Aplicando um sistema de execução de programas no suporte a linguagens paralelas orientadas a objetos.[1]

Hermes Senger[¥] Líria Matsumoto Sato[†] Laís do Nascimento Salvador[¥]

[†]PCS - Departamento de Engenharia de
Computação e Sistemas Digitais
e-mail : liria@pcs.usp.br

[¥]LSI-Laboratório de Sistemas Integráveis
e-mail : hermes@lsi.usp.br

Escola Politécnica da Universidade de São Paulo
Edifício de Engenharia de Eletricidade
Av. Prof. Luciano Gualberto, trav. 3, n° 158
05508-900 - São Paulo, SP
Tel. (011) 818-55589/818-5662

RESUMO

Este trabalho apresenta a maneira como um sistema de execução de programas distribuído pode ser utilizado, no sentido de fornecer o suporte à execução de programas escritos em uma linguagem paralela orientada a objetos. Nesse sentido, o trabalho apresenta a maneira como os programas escritos em uma linguagem específica podem ser traduzidos para o modelo de computação distribuída implementado no sistema.

ABSTRACT

This paper presents how a specific distributed run-time system can be used, supporting the execution of programs written in a parallel object-oriented language. As an example, it is presented the mapping of programs written in a specific language to the distributed computing model, offered by the system.

[1] Este trabalho foi financiado pelos projetos FINEP/PAD (processo n° 5.6.94.0260.00), e FINEP/ RECOPE (processo 3607/96).

1. Introdução

Nos aglomerados de estações de trabalho e microcomputadores, interligados através de uma rede local, grande parte dos ciclos de *clock* são desperdiçados [1]. Assim, uma estratégia bastante atraente consiste em particionar tarefas computacionalmente complexas, ou de grandes dimensões, distribuindo-as entre os computadores de uma rede local.

Por outro lado, o paradigma de orientação a objetos tem recebido a atenção de diversos projetos de linguagens e ferramentas de processamento distribuído [2,3], pois as unidades de particionamento de programas podem ser os próprios objetos, que são distribuídos entre os nós de processamento. O encapsulamento de dados [4] inibe as tentativas de um objeto ter acesso direto sobre os dados de outros objetos (que podem residir em outro nó), e portanto, reduz a carga de comunicação gerada sobre a rede.

Os aspectos do projeto e da implementação do sistema encontram-se presentes de maneira mais detalhada em [5]. Aqui serão apresentados os aspectos relativos à aplicação desse sistema no suporte à execução de programas escritos na linguagem baseada em objetos denominada ÁGATA [6]. Além desta aplicação, o sistema oferece suporte à execução de programas escritos em outras linguagens orientadas a objetos, como a C++[7] por exemplo, desde que sejam observadas algumas condições, também discutidas no decorrer deste trabalho.

1.1. O contexto do projeto.

Uma das finalidades primordiais do projeto desse sistema é oferecer suporte à execução de programas escritos em linguagens orientadas a objetos paralelas, em particular à linguagem ÁGATA. A linguagem ÁGATA possui sintaxe semelhante à da C++ para expressar o paradigma de objetos. Em ÁGATA, existem construções que permitem explicitar aspectos de paralelismo dos objetos. Uma primeira versão do compilador foi implementada para uma arquitetura multiprocessadora com memória compartilhada. Para a implementação de aplicações, a linguagem ÁGATA oferece três tipos de objetos:

•Objetos totalmente paralelos : permitem a execução paralela de seus métodos

•Objetos parcialmente paralelos : alguns de seus métodos são do tipo ***mutex***, isto é, não podem ser executados em paralelo com outros métodos do mesmo objeto. Os demais métodos podem ser executados de forma paralela.

•Objetos seqüenciais : são os objetos de classes do tipo ***monitor***. É como se todos os seus métodos fossem do tipo ***mutex***, ou seja, não permitem qualquer paralelismo entre os seus métodos.

O trabalho apresenta uma descrição resumida do modelo de execução de programas distribuídos em aglomerados de estações, implementado pelo sistema, como também uma proposta de mapeamento de programas escritos em ÁGATA para o modelo distribuído.

1.2. Aglomerados de estações de trabalho.

Um dos principais fatores que motivaram o projeto desse sistema, foi o fato de que já se encontra implementada uma primeira versão de um compilador ÁGATA, com um sistema de execução de programas para uma arquitetura multiprocessadora, com memória compartilhada. O próximo passo é implementar um compilador para arquiteturas com memória distribuída. Neste sentido, o sistema aqui apresentado gerencia a execução dos métodos dos objetos no aglomerados de estações.

Para se obter alto desempenho, é preciso distribuir corretamente a carga de trabalho entre os nós do aglomerado. Assim, o sistema detecta a existência de recursos ociosos (processadores livres ou com baixa utilização), também como forma de se evitar a sobrecarga de algumas estações que já estejam completamente ocupadas com tarefas de seus usuários locais. Outro fator considerado pelo sistema é a heterogeneidade, pois podem coexistir máquinas de diversos modelos e fabricantes no mesmo aglomerado, cada qual com diferentes capacidades de desempenho.

2. O suporte oferecido pelo sistema de execução de programas paralelos

No modelo orientado a objetos, um programa paralelo é constituído de um conjunto de objetos que podem permanecer residentes ou não, em um determinado nó. Em um dado instante vários métodos desses objetos podem estar em execução, de acordo com as três modalidades citadas na seção 1.1.

Os objetos são constituídos de métodos associados a um estado. No contexto de programação distribuída, o sistema discutido neste trabalho prevê a existência de dois tipos distintos de objetos : os ***regulares*** e os ***de estado volátil***.

Se uma classe é definida como *regular*, 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 *regulares* pode ser feito automaticamente por um escalonador que irá alocar o nó que oferece maior oferta de processamento nesse momento. O sistema permite também a possibilidade de explicitar o nó onde cada um dos objetos *regulares* 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 *de estado volátil* não terão seus atributos mantidos entre uma chamada e outra de seus métodos. Isto permite que cada execução de seus métodos possa ser processada em um nó diferente da chamada anterior, inclusive ao mesmo tempo. O sistema faz o mapeamento de objetos *de estado volátil* quando é chamado um de seus métodos pela primeira veez, de acordo com a oferta de processamento de cada nó do aglomerado.

O modelo de orientação a objetos apresenta aspectos interessantes à exploração do paralelismo. O encapsulamento determina que um método só poderá ter acesso direto aos dados pertencentes ao seu objeto. Isto torna cada objeto um módulo estanque, de forma que grande parte das dependências de dados são naturalmente eliminadas. Cada método, para ser executado, dependerá apenas dos argumentos que lhe forem passados além dos atributos do seu objeto. O sistema não prevê o compartilhamento de variáveis entre objetos residentes em diferentes nós. Dessa forma, o sistema não permite a utilização de variáveis globais ou mesmo de classes com dados públicos. Uma alternativa consiste em utilizar objetos com métodos públicos da classe, que fazem o acesso a dados privados. A chamada a métodos dos objetos pode ser feita com o envio de parâmetros e o recebimento do resultado.

2.1. O mapeamento dos objetos

Para obter um bom desempenho, é preciso obter informações sobre a existência de recursos ociosos no aglomerado, e distribuir os objetos de um programa paralelo em tempo de execução, de acordo com essas informações (Figura 1). A essa distribuição dá-se o nome de mapeamento [2].

O sistema efetua o mapeamento no instante em que ocorre a primeira chamada a um método qualquer de um determinado objeto, em função da situação de carga do sistema. A partir da segunda chamada, o sistema irá direcionar o pedido seguindo o mapeamento anteriormente estabelecido. Para os objetos *de estado volátil*, cujo estado não precisa ser mantido entre uma chamada e outra de qualquer de seus métodos, o mapeamento pode ser feito novamente a cada chamada. Isto favorece a manutenção do balanceamento de carga.

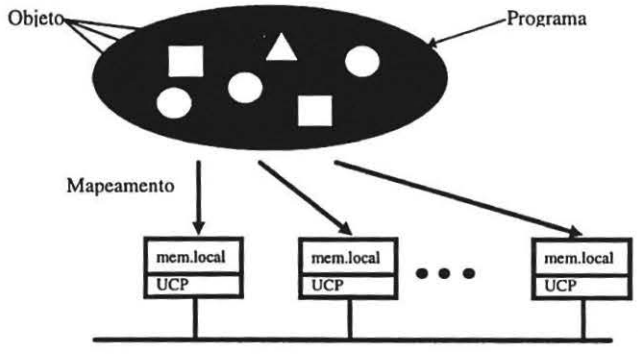

Figura 1 - Programa paralelo composto de objetos.

O usuário do sistema pode escolher o nó em que o objeto deverá residir, aproveitando eventuais funcionalidades especiais de certos nós do aglomerados como processamento vetorial, unidade de discos local, etc., e assim aumentar o desempenho da aplicação.

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 *regular* B em um nó *j*, o mapeamento será feito localmente, pelo próprio nó *i*, que 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 *regular*. Atualmente, o sistema não utiliza nenhum mecanismo de tolerância a falhas.

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 gerencia e executa as chamadas da fila local. Isto torna o mecanismo de mapeamento mais eficiente.

A estratégia de mapeamento, vista sob o aspecto do sistema operacional, leva em conta a carga de trabalho de cada nó do aglomerado (informação que é fornecida pelo núcleo local de cada nó), bem como uma estimativa da granularidade implícita no objeto, e um fator de desempenho de cada nó de processamento[5]. O mecanismo que implementa a estratégia de mapeamento é disponibilizado sob a forma de um conjunto de rotinas que promovem a chamada e gerenciam a execução dos métodos dos objetos.

3. Aplicações do sistema

O sistema apresentado pode ser utilizado de duas formas diferentes : como um sistema de execução, acoplado ao código gerado por um compilador de linguagem, ou ainda, como um conjunto de rotinas que pode ser utilizado diretamente por um programador de aplicações. O sistema não se limita a oferecer suporte apenas para a linguagem ÁGATA, mas para toda uma classe de linguagens que observem algumas características e restrições :

- A linguagem pode permitir o uso de variáveis globais, mas o programador deve ser informado de que o sistema não faz qualquer consistência entre as várias cópias existentes nos diversos nós.
- Não deve permitir o uso de dados públicos, mas apenas chamadas aos métodos públicos da classe.
- A linguagem deve permitir fazer uso de funções.
- A linguagem pode prever exploração de paralelismo entre métodos de um mesmo objeto e também dentro de um mesmo objeto através de *loops* paralelos. Na sua primeira versão o sistema não oferece suporte a tais facilidades, entretanto, isso está previsto como trabalho futuro no qual se pretende utilizar estações com recursos de multiprocessamento como nós do aglomerado.

3.1. A Linguagem ÁGATA

A linguagem ÁGATA foi proposta inicialmente com base em uma arquitetura multiprocessadora com memória compartilhada. Por outro lado, criou-se um novo modelo de execução baseado na distribuição dos objetos entre diversos nós de um aglomerado de estações, que não compartilham fisicamente uma área de memória. Apesar de existirem algumas diferenças entre o modelo original da ÁGATA e o modelo de execução distribuída aqui proposto, pode-se implementar um compilador de programas ÁGATA para um aglomerado de estações de trabalho, visando utilizar-se do suporte oferecido pelo sistema de execução. Para tal, é necessário estabelecer um mapeamento do modelo original da linguagem ÁGATA para o modelo distribuído.

3.1.1. Características da linguagem

Sintaticamente, a linguagem ÁGATA segue o padrão ANSI na declaração de tipos, atribuições, cálculo de expressões, declaração e chamada de funções, etc. Para expressar o paradigma de objetos, a linguagem segue a sintaxe de C++ [7]. A linguagem possui algumas características importantes, bastante propícias para a execução em uma arquitetura distribuída :

- Não permite a declaração de variáveis compartilhadas, isto é, visíveis a todos os objetos. A existência de variáveis compartilhadas aumentaria a responsabilidade do sistema de execução, que deveria oferecer o suporte necessário para que os objetos residentes em qualquer nó pudessem ler e escrever sobre tais variáveis, de maneira consistente. A utilização de variáveis compartilhadas poderia provocar dependências de dados entre objetos, comprometendo o desempenho dos programa.

- As classes não podem ter dados públicos, isto é, visíveis a outros objetos. Muito embora linguagens como C++ possuam tal característica, isto vai contra o princípio de encapsulamento previsto no paradigma de objetos. Se um objeto possui dados públicos, todos os seus usuários poderão ler e escrever sobre esses dados. Se o objeto e seu usuário residirem em nós distintos que não compartilham memória, o sistema de execução é quem deveria prover esse acesso remoto. Na hipótese de haver mais de um usuário tentando escrever sobre o mesmo dado público simultaneamente, o sistema deveria oferecer o suporte necessário para exclusão mútua em ambiente distribuído. Por todos esses motivos, o encapsulamento de dados é uma das principais características inerentes ao paradigma de objetos que favorece o uso de paralelismo.

•A resolução de funções é feita de forma seqüencial, sendo portanto, resolvida localmente no mesmo nó onde foi feita a chamada.

•As variáveis compartilhadas e os dados públicos podem ser substituídos por vários outros recursos existentes no paradigma.

3.1.2. O mapeamento de programas ÁGATA para o modelo de execução distribuída

A linguagem ÁGATA utiliza as palavras reservadas ***monitor*** e ***mutex*** para fazer certas restrições ao paralelismo entre métodos de um mesmo objeto (paralelismo intra-objeto). Quando os métodos alteram o estado do objeto, algumas vezes é necessário dispor de mecanismos de exclusão mútua, para evitar um possível estado inconsistente do objeto.

Se um método for declarado com a palavra reservada ***mutex***, então esse método não poderá ser executado em paralelo com mais nenhum outro método do mesmo objeto. A declaração de um objeto como ***monitor*** equivale a dizer que todos os seus métodos são do tipo ***mutex***, ou seja, não admitem qualquer paralelismo entre si.

```
par class C1 {
        ...
  public :
        void metodo1 ( );
        . . .
        int metodoN ( );
};
```

Figura 2 - Declaração de uma classe paralela.

Originalmente, a linguagem ÁGATA assume implicitamente o paralelismo inter-objetos. Nas arquiteturas que possuem memória distribuída, entretanto, o paralelismo é obtido através de métodos executando em nós remotos, o que implica em custos adicionais bastante elevados. Em função disto, nessas arquiteturas é conveniente que o usuário informe de maneira explícita quais são as classes cujos métodos possuem complexidade computacional elevada (e conseqüentemente, granularidade grossa) o suficiente para tornar a execução paralela viável. Assim, recomenda-se acrescentar o prefixo ***par*** ao vocabulário da linguagem, para indicar aquelas classes (Figura 2) cujas instâncias poderão ser mapeadas em nós remotos, através de chamadas às rotinas

_exec_regular_map ou ***_exec_volatile_st_map*** [5] que realizam o mapeamento automático, com balanceamento de carga. Para as demais classes, o compilador deverá gerar uma chamada à rotina ***_exec_regular*** ou ***_exec_volatile_st*** [5], que promovem a execução de um método de um objeto de classe *regular* e classe *de estado volátil*, respectivamente. Tais rotinas permitem que se especifique o nó onde o objeto deve ser mapeado, indicando explicitamente que tais objetos devem residir e ter suas chamadas executadas no próprio nó local.

Em sua primeira versão, o sistema de execução não prevê a existência de multiprocessamento nos nós do aglomerado. Todos os pedidos de execução de métodos enviados a um determinado nó são colocados em uma fila local de execução e processados seqüencialmente, seguindo a ordem de chegada. Uma vez que o sistema se limita ao monoprocessamento, e mecanismos do tipo *multithreading* também não são implementados numa primeira versão, torna-se desnecessária a implementação de mecanismos de suporte à exclusão mútua para a linguagem ÁGATA.

Todo objeto *regular* se comporta como se fosse do tipo ***monitor***, ou seja, terá seus métodos executados sempre seqüencialmente. Já os objetos de classe *de estado volátil*, não podem ser declarados com o tipo *monitor*, e nem tampouco possuir métodos *mutex*, uma vez que tais objetos podem ter seus métodos executados em nós diferentes do aglomerado, inclusive simultaneamente. Tal consistência deve ser realizada pelo compilador. Seria conveniente acrescentar as construções ***_regular*** e ***_volatile_state*** para declarar classes de objetos do tipo *regular*, e *de estado volátil*, respectivamente. A Figura 3-a ilustra a declaração de uma classe *monitor*.

```
monitor class C2 {
  public :
       void metodo1 ( );
       . . .
       int metodoN ( );
};
```

a)

```
class C3 {
  public :
       mutex void metodo1( );
       . . .
       int metodoN ( );
};
```

b)

Figura 3 -a) Declaração de classe monitor - b) Classe com método mutex.

Por uma questão de simplicidade e facilidade para o programador, o tipo *regular* deve ser considerado *default*, em caso de omissão. No exemplo da Figura 3-b a classe C3 define objetos *regulares*, que contém um método do tipo ***mutex***. O usuário deverá

declarar de forma explícita, como no exemplo da Figura 4, as classes *de estado volátil.*

Cabe salientar que, neste caso, a classe C4 não poderia ser do tipo monitor, e nem mesmo possuir métodos *mutex*.

```
volatile_state class C4{
  public :
        void metodo1 ( );
        . . .
        int metodoN ( );
};
```

Figura 4 - Classe de objetos de estado volátil.

Por razões de desempenho, os objetos de classes declaradas com o prefixo ***par*** serão mapeados segundo a estratégia de balanceamento de carga oferecida pelo sistema, por *default*. Sempre que ocorrer a primeira chamada a um de seus métodos este objeto será mapeado em um nó, onde deverá residir e ser executado, até sua destruição ou até o término do programa. O sistema de execução irá dirigir de forma automática todas as chamadas subseqüentes a métodos desse objeto para o nó onde o mesmo reside.

Opcionalmente, o usuário poderá explicitar, por ocasião da primeira chamada, um nó para o mapeamento. Dessa forma é possível aproveitar algumas funcionalidades específicas previamente conhecidas de certos nós do aglomerado. Por exemplo, um objeto cujos métodos realizam acesso a disco com grande freqüência, poderia residir em um nó que é o servidor dos arquivos envolvidos. No exemplo da Figura 5, o operador especial @ indica o mapeamento a ser feito.

Nos casos de paralelismo intra-método, quando um método faz uma chamada síncrona a outro método do mesmo objeto, o compilador não deverá gerar uma chamada às rotinas do sistema, mas ao invés disso, efetuar a chamada naturalmente, de maneira seqüencial. Isto deve-se ao fato de que os dois métodos pertencem ao mesmo objeto, e portanto, residem em um mesmo nó. Tratar uma chamada síncrona de forma seqüencial é conveniente, até por uma questão de simplicidade, em qualquer situação em que o objeto do método chamador e o objeto do método chamado residam no mesmo nó do aglomerado.

deverá bloquear temporariamente a execução de **A.met2 ( )** até que o método chamado envie o valor de retorno. Durante o tempo de espera por esse retorno, o sistema irá verificar o estado da fila local, e executar as chamadas que estão aguardando sua vez (**A.met1( ), B.met2( )**, etc.). A Figura 7 ilustra como ficaria o fluxo de execução nos dois nós envolvidos.

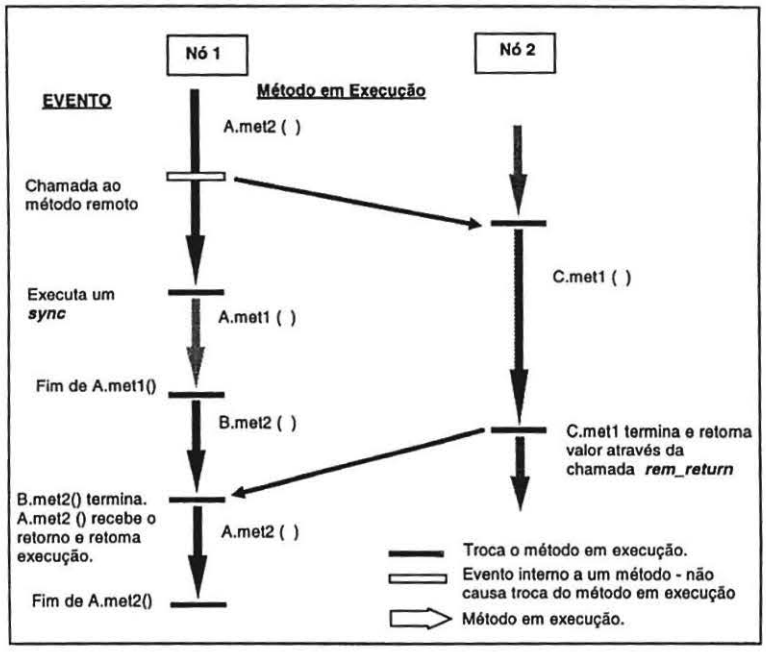

Figura 7 - Esquema de execução de uma chamada, com espera pelo resultado.

Neste ponto, é necessário deixar claro como fica a tradução de programas ÁGATA, utilizando os recursos oferecidos pelo sistema de execução. Isto é ilustrado na Figura 8.

No exemplo da Figura 8-b, o compilador deverá retardar ao máximo a ocorrência da chamada ***_sync***, para que seja utilizada somente quando o resultado for necessário. Uma chamada ***_sync*** deve conter a identificação do método chamado, e um endereço para o armazenamento do valor de retorno. O sistema cuidará de identificar o nó onde reside o objeto, cujo método foi chamado, e aguardar pelo respectivo retorno.

3.1.3. Comunicação.

A forma mais natural de comunicação entre objetos em ÁGATA se dá pela passagem de parâmetros e devolução de resultado nas chamadas a métodos. Existe

deverá bloquear temporariamente a execução de **A.met2 ( )** até que o método chamado envie o valor de retorno. Durante o tempo de espera por esse retorno, o sistema irá verificar o estado da fila local, e executar as chamadas que estão aguardando sua vez (**A.met1( ), B.met2( )**, etc.). A Figura 7 ilustra como ficaria o fluxo de execução nos dois nós envolvidos.

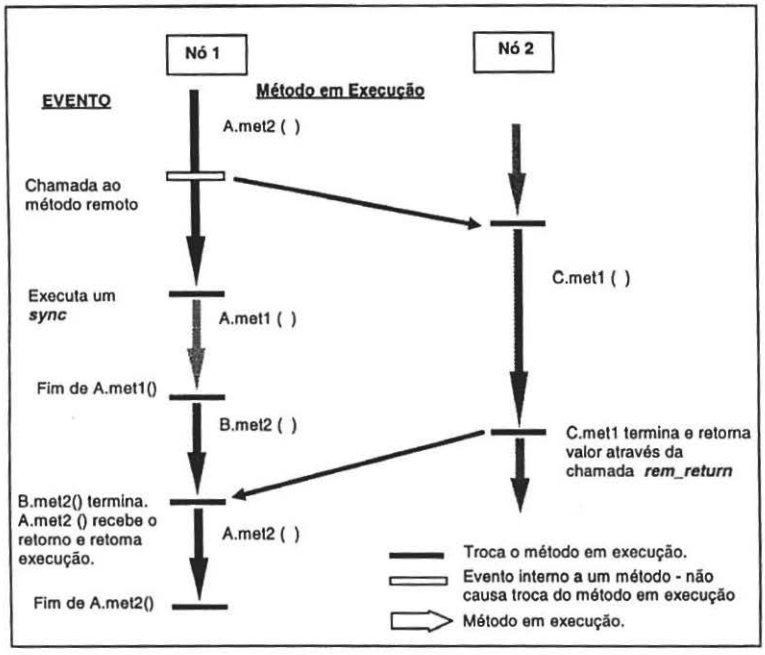

Figura 7 - Esquema de execução de uma chamada, com espera pelo resultado.

Neste ponto, é necessário deixar claro como fica a tradução de programas ÁGATA, utilizando os recursos oferecidos pelo sistema de execução. Isto é ilustrado na Figura 8.

No exemplo da Figura 8-b, o compilador deverá retardar ao máximo a ocorrência da chamada ***_sync***, para que seja utilizada somente quando o resultado for necessário. Uma chamada ***_sync*** deve conter a identificação do método chamado, e um endereço para o armazenamento do valor de retorno. O sistema cuidará de identificar o nó onde reside o objeto, cujo método foi chamado, e aguardar pelo respectivo retorno.

3.1.3. Comunicação.

A forma mais natural de comunicação entre objetos em ÁGATA se dá pela passagem de parâmetros e devolução de resultado nas chamadas a métodos. Existe

ainda, uma forma alternativa de comunicação entre objetos de classes diferentes através da passagem do endereço de um objeto como argumento para um método. O método chamado, apesar de receber esse endereço, não consegue ter visibilidade sobre os dados privados do outro objeto, mantendo assim o encapsulamento.

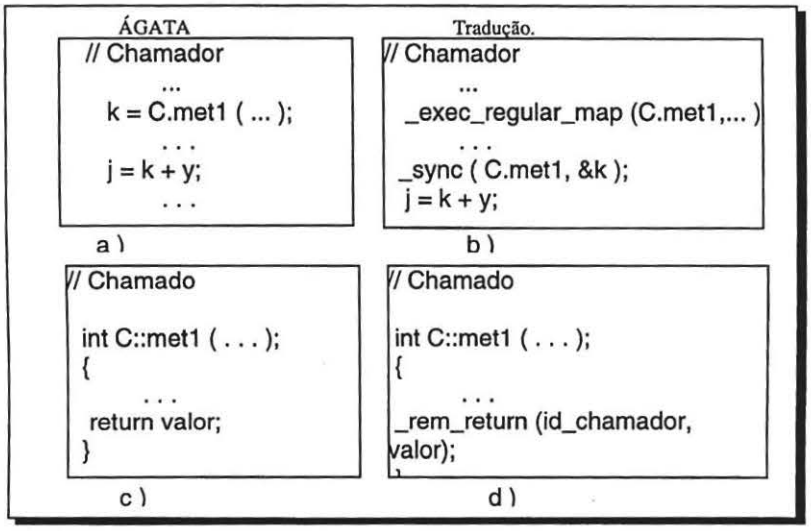

Figura 8 - Tradução de ÁGATA para rotinas do sistema.

Em um ambiente cujos objetos residem na memória local de diversos nós do aglomerado, e não em uma memória compartilhada, a prática de enviar endereços deve ser inibida, ou pelo menos controlada, pois um ponteiro só tem significado na memória local onde o objeto apontado reside. Verificar se dois objetos residem no mesmo nó em tempo de execução gera um esforço computacional extra, e inócuo na maioria das vezes. Isto torna aconselhável a um compilador ÁGATA para tal arquitetura, impedir a prática de comunicação entre objetos residentes em nós distintos através do envio de endereços.

3.2. Algumas considerações sobre o sistema

Além de fornecer suporte à execução de programas escritos em linguagem ÁGATA e de outras linguagens conforme mencionado na seção 4, o sistema pode ser utilizado também como uma biblioteca de rotinas, chamadas diretamente em programas fontes que implementam aplicações que exigem um esforço computacional maior do que aquele oferecido por uma única máquina, e que dispõe de uma rede local de estações de trabalho ou de PC's com uma taxa de utilização baixa.

Para se obter ganhos com o sistema conforme proposto neste trabalho, é preciso considerar a granularidade dos objetos. Certamente o mapeamento gera um *overhead* que deve ser evitado, quando o volume de processamento contido nos métodos de um objeto não for suficientemente grande. Dessa forma, os objetos com granularidade muito fina devem ser executados no nó local, isto é, no mesmo nó onde reside o objeto (ou programa principal) que fez a primeira chamada a um de seus métodos.

Se o sistema for utilizado por um compilador, a decisão sobre utilizar ou não o mapeamento poderá ser auxiliada por uma ferramenta especial para essa finalidade. Tal ferramenta teria de analisar as características do aglomerado e sugerir a granularidade mínima para objetos a serem mapeados. As principais características a serem analisadas são a relação entre velocidade de comunicação e capacidade de processamento dos nós (tempo médio para transmissão de uma mensagem, em ciclos de máquina) ou ainda, a situação de carga da rede em um dado instante.

3.3. Desempenho obtido.

Os detalhes sobre o projeto do sistema, sobre os resultados relativos ao balanceamento de carga, bem como o *speedup* obtido foram anteriormente apresentados em [5]. Aqui pretende-se ilustrar o desempenho de forma bastante sucinta. A título de exemplo, um programa ÁGATA que executa 100 vezes o método de multiplicação de matrizes foi traduzido para o ambiente de execução do sistema. Cada multiplicação envolve dois operandos matriciais de ordem 200. O ambiente utilizado para testes foi uma rede local com 4 estações de trabalho, sendo 3 estações SGI-Indy e uma SGI-Indigo, todas elas dotadas de CPU MIPS R4000 e FPU R4010. Uma estação Indy possuía 32 Mbytes (nó 3) de memória principal, enquanto que as demais (nós 0, 1 e 2) tinham 64 Mbytes. O quadro abaixo confronta o tempo médio gasto para executar todas as 100 multiplicações de forma seqüencial em cada um dos nós, com o tempo gasto quando as 100 multiplicações foram distribuídas.

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

O parâmetro **t-seq** representa o tempo que cada nó levou para concluir a tarefa de forma sequencial. Em seguida, **t-par** foi obtido distribuindo-se as 100 operações de multiplicação entre os 4 nós do aglomerado em situações normais de utilização, nas

quais cada nó poderia eventualmente estar executando tarefas de seus usuários interativos. Uma vez que os nós apresentam características ligeiramente diferentes, o *speedup* foi determinado com relação a cada um dos nós. Todos os valores representam a média de trinta execuções.

3.4. Comparação com outros sistemas

Existem diversos trabalhos propondo linguagens paralelas orientadas a objetos e seus modelos de computação paralela na literatura. Um dos projetos que obtiveram relativo destaque e apresenta algumas similaridades com a proposta aqui apresentada é o sistema que implementa a linguagem MENTAT [3].

Uma das semelhanças consiste na divisão estrutural no projeto, em duas partes; uma que trata da definição da linguagem com suas construções, sintaxe e semântica, e outra, que trata do sistema de execução (*run-time system).* Outro aspecto em comum é o fato de que tanto o projeto da MENTAT e seu sistema de execução quanto o sistema aqui apresentado, tratam de classes de objetos com estado volátil e não-volátil.

Na linguagem MENTAT o programador deve explicitar o paralelismo em seu programa fonte. No caso do sistema aqui apresentado, o paralelismo pode ser explicitado igualmente pelo programador, ou ainda, determinado por um compilador que realize a detecção automática de paralelismo. A semelhança portanto, reside no fato de que, tanto no sistema de execução proposto, quanto no RTS da linguagem MENTAT, a decisão sobre quando utilizar ou não utilizar o paralelismo não é de responsabilidade do sistema de execução.

O modelo de execução do sistema aqui proposto, por sua vez, difere totalmente do modelo usado pela linguagem MENTAT, que se baseia na construção de um grafo que modela macro dependências de dados entre os objetos. Esse grafo orienta as decisões sobre o escalonamento dos objetos, determinando quais chamadas a métodos estão prontas para serem executadas. O sistema proposto não focaliza o aspecto de escalonamento propriamente, mas ao invés, promove a execução das chamadas em função de sua demanda, e da disponibilidade de recursos de processamento ociosos existentes no aglomerado.

No projeto da linguagem MENTAT alguns aspectos não tiveram grande ênfase, tais como a heterogeneidade do aglomerado, permissão de que o programador explicite

a localização física dos processos, e finalmente, o balanceamento de carga como forma de se obter alto desempenho.

O modelo de execução difere totalmente do modelo usado pela linguagem Mentat, que se baseia na construção de um grafo que modela macro dependências de dados entre os objetos. Esse grafo orienta as decisões sobre o escalonamento dos objetos, determinando quais chamadas a métodos estão prontas para serem executadas. O sistema proposto não focaliza o aspecto de escalonamento propriamente, entretanto, promove a execução das chamadas em função de sua demanda, e da disponibilidade de recursos de processamento ociosos existentes no aglomerado.

4. Conclusões

O trabalho apresenta como principal contribuição uma solução para a tradução de programas escritos em uma linguagem baseada em objetos para um modelo de execução distribuída que provê balanceamento de carga. Essa tradução poderá ser utilizada na implementação de um compilador da linguagem para aglomerados de estações. Verificou-se que a tradução de programas nesta linguagem para o modelo de execução do sistema é viável, com pequenas modificações e acréscimos à proposta original da linguagem. O desempenho obtido por um programa teste foi satisfatório.

BIBLIOGRAFIA

[1] DUKE, D.W; ELIAS, D.; LIVNY M.& TURCOTTE, L. **Clustered Workstation Environments.** Tutorials of the Supercomputing 93. ACM SIGARCH & IEEE Computer Society. Nov. 1993.

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

[3] WILSON, G. V. and LU, p. **Parallel Programming Using C++.** London, MIT Press, 1996.

[4] MEYER, Bertrand. **Object Oriented Software Construction.** Englewood-Cliffs, Prentice-Hall, 1988.

[5] SENGER, H. **Um sistema de processamento para programas paralelos em sistemas distribuídos.** Dissertação de mestrado apresentada à Escola Politécnica da USP. 1996.

[6] SALVADOR, L.N., SATO, L.M.;**Aspectos de paralelismo de uma linguagem de programação.** Dissertação de mestrado apresentada à Escola Politécnica da USP. 1995.

[7] STROUSTRUP, B.**The C++ programming language**. Addison-Wesley, 1986.

[8] BEN-ARI, M. **Principles of Concurrent and Distributed Programming.** Prentice-Hall. Cambridge, 1990.