaphore* em java é usada para casos de uso exclusivo de uma única *thread* ao mesmo tempo, ela é do tipo *static*, pois não queremos que em cada instancia dessa classe seja criado um novo *hashi* e sim que eles possam ser compartilhados entre todos os objetos de Filosofo. A classe *Semaphore* como construtor exige que seja passado o numero de sessões que poderão usar esse recurso ao mesmo tempo, nesse caso cada *hashi* poderá ser usado apenas por um filósofo por vez.  O código 6 mostra o método run() da classe Filosofo.java, nessa classe ele irá chamar os métodos na ordem de sua execução.

```
67      public void run() {
68
69          while (true) {
70              try {
71                  faminto();
72                  pegarHashi();
73                  comer();
74                  soltarHashi();
75                  pensar();
76              } catch (Exception e) {
77              }
78
79          }
80
81      }
82
```

**Código 6 – jantar dos filósofos run()**

Nesse trecho de código podemos ver o método *run(),* que deve ser implementado em todas as classes que herdam de *Thread(.)* Esse método irá ser executado conforme instanciado um objeto dessa classe e chamado o método *start()* da classe Filosofo.java. No Apêndice B - Código fonte Jantar.java - temos o código da classe que irá ter o método main() e irá instanciar cinco objetos Filósofos e iniciá-los. Na figura 12 temos um resultado possível para a execução dessas classes, porém, como o escalonamento dos processos e as prioridades são dadas pelo próprio processador, além da própria aleatoriedade empregada no código, em cada execução desse programa poderemos esperar um resultado diferente, ainda levando em consideração que os tempos de pensar e comer de cada filósofo são atribuídos aleatoriamente através do método Math.random(), o que torna os seus resultados ainda mais imprevisíveis.

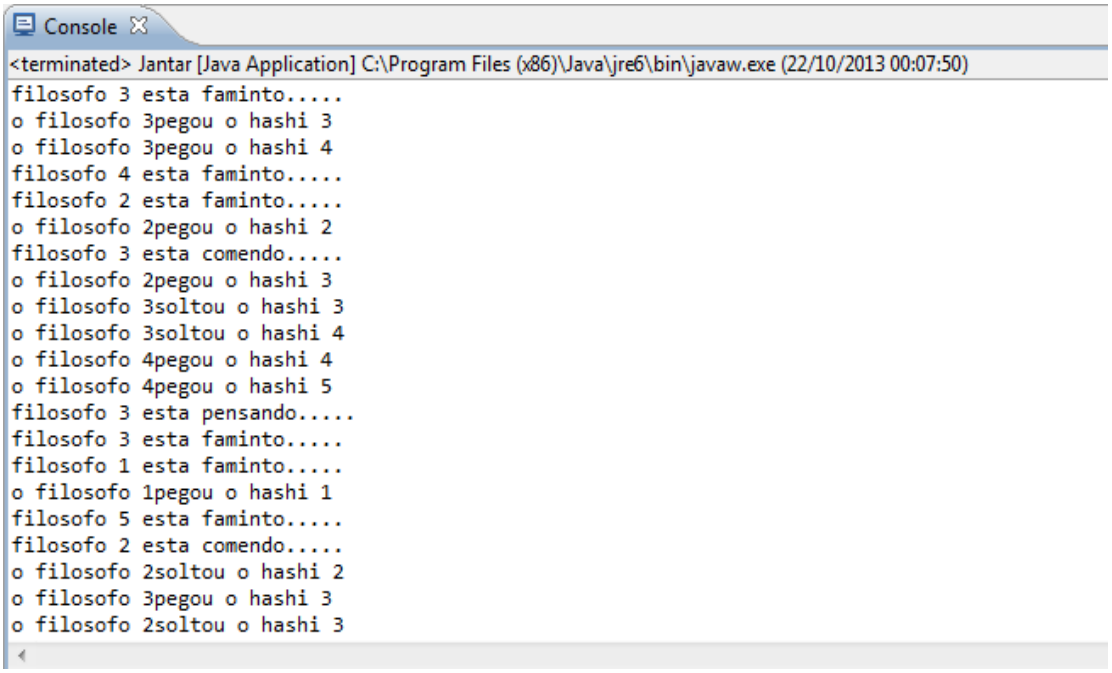


**Figura 12 – jantar dos filósofos resultado**

No resultado apresentado na figura 12 podemos ver uma execução da classe Jantar.java onde cinco filósofos são instanciados e iniciados. Podemos perceber a execução paralela dos filósofos que coexistem ao mesmo tempo na memória e disputam os mesmos recursos passando dos estados faminto, pensando e comendo. Quando um dos filósofos com fome ele logo procura pelos *hashis* disponíveis e depois de utilizá-lo devolve para a mesa liberando o recurso para que outro filósofo possa usar.

# 6. CONCLUSÃO

Diante dos estudos realizados pode-se concluir que a computação paralela é mais que uma forma e conceito de arquitetura ou de programação, mas sim uma tendência e uma grande evolução para a velocidade de processamento e poder computacional. Apesar de aumentar exponencialmente a dificuldade e a complexidade de um código, o paralelismo em nível de programação, acarreta num grande aumento na velocidade e uso correto de todos os processadores. Mesmo o S.O. realizando a divisão de tarefas da forma mais adequada possível entre os processadores, núcleos ou mesmo nós de um computador, Nota-se esta melhora real, quando existe uma evolução conjunta de arquitetura e programação.

## 6.1. TRABALHOS FUTUROS

Com base nos resultados obtidos e os estudos realizados abre-se espaço para o desenvolvimento de aplicações mais complexas que façam uso de paralelismo para diversos tipos de sistemas operacionais e arquiteturas, também se torna possível o estudo do emprego prático real de paralelismo e computação distribuída para o cotidiano de uma empresa.

## REFERÊNCIAS

ANDREWS, Gregory R. **Foundations Of Multithreaded, Parallel, And Distributed Programming** , Addison-Wesley; 1 edition, 1999.

CAMPBELL, Colin; MILLER, Ade;. **Design Patterns for Decomposition and Coordination on Multicore Architectures**, Microsoft Press, 2011.

DEITEL, Harvey M.; DEITEL, Paul J. **Java Como Programar**, oitava edição, Pearson Education, 2010.

FAUSTINO JÚNIOR, Esli Pereira; FREITAS, Reinaldo Borges de; **Construindo supercomputadores com Linux**, Centro Federal De Educação Técnologica de Goiás, Goiás, Gôiania, 2005.

KERNIGHAN , Brian W. ; RITCHIE, Dennis M. , **The C programming Language**, Prentice-Hall , 1988.

OLIVEIRA, Rômulo Silva de; CARISSIMO, Alexandre da Silva; TOSCANI, Simão Sirineio; **Sistemas Operacionais**, Revista de Informática Teórica e Aplicada – RITA – VOLUME VIII (3), 2001

PINHO, Eduardo Gurgel. **Uma Linguagem De Programação Paralela Orientada A Objetos Para Arquiteturas De Memória Distribuída** – Universidade Federal do Ceará, Fortaleza, Ceará, 2012.

ROSE, César A.F. de; NAVAUX, Philippe O.A. **Arquiteturas Paralelas**: BOOKMAN,2008.

SCHEPKE, Claudio. **Ambientes de Programação Paralela**. Pós-Graduação Em Computação - Universidade Federal Do Rio Grande Do Sul, Porto Alegre, Brasil, 2009.

SCHIOZER, Denis José. **Computação Paralela Aplicada a Simulação Numérica de Reservatórios** - Universidade Estadual De Campinas Faculdade De Engenharia Mecânica, Campinas, São Paulo, 2003.

## APÊNDICE A - Código fonte Filosofo.java

```
import java.util.concurrent.Semaphore;

public class Filosofo extends Thread {

	private int id;
	private String nome;
	private static Semaphore hashis[] = new Semaphore[6];

	public Filosofo(String nome, int id) {
		for (int i = 1; i <= 5; i++) {
			hashis[i] = new Semaphore(1);

		}
		this.setNome(nome);
		this.setId(id);

	}

	public void pegarHashi() throws InterruptedException {
		hashis[id].acquire();
		System.out
				.println("o filosofo " + this.getId() + "pegou o hashi "
+ id);
		if (id < 5) {
			hashis[id + 1].acquire();
			System.out.println("o filosofo " + this.getId() + "pegou o
hashi "
					+ (id + 1));
		} else {
			hashis[1].acquire();
			System.out.println("o filosofo " + this.getId() + "pegou o
hashi "
					+ 1);

		}

	}

	public void soltarHashi() throws InterruptedException {
		hashis[id].release();
		System.out.println("o filosofo " + this.getId() + "soltou o hashi "
				+ id);
		if (id < 5) {
			hashis[id + 1].release();
			System.out.println("o filosofo " + this.getId() + "soltou o
hashi "
					+ (id + 1));
		} else {
			hashis[1].release();
			System.out.println("o filosofo " + this.getId() + "soltou o
hashi "
					+ 1);
```

```
	}
}

public void comer() throws InterruptedException {
	this.sleep((long) (Math.random() * 1000));
	System.out.println(nome + " esta comendo.....");
}

public void pensar() throws InterruptedException {
	this.sleep((long) (Math.random() * 1000));
	System.out.println(nome + " esta pensando.....");
}

public void faminto() throws InterruptedException {
	this.sleep((long) (Math.random() * 1000));
	System.out.println(nome + " esta faminto.....");
}

public void run() {

	while (true) {
		try {
			faminto();
			pegarHashi();
			comer();
			soltarHashi();
			pensar();
		} catch (Exception e) {
		}

	}

}

public String getNome() {
	return nome;
}

public void setNome(String nome) {
	this.nome = nome;
}

public long getId() {
	return id;
}

public void setId(int id) {
	this.id = id;
}

public static Semaphore[] getHashis() {
	return hashis;
}

public static void setHashis(Semaphore[] hashis) {
	Filosofo.hashis = hashis;
}
```

```
}
```

## APÊNDICE B - Código fonte Jantar.java

```
public class Jantar {

	public static void main(String[] args) {

		try {
			Filosofo f1 = new Filosofo("filosofo 1", 1);
			Filosofo f2 = new Filosofo("filosofo 2", 2);
			Filosofo f3 = new Filosofo("filosofo 3", 3);
			Filosofo f4 = new Filosofo("filosofo 4", 4);
			Filosofo f5 = new Filosofo("filosofo 5", 5);

			f1.start();
			f2.start();
			f3.start();
			f4.start();
			f5.start();
		} catch (Exception e) {

			System.out.println(e.getMessage());
		}

	}

}
```