Um núcleo de sistema distribuído para a simulação a eventos discretos

Carlos Maziero

Laboratório de Controle e Microinformática
Departamento de Engenharia Elétrica
Universidade Federal de Santa Catarina
88040-900 Florianópolis, Santa Catarina

E-mail: maziero@lcmi.ufsc.br

Resumo

O advento de máquinas paralelas com memória distribuída torna possível a concepção e implantação de ambientes dedicados ao suporte de classes de aplicações distribuídas distintas. Este artigo apresenta um núcleo de sistema distribuído cujo objetivo é fornecer um ambiente de tempo virtual para uma classe de aplicações cujo paradigma é a simulação a eventos discretos. Após apresentar as características fundamentais do conceito de tempo virtual, são estudadas a implantação e a utilização desse núcleo. Medidas de desempenho são igualmente fornecidas.

Abstract

The advent of distributed memory parallel machines turns feasible the design and implementation of dedicated environments for distributed applications. This paper presents a distributed kernel whose aim to provide a virtual time environment for application programs (distributed discrete-event simulation is the most known of the applications based on a such virtual time, and consequently constitutes the paradigm of this class). This paper presents first basic features of the virtual time concept and then describes time-related aspects of the implementation of the distributed kernel. Some measure results are also presented.

1 Introdução

Podemos dividir as aplicações informáticas em duas classes : aquelas nas quais o tempo executa um papel relevante e as demais, nas quais o tempo é somente um recurso servindo à execução. As aplicações de tempo-real constituem um exemplo da primeira classe : nestas o tempo físico é um objeto de programação utilizado pela aplicação e que coordena sua execução [1]. As características específicas de cada uma destas classes de aplicações devem ser levadas em conta pelos sistemas operacionais que lhes servem de suporte.

Nas aplicações tempo-real, o tempo físico definido pelo ambiente externo constitui o ponto de referência comum ao conjunto de processos, que os permite de sincronizar de maneira a realizar as funções desejadas. Essa utilização do tempo, estendida ao tempo

lógico, dá origem à noção de *tempo virtual* [9, 12]. Nesse contexto existe um relógio virtual global ao conjunto de processos, cuja evolução é cadenciada por regras próprias à aplicação e não mais por um fenômeno físico exterior. Esse relógio permite organizar o cálculo, datar eventos, etc. A simulação a eventos discretos é um exemplo característico de aplicação que utiliza a noção de tempo virtual [2, 7, 9].

Neste artigo nos interessamos à definição de um núcleo de sistema operacional, implementado em uma máquina paralela com memória distribuída, que oferece primitivas para a execução de aplicações distribuídas cuja evolução é baseada em um tempo virtual. O artigo divide-se em 6 seções : a seção 2 caracteriza o ambiente de tempo virtual oferecido pelo núcleo ; a seção 3 apresenta em detalhes a construção do núcleo ; a seção 4 apresenta exemplos de uso e finalmente a seção 5 apresenta medidas de desempenho do protótipo.

2 Um ambiente de tempo virtual

As aplicações consideradas por este núcleo são constituídas de um conjunto de processos conectados por canais de comunicação unidirecionais *fifo* (*first-in, first-out*), com uma topologia qualquer (figura 1). Cada processo possui um certo número de canais de entrada, sobre os quais ele recebe mensagens, e de saída, sobre os quais ele pode emitir mensagens. As mensagens constituem a única forma de interação entre os processos.

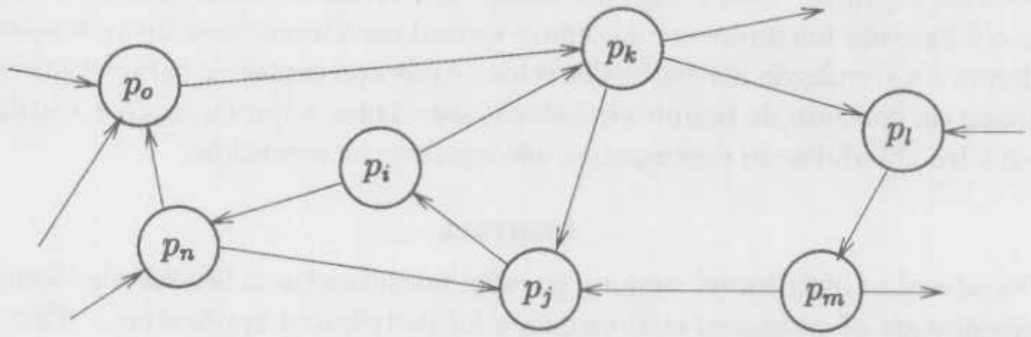

Figura 1: estrutura de uma aplicação

2.1 Propriedades do tempo virtual

Os processos da aplicação evoluem em um tempo virtual do qual todos têm a mesma percepção, representada por um relógio virtual global t_v. O sincronismo lógico dos processos, que permite ordenar todos os eventos produzidos no sistema, pode ser sintetizado pela seguinte propriedade :

$\mathcal{P}_1$: **Unicidade do tempo virtual** : o tempo virtual é único e evolui à mesma velocidade (lógica) para todos os processos.

Enquanto os processos executam ações que têm uma duração positiva ou nula no tempo virtual, suas interações são caracterizadas pela seguinte propriedade :

$\mathcal{P}_2$: **Sincronismo das comunicações** : no tempo virtual toda mensagem enviada no instante t é recebida por seu destinatário no mesmo instante t.[1]

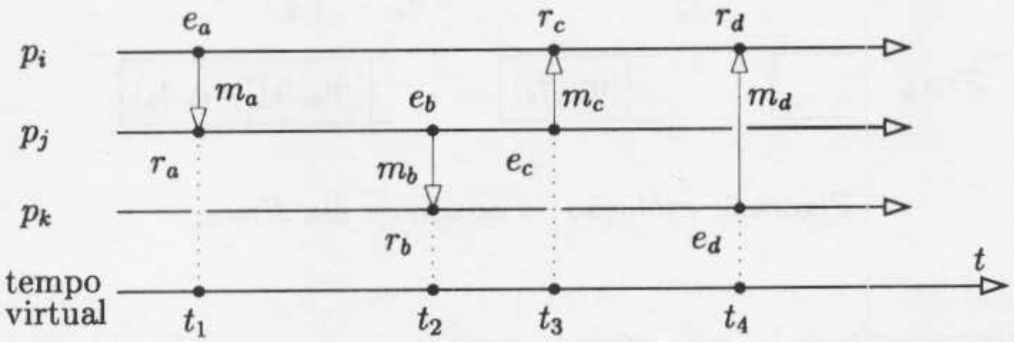

Figura 2: comunicações síncronas no tempo virtual

Neste modelo as comunicações são portanto síncronas no sentido expresso pela propriedade $\mathcal{P}_2$, como mostra a figura 2 (para cada mensagem m_x, e_x indica sua emissão e r_x sua recepção). Entretanto, para não restringir o contexto das aplicações suportadas pelo núcleo, consideramos que os processos são assíncronos em relação ao *consumo* das mensagens : se uma mensagem é recebida no instante t, seu destinatário pode consumí-la em $t' > t$. Este assincronismo define a propriedade $\mathcal{P}_3$:

$\mathcal{P}_3$: **assincronismo dos processos** : quando uma mensagem chega a um processo, este pode consumí-la imediatamente ou em uma data posterior ; a data de consumo é definida unicamente pelo comportamento do processo.

Essa propriedade pode ser facilmente implementada : as mensagens recebidas por um processo p_i são estocadas na ordem de chegada e com suas datas de emissão em uma fila chamada $\mathcal{F}rec$ (figura 3).

2.2 A interface do núcleo

O núcleo é visto pelos processos como uma máquina oferecendo o tempo virtual definido pelas propriedades $\mathcal{P}_1$, $\mathcal{P}_2$ e $\mathcal{P}_3$. O tempo virtual é utilizado pelos processos da aplicação

[1] No lugar da propriedade $\mathcal{P}_2$, a propriedade $\mathcal{P}_2'$ poderia ser enunciada :

$\mathcal{P}_2'$: quando uma mensagem m é enviada no instante t, seu emissor estipula a duração δ_m de sua transferência ; a mensagem será recebida por seu destinatário no instante $t + \delta_m$.

As propriedades $\mathcal{P}_2$ e $\mathcal{P}_2'$ têm poder de expressão equivalente. Tomando $\delta_m = 0\ \forall m$, a propriedade $\mathcal{P}_2'$ se reduz a $\mathcal{P}_2$. Considerando $\mathcal{P}_2$, sempre é possível introduzir um processo entre o emissor e o receptor de uma mensagem, que irá conservá-la durante δ_m unidades de tempo, o que é equivalente a $\mathcal{P}_2'$ entre o emissor e o receptor da mensagem.

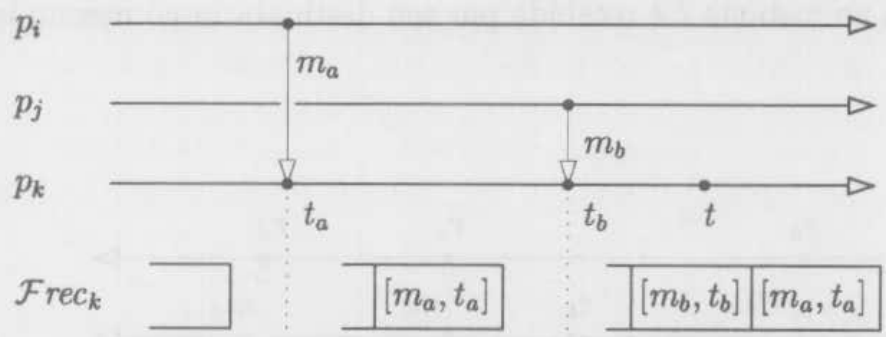

Figura 3: evolução do estado da fila $\mathcal{F}rec_k$

através dos serviços oferecidos pelo núcleo ; esses serviços, materializados por primitivas que podem ser chamadas pelos processos, são os seguintes :

- Leitura do relógio virtual t_v, que indica o instante presente do processo no tempo virtual, através da primitiva `Tempo`.
- Esperar o relógio global t_v atingir uma data t, através da primitiva `EsperaTempo(t)`.
- Esperar a chegada de uma nova mensagem em $\mathcal{F}rec_k$, com data-limite t, através da primitiva `EsperaMsg(t)`. Trata-se do análogo da primitiva de espera de evento com data-limite geralmente utilizada em sistema de tempo real.

Além dos serviços diretamente ligados ao tempo virtual, outros serviços são oferecidos aos processos. Para acessar as mensagens presentes na fila $\mathcal{F}rec_k$ (ou seja, recebidas e não consumidas), um processo pode usar as seguintes primitivas :

- `Vazia` : verifica se a fila $\mathcal{F}rec_k$ está vazia.
- `Primeira` : devolve uma referência sobre a mensagem mais antiga presente em $\mathcal{F}rec_k$, ou `nil` se a fila está vazia.
- `Última` : idem, para a mensagem mais recente.
- `Próxima(m)` : idem, para a mensagem que sucede `m` na fila.
- `Anterior(m)` : idem, para a mensagem que precede `m` na fila.
- `Pegar(m)` : retira a mensagem `m` da fila.

Outras primitivas são oferecidas pelo núcleo, dentre as quais podemos ressaltar :

- A primitiva `NovaMsg(tipo)` permite criar uma nova mensagem do tipo indicado.
- `Enviar(m,cs)` permite enviar uma mensagem `m` sobre um canal de saída `cs`.

Todas as ações efetuadas por um processo têm duração nula no tempo virtual, com exceção das primitivas `EsperaTempo(t)` e `EsperaMsg(t)`. Essas duas primitivas, por suas definições, "consomem tempo virtual".

3 A construção do núcleo

O núcleo de sistema apresentado foi construído para operar sobre uma máquina paralela MIMD com memória distribuída. Nesse tipo de máquina cada processador dispõe de um espaço de memória exclusivo e de canais de comunicação servindo à troca de mensagens com os demais. A troca de mensagens, com atrasos arbitrários mas finitos, constitui a única forma de comunicação entre os processadores. Pode-se considerar que os canais de comunicação são *fifo* e que todos os processadores estão diretamente conectados (isto pode ser assegurado com a ajuda de protocolos adequados). A realização de um núcleo de sistema para oferecer a noção de tempo virtual sobre esse tipo de máquina impõe a resolução de três problemas :

- É necessário efetuar a distribuição da aplicação sobre uma máquina com um número fixo de processadores. Como o número de processos da aplicação pode ser superior ao número de processadores da máquina, vários processos devem ser colocados sobre um mesmo processador. Cabe ao núcleo "mascarar" essa distribuição, tornando transparentes a troca de mensagens e o compartilhamento de processadores.

- De acordo com sua definição, a máquina virtual oferece a cada processo um serviço de troca de mensagens instantâneo no tempo virtual (propriedade $\mathcal{P}_2$), que garante que uma mensagem emitida por um processo no instante t será recebida por seu destinatário em t. Para tal, o núcleo deve estampilhar as mensagens enviadas pelos processos com suas datas de emissão, e controlar sua entrega aos destinatários, usando métodos de sincronização adequados.

- Finalmente, é necessário controlar a evolução dos processos de modo a respeitar as propriedades $\mathcal{P}_1$ e $\mathcal{P}_3$. A propriedade $\mathcal{P}_1$ assegura à aplicação uma visão coerente do tempo virtual, sob a forma de um relógio virtual comum ao conjunto de processos. Como as interações entre processos se limitam à troca de mensagens, é suficiente assegurar que, ao receber uma mensagem, um processo não possa detectar que seu emissor se encontra em um outro instante no tempo virtual. A propriedade $\mathcal{P}_3$ permite a cada processo o acesso às mensagens que lhe foram enviadas (que pertencem ao seu passado). Essa propriedade é assegurada pela fila $\mathcal{F}rec$, cujo acesso é controlado pelo núcleo. Ele deve permitir o acesso somente às mensagens pertencentes ao passado do processo (cujas estampilhas são inferiores ao relógio global t_v), impedindo que alguma mensagem do futuro do processo possa ser vista.

Estes problemas são relativamente independentes e podem ser resolvidos separadamente. Além disso, eles podem ser estruturados de modo hierárquico : para controlar a evolução dos processos, é necessário assegurar as comunicações instantâneas no tempo virtual ; para implementar esse serviço de comunicação é necessário resolver os problemas ligados à distribuição. Estas constatações nos levaram a estruturar o núcleo em três camadas (figura 4), que correspondem a cada um dos problemas acima enunciados : o *controle da distribuição*, que gerencia a distribuição de mensagens e oferece às camadas superiores um serviço de troca de mensagens transparente ; o *controle das comunicações no tempo virtual*, que contrói um serviço de troca de mensagens sincronizado em relação ao tempo virtual, e o *controle da evolução dos processos*, que implementa a interface do núcleo e coordena a evolução dos processos no tempo virtual. A concepção modular da estrutura

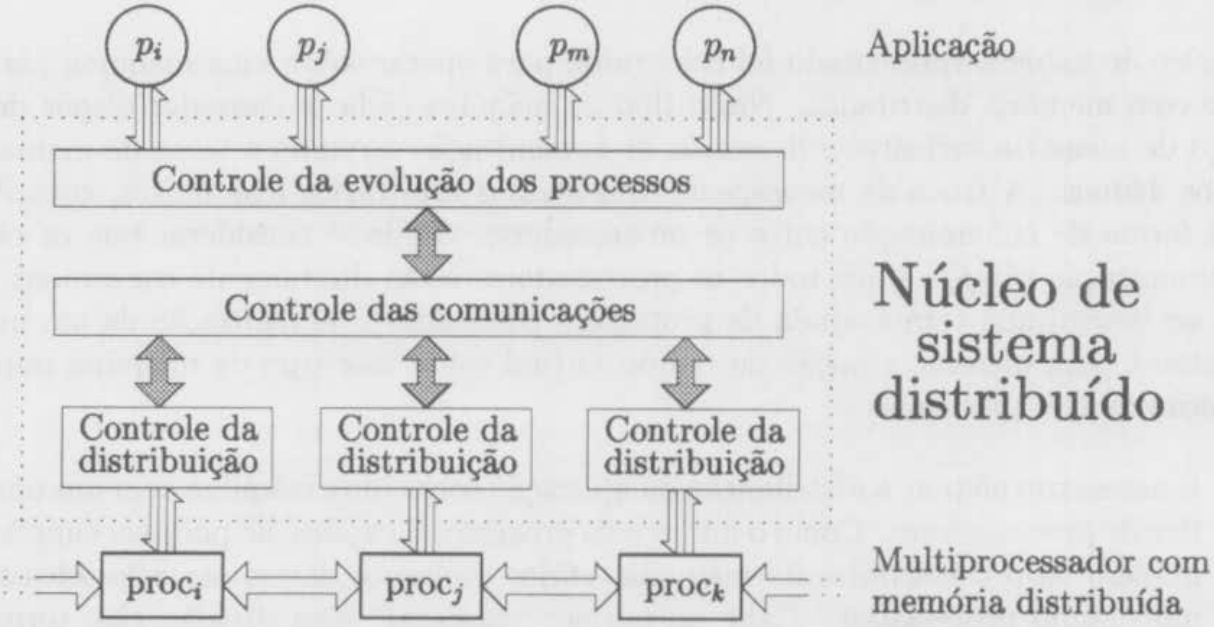

Figura 4: Estrutura do núcleo de sistema distribuído

do núcleo facilita testes e modificações, e permite oferecer aos processos da aplicação uma interface homogênea e independente das técnicas de sincronização utilizadas para a implementação do núcleo, e também da topologia da máquina utilizada.

3.1 Controle da distribuição

Na execução distribuída de uma aplicação, cada processador da máquina se encarrega de executar um grupo de processos. A aplicação não deve entretanto se ocupar dos problemas ligados a essa distribuição, em particular o encaminhamento das mensagens. Quando da definição da rede de processos, esta camada do núcleo memoriza o grafo de interconexões dos processos e suas localizações, para oferecer às camadas superiores do núcleo (e em conseqüência à aplicação) a visão de uma máquina abstrata única sobre a qual se encontram todos os processos.

3.2 Controle da comunicação no tempo virtual

A função básica desta camada é garantir que toda mensagem enviada em uma data t no tempo virtual será recebida por seu destinatário na mesma data (propriedade $\mathcal{P}_2$). Para tal, cada mensagem é estampilhada com sua data de emissão, e as mensagens recebidas por um processo lhe são entregues na ordem dessas estampilhas. Essa ordem permite garantir que a evolução do tempo virtual sobre cada processo será coerente (um processo não pode "voltar no tempo virtual" recebendo uma mensagem enviada a t seguida de outra enviada a $t' < t$) [7, 9, 11].

Como as comunicações reais são assíncronas, são necessários mecanismos de controle adequados para garantir a entrega das mensagens na ordem das estampilhas. Consideremos a situação da figura 5, na qual o processo p_i recebe três mensagens tais que $t_1 < t_2 < t_3$.

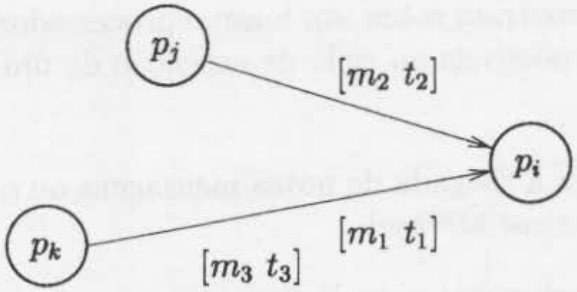

Figura 5: Entrega de mensagens na ordem das estampilhas

As mensagens $[m_1\ t_1]$ e $[m_2\ t_2]$ podem ser entregues a p_i, o que não ocorre com $[m_3\ t_3]$, pois o processo p_j pode enviar a p_i uma mensagem $[m'_2\ t'_2]$ tal que $t'_2 < t_3$, e neste caso m'_2 deverá ser entregue antes de m_3. Duas classes de técnicas foram propostas para resolver este problema : as técnicas *otimistas* [4] e as técnicas *pessimistas* [2]. Nosso núcleo utiliza a técnica pessimista de prevenção de interblocagem [11] : quando um processo p_j sabe que não irá enviar nenhuma mensagem sobre um canal antes de uma certa data futura t $(t > t_v)$, ele emite sobre esse canal uma mensagem especial $[null\ t]$, para informar seu sucessor. Essa previsão sobre o futuro de um canal de saída exerce uma grande influência sobre o desempenho desta técnica [6, 7, 8]. O núcleo oferece uma primitiva suplementar `Previsão(tprev)` que permite a um processo informar ao núcleo sua previsão corrente sobre o futuro de seus canais de saída.

Para realizar a entrega da mensagens aos processos na ordem correta, a camada de controle das comunicações no tempo virtual gerencia vários "relógios" : para cada canal c_{ij} define-se um *relógio do canal* tc_{ij}, que contém a estampilha da última mensagem (real ou *null*) que por ele circulou. Estes relógios permitem definir um relógio de entrada de um processo p_i, te_i, que é o mínimo dos relógios dos canais de entrada desse processo : $te_i = min_k(tc_{ki})$. Desta forma é possível entregar ao processo p_i toda mensagem $[m\ t]$ tal que $t \leq te_i$, pois esse relógio representa a data até a qual o conteúdo dos canais de entrada do processo p_i é inteiramente conhecido e pode lhe ser entregue (colocado à sua disposição em $\mathcal{F}rec_i$). Está demonstrado em [2, 11] que, se cada processo evolui no tempo virtual, esse mecanismo garante a evolução coerente do conjunto de processos.

3.3 Controle da evolução dos processos

Como o número de processos da aplicação é a priori maior que o número de processadores, vários processos devem executar sobre o mesmo processador. Uma possível solução a este problema consiste em considerar que existe sobre cada processador um simulador seqüencial clássico que coordenaria a evolução dos processos que se encontram nesse processador. Neste caso existiria somente um relógio virtual por processador e os algorítmos de sincronização vistos na seção 3.2 se aplicariam somente entre processadores. Outra solução consiste em manter um relógio por processo e então aplicar os algorítmos de sincronização entre todos os processos da aplicação. Esta última solução permite um melhor aproveitamento do paralelismo potencial do modelo : tendo em vista o grafo de processos e as interações possíveis entre processos sobre processadores distintos, é possível que mensagens com estampilhas superiores à menor estampilha sobre um processador possam

ser tratadas por seus destinatários.

Os processos que se encontram sobre um mesmo processador disputam o mesmo para sua execução. Os estados possíveis no ciclo de execução de um processo p_i da aplicação são os seguintes :

- *Bloqueado* : ele espera a chegada de novas mensagens ou o avanço de seu relógio de entrada te_i para tornar-se ativável.
- *Ativável* : ele pode se executar, mas deve esperar que o processador esteja disponível.
- *Ativo* : é o processo que se executa. Ele pode ser bloqueado pela espera de novas mensagens, através da chamada à primitiva `fim` ou quando $t_v > t_{max}$).
- *Terminado* : o processo encerrou sua execução ; sua previsão de próximo envio de mensagem é fixada a $+\infty$, para informar a seus sucessores que ele não irá mais enviar mensagens.

3.3.1 Coordenação dos relógios dos processos

Cada processo da aplicação é dotado de uma representação local tp_i do relógio virtual global t_v, que ele pode consultar e que avança à medida em que ele se executa. Como uma sincronização forte desses relógio locais ($\forall\ i, j\ tp_i = tp_j$) seria prejudicial ao desempenho, decidimos permitir sua dessincronização ($tp_i \neq tp_j$). Torna-se então necessário assegurar que, durantes as comunicações, os processos não possam constatar divergências entre seus relógios locais, garantindo assim a propriedade $\mathcal{P}_1$. O serviço de comunicação no tempo virtual (seção 3.2) garante que não haverá recepção de novas mensagens pertencendo ao passado do processo (*i.e.* toda nova mensagem recebida terá $t_m \geq tp_i$). Por outro lado, o serviço de acesso à fila de mensagens $\mathcal{F}rec_i$ (*i.e.* as primitivas descritas na seção 2.2) deve impedir acesso do processo às mensagens de seu futuro ($t_m > tp_i$) que estiverem presentes em seus canais de entrada (mensagens enviadas por um processo mais adiantado no tempo virtual).

É possível que um processo não tenha necessidade de conhecer todas as mensagens que lhe foram enviadas até o instante presente para apresentar um comportamento correto. Por exemplo, um processo simulando um servidor com comportamento *fifo* e com mensagens (clientes) ainda não tratadas em seus canais de entrada, pode tratá-los sem precisar tomar conhecimento da chegada de novos clientes. O bloqueio sistemático desse processo (para esperar que $te_i \geq tp_i$) introduz uma sincronização pesada e inútil. Desta forma, o núcleo autoriza $tp_i > te_i$ (ou seja, possivelmente existem mensagens emitidas a esse processo com $t_m < tp_i$ mas que ainda não chegaram a seus canais de entrada), mas somente quando as propriedades $\mathcal{P}_1$ e $\mathcal{P}_2$ forem preservadas. O processo é imediatamente bloqueado se ele tentar acessar essas mensagens.

A próxima seção mostra como a implementação das primitivas leva em conta essa possibilidade de "afrouxamento" da sincronização no tempo virtual.

3.3.2 Implementação das primitivas do núcleo

As primitivas apresentadas na seção 2.2 permitem aos processos da aplicação acessar a máquina virtual construída pelo núcleo. Algumas dessas primitivas são fortemente dependentes dos critérios utilizados para a sincronização dos processos, como é o caso

das primitivas de espera no tempo virtual (`EsperaTempo(t)`) e de espera de mensagem (`EsperaMsg(t)`), e também das primitivas de acesso à fila de mensagens recebidas $\mathcal{F}rec_i$ (`Primeira`, `Última`, etc.).

Examinemos a primitiva de espera : em princípio, um processo p_i que efetua `EsperaTempo(t)` deveria esperar para que seu relógio de entrada evolua até a data t ($te_i \geq t$) antes de continuar sua execução (com o relógio local atualizado : $tp_i = t$). Entretanto, como vimos na seção anterior, não é sempre necessário a um processo conhecer completamente o conteúdo de seus canais de entrada para continuar sua execução. Portanto, a implementação da primitiva `EsperaTempo(t)` será efetuada pela simples atualização do relógio local do processo, sem bloqueá-lo.

Por sua definição, a fila $\mathcal{F}rec_i$ contém, no instante virtual t, todas es mensagens enviadas ao processo p_i e ainda não consumidas. Com a implementação proposta para a primitiva `EsperaTempo(t)`, essa propriedade da fila pode ser violada. Entretanto ela pode ser reestabelecida sem que essa inconsistência temporária seja percebida pelo processo. As primitivas de acesso à fila deverão levar em conta a possibilidade de inconsistência na fila $\mathcal{F}rec_i$, como mostra a análise das duas situações possíveis para os relógios local e de entrada do processo p_i :

- $tp_i < te_i$: todas as mensagens $[m\ t_m]$ pertencentes ao passado do processo ($t_m \leq tp_i$) ainda não consumidas são realmente disponíveis na fila $\mathcal{F}rec_i$; o processo pode acessá-las livremente. Esta situação corresponde à apresentada na figura 6.

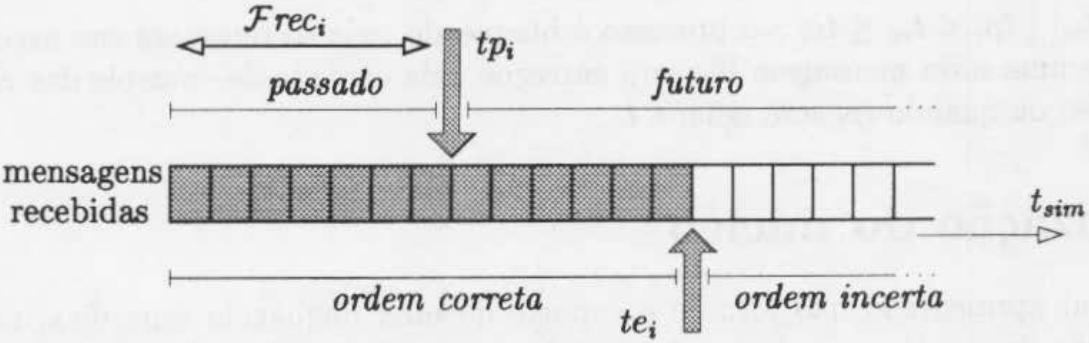

Figura 6: Espera no tempo virtual, com $tp_i < te_i$

- $tl_i \geq te_i$: uma parte das mensagens relativas ao passado de p_i (as mensagens $[m\ t_m]$ tais que $te_i < t_m \leq tp_i$) não é realmente disponível, pois a ordem da fila não é inteiramente determinada (a parte da fila cuja ordem é indeterminada está representada pela zona branca de $\mathcal{F}rec_i$ na figura 7). Somente as mensagens $[m\ t_m]$ tais que $t_m \leq te_i$ são realmente disponíveis em $\mathcal{F}rec_i$. Desta forma o processo p_i pode continuar sua execução, mas corre o risco de ser bloqueado em chamadas às primitivas de acesso à fila $\mathcal{F}rec_i$, se tentar acessar a parte ainda indeterminada dessa fila. Sua execução só será retomada após o avanço do relógio de entrada te_i, quando a mensagem em questão puder ser efetivamente acessada ($te_i \geq t_m$).

Resta examinar a implementação da primitiva de espera de mensagem. Uma chamada à primitiva `EsperaMsg(t)` entrega ao processo uma nova mensagem, além das que ele já

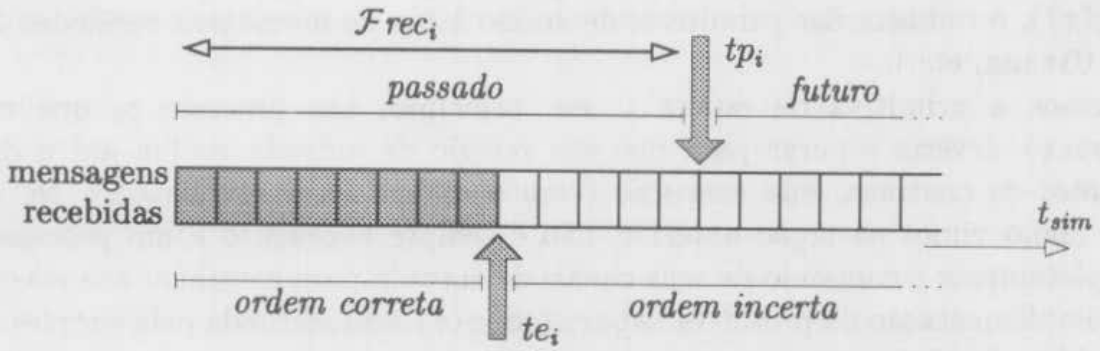

Figura 7: Espera no tempo virtual, com $tp_i > te_i$

conhecia (que possuíam $t_m \leq tp_i$), ou nada, se nenhuma mensagem chegar antes da data limite de espera t. O relógio local do processo deve então avançar até a data de chegada da nova mensagem, ou até t, caso contrário. Assim, a execução de uma espera de mensagem por um processo p_i implica sempre no avanço de seu relógio local tp_i. Ela pode também impor ou não um bloqueio do processo, caso exista ou não uma mensagem m disponível sobre um canal de entrada, dotado de uma estampilha t_m inferior à data limite t :

- $\exists[m\ t_m] \mid tp_i < t_m \leq te_i$: neste caso a mensagem mais antiga $[m, t_m]$ é integrada à fila $\mathcal{F}rec_i$, tp_i avança até t_m e o processo retoma sua execução.
- $\neg\exists[m\ t_m] \mid tp_i < t_m \leq te_i$: o processo é bloqueado ; ele só retomará sua execução quando uma nova mensagem lhe será entregue pela camada de controle das comunicações, ou quando tp_i será igual a t.

4 Utilização do núcleo

O núcleo aqui apresentado não fornece o suporte de uma linguagem específica, nem a implementação de um tipo particular de modelo de simulação. Seu objetivo é fornecer um conjunto de primitivas suficientemente gerais para suportar a realização de ambientes correspondentes a linguagens ou modelos de simulação diversos. Buscando verificar se esse objetivo foi atingido, estudamos a implementação de um certo número de modelos clássicos ; esse estudo nos leva a crer que as primitivas propostas permitem uma implementação simples e razoavelmente eficaz de modelos diversos.

Consideremos, para ilustrar a utilização do núcleo, a implementação de uma instrução hipotética oferecida por uma linguagem ao usuário : a instrução *pegar_lifo(m)*, que entrega ao processo p_i que a chama a mensagem mais recente. Essa instrução pode ser construída pela seguinte seqüência de chamadas às primitivas do núcleo :

```
pegar_lifo(m) :
   se Vazio então EsperaMsg(∞) fimse ;     /* se Frec_i vazia, esperar nova mensagem */
   m ← Pegar (Última) ;                    /* pegar a última msg que chegou */
```

Examinemos um exemplo mais elaborado de uso das primitivas do núcleo : a definição de um servidor de redes de fila de espera com política *quantum* (as mensagens em circulação correspondem aos clientes). Em um servidor *quantum* clássico os clientes são atendidos sucessivamente em fatias de tempo de duração fixa (chamadas *quantum*), e a ordem de atendimento é *fifo*. Uma solução direta, embora ineficaz, para a implementação desse servidor seria escrever todo o seu código no interior de um processo. Podemos adotar no entanto outra solução, mais interessante : quando o servidor termina o *quantum* de um cliente, este pode ser re-enviado à entrada do servidor por meio de um canal auxiliar (canal "retorno", na figura 8), caso ele ainda requeira tempo de tratamento. Nesse caso ele será ordenado com os novos clientes na entrada do servidor. Assim que terminar seu tratamento, o cliente abandona o servidor pelo canal *saída*.

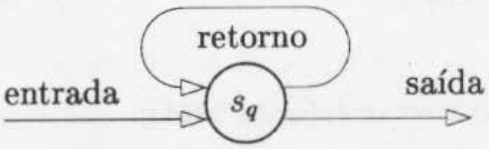

Figura 8: Servidor *quantum*

O código descrito abaixo representa o comportamento do servidor. A variável *no_retorno*, que conta o número de clientes sobre o canal *retorno*, é utilizada para o mecanismo de cálculo da previsão a fornecer ao núcleo. Como o servidor só se bloqueia na ausência de clientes (especialmente quando o canal *retorno* está vazio), não há interesse em calcular previsões enquanto houverem clientes sobre esse canal. Quando o canal *retorno* estiver vazio, a previsão é dada pelo mínimo entre o próximo tempo de serviço t_{prox} e a duração do *quantum* t_{quant}. O atributo t_{rest} de cada cliente indica o tempo de tratamento que lhe resta a efetuar. A primitiva `MsgCanal(`m`)` indica o canal de onde provém a mensagem.

```
Servidor quantum :
  t_prox ← F_aleat ;                          /* tempo de serviço para o primeiro cliente */
  no_retorno ← 0 ;
  repetir
    se no_retorno = 0 então Previsão (min (t_prox, t_quant))     /* estabelecer previsão */
    senão Previsão (0.0) fimse ;
    se Vazia então EsperaMsg(∞) fimse ;
    cliente ← Pegar (Primeira) ;                  /* pegar próximo cliente a tratar */
    se MsgCanal (cliente) = retorno então no_retorno ← no_retorno −1
    senão
      cliente.t_rest ← t_prox ;                   /* tempo de serviço do novo cliente */
      t_prox ← F_aleat ;
    fimse ;
    t_serv ← min (cliente.t_rest, t_quant) ;          /* tratar e expedir o cliente */
    EsperaTempo (t_serv) ;
    cliente.t_rest ← cliente.t_rest - t_serv ;
    se cliente.t_rest = 0.0 então Enviar (cliente, saída)
    senão
      Enviar (cliente, retorno) ;
      no_retorno ← no_retorno +1 ;
    fimse ;
  até Tempo ≥ t_max ;                                 /* fim da simulação */
```

5 Estudo do desempenho do núcleo

Ao avaliar o desempenho do núcleo de sistema construído, nos interessamos basicamente em estudar seu uso na execução de simulações distribuídas. Sob um ponto de vista quantitativo, é importante avaliar se as técnicas empregadas permitem acelerar simulações a eventos discretos através de sua distribuição. O impacto das decisões tomadas sobre a sincronização dos processos também deve ser avaliado. Finalmente, é interessante analisar a influência das características do modelo sobre o desempenho do núcleo.

A implementação do núcleo e as experiências foram efetuadas em um INTEL iPSC/2, uma máquina paralela com 64 processadores x386, sem memória comum e com uma rede de interconexão hipercúbica. Dois modelos foram profundamente estudados : um toróide de 16 × 16 processos, no qual cada processo possui dois canais de entrada e dois canais de saída, e um modelo com topologia variável chamado *rev* (figura 9), constituido por uma matriz 16 × 16 de processos, na qual é possível fazer variar o número de canais de entrada (e portanto de saída) de cada processos (em uma rede com k entradas, chamada rev_k, o processos $p_{i,j}$ é conectado aos processos $p_{i+1,j+1}$, $p_{i+2,j+1}$, ..., $p_{i+k,j+1}$, todos os índices sendo calculados módulo 16). Estas redes contém um grande número de processos ; cremos ser este efetivamente o tipo de modelo a estudar, visto que as simulações que se deseja paralelizar são normalmente de grandes dimensões e apresentam boa quantidade de paralelismo potencial. Escolhemos redes regulares para simplificar a análise dos resultados das experiências, limitando os efeitos devidos à distribuição dos processos sobre os processadores.

Em uma experiência todos os processos têm um comportamento similar, conforme descrito abaixo (*nbmsg* indica o número de mensagens iniciais por processo, *carga* indica

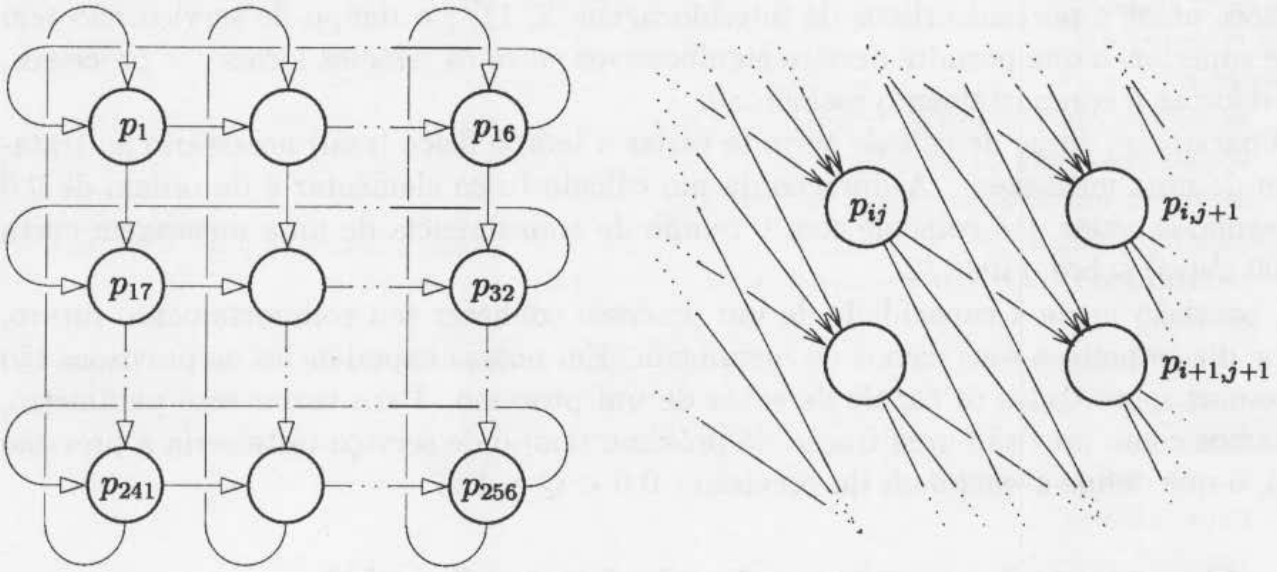

Figura 9: Modelos usados nas experiências

a carga de cálculo real necessária ao tratamento de uma mensagem, e t_{max} a data de fim de simulação).

```
Processos do modelo :
  para i ← 1 até nbmsg faça                    /* enviar as nbmsg mensagens iniciais */
    Msg ← NovaMsg (tipo_msg) ;
    Enviar (Msg, Saída) ;
  fimpara ;
  repetir
    t_serv ← 1.0 + exp (9.0) ;                 /* próximo tempo de serviço e previsão */
    Previsão (Q * t_serv) ;
    se Vazio então EsperaMsg(∞) fimse ;        /* pegar a próxima mensagem */
    Msg ← Pegar ( Primeira | Última) ;
    EsperaTempo (t_serv) ;                     /* esperar t_serv unidades de tempo virtual */
    para i ← 1 até carga faça cálculo unitário ;   /* carga de cálculo real */
    Enviar (Msg, Saída) ;                      /* enviar a mensagem sobre uma das saídas */
  até Tempo ≥ t_max ;
```

Estudamos duas políticas para a ordem de tratamento das mensagens : *fifo* (a mensagem mais antiga primeiro) e *lifo* (a mensagem mais recente primeiro). Estes dois casos representam as situações extremas para as técnicas de sincronização dos processos (conforme a seção 3.3) : *fifo* é a mais favorecida (um processo só é bloqueado se não houver mensagem a consumir) e *lifo* é a menos favorecida (um processo não pode consumir a mensagem mais recente enquanto $te_i < tp_i$, permanecendo bloqueado).

O tempo de serviço corresponde à duração, no tempo virtual, do tratamento de uma mensagem, e é dado por uma variável aleatória de lei $a + exp\ (b) \mid a, b > 0$[2]. Duas razões

[2]A função $exp(x)$ gera um valor aleatório de distribuição exponencial negativa e média x.

nos levaram a escolher essa lei : o tempo de serviço mínimo é maior que zero, para evitar previsões nulas e portanto riscos de interblocagem [2, 11] ; o tempo de serviço não tem limite superior, o que permite desvios significativos entre os relógios locais dos processos, para reforçar o comportamento assíncrono.

O parâmetro *carga de cálculo* permite variar o tempo físico (real) necessário ao tratamento de uma mensagem. A duração de um cálculo físico elementar é da ordem de 0.6 milisegundos, valor que coincide com o tempo de transferência de uma mensagem curta (< 100 bytes) sobre o iPSC/2.

A previsão mede a capacidade de um processo conhecer seu comportamento futuro, no que diz respeito a seus envios de mensagem. Em nossas experiências as previsões são as mesmas sobre todos os canais de saída de um processo. Para variar esse parâmetro, utilizamos como previsão uma fração do próximo tempo de serviço (este seria a previsão ideal), o que define a *qualidade* da previsão : $0.0 < Q \leq 1.0$.

5.1 Comparação com um simulador seqüencial

Para uma avaliação preliminar do núcleo, comparamos sua execução seqüencial com a de um simulador clássico (o simulador contido no sistema de análise de redes de filas *Qnap2* [13]), para modelos diversos de redes de filas, sobre uma mesma máquina monoprocessador.

Foi possível constatar que, para modelos de dimensões significativas (256 processos), nosso núcleo é consideravelmente mais rápido que *Qnap2*, exceto em condições muito penalizantes para as técnicas de sincronização empregadas (muitos canais de entrada, poucas mensagens em circulação e previsões mínimas). No entanto, modelos de pequenas dimensões (16 processos) são simulados mais rapidamente com *Qnap2*.

Podemos concluir que nosso núcleo é eficiente na simulação de modelos cujo tratamento seria pesado para os simuladores seqüenciais. Isto pode ser atribuído à dificuldade de gerenciamento de um escalonador excessivamente longo em um simulador convencional. Em nosso núcleo o conceito de escalonador global deixa de existir, para dar lugar às filas de entrada de cada processo, bem mais curtas e de fácil manipulação.

5.2 Aceleração obtida pela distribuição

A aceleração foi calculada como sendo, para um modelo dado, a relação entre a duração da execução de uma simulação sobre um processador e a duração da mesma simulação sobre n processadores. A figura 10 mostra os resultados obtidos com o toróide 16×16, com processos *fifo* e carga de cálculo variando de 0 a 16. Em todos os casos o tempo de execução diminui quando o número de processadores aumenta, e a aceleração é mais forte quando a carga de cálculo aumenta.

Evidentemente, quanto maior o número de processadores utilizados para realizar um cálculo distribuído, menor será a eficiência de uso de cada um deles, por causa dos custos associados à comunicação e à sincronização. Mas uma queda no tempo global de cálculo é observada (para os modelos estudados), e este é um dos principais objetivos do uso de máquinas paralelas.

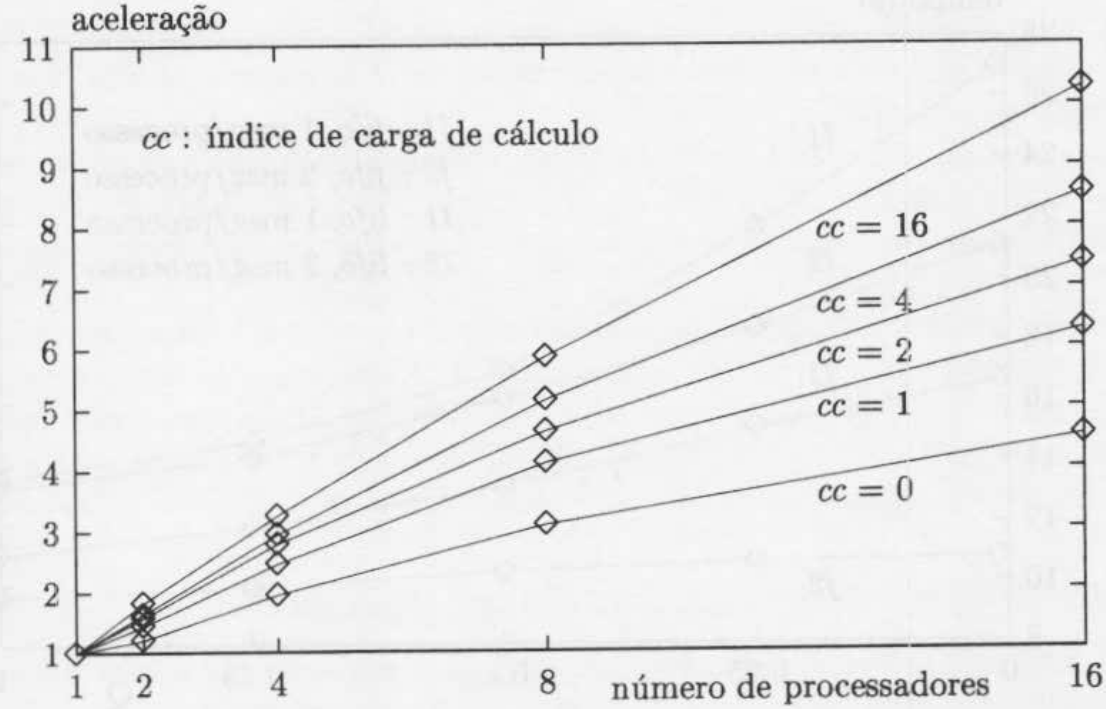

Figura 10: Aceleração = $\mathcal{F}$ (#processadores) para o toróide com processos *fifo*

5.3 Influência da política de serviço

Para avaliar a influência do comportamento dos processos do modelo sobre o desempenho do núcleo, dois tipos de comportamentos foram estudados : processos *fifo* e processos *lifo*. Lembramos que esses dois comportamento representam as situações extremas para as sincronizações do núcleo. Em cada experiência todos os processos são do mesmo tipo.

Foi possível constatar que a execução de um modelo com processos *fifo* sempre é mais rápida que a mesma execução com processos *lifo* (figura 11). Essa diferença indica claramente que a dessincronização introduzida entre os relógios local tp_i e de entrada te_i de cada processo (ver seção 3.3) é útil, permitindo ao núcleo tirar vantagem do comportamento dos processos.

A diferença entre os tempos de execução *fifo* e *lifo* se acentua quando o número de mensagens em circulação aumenta (figura 11). Esse fenômeno indica a eficiência no controle da execução dos processos, pois o fato de só bloquear um processo se ele tenta efetivamente acessar uma mensagem inexistente provoca uma redução considerável no número de bloqueios para processos *fifo* quando o número de mensagens aumenta. A figura 11 mostra também que boas previsões diminuem a diferença entre os tempos de execução observados.

5.4 Influência da previsão

Um estudo feito por Fujimoto [5] sobre as técnicas de sincronização pessimistas indicava que a qualidade da previsão é um parâmetro importante para o desempenho das simulações paralelas. Esse estudo foi realizado sobre uma máquina a memória compartilhada, e é interessante verificar se seus resultados continuam válidos para máquinas com memória distribuída.

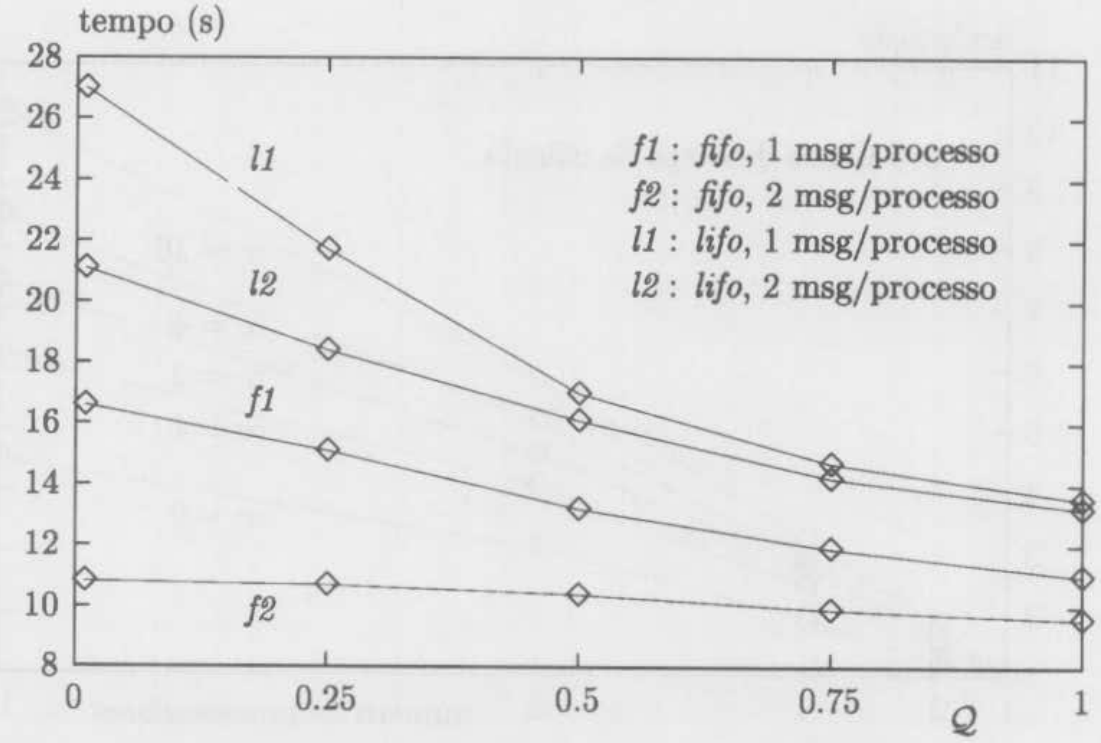

Figura 11: Tempo = $\mathcal{F}$ (qualidade da previsão) para o toróide

A figura 11 indica o tempo de simulação em função da qualidade Q da previsão, para um toróide 16 × 16. Ela permite constatar que a previsão exerce mais influência sobre processos *lifo* que *fifo*. Isto é uma conseqüência da otimização dos bloqueios de processos efetuada no núcleo. Se há muitas mensagens em circulação, os processos *fifo* quase não são bloqueados, e a previsão tem pouca influência. Os resultados obtidos para os processos *lifo* conferem com os obtidos por Fujimoto.

Sobre a rede com topologia variável *rev* foi observado que a qualidade da previsão é um fator importante, mesmo para processos *fifo*, quando o número de canais de entrada é elevado. Entretanto esse efeito tende a se atenuar quando o número de mensagens em circulação aumenta (como mostram as seções 5.5 e 5.6).

5.5 Influência do número de mensagens

A população de mensagens reais permanece constante durante uma simulação ; ela é dada pelo número de mensagens criadas inicialmente por cada processo (em nosso modelo o tratamento de uma mensagem provoca sempre a emissão de outra mensagem). As experiências foram efetuadas com uma duração fixa no tempo virtual, e a mesma lei foi utilizada para os tempos de serviço ; como conseqüência, o número de tratamentos de mensagens efetuados permanece constante.

O aumento do número de mensagens em circulação é penalizante para um simulador seqüencial, devido ao aumento da fila de eventos futuros. Em uma simulação distribuída a dimensão das filas também aumenta, mas esse fenômeno tem menos conseqüências devido à existência de uma fila para cada processo, no lugar de uma longa e única fila global. Além disso, esse efeito é mais influente em processos *lifo* (é necessário percorrer toda a fila para acessar a última mensagem). Entretanto, como o algorítmo de controle das comunicações espera a presença de uma mensagem sobre cada canal de entrada de um processo para

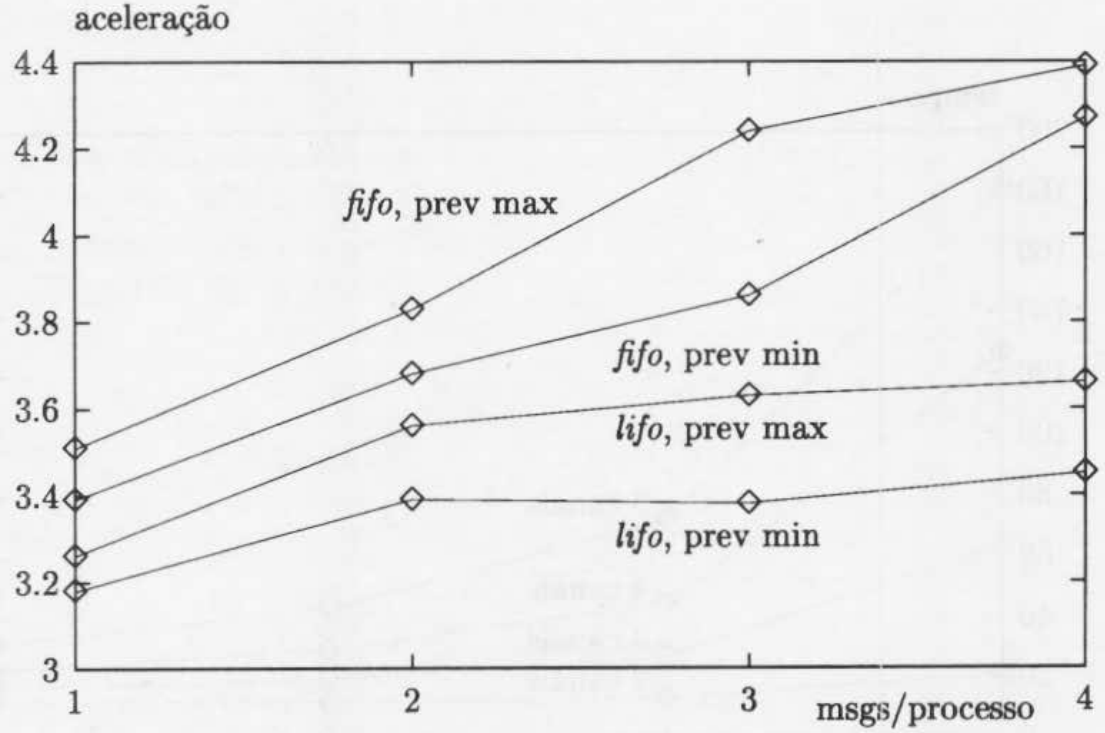

Figura 12: Aceleração = $\mathcal{F}$ (número de mensagens) para o toróide

lhe entregar uma mensagem, o aumento do número de mensagens deveria conduzir a uma melhoria do desempenho da simulação, graças a uma redução das esperas. A figura 12, que apresenta a aceleração para uma simulação do toróide com 8 processadores, confirma parcialmente essa intuição. A melhoria se produz efetivamente para processos *fifo*, mas no entanto não há praticamente ganho para processos *lifo* com mais de duas mensagens.

A figura 13 indica os tempos de simulação sobre o modelo com número variável de entradas, em função do número de mensagens por processo. Pode ser igualmente constatada uma redução do tempo de simulação, mais perceptível quando o número de canais de entrada de cada processo é grande.

5.6 Influência da topologia

Avaliar o impacto da forma da rede de processos sobre o desempenho de um simulador não é uma tarefa trivial, pois é difícil quantificar esse parâmetro. Para evitar efeitos devidos à disposição dos processos sobre os processadores, consideramos somente redes regulares. Desta forma nosso estudo se baseou essencialmente sobre o número de canais de entrada por processo, que nos parece um parâmetro importante. Outras características, como o número e a dimensão dos circuitos presentes na rede, não foram estudadas.

A figura 14 indica o tempo de simulação para redes cujo número de entradas varia de 1 a 15, para processos *fifo* e *lifo*. Constatamos um aumento considerável do tempo de simulação com o número de canais de entrada por processo, sobretudo quando as previsões são de baixa qualidade. Esse fenômeno é relativamente independente do comportamento dos processos (*fifo* ou *lifo*).

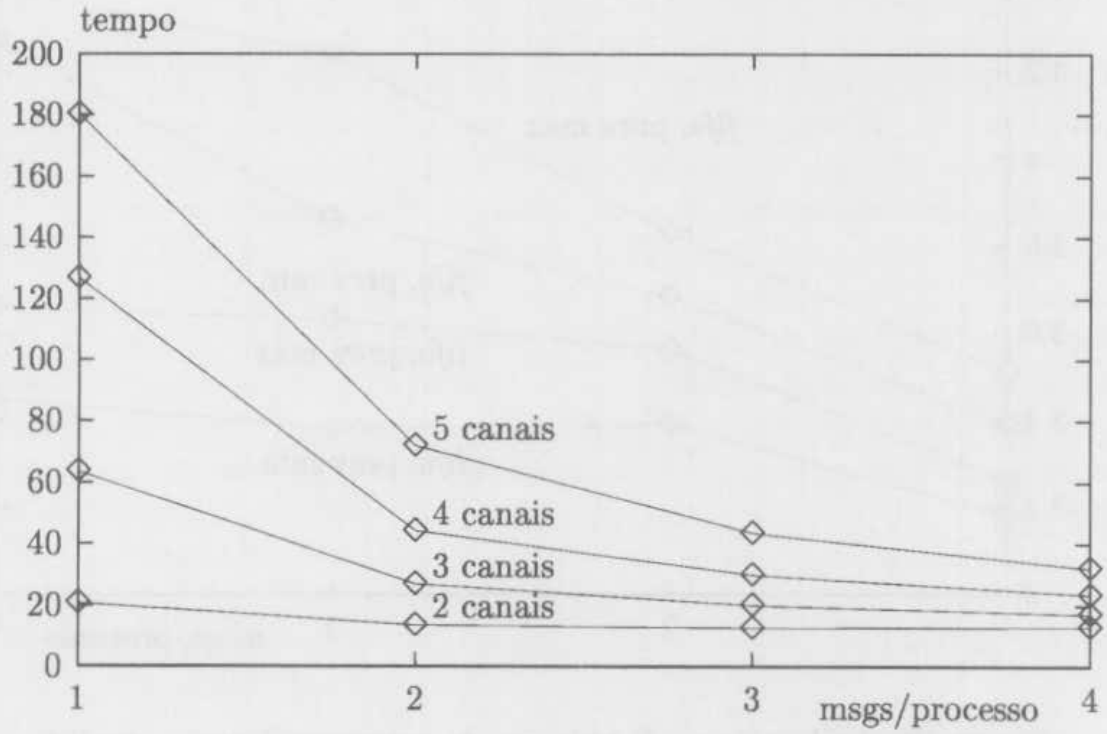

Figura 13: Tempo = $\mathcal{F}$ (número de mensagens) para o *rev*, processos *fifo*

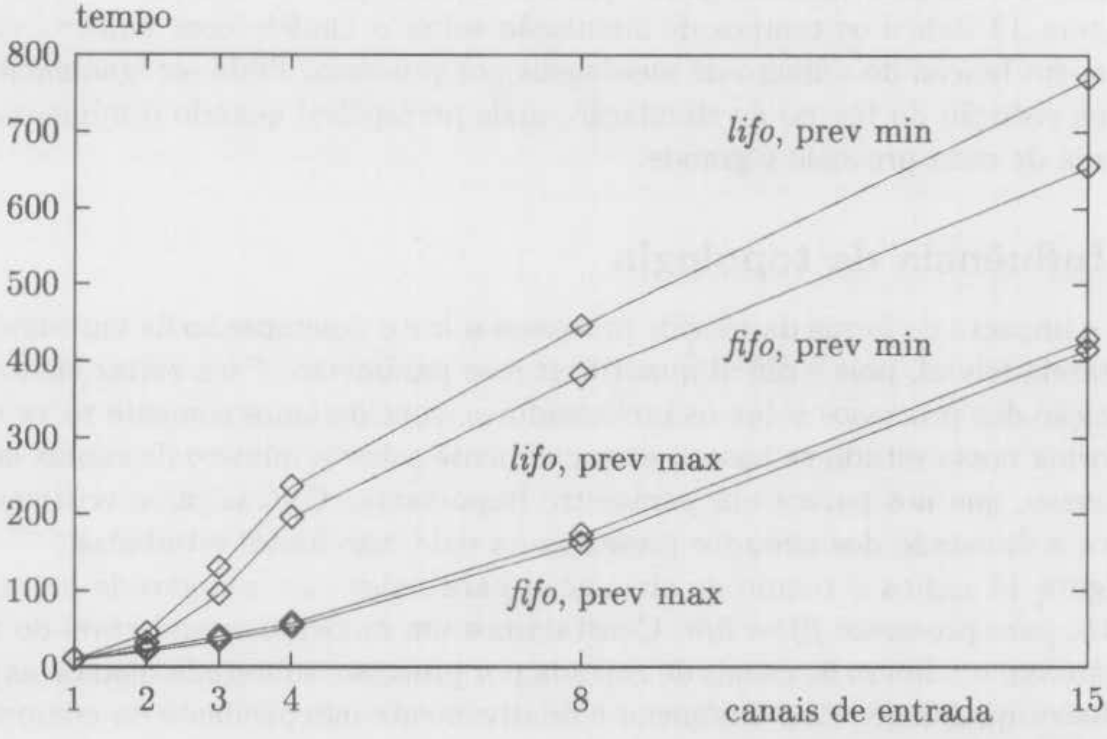

Figura 14: Tempo = $\mathcal{F}$ (número de canais de entrada) para o *rev*

5.7 Análise dos resultados obtidos

As experiências realizadas mostram que é efetivamente possível obter uma redução do tempo de simulação distribuindo uma simulação com um algorítmo de controle de tipo pessimista. Os resultados obtidos indicam que o desempenho das simulações realizadas sobre o núcleo construído é beneficiado pelo aumento na carga de cálculo e no número de mensagens em circulação. Esses dois fenômenos são prejudiciais aos simuladores seqüenciais, mas constituem uma fonte considerável de paralelismo potencial. Portanto as simulações que melhor se prestam à uma execução paralela são justamente aquelas que justificam uma paralelização.

Foi possível verificar a utilidade das dessincronizações introduzidas na gestão dos relógios locais e de entrada dos processos. A diferença considerável entre os tempos de simulação de modelos contendo processos *fifo* e *lifo* demonstra seu interesse. Os resultados obtidos na simulação de modelos com processos *lifo* correspondem aos apresentados por outros autores [6, 7] para processos genéricos. Para processos *fifo* o desempenho observado com nosso núcleo é claramente superior, o que demonstra seu potencial.

Outros resultados interessantes dizem respeito à qualidade das previsões fornecidas pelos processos : elas ganham importância quando as demais características do modelo são desfavoráveis (número de canais de entrada, número de mensagens, etc.). De maneira geral, a qualidade das previsões exerce uma influência maior sobre redes de processos *lifo* que sobre *fifo*. Encontramos até mesmo situações onde a qualidade das previsões não exerce praticamente influência sobre a duração da simulação, como indica a figura 11. Mais uma vez, esses resultados são devidos às otimizações na gestão dos acessos às mensagens que chegam aos processos.

6 Conclusão

Neste artigo nós apresentamos a realização e a avaliação de um núcleo de sistema distribuído destinado à execução de programas de simulação a eventos discretos sobre máquinas paralelas com memória distribuída. A estrutura proposta para o núcleo é composta de três camadas, cada uma resolvendo uma parte precisa do problema de gestão de uma simulação distribuída : a distribuição do modelo, transferência instantânea de mensagens e a evolução dos processos no tempo simulado.

Contrariamente às técnicas convencionais utilizadas para a implementação de simulações paralelas, em nosso núcleo a gestão das comunicações no tempo simulado foi separada do controle de evolução dos processos. Apesar dessa separação, as experiências mostraram um bom desempenho. A introdução de dessincronizações suplementares entre a execução de um processo e o estado de seus canais de entrada permitiu a redução do tempo de execução das simulações e também a atenuação da influência das previsões sobre o desempenho.

Em um contexto mais abrangente, a concepção de um núcleo de sistema distribuído para aplicações pilotadas por um tempo virtual é interessante pelas restrições de sincronização que ela define sobre as comunicações : toda mensagem emitida na data virtual t é recebido nessa mesma data t. Podemos encontrar uma estratégia análoga nos mecanismos de controle de acesso a dados que usam uma ordem criada por estampilhagem [10]. Isto sugere um estudo mais profundo das propriedades ligadas à comunicação oferecidas por núcleos de sistemas distribuídos [3]. Desde ambientes totalmente assíncronos até o

ambiente restrito definido pela propriedade $\mathcal{P}_2$, nos parece importante destacar tipos de comunicação de base e caracterizar corretamente as aplicações que deles necessitam. O núcleo aqui apresentado é um passo concreto nessa direção.

Referências

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), September 1991.

[2] K.M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[3] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. Technical Report 91-55, LITP - Paris 7 - France, September 1991.

[4] D. Jefferson et al. Distributed simulation and the *time-warp* operating system. In *11^{th} ACM Symposium on Operating Systems Principles*, pages 77–93, Austin - Texas, November 1987.

[5] R. M. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, pages 34–41, Pennsylvania, August 1988.

[6] R. M. Fujimoto. Performance measurements of distributed simulation strategies. In *SCS Conference on Distributed Simulation*, pages 14–20, San Diego - California, February 1988.

[7] R.M. Fujimoto. Parallel discret-event simulation. *Communications of the ACM*, pages 31–53, October 1990.

[8] Ph. Ingels and M. Raynal. Simulation répartie : schémas d'exécution pour un modèle à processus. *Techniques et Sciences Informatiques*, 9(5):383–397, 1990.

[9] D. Jefferson. Virtual time. *ACM Toplas*, 7(3):404–425, July 1985.

[10] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft. *Operating systems: advanced concepts*. The Benjamin/Cummings Pub. Co., 1987. 414 pgs.

[11] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[12] G. Neiger and S. Toueg. Substituting for real time and common knowledge in distributed systems. In *6^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 281–293, August 1987.

[13] D. Potier and M. Veran. *Qnap2*: a portable environment for queueing systems modelling. In *International Conference on Modelling Techniques and Tools for Performance Evaluation*, 1984.