# Um sistema de execução de programas paralelos orientados a objetos em sistemas distribuídos.

Hermes Senger
Líria Matsumoto Sato

Laboratório de Sistemas Integráveis
Edifício de Engenharia de Eletricidade
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, trav. 3, nº 158
05508-900 - São Paulo, SP
Tel. (011) 818-5270/818-5589
e-mail: {hermes,liria}@lsi.usp.br

## ABSTRACT

This paper presents a run-time library for execution of object-oriented parallel programs, on workstation clusters. In order to improve load balancing of the programs, the library offers a mechanism to distribute the objects among the nodes automatically. The strategy adopted takes into account the heterogeneity of the cluster, and other users.

## RESUMO

Este trabalho apresenta um conjunto de rotinas para execução de programas paralelos, implementados em uma linguagem orientada a objetos, sobre aglomerados de estações de trabalho. A estratégia adotada na distribuição dos objetos do programa paralelo, leva em conta a heterogeneidade e a presença de outros usuários no aglomerado.

## 1 - INTRODUÇÃO

A exploração de paralelismo em sistemas distribuídos, tais como aglomerados de estações de trabalho é uma alternativa promissora na área de processamento de alto desempenho.

Diversas linguagens e sistemas de programação surgiram com o intuito de solucionar ou amenizar os problemas inerentes à programação distribuída. Com relação à expressividade do paralelismo, surgiram linguagens baseadas em paradigmas como funcional [HUDAK86], procedural ( baseado em processos ou em *statements* como unidade de paralelismo ) [BURNS87] [GEHAN92] [INMOS84], modelo de paralelismo de dados, e no paradigma de orientação a objetos. Também é possível classificar as linguagens existentes sob aspectos como o da comunicação usando modelos como passagem de mensagem explícita ou modelo de compartilhamento de dados, ou ainda modelos tolerantes a falhas.

O paradigma de orientação a objetos apresenta duas características bastante atraentes à programação paralela, e em particular a distribuída :

- o encapsulamento das informações determina que cada objeto só pode ter acesso aos seus dados.

- o acesso a dados de outros objetos só pode ser feito através de mensagens aos métodos de tais objetos.

Tais características proporcionam algumas facilidades na distribuição dos objetos de um programa paralelo sobre uma arquitetura MIMD com memória distribuída. Esta arquitetura se baseia no modelo em que cada processador tem acesso direto a um espaço de endereçamento local, e se comunica com os demais processadores por meio de um sistema de interconexão que permite o envio e recebimento de mensagens. O tempo de acesso a dados que não pertençam à memória local de um nó, é pelo menos uma ordem de grandeza maior do que um acesso local.

O encapsulamento oferecido pelo paradigma de objetos impede que um nó que esteja executando um método de um certo objeto tente acessar diretamente os dados de outro objeto, que poderiam estar na memória de outro nó. Além disto, o paradigma estabelece um mecanismo bastante claro para os acessos a dados de outros objetos ( que neste caso pode residir em outro nó ), através de chamadas a seus métodos públicos. Caso os dois objetos envolvidos estejam em nós distintos, tanto o processo de chamada do método público quanto a devolução do dado solicitado, serão feitos através do sistema de interconexão dos nós.

Um problema inerente à programação distribuída, consiste em que um programa distribuído só termina quando a última de suas unidades terminar. É necessário portanto, saber distribuir corretamente a carga nos nós. A tarefa de promover o balanceamento de carga torna-se mais complexo, quando o sistema distribuído for uma rede heterogênea de estações de trabalho com diversos usuários que compartilham virtualmente diversas máquinas.

Este trabalho apresenta a proposta de um conjunto de rotinas para a execução de programas paralelos implementados em linguagens orientadas a objetos, em aglomerados de estações de trabalho heterogêneas, levando em consideração a existência de outros usuários em cada um dos nós do aglomerado.

A seção 2 descreve sucintamente a linguagem Ágata que será utilizada na seção 4 para mostrar um exemplo real e seus resultados. A seção 3 descreve a proposta do conjunto de rotinas para um sistema de execução de programas paralelos, e a estratégia adotada para o mapeamento dos objetos em um aglomerado de estações de trabalho.

## 2 - A LINGUAGEM ÁGATA

A linguagem ÁGATA foi projetada dentro do paradigma de orientação a objetos, como uma ferramenta de programação paralela inicialmente voltada à programação de multiprocessadores com memória compartilhada [SALVA94].

### 2.1 - Características da linguagem.

Neste item segue uma descrição sucinta da linguagem. Mais detalhes podem ser encontrados em [SALVADOR94].

- A sintaxe para expressão do paradigma OO segue o padrão C++;

- O paralelismo é obtido através da execução em paralelo de chamadas a métodos de objetos;

- A linguagem não permite a declaração de dados públicos, mas apenas chamadas aos métodos públicos de uma classe;

- A linguagem não permite a declaração de variáveis globais;

- As construções sintáticas básicas como definição de tipos ( que não classes ), atribuições, cálculo de expressões e uso de funções seguem o modelo ANSI C [KERN90].

A definição da linguagem ÁGATA permite a exploração de paralelismo em 3 níveis:

• O Paralelismo Inter-Objetos prevê a execução paralela de métodos de objetos distintos.

• O Paralelismo Intra-Objetos, que se divide em dois casos distintos:

- Paralelismo Inter-Métodos prevê a execução paralela de diferentes métodos de um mesmo objeto, e

- Paralelismo Intra-Método que prevê a utilização de loops paralelos dentro de um mesmo método.

## 3 - O SISTEMA DE EXECUÇÃO.

Sistemas baseados em arquiteturas MIMD com memória distribuída oferecem maior escalabilidade em relação aos sistemas com memória compartilhada.

Em particular, os aglomerados de estações de trabalho podem ser enquadrados no modelo MIMD com memória distribuída, uma vez que cada máquina da rede possui processador e memória local, e dispõe de um sistema de comunicação que permite a troca de mensagens entre os nós.

Os avanços em novas tecnologias de interconexão ponto-a-ponto como os padrões ATM, SCI, Fibre Channel, HIPPI, entre outros, proporcionam velocidades de comunicação de 10 a 1000 Mbytes/s nas redes, o que viabiliza a execução de aplicações de granularidade[1] grossa e média. Um breve panorama sobre o estado da arte em padrões de comunicação ponto-a-ponto de alta velocidade pode ser visto em [KOFUJ94].

O sistema de execução de programas paralelos apresentado, tem como principal objetivo, permitir a execução paralela de métodos dos objetos chamados. O sistema também permite que o usuário indique quais objetos devem residir e ter seus métodos executados localmente, e quais podem ser distribuídos pelos outros nós.

Existem dois tipos distintos de objetos que o sistema prevê : os objetos **persistentes** e os **não-persistentes**.

Se uma classe é definida como persistente, isto implica que o objeto deve conservar seus atributos entre uma execução e outra de qualquer de seus métodos. Quando ocorre a primeira chamada a um de seus métodos, o objeto deve ser associado a um nó e aí permanecer até a sua destruição ou até que o programa termine. O mapeamento de objetos persistentes pode ser feito automaticamente por um escalonador que irá alocar o nó que oferece maior oferta de processamento nesse momento. O usuário tem a possibilidade de explicitar o nó onde cada um de seus objetos persistentes deve residir durante toda sua existência. Isto permite aproveitar possíveis funcionalidades particulares de certas máquinas da rede como por exemplo maior quantidade de memória, processadores vetoriais, etc.

Os objetos de classes declaradas como não-persistentes não têm seus atributos mantidos entre uma chamada e outra de seus métodos. Isto permite que cada execução de seus métodos possa ser processada em uma máquina diferente da chamada anterior, inclusive ao mesmo tempo. Isto confere um grau adicional de paralelismo aos programas através da replicação de objetos não-persistentes. O sistema faz o mapeamento de objetos não-persistentes a cada chamada de um de seus métodos, de acordo com a oferta de processamento de cada nó do aglomerado.

### 3.1. O mapeamento dos objetos no sistema distribuído

Nos sistemas centralizados convencionais, um *escalonador* [TANEN92] trata da divisão adequada do processador entre os diversos processos. O escalonamento em sistemas distribuídos envolve dois aspectos do problema, quanto à localidade : estratégias *locais* e *globais de escalonamento*.

**Estratégias globais de escalonamento ( mapeamento )**

Sob o aspecto de *estratégia global de escalonamento*, o problema da *alocação de processadores* ("processor allocation") [TANEN92] também chamada de

[1] A granularidade indica o volume de processamento que uma máquina deve executar, até que necesite trocar uma nova mensagem com outra máquina.

*mapeamento*[2] ("mapping") [BAL 90] quando se trata de unidades de um mesmo programa, trata da decisão sobre quais processos serão executados em quais processadores ou nós de um aglomerado. O objetivo do mapeamento é minimizar o tempo de execução de um programa e pode ser feito em diversas fases diferentes :

- O mapeamento *estático* é feito durante a fase de compilação do programa paralelo. Apesar de ser menos flexível, o mapeamento estático permite que o programador conheça previamente quais das unidades residem no mesmo processador. Isto permite tirar vantagens como a redução do custo da comunicação entre tais unidades. As linguagens **occam** [INMOS84]e **StarMod** [COOK 80] são exemplos que utilizam mapeamento estático.

- Com o mapeamento dinâmico *fixo em tempo de execução* o sistema determina o mapeamento no momento da criação do processo, em tempo de execução. Uma vez estabelecido, o mapeamento não pode ser alterado até que o processo termine, ou que o programa chegue ao final. As linguagens ParAlfl [HUDAK86] e Concurrent Prolog [SHAPI84] [TAYLOR84].

- No mapeamento dinâmico *livre com migração de processos*, o mapeamento também é criado em tempo de execução. A diferença é que um processo pode ser executado em diversos locais durante a sua existência [HSIEH93]. As linguagens Emerald [JUL 88] e DOME [BEGUE94], os sistemas operacionais DEMOS/MP [POWEL83] utilizam migração de processos ou objetos. Além destes, uma coleção de outros sistemas e linguagens são comentados em [NUTAL94].

Trabalhos como [EAGER88] mostram que não há ganhos significativos quando se faz migração de processos ativos, pois o *overhead* criado para fazer a migração é grande. Esta idéia é compartilhada por [TANEN92], que argumenta que é bem melhor utilizar algoritmos simples e baratos, que não geram grande *overhead*, do que utilizar algoritmos complicados e caros computacionalmente. Ganhos significativos de desempenho podem ser obtidos através de migração em certos tipos de aplicações com objetos persistentes de longa duração, e requisitos estreitos quanto ao tempo de resposta. Neste caso, é possível que ocorram situações onde o sistema permaneça com a carga desbalanceada durante um período longo, significativamente maior do que o overhead gerado pela migração.

**Estratégias locais de escalonamento.**

O outro aspecto trata da *estratégia local de escalonamento*, no âmbito interno de um dado processador. Esta questão apresenta uma similaridade apenas aparente com o problema de escalonamento em sistemas centralizados. Entretanto, existem aspectos relativos à ocorrência de eventuais comunicações ou sincronizações com processos residentes em outros processadores [TANEN92] e que poderão ser levados em conta dependendo da abordagem que se dê ao problema .

Os compiladores tratam do mapeamento de diferentes processos de um mesmo programa, visando obter máximo desempenho através do paralelismo, ou máxima confiabilidade através de replicações. Um sistema operacional distribuído, ao contrário,

---

[2] Neste trabalho o termo mapeameto será utilizado para indicar a relação de associação entre os objetos da aplicação e os processadores físicos, que determina onde cada objeto reside e onde seus métodos devem ser executados.

deve criar um mapeamento para processos de diferentes programas e diferentes usuários. Existem diferenças conceituais, pois enquanto que nos sistemas operacionais distribuídos os processos competem pelos recursos, nos compiladores os processos trabalham de maneira cooperativa. É oportuno ressaltar que a preocupação deste trabalho está na visão do problema sob o ponto de vista de compiladores.

### 3.2. O mecanismo de mapeamento utilizado

Sempre que um método de um objeto pertencente a uma classe não-persistente for chamado, o sistema escolhe qual o melhor nó do aglomerado para atender a chamada. No caso de objetos de classes persistentes, essa escolha deverá ser feita apenas quando ocorrer a primeira chamada a um de seus métodos.

O mecanismo de mapeamento é descentralizado. Assim, quando um método do objeto A que esteja em execução no nó *i*, invocar um método do objeto persistente B em um nó *j*, o mapeamento é feito localmente, pelo próprio nó *i*. O nó *i* deverá atualizar uma tabela de mapeamento local, e informar a todos os demais nós do aglomerado, sobre o mapeamento feito. A tabela de mapeamento é composta pela identificação do objeto e a identificação do nó alocado.

Uma vez feito o mapeamento, o nó escolhido passará a atender todas as futuras chamadas a métodos desse objeto persistente. Atualmente, o sistema não utiliza nenhum mecanismo rigoroso de consistência entre as réplicas das tabelas.

O sistema mantém uma fila local de chamadas a métodos, para cada um dos nós. Mapear um objeto consiste em determinar em qual das filas a chamada deve ser colocada. É importante ressaltar que o sistema não cria um processo para cada objeto, mas possui um único processo em cada nó que gerência e executa as chamadas da fila local. Isto torna o mecanismo de mapeamento mais eficiente.

### 3.3. A Estratégia de mapeamento adotada

A escolha da máquina menos carregada da rede, é feita através de um parâmetro $X_i$ que indica a carga sobre cada máquina, onde :

$$X_i = \frac{C_i}{1-O_i} \quad ; \; i \in \{ 0, .., n-1 \}$$

Onde :

$O_i$ é um índice para medir a ocupação de CPU do nó $i$ no último intervalo de medição. Valores de $0 \leq O_i < 1$ indicam a que a CPU permaneceu algum tempo livre no último intervalo de medição. Valores a partir de 1 indicam sobrecarga do nó. Este valor é um parâmetro fornecido pelo sistema operacional de cada nó.

$C_i$ é a carga do nó $i$ em um dado instante. $C_i$ é dado pela somatória :

$$C_i = \sum_{j=1}^{k} t_j ,$$

onde :

$k$ é a quantidade de métodos na fila de execução do nó $i$.

$t_j$ é o tempo que o nó $i$ gastaria para executar o $j$-ésimo método da fila local de chamadas de métodos, dado por :

$$t_j = tb_j \cdot \frac{1}{M_i}$$

$M_i$ é o coeficiente de desempenho de cada nó $i$ do aglomerado, em relação ao nó padrão ( nó de número 0 ).

sendo que $tb_j$ é o tempo que p nó padrão gasta para executar o método $j$

Assim, a melhor escolha será o nó $i$, tal que o seu valor $X_i$ seja o mínimo para $i = 0, .. , n-1$. Além de levar em conta a capacidade de cada nó, a estratégia dá preferência a um nó com maior ociosidade, ou menor sobrecarga, nesta ordem. Mais precisamente, esta escolha é feita pela função de mínimo dada por :

$$G : A \times A \rightarrow \mathbb{R}, \text{ onde :}$$

$$G(X_i, X_j) = \begin{cases} \min\{X_i, X_j\}, & \text{se } X_i \cdot X_j > 0 \\ \max\{X_i, X_j\}, & \text{se } X_i \cdot X_j < 0 \\ X_i, & \text{se } X_i = 0 \\ X_j, & \text{se } X_j = 0 \text{ e } X_i \neq 0 \\ X_i, & \text{se } X_i > 0 \text{ e } X_j = \infty \\ X_j, & \text{se } X_j > 0 \text{ e } X_i = \infty \\ \infty, & \text{nos demais casos} \end{cases}$$

$$i, j \in \{ 0, 1, ..., n - 1 \}$$

$$A = \{ \; X_i = \frac{C_i}{1 - O_i} \mid i = 0, ..., n\text{-}1 \; \}$$

onde $n$ é a quantidade de nós do aglomerado.

### 3.2. O conjunto de rotinas para execução de programas paralelos

Este item apresenta uma descrição da biblioteca que o sistema oferece para executar chamadas a métodos dos objetos de um programa paralelo.

**a ) _executa_np**

Executa método de objeto não-persistente em um nó específico do aglomerado.

Sintaxe :

```
int info = _executa_np ( int no, int obj, unsigned long
metodo,int tam_lista, ... )
```

Argumentos :

| | |
|---|---|
| `no` | número do nó, onde o objeto não-persistente deverá residir ser executado. |
| `obj` | número do objeto na tabela de objetos |
| `metodo` | endereço do método a ser executado |
| `tam_lista` | tamanho da lista de argumentos a ser passada para o método. |

Descrição :

Logo após o argumento `tam_lista` deve seguir a lista de argumentos a ser enviada ao método a ser chamado. Caso `no` seja igual a -1, o sistema irá executar o método especificado localmente. Esta rotina não altera a tabela de mapeamento dos objetos.

b) `_executa_np_map`

Executa método de objeto não-persistente em um nó do aglomerado, utilizando a função de mapeamento.

Sintaxe :

```
int info = _executa_np_map ( int obj, unsigned long
metodo,
                             int tam_lista, ...)
```

Argumentos :

| | |
|---|---|
| `obj` | número do objeto na tabela de objetos |
| `metodo` | endereço do método a ser executado |
| `tam_lista` | tamanho da lista de argumentos a ser passada para o método. |

Descrição :

Logo após o argumento `tam_lista`, deve seguir a lista de argumentos a ser enviada ao método a ser chamado. Esta rotina verifica qual o nó do aglomerado com melhores condições para executar o método e envia uma chamada remota. Esta rotina não altera a tabela de mapeamento dos objetos.

c) `_executa_p`

Executa método de objeto persistente um nó específico do aglomerado.

Sintaxe :

```
int info = _executa_p ( int no, int obj, unsigned
                        long metodo,int tam_lista, ... )
```

Argumentos :

| | |
|---|---|
| `no` | número do nó do aglomerado, onde o objeto persistente deverá residir permanentemente. |
| `obj` | número do objeto na tabela de objetos |
| `metodo` | endereço do método a ser executado |
| `tam_lista` | tamanho da lista de argumentos a ser passada para o método. |

Descrição :

Logo após o argumento `tam_lista`, deve seguir a lista de argumentos a ser enviada ao método a ser chamado. Caso **no** seja igual a **-1**, o sistema irá executar o método especificado localmente. Caso seja a primeira chamada a um método do objeto, a rotina registra que o objeto deve residir no nó especificado, e realiza a chamada. Caso não seja a primeira chamada a um método do objeto, o sistema ignora o parâmetro **no**, e obedece o que foi registrado na tabela de mapeamento.

d ) `_executa_p_map`

Executa método de objeto persistente um nó específico da rede, utilizando a função de mapeamento.

Sintaxe :

```
int info = _executa_p_map ( int obj, unsigned long
                            metodo, int tam_lista, ...)
```

Argumentos :

| | |
|---|---|
| `no` | número do nó do aglomerado, onde o objeto persistente deverá residir. |
| `obj` | número do objeto na tabela de objetos |
| `metodo` | endereço do método a ser executado |
| `tam_lista` | tamanho da lista de argumentos a ser passada para o método. |

Descrição :

Logo após o argumento `tam_lista` deve seguir a lista de argumentos a ser enviada ao método a ser chamado. Caso seja a primeira chamada a um método do objeto, o sistema irá determinar um mapeamento em função das condições de cada nó do aglomerado, e registrar essa informação na tabela de mapeamento dos objetos. Caso já exista um mapeamento para o objeto especificado, o sistema simplesmente obedece o mapeamento existente.

### 3.3. A semântica de passagem de parâmetros utilizada

As chamadas remotas a métodos seguem a semântica de passagem de parâmetros por valor. Passagens de parâmetro por referência são dependentes do

contexto local, e perdem completamente o sentido para uma execução remota. A passagem de um método como parâmetro na chamada de outro método deve ser inicialmente subdividida em duas etapas. Ex.:

objB.met1( objA.met1 ( ) ):

Deve ser transformada em algo do tipo :

```
temp = objA.met1 ( );
objB.met1 ( temp );
```

### 3.4. Chamadas síncronas

Chamadas síncronas são aquelas onde o método chamado deve devolver algum valor. Neste caso, se o método chamado é local, o sistema irá promover a sua execução e a devolução do resultado sem maiores problemas.

Caso o método resida em outro nó, é necessário esperar que sua execução termine e que o resultado seja devolvido para então continuar o processamento local. A espera pelo retorno, entretanto, é sem dúvida nenhuma um limitante do paralelismo. Atualmente o sistema está sendo modificado no sentido de permitir que o bloqueio de um único objeto causado por uma chamada síncrona, não bloqueie também as outras chamadas que aguardam na fila local de execução.

## 4 - UM EXEMPLO REAL

Esta seção apresenta um exemplo real para avaliação do desempenho do sistema proposto. O sistema proposto foi implementado utilizando a biblioteca PVM ( *Parallel Virtual Machine* ) [GUEIS94], que oferece portabilidade sobre um conjunto grande de fabricantes e modelos de máquinas. O sistema PVM oferece um conjunto de rotinas de comunicação entre máquinas com diferentes formatos de armazenamento de dados, rotinas para criação e gerenciamento de processos, além de uma ferramenta de monitoramento das aplicações com interface gráfica. A figura 1 mostra a visão que o sistema dá ao usuário, sobre o aglomerado de estações.

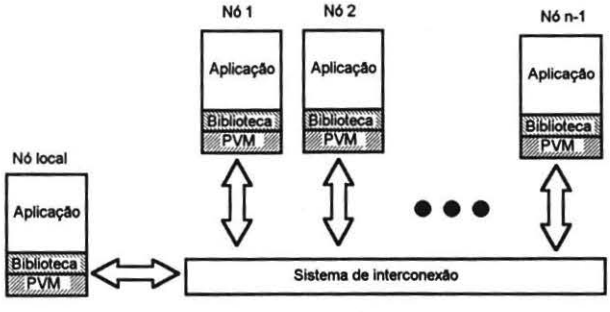


figura 1

### 4.1. O exemplo usado para teste

O código abaixo é equivalente ao que foi executado para medir o desempenho obtido.

```
#define size 200
#define times 100
class matriz {
        double a[size][size], b[size][size], c[size][size];
public :
        matriz ( );
        void mult( );
};
matriz :: matriz( )
{
        int i, j;
        for(i=0; i<size; i++)
                for(j=0; j<size;j++) {
                        a[i][j] = (double) i + j;
                        b[i][j] = (double) i - j;
                        c[i][j] = 0.0;
                }
}
void matriz::mult( )
{
        int i, j, k;
        for (i=0; i<size;i++)
                for(j=0; j<size; j++) {
                        a[i][j] = (double) i + j;
                        b[i][j] = (double) i - j;
                        c[i][j] = 0.0;
                }
        for (i=0; i<size;i++)
                for(j=0; j<size; j++)
                        for(k=0; k<size;k++)
                                c[i][j] += a[i][k] * b[k][j];
}
void main()
{
        matriz Mat1;
        int i;

        for  (i=0; i<times;i++) {
                Mat1.mult( );
                delay(1); /* simula um trecho sequencial do programa do usuário */
        }
}
```

### 4.2. O ambiente usado para teste

O ambiente utilizado para os testes foi uma rede local com 4 estações de trabalho cujas características são descritas a seguir :

| No | Modelo | CPU/FPU | Memória |
|---|---|---|---|
| 0 | SGI-Indigo | R4000/R4010 | 64 Mbytes |
| 1 | SGI-Indy | R4000/R4010 | 64 Mbytes |
| 2 | SGI-Indy | R4000/R4010 | 64 Mbytes |
| 3 | SGI-Indy | R4000/R4010 | 32 Mbytes |

O teste gerou os seguintes resultados :

| Nó | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| t-seq (segundos) | 1065 | 1090 | 1082 | 1482 |
| t-par (segundos) | 301.5 | 301.5 | 301.5 | 301.5 |
| speedup | 3.53 | 3.62 | 3.59 | 4.92 |

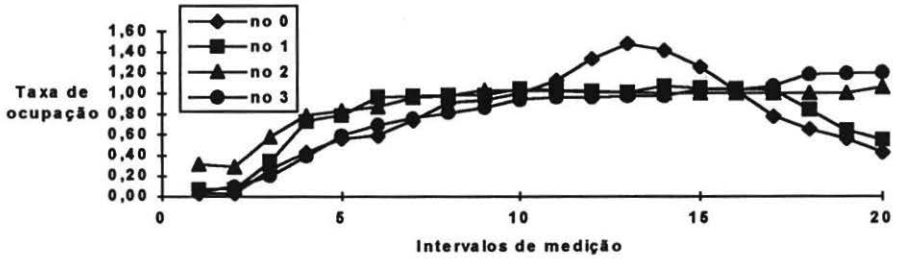


**figura 2 - situação de carga dos nós durante o teste**

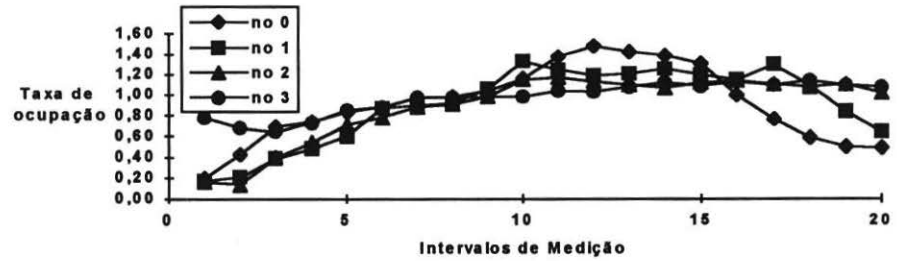


**figura 3 - situação de carga nos nós**

As figuras 2 e 3 ilustram a situação de carga nos nós, ao longo do tempo de execução do programa exemplo. O valor da medida indica a carga média de cada nó

nos últimos sessenta segundos. O teste mede a situação de carga em cada nó, a cada intervalo de dez segundos.

Os gráficos apresentam um período ascendente até a sexta medida. A partir daí, existe um período de estabilidade que mostra que a carga foi balanceada. Como o mecanismo utilizado para medir a carga fornece a média sobre o último minuto, existe um atraso nas informações, no início da execução.

A figura 2 mostra uma elevação na carga de um dos nós, causada pela interferência de outros usuários.

## 5 - CONCLUSÕES

O trabalho apresenta como contribuição uma estratégia de distribuição de objetos em ambientes distribuídos heterogêneos.

A estratégia proposta para manter o balanceamento de carga dos nós se mostrou adequada, promovendo resultados satisfatórios nos testes realizados. O *speedup* conseguido ficou próximo do máximo possível para o aglomerado de estações utilizado.

Algumas otimizações estão sendo implementadas no sentido de obter dados mais significativos sobre a carga dos nós, proporcionando decisões mais rápidas quando ocorrem alterações abruptas causadas pelos outros usuários.

## BIBLIOGRAFIA


[BAL 90] BAL, H.; **Programming Distributed Systems**. Silicon Press, Summit, New Jersey, Prentice-Hall, 1990.

[BEGUE94] BEGUELIN, A., SELIGMAN, E., STARKEY, M.; **"DOME: Distributed Object Migration Environment".** Tech. Report, School of Computer Science, Carnegie Mellon University, CMU-CS-94-153, Pittsburgh, 1994.

[BEN-A90] BEN-ARI, M.;**Principles os Concurrent and Distributed Programming**. Prentice-Hall, Cambridge, UK, 1990.

[BURNS87] BURNS, A., LISTER, A.M., WELLING, A.J.; **"A review of Ada Tasking"** *Lecture Notes in Computer Science*, 262. Springer-Verlag, Berlin.

[COOK 80] COOK, R.P. **"*MOD- A language for Distributed Programming"**. *IEEE Trans. Soft. Engin.* , vol. SE-6, N° 6, p. 563-571.

[EAGER88] EAGER, D.L., LAZOWSKA, E.D., e ZAHORJAN, J.; **"The Limited Performance Benefits of Migrating Active Processes for Load Sharing"**.Proc. 1988 SIGMETRICS Conference on Mensurement and Modeling of Computer Syetems, Performance of Evaluation Review. Maio 1988.

[GEHAN92] GEHANI, N.C, ROOME, W.D.; **Implementing Concurrent C.** Software-Practice and Experience, vol. 22 março de 1992.

[GUEIS94] GUEIST, A. et. al. **PVM: Parallel Virtual Machine.A User's Guide and Tutorial for Networked Parallel Computing.** MIT Press, Massachusetts, 1994.

[HSIEH93] HSIEH, W.C., WANG, P., and WEIHL, W.E., **Computation Migration Locality for Distributed-Memory Parallel Systems".** *4th ACM SIGPLAN Symp. on Principles & Pratices of Parallel Programming - POPPS*,San Diego, CA, Maio, 1993.

[HUDAK86] HUDAK, P. **"Para-Functional Programming"***IEEE Computer*, Vol. 19 N°8, p. 60-70.

[INMOS84] **Occam Programming Manual**. Prentice-Hall, Englewood Clifs, NJ, 1984.

[KOFUJ94] KOFUJI, S.T.; **Considerações de Projeto e Análise do SPADE - um Multiprocessador de Larga Escala baseado no padrão ANSI/IEEE-SCI** - Tese de Doutoramento - Depto. de engenharia Eletrônica da Escola Politécnica da USP. São Paulo, 1994.

[NUTAL94] NUTALL, M., **"A brief survey of systems providing process or object migration facilities".** *Operating System Review*, Outubro, 1994.

[POWEL83] POWELL, M.L., MILLER, B.P., **"Process Migration in DEMOS/MP".** *Proc. of 9th ACM Symp. on Operating Systems Pinciples*.Bretton Woods, NH, USA, Outubro de 1983.

[SALVA94] SALVADOR, L.N., SATO, L.M.; **"Uma Linguagem de Programação Orientada a Objetos Para Ambientes Paralelos".** In: Anais do VI Simpósio Brasileiro de Arquitetura de Computadores. Caxambú, Agosto, 1994.

[SHAPI84] SHAPIRO, E., MIEROWSKY, C. **"Fair, Biased, and Self-Balancing Merge Operations : Their Specifications and Implementation in Concurrent Prolog".** Journal of New Generation Computing, Vol. 2 N° 3, p. 221-240.

[TANEN92] TANENBAUM, A.S. **Modern Operating Systems.** Prentice-Hall, New Jersey, 1992.

[TAYLO87] TAYLOR, S., AV-RON, E., SHAPIRO, E. **"A Layered Method for Process and Code Mapping".** Journal of New Generation Computing, Vol.5,N°2, p. 185-205., 1987.