Análise de Custos, Desempenho e Balanceamento de Carga de Algoritmos de Junção em Ambientes com Memória Distribuída[1]

Alberto Lerner[2] **Sérgio Lifschitz**[3] **Alexandre Plastino**[4]

E-mail: {lerner, lifschit, plastino}@inf.puc-rio.br
PUCRioInfMCC16/96

Abstract- We study here the practical and analytical behavior of some parallel join algorithms implemented in a shared-nothing environment. Particularly, we have considered the parallel approaches for the nested-loops algorithm and the hashing algorithm. The implementations are done in a IBM SP2, using the MPI and PVM tools. We first present a cost model for both algorithms and some comments on the potential performances before discussing the practical results achieved. Then we discuss two important issues: the communication cost and workload balancing. With respect to the latter, we suggest and implement a parallel strategy that dynamically balance the workload based on a better use of the computational resources available. We also show some preliminary results and a performance discussion for this strategy.

Keywords- Databases, Parallelism, Join, Behavior Analysis, Load Balancing.

Resumo- Neste trabalho, estudamos de forma analítica e prática o desempenho de algoritmos paralelos para implementação da junção em arquiteturas com memória distribuída. Em particular, consideramos as abordagens de paralelização dos algoritmos de iterações aninhadas e de *hashing*. As implementações são feitas em uma máquina IBM SP2, utilizando ferramentas como o MPI e o PVM. Num primeiro momento, é apresentado um modelo de custos e os desempenhos induzidos por este são comentados. Em seguida, os resultados obtidos na prática são apresentados e analisados, com ênfase em dois pontos importantes: o custo de comunicação e o balanceamento de carga. Especificamente quanto a este último aspecto, propomos e implementamos uma estratégia de paralelização com equilíbrio dinâmico de carga baseada em uma melhor utilização dos recursos computacionais disponíveis. Os resultados iniciais desta implementação são também apresentados e discutidos.

Palavras-chave- Bancos de dados, Paralelismo, Junção, Análise de Comportamento, Equilíbrio de Carga

---

[1] Este trabalho foi parcialmente desenvolvido com o apoio do Laboratório Nacional de Computação Científica/CNPq
[2] Mestrando, bolsista CAPES
[3] Professor Assistente, parcialmente financiado pelo CNPq/PQ 300048/94-7
[4] Doutorando, Professor Assistente do Departamento de Ciência da Computação da UFF

1. Introdução

A operação de junção é, sem dúvida, uma das operações mais realizadas e mais importantes em sistemas de bancos de dados relacionais. Dado que tem um custo bastante elevado, diversas propostas de algoritmos são conhecidas na literatura, todas visando desenvolver técnicas eficientes para realizar esta operação (*e.g.* [Bra84][ME92] [HR96]).

Com o advento dos bancos de dados paralelos [PMC+90][DG92], o estudo destes algoritmos foi revisto para viabilizar a execução da junção por vários processadores simultaneamente [SD89][LOT94]. Entre as várias arquiteturas possíveis, aquela baseada em memória distribuída (*shared-nothing* ou *message-passing*), onde cada processador tem acesso a recursos próprios não compartilhados, tem sido considerada a que melhor se adapta ao contexto de sistemas de bancos de dados [Sto86][DG92].

Este trabalho tem por objetivo realizar um estudo sobre o comportamento de alguns algoritmos de junção em um ambiente com memória distribuída. São escolhidos os algoritmos de iterações aninhadas (*nested loops*) e de *hashing* [EN94]. Ao invés de realizarmos um conjunto completo de testes computacionais, optamos por estudar e comparar os desempenhos de situações que ilustrem certas características do comportamento dos algoritmos. Em particular, estamos interessados em dois aspectos: o custo de comunicação, fundamental para algoritmos utilizando a arquitetura em questão, e o balanceamento de carga, problema ainda pouco explorado até o momento.

Ao descrever os algoritmos selecionados, realizamos uma análise dos custos da junção em paralelo, o que inclui os custos de transmissões de dados entre os processadores. Visando obter uma idéia mais precisa do comportamento dos algoritmos, resolvemos também implementá-los e testá-los em um ambiente real de desenvolvimento paralelo: o sistema IBM SP2 [AMM+95]. Como será descrito, utilizamos ferramentas de paralelização de processos como o PVM [GBD+94] e o MPI [MPI95]. No referido ambiente, o PVM utiliza a rede *Ethernet* para interligar os processadores, enquanto o MPI lança mão de um *switch* de alta velocidade para tanto. Esta diferença entre as duas ferramentas permitiu que se fizessem comparações quanto ao custo de comunicação que cada tipo de interligação conferia à implementação.

Por fim, analisamos o problema de equilíbrio de carga entre os processadores, que pode ter efeito negativo em uma execução paralela. A maior parte dos trabalhos existentes nesta área propõem métodos de balanceamento que visam dividir o trabalho de processamento em partes iguais. Estas sub-tarefas geralmente são distribuídas *a priori* entre os processadores participantes da operação e um eventual balanceamento é feito após detecção de desequilíbrios [DNS+92][RM93][RM95]. Porém, os processadores de uma máquina paralela podem não ter capacidade de processamento uniforme, podem estar dedicados a mais de uma tarefa concorrentemente ou podem até mesmo ter distribuição de dados não ser uniforme entre eles, fazendo com que estas tentativas de equilíbrio não sejam eficientes em alguns casos. Nesse contexto, propomos e analisamos uma estratégia de balanceamento dinâmico de carga que, aplicada à ambos os algoritmos, promete um melhor desempenho na prática.

Após essa rápida introdução, o texto segue organizado da seguinte forma: na próxima seção, apresentamos uma breve revisão sobre a operação de junção. Na Seção 3, também revisamos os algoritmos paralelos de junção escolhidos e comentamos as análises de custos normalmente

considerados. Na Seção 4, são descritas em detalhes e comparadas as implementações dos algoritmos escolhidos como objetos de estudo. Nesta mesma seção, são detalhados o ambiente de desenvolvimento e características das ferramentas de paralelização aplicadas. Em seguida, na Seção 5, apresentamos e comparamos os resultados práticos com aqueles esperados. Também nesta seção chamamos a atenção para os problemas de custo de comunicação e de balanceamento de carga que os algoritmos apresentam. Na Seção 6, descrevemos nossa proposta de estratégia de paralelização com balanceamento dinâmico de carga e aplicamos a mesma a um dos algoritmos relatando os resultados inicialmente coletados. Concluímos, na Seção 7, citando trabalhos em desenvolvimento e futuros.

2. A Operação de Junção

Em [EN94], a junção é definida como a operação da álgebra relacional utilizada para combinar pares de tuplas correlatas de duas relações em tuplas únicas. Esta operação é essencial para qualquer sistema de banco de dados que contenha mais de uma relação, pois permite explorar o conceito de relacionamento entre estas.

R	A1	A2	A3
	001	aaaa	1/02/96
	003	bbbb	3/06/84
	005	cccc	23/04/92
	005	dddd	17/09/87
	009	eeee	2/05/95
	012	ffff	1/03/93

S	B1	B2
	001	AAAA
	002	BBBB
	005	CCCC
	011	DDDD

Q	A1	A2	A3	B1	B2
	001	aaaa	1/02/96	001	AAAA
	005	cccc	23/04/92	005	CCCC
	005	dddd	17/09/87	005	CCCC

Figura 1 - Exemplo de uma operação de junção entre a relação R e S gerando Q como resultado

A forma genérica da junção, dadas duas relações $R(A_1, A_2, ..., A_n)$ e $S(B_1, B_2, ..., B_m)$, é expressa como:

$$R \text{ X}_{<condição\ de\ junção>} S \quad (2.1)$$

O resultado da junção é uma relação Q com $n + m$ atributos na ordem $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, onde Q possui uma tupla para cada caso onde se possa combinar o par $<r, s>$ segundo a condição de junção, sendo r uma tupla da relação R e s uma tupla da relação S. Na figura 1, temos ilustrada a junção entre duas relações $R(A_1, A_2, A_3)$ e $S(B_1, B_2)$, com relação-resultado $Q(A_1, A_2, A_3, B_1, B_2)$, fruto da condição de junção $A_1 = B_1$.

A operação de junção é algo similar ao produto cartesiano, como definido na Teoria de Conjuntos. No produto cartesiano, cada tupla da relação R é combinada com todas as tuplas da relação S. Supondo $\|R\|$ e $\|S\|$ o número de tuplas das relações R e S, respectivamente, o produto cartesiano dá origem a uma relação resultante de $\|R\| \ . \ \|S\|$ tuplas. Pode-se dizer também que a operação de junção é composta pelo produto cartesiano de R e S seguido de uma seleção das tuplas que atendam à condição de junção.

A condição de junção pode ser expressa na forma $R(a)\ \theta\ S(b)$ ou $R(a)\ \theta\ c$, onde θ é um operador que define uma comparação entre o atributo a da relação R e o atributo b de S ou entre a e uma constante c. Essa junção genérica é chamada de θ-junção. O operador θ poderia ser um dos seguintes operadores de comparação: $=, \neq, <, >, \leq, \geq$.

Equijunção é a denominação específica atribuída à junção onde θ é o operador de igualdade. A junção natural é a variante da equijunção na qual os atributos de junção têm o mesmo nome em ambas as relações envolvidas na operação. Neste caso, o resultado da junção omite o atributo de junção de uma das relações.

O número de tuplas da relação resultante de uma junção, representado por $\|RXS\|$, será sempre menor ou igual ao número de tuplas gerado pelo produto cartesiano das mesmas relações. À razão entre estas quantidades de tuplas dá-se o nome de fator de seletividade *fs,* onde:

$$fs = \frac{\|RXS\|}{\|R\|.\|S\|} \quad (2.2)$$

O fator de seletividade irá variar de 0 a 1 indicando, respectivamente, um menor ou maior número de tuplas geradas para uma dada operação de junção.

3. Descrição dos Algoritmos

3.1 Algoritmos Seqüenciais

Existem três principais estratégias para a resolução da equijunção: a junção por iterações aninhadas (*nested-loops*), a junção através de ordenação e fusão (*sort-merge*) e a junção baseada em *hash* (*hash-based*) [EN94].

A junção por iterações aninhadas é a mais simples e intuitiva dentre elas. A junção é feita

comparando-se cada tupla *r* da relação *R* com todas as tuplas *s* da relação *S*. Quando a condição de junção é satisfeita para um par <*r*, *s*>, este é incluído na relação-resultado. O nome do algoritmo vem do fato dele ser comumente implementado através de duas iterações aninhadas.

A junção através da ordenação e fusão é feita, como sugere o nome, em duas fases: na primeira, as tuplas das relações *R* e *S* são ordenadas pelo atributo sobre o qual será feita a equijunção e, na segunda, as duas relações ordenadas são percorridas simultaneamente, tendo os pares <*r*, *s*> que satisfazem a condição de junção inseridos na relação-resultado.

A junção baseada em *hashing* é o método que mais variações oferece [LOT94]. Este método procura identificar as tuplas da primeira relação que possam satisfazer a condição de junção com uma determinada tupla da segunda relação. Desta forma, cada tupla da segunda relação é comparada com um conjunto menor de tuplas da primeira.

A identificação das tuplas é feita através da técnica de *hashing*. Nesta técnica, uma função mapeia os valores de um domínio A num domínio A', este último assumindo um número de valores bem menor do que o domínio A. É portanto provável que valores distintos do domínio A assumam o mesmo valor no domínio A', caracterizando uma colisão.

Aplicando a função de *hashing* ao atributo de junção de todas as tuplas da relação *R* obtemos conjuntos disjuntos de tuplas com valores idênticos em A'. A partir daí, cada tupla de *S* é comparada apenas com aquele conjunto que assumiu em A' o mesmo valor que o seu atributo de junção ao aplicar a função.

3.2 Análise de Custos e Comparação entre os Algoritmos Sequenciais

Numa abordagem simplificada, a complexidade dos três algoritmos poderia ser avaliada através do número de comparações entre tuplas que cada um realiza. A junção por iterações aninhadas claramente realiza $||R|| \cdot ||S||$ comparações. No caso da seletividade de uma dada junção ser baixa, o algoritmo de junção por iterações aninhadas estaria fazendo um grande número de comparações desnecessárias.

O algoritmo de junção pela ordenação e fusão tenta reduzir este número de comparações desnecessárias ordenando as relações *R* e *S* antes de compará-las. O custo desta ordenação é $||R||.\log||R|| + ||S||.\log||S||$. Considerando o caso mais simples onde os atributos envolvidos na condição de junção não contenham repetição de valores, seriam necessários apenas $||R||+||S||$ comparações para que se gerasse a relação-resultante. A complexidade final do algoritmo seria então $||R||.\log ||R|| + ||S||.\log||S|| + ||R|| + ||S||$.

A idéia do método por *hashing* é a de obter exatamente o mesmo número de comparações $||R|| + ||S||$ do algoritmo de junção por ordenação e fusão, porém sem o custo de uma ordenação das relações envolvidas. No suposto caso de haver um função de *hashing* determinística, onde não haveriam colisões de tuplas em um mesmo valor de *hash*, o custo para a construção da tabela de *hash* seria $||R||$ e o custo de comparação do valor de *hash* de cada tupla *s* da relação *S* com os valores da tabela seria $||S||$. Desta forma, o algoritmo de junção por *hash* geraria o menor número de comparações dentre os três.

Entretanto, eleger um dos algoritmos como melhor requer uma análise da complexidade mais minuciosa do que a exposta até aqui. Em primeiro lugar, é perfeitamente plausível que um atributo envolvido na condição de junção assuma valores duplicados, uma vez que nem sempre

este é um atributo chave. Neste caso, a comparação feita na segunda etapa do algoritmo de junção por ordenação e fusão teria de levar isto em conta, implicando em um número de comparações na prática maior do que $\|R\| + \|S\|$. De fato, no pior caso, onde o fator de seletividade da junção fosse próximo a 1 e houvesse grande taxa de repetição de valores, o custo da comparação seria bem maior que $\|R\| + \|S\|$, tendendo a $\|R\| \cdot \|S\|$ como um limite superior. A ordenação, neste caso, teria sido em vão.

É também provável que nem sempre uma dada função de *hashing* produza valores distintos (*i.e.*, sem colisão) para cada valor no domínio de entrada. Isto porque as funções utilizadas para o *hashing* não são determinísticas, fazendo com que a distribuição de valores seja dependente dos dados de entrada. Uma má escolha da função de *hashing* no último algoritmo descrito poderia, então, levar a um efeito semelhante. A distribuição das tuplas da relação *R* pode ser tão imperfeita na tabela de *hash* que, a cada tupla *s* da relação *S*, sejam necessárias muitas comparações com valores relativos à colisões.

Em segundo lugar, deve-se considerar que um sistema de banco de dados relacional provê estruturas de auxílio à acesso aos dados, tais como os índices. A existência ou não de índices pode mudar bastante a eficiência de cada um dos algoritmos expostos, gerando variantes de algoritmos que apresentam menor custo do que os algoritmos básicos descritos.

Em terceiro lugar, estudos de algoritmos para implementar os operadores propostos pela álgebra relacional estão baseados em uma premissa muito importante: o somatório de espaço necessário para acomodar tuplas de relações envolvidas numa dada operação é sempre maior que a quantidade de memória disponível [LOT94]. Numa análise rápida, isto significa que a ordenação, no caso da junção por ordenação e fusão, deve ser externa (*i.e.*, não pode ser feita somente em memória primária) e que o algoritmo por *hashing* deve prever *overflow* em disco da tabela de *hash* construída em memória. O próprio algoritmo de junção baseado em iterações aninhadas tem variações sofisticadas de utilização de memória no sentido de reduzir ao máximo o número de acessos ao disco.

Em condições reais, além de terem um componente de tempo ligado à comparação de tuplas, os algoritmos são também fortemente influenciados pelo componente de entradas e saídas (E/S). O custo muitas vezes superior de E/S em relação ao custo de uma comparação em memória fará com que a complexidade dos algoritmos de junção seja dominada pelo primeiro.

Sendo assim, podemos concluir que não há um algoritmo melhor que os demais em termos absolutos. Cada qual, mesmo o mais simples, é mais adequado a uma certa gama de situações. A diferença entre eles é quão frequente, em termos reais, estas situações se aplicam. Isto sugere que, também no ambiente paralelo, várias alternativas para a solução do problema sejam disponibilizadas.

3.3 Algoritmos Paralelos

Uma das formas de se conseguir melhorias no tempo de resposta do processamento da junção é realizá-la em um ambiente composto de p ($p > 1$) processadores que atuem cooperativamente e coordenadamente. De forma similar a outros algoritmos, o processamento paralelo da junção envolve normalmente três passos: decomposição da junção em n tarefas, atribuição das n tarefas aos p processadores e montagem dos resultados parciais das tarefas. Intuitivamente, a junção de sub-relaÁ¦es das duas relações operandas, obtidas através do processo de replicação (ou fragmentação) das tuplas envolvidas na junção, seria uma boa decomposição da junção em

sub-tarefas.

O processo de replicação (ou fragmentação) das tuplas das relações *R* e *S* numa arquitetura com memória distribuída é uma etapa composta de envios de tuplas entre os processadores. Vale ressaltar que esta etapa se distingue da etapa de distribuição inicial dos dados entre os processadores, na qual cada processador recebe os dados que deverá armazenar, enquanto que no processo de replicação cada processador recebe as tuplas de *R* e *S* que deverá processar. Para os cálculos locais a cada processador, pode-se usar qualquer um dos algoritmos seqüenciais conhecidos na literatura [EN94]. Antes de descrever os algoritmos estamos assumindo que:

- As relações *R* e *S* são inicialmente particionadas uniformemente por todos os processadores utilizados no processamento da junção, isto é, todo processador que participa da operação armazena aproximadamente $\|R\|/p$ tuplas de *R* e $\|S\|/p$ de *S*;
- Não são conhecidos os critérios pelos quais as relações foram particionadas, de modo que não se pode saber se há relação entre o(s) atributo(s) que compõe(m) a condição de junção e o(s) atributo(s) sobre os quais a partição foi feita. Em outras palavras, não se sabe se a junção é HPJA (*Hash-Partitioned on their Join Atributes*) ou não-HPJA, de acordo com [SD89];
- Não é garantido que haja memória suficiente em cada um dos nós que participam do algoritmo para o processamento de todas as tuplas lá armazenadas de uma só vez.

3.3.1 Junção Paralela por Iterações Aninhadas

O algoritmo sequencial de junção por iterações aninhadas é facilmente paralelizável [LOT94]. Como cada tupla *s* da relação *S* será comparada com todas as tuplas *r* da relação *R* - a menor entre as duas, para efeito de redução do número de E/S [EN94] - esta última pode ser replicada por todos os nós que participam da operação. Esta replicação é necessária pelo fato de, em arquiteturas com memória distribuída, cada processador apenas acessar disco e memória locais.

O algoritmo 3.1 descreve a junção por iterações aninhadas paralela. A iteração nas linhas 1-3 é a responsável pela transmissão de cada um dos fragmentos R_i, $0 < i < p$-1 da relação *R* entre todos os nós *j*, $j \neq i$. Ao final deste passo, cada nó *i* terá recebido *p*-1 fragmentos de *R* que, fundidos ao seu próprio fragmento R_i, formarão uma réplica de *R* em cada nó. As linhas 2-3 representam a operação de comunicação de todos-para-todos. Na iteração nas linhas 4-5, um algoritmo de junção seqüencial é utilizado para realizar a junção local entre o fragmento da relação *S* previamente armazenado naquele nó e a recém-adquirida réplica da relação *R*.

```
ITERAÇÕES-ANINHADAS-PARALELO
1  para todo nó i em paralelo
2    "broadcast" Ri para os outros nós
3    receba os fragmentos dos nós Rj , j ≠ i
4  para cada nó i em paralelo
5    JUNCAO-LOCAL(réplica de R, fragmento Si)
```

Algoritmo 3.1

No caso, e apenas por facilidade de implementação, o algoritmo de junção local escolhido foi uma variação do algoritmo básico de iterações aninhadas chamado de iterações aninhadas por blocos [EN94]. Neste algoritmo, uma heurística de utilização de memória é aplicada para otimizar o número de acessos a disco. Pela heurística, a relação armazenada no menor número de páginas de disco é utilizada na iteração mais externa e consome todas as páginas de memória reservadas ao processamento de tuplas das relações menos uma página. Esta última será utilizada apenas para conter tuplas da outra relação envolvida na junção.

O algoritmo 3.2 descreve o processo de junção local. As iterações iniciando nas linhas 1 e 3 carregam para memória o máximo de páginas de disco possível das relações *R* e *S*, respectivamente. A partir daí, as iterações aninhadas das linhas 5-6 fazem as comparações entre as tuplas atualmente em memória. As condições das linhas 9-10 garantem que todas as tuplas das duas relações serão lidas e processadas pelo algoritmo.

```
JUNCAO-LOCAL(R, S)
1   repita
2     leia (M - 1) páginas da relação R em memória
3     repita
4       leia 1 página da relação S em memória
5       para cada tupla r da relação R em memória faça
6         para cada tupla s da relação S em memória faça
7           se (condição de junção) for verdadeira
8             junção(r,s)
9     até que todas as páginas da relação S tenham sido lidas
10  até que todas as páginas da relação R tenham sido lidas
```

Algoritmo 3.2

3.3.2 Análise de Custos da Junção Paralela por Iterações Aninhadas

Para analisar o tempo gasto por este algoritmo, além de considerar o tempo das operações em memória, é necessário também levar em consideração os custos de E/S e de comunicação entre os processadores. Para simplificar o cálculo, o algoritmo será dividido em duas fases: a de replicação dos fragmentos da relação *R* (linhas 1-3 do algoritmo 3.1) e a de junção local (linhas 4-5 do algoritmo 3.1 e algoritmo 3.2). Sendo assim, o tempo total do algoritmo, representado por T_{IA} na expressão 3.1, será dado pela soma dos tempos da fase de replicação, $T_{replicação}$, e na fase de junção local, $T_{junção}$.

$$T_{IA} = T_{replicação} + T_{junção} \qquad (3.1)$$

O tempo gasto na fase de replicação é puramente o tempo utilizado na operação de comunicação de todos-para-todos. Porém, o custo desta operação depende do tipo de rede de interconexão na qual os processadores estão ligados [KGG+94]. Sendo assim, a análise desta operação será tratada como uma série de comunicações entre os processadores, isto é, cada processador envia sua réplica da relação *R* para os outros *p*-1 processadores e recebe *p*-1 fragmentos desta relação, gravando-os em disco. Este resultado é dado na expressão 3.2.

É válido ressaltar que as operações de E/S têm seu custo medido em função do número de páginas de disco lidas ou gravadas. Para uma dada relação *R*, o número mínimo de páginas

necessárias para armazená-la por inteiro é dado pela representação $|R|$. Também para efeito de cálculo de tempo, são definidas as constantes op_c, op_d e op_m,, representando o tempo médio gasto respectivamente em uma operação de transmissão de uma tupla de um nó para o outro, em uma operação de leitura ou gravação de uma página em disco e em uma operação envolvendo tuplas em memória.

$$T_{replicação} = (op_d \,.\, |R|/p) + (op_c \,.\, ||R||/p \,.\, (p\text{-}1)) + (op_d \,.\, |R|/p \,.\, (p\text{-}1)) \qquad (3.2)$$

O primeiro termo da expressão 3.2 diz respeito ao tempo gasto na leitura do fragmento da relação R armazenado em cada nó, sendo este custo medido como função do número de páginas da relação. O segundo termo expressa o tempo gasto na transmissão do fragmento da relação R para os outros p-1 nós, medido como função do número de tuplas do fragmento. Finalmente, o último termo reflete o tempo utilizado para armazenar em disco todas as tuplas recebidas de outros nós, novamente medido como função do número de páginas.

De fato, esta aproximação para a operação de comunicação de todos-para-todos reflete o tempo dado em [KGG+94] para redes em anel, malha e hipercubo, como nas expressões 3.3, 3.4 e 3.5, respectivamente. Seja t_s o tempo de inicialização de um processador, t_w o tempo de transmissão de uma unidade de mensagem e m o número de unidades a serem transmitidas, os tempos em [KGG+94] para roteamento por *store-and-forward* são:

$$T_{all_to_all_anel} = (t_s + t_w m)(p\text{-}1) \qquad (3.3)$$

$$T_{all_to_all_malha} = 2t_s(\sqrt{p} - 1) + t_w m(p\text{-}1) \qquad (3.4)$$

$$T_{all_to_all_cubo} = t_s \log p + t_w m(p\text{-}1) \qquad (3.5)$$

Se a comparação $t_s << t_w m$ for válida, os três tempos acima se reduzem a $t_w m(p\text{-}1)$. Supondo para o caso da replicação que $t_w = op_c$ e $m = ||R||/p$, tem-se exatamente o mesmo resultado que o dado pelo segundo termo de $T_{replicação}$, na expressão 3.2. A diferença está no fato de, em [KGG+94], se buscar e armazenar dados em memória, ao passo que este trabalho segue a premissa de que o número de tuplas a serem manipuladas é muito maior que o número de tuplas que podem ser acomodadas em memória.

A análise do tempo gasto na fase de junção local, descrita pela expressão 3.6, deve levar em conta os acessos à disco necessários para a carga em memória das tuplas da réplica da relação R e do fragmento da relação S, o número de comunicações necessárias para concentrar as tuplas que sofrerão a junção num só processador e o tempo de CPU para a comparação entre as tuplas na memória. O tempo de junção será dependente da quantidade M de páginas de memória disponíveis para tal.

$$T_{junção} = (op_d \,.\, (\,|R| + \lceil |R|/(M\text{-}1) \rceil \,.\, |S|/p)) + (op_c \,.\, fs \,.\, ||R|| \,.\, ||S||/p) + (op_m \,.\, ||R|| \,.\, ||S||/p) \qquad (3.6)$$

No que diz respeito à operações em disco, cada página da réplica da relação R será lida uma vez e cada página do fragmento da relação S será lida tantas vezes quantas forem necessária para compara-la com todas as páginas de R, isto é, $\lceil |R|/(M\text{-}1) \rceil$ vezes, como representado no primeiro termo. O segundo termo reflete o número de comunicações entre processadores,

necessárias quando é feita a junção de duas tuplas. Neste caso, o número de comunicações é proporcional ao fator de seletividade *fs* da junção da réplica da relação *R* com o fragmento da relação *S*. Por fim, no último termo é dado o número de comparações de tuplas em memória que o algoritmo faz.

É importante notar que, pela heurística utilizada, a relação com menor número de tuplas será utilizada na iteração mais externa. Por convenção, a relação *R* é a menor das duas relações. Porém, apesar de $||R|| < ||S||$ ser uma afirmação válida, é possível que $||R|| > ||S||/p$ seja também verdadeira. Neste caso, a relação *R* do algoritmo de junção local passará a ser o fragmento da relação *S* inicial e a relação *S* deste algoritmo será a réplica da relação *R* original, fazendo com que o primeiro termo de $T_{junção}$ passe a ser como o descrito na expressão 3.7.

$$op_d \,.\, (\,|S|/p + \lceil |S|/(p(M-1)) \rceil \,.\, |R|) \qquad (3.7)$$

Do ponto de vista de aceleração *(speedup)*, este algoritmo não trará um aumento de velocidade proporcional à adição de outros processadores à tarefa. Isto porque as variáveis das expressões de tempo $T_{replicação}$ e $T_{junção}$, 3.2 e 3.6 (ou 3.7) respectivamente, não necessariamente são inversamente proporcionais ao número de processadores *p*.

Supondo *p* suficientemente grande, tal que $(p - 1) \approx p$, poderia-se simplificar a expressão 3.2 para

$$T_{replicação} = (\, op_d \,.\, |R|/p \,) + (\, op_c \,.\, ||R|| \,) + (\, op_d \,.\, |R| \,) \qquad (3.8)$$

onde, do ponto de vista de operações de comunicação e de disco, a fase de replicação não fornece aceleração linear. Também pelo fato da relação *R* ser replicada por todos os processadores, a fase de junção local, na expressão 3.6, apresenta termos como $||R||$ e $|R|$ não proporcionais a *p*.

3.3.3 Junção Paralela por *Hashing*

O algoritmo sequencial de junção por *hash* é o melhor, dentre os métodos de junção, para a aplicação de paralelismo [LOT94]. A idéia básica do algoritmo é agrupar tuplas tanto da relação *R* como de *S* em *buckets*. Os *buckets* são conjuntos disjuntos de tuplas com alta probabilidade de junção entre si, isto é, as tuplas com o mesmo valor de *hash* sobre o atributo de junção estão agrupadas. Após a geração dos *buckets*, cada um destes conjuntos sofre uma junção local. Isto sugere que tanto a fase de partição em *buckets* como a posterior junção podem ser feitas em paralelo.

```
HASH-PARALELO
1  para todo nó i em paralelo
2    para cada tupla r do fragmento R_i e s do fragmento S_i
3      j ← hash(atributo de junção)
4      envia r ou s para nó j
5    recebe os r ou s enviados para i
6  para cada nó i em paralelo
7    JUNCAO-LOCAL(bucket R_i, bucket S_i)
```

Algoritmo 3.3

A junção paralela por *Hash* é descrita no algoritmo 3.3. A iteração das linhas 1-5 é a responsável pela formação de *p buckets* contendo tuplas tanto da relação *R* como de *S*. Cada *bucket* é enviado para um processador específico e por ele será tratado. Ao final deste passo, cada processador estará armazenando um conjunto de tuplas de *R* e *S* que tenham atributos de junção com o mesmo valor de *hash*, isto é, com alta probabilidade de junção. Na iteração das linhas 6-7, um algoritmo de junção sequencial é utilizado para realizar a junção local das tuplas da relação *R* e das de *S* que façam parte de um mesmo *bucket*. Também aqui será utilizado o algoritmo 3.2, a variação do algoritmo de iterações aninhadas seqüencial descrito anteriormente.

3.3.4 Análise de Custos da Junção Paralela por *Hashing*

Para analisar o tempo gasto por este algoritmo, além de considerar o tempo de operações em memória, é necessário também levar em consideração os custos de E/S e de comunicação entre processadores, assim como feito para os algoritmos 3.1 e 3.2. Para efeito deste cálculo, o algoritmo será dividido em duas fases: a de partição em *buckets* (linhas 1-5 do algoritmo 3.3) e a de junção local (linhas 6-7 do algoritmo 3.3 e algoritmo 3.2). Sendo assim, o tempo total do algoritmo, representado por T_{hash} na expressão 3.9, será dado pela soma dos tempos da fase de replicação, $T_{partição}$, e na fase de junção local, $T_{junção}$.

$$T_{hash} = T_{partição} + T_{junção} \tag{3.9}$$

O tempo gasto na fase de partição está associado ao cálculo do valor de *hash* de cada tupla armazenada nos fragmentos de *R* e *S* e seu envio ao processador responsável pelo tratamento de sua junção. A expressão 3.10 reflete o cálculo deste custo.

$$T_{partição} = (op_d \,.\, (|R|/p + |S|/p)) + (op_m \,.\, (\|R\|/p + \|S\|/p)) + (op_c \,.\, (\|R\|/p + \|S\|/p)) + (op_d \,.\, (|R|/p + |S|/p)) \tag{3.10}$$

O primeiro termo da expressão 3.10 reflete o tempo gasto na leitura, por parte de cada processador, do fragmento da relação *R* e da relação *S* aí armazenados. Como visto no cálculo de tempo dos algoritmos 3.1 e 3.2, os tempos de acesso a disco são dados em função do número de páginas de cada relação. O segundo termo considera que o cálculo do valor de *hash* para uma dada tupla pode ser feito em uma unidade de computação em memória. O tempo gasto na transmissão das tuplas entre os processadores é visto no terceiro termo da mesma expressão.

O quarto termo da expressão 3.10 diz respeito ao armazenamento das tuplas de um *bucket* em disco. Neste caso, é assumido que a função de *hashing* utilizada produza *p buckets*, cada qual contendo exatamente $\|R\|/p$ tuplas de *R* e $\|S\|/p$ tuplas de *S*. Trata-se do caso ideal, pois no pior caso, a distribuição que a função de *hash* imprimiria a um dado conjunto de tuplas poderia chegar a uma configuração de apenas um *bucket* contendo todas as tuplas de *R* e *S*. Isto provavelmente ocorreria no caso em que os valores do atributo de junção de todas as tuplas fossem iguais. Então, no pior caso, o quarto termo da expressão 3.10 poderá chegar a

$$(op_d \,.\, (|R| + |S|)) \tag{3.11}$$

Como visto na expressão 3.6, o algoritmo seqüencial utilizado para a junção local tem seu tempo intimamente ligado ao número de tuplas envolvidos na junção e ao fator de seletividade. Como a distribuição de tuplas é feita através de *hashing*, o cálculo do tempo consumido na fase de junção local deve levar em conta a possibilidade ou não das tuplas terem sido distribuídas uniformemente entre os *buckets*. A expressão 3.12 reflete o cálculo deste tempo no caso onde a distribuição é perfeitamente uniforme. Já a expressão 3.13 assume o pior caso, onde todas as tuplas, tanto de *R* como de *S,* foram enviadas para um mesmo *bucket*.

$$T_{junção} = (op_d \,.\, (\, |R|/p + \lceil |R|/p(M-1) \rceil \,.\, |S|/p)) + (op_c \,.\, fs \,.\, ||R||/p \,.\, ||S||/p) + (op_m \,.\, ||R||/p \,.\, ||S||/p) \quad (3.12)$$

$$T_{junção} = (\, op_d \,.\, (\, |R| + \lceil |R|/(M-1) \rceil \,.\, |S|) \,) + (\, op_c \,.\, fs \,.\, ||R|| \,.\, ||S|| \,) + (\, op_m \,.\, ||R|| \,.\, ||S|| \,) \quad (3.13)$$

Vale ressaltar que o tempo dado em 3.13 é similar ao tempo gasto por apenas um processador para realizar a junção de toda a relação *R* com a relação *S*, se não for considerado o custo de comunicação. Sendo assim, o algoritmo por *hashing* deve ter uma forte preocupação com balanceamento de carga entre os processadores, sob pena de grande desperdício de eficiência computacional, pois o tempo da fase de junção do algoritmo paralelo por *hashing* será dominado pelo processador que contiver o maior número de tuplas de *R* e de *S* juntas.

Levando-se em conta o caso ideal do algoritmo, a aceleração obtida através da adição de novos processadores seria linear. Em ambos os tempos $T_{partição}$ e $T_{junção}$ as variáveis $|R|$, $||R||$, $|S|$ e $||S||$ sempre aparecem inversamente proporcionais ao número de processadores p. Em tese, provavelmente a aceleração do algoritmo se apresentará tão linear quanto for a capacidade da função de *hash* em dividir as tuplas de *R* e *S* em *buckets* de mesmo tamanho.

3.4 Comparação entre os Algoritmos Paralelos

Os algoritmos paralelos podem ser comparados segundo critérios de alocação de dados aos diferentes processadores, o custo da comunicação entre eles e, finalmente, a divisão de trabalho entre os processadores.

3.4.1 Divisão de Dados entre os Processadores

Apesar de utilizarem o mesmo algoritmo de junção local, os dois algoritmos diferem bastante entre si. Esta diferença reside na forma como os dois dividem as tuplas envolvidas na junção entre os vários processadores.

A figura 2 ilustra a forma como o algoritmo por iterações aninhadas paralelo faz a partição das relações *R* e *S* em vários fragmentos e a alocação destes entre os processadores. A relação *R* está representada como o eixo horizontal e a relação *S* como o eixo vertical. Na figura 2(a), uma sugestão de distribuição de dados entre os processadores antes da replicação da relação *R*. Na figura 2(b), a distribuição dos dados antes do início da junção local.

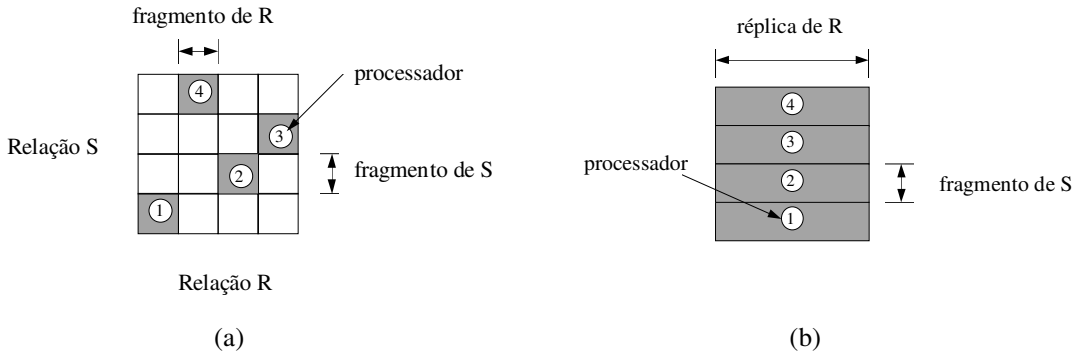


Figura 2 - Ilustração da sub-divisão em tarefas do algoritmo paralelo de iterações aninhadas (a) antes e (b) depois da replicação de *R*

Da mesma forma, a figura 3 sugere como o algoritmo paralelo por *hashing* faz a partição das relações *R* e *S*. Na figura 3(a) está representado uma sugestão de fragmentação das tuplas das duas relações entre os vários nós antes da fase de *hashing*. A figura 3(b) demonstra que tuplas com o mesmo valor de *hash* sobre o atributo de junção estarão juntas em um mesmo processador. Notar a não uniformidade na divisão de tuplas representado pelo lado direito da figura.

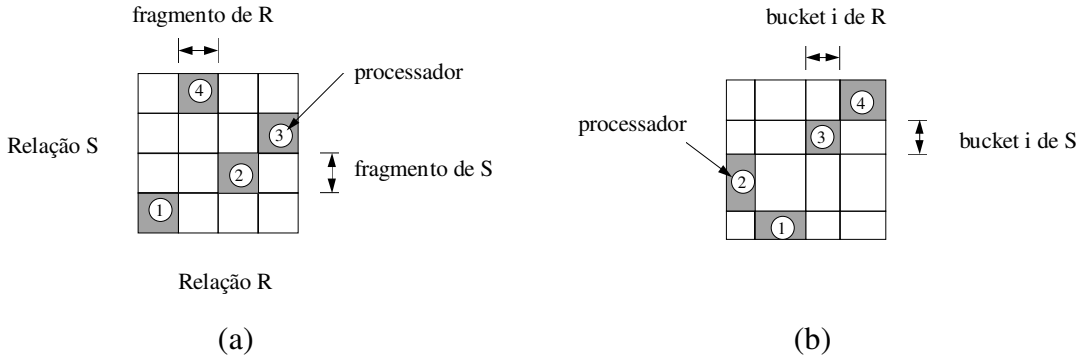


Figura 3 - Ilustração da sub-divisão em tarefas do algoritmo paralelo de junção por hash (a) antes e (b) depois da fase de agrupamento de tuplas em *buckets*

3.4.2 Custo de Comunicação entre Processadores

Uma vez que a divisão de dados entre os processadores foi estabelecida, cada algoritmo terá um custo para movimentar os dados de forma que a junção possa ter início. O algoritmo de iterações aninhadas em paralelo se baseia no método de replicação de tuplas em oposição ao algoritmo de *hash* em paralelo, que utiliza uma estratégia de redistribuição de tuplas.

Intuitivamente, pode-se dizer que a replicação de dados leva a um custo de comunicação entre processadores maior do que uma distribuição dos mesmos. Na distribuição, a mesma quantidade de dados é movimentada independente do número de processadores envolvidos -

apenas o seu local de armazenamento é alterado. Na replicação, a cada novo processador acrescentado ao ambiente, uma nova réplica de *R* deve ser transmitida para o novo participante, fazendo com que esta tarefa tenha o seu custo proporcional ao número de processadores. De fato, como será visto mais adiante, este fator contribui para que o primeiro algoritmo não apresente uma taxa de aceleração linear.

Porém, há de se levar em conta que, para um dado tamanho de *R* em relação a *S* e para um dado número de processadores, pode ser mais vantajoso replicar *R* do que redistribuir *R* e *S*. Isto é, para $||R||$ muito menor que $||S||$ - o quanto menor dependerá do número de processadores envolvidos - é menos custoso em termos de comunicação entre processadores fazer *p* cópias de *R* do que movimentar todas as tuplas de *R* e *S*.

É isto que demonstra o gráfico da figura 4. Levados em conta apenas os custos de comunicação da primeira fase do algoritmo, temos que, para a área acima da curva, o algoritmo por iterações aninhadas é melhor que o por *hashing*. O formato exponencial da curva sugere que a região abaixo dela, onde o algoritmo por *hashing* é mais eficiente, será bem maior que a área acima da mesma.

O gráfico apresenta alguns pontos de interesse. Para uma configuração de dois processadores, quaisquer que sejam o número de tuplas de *R* e de *S*, o algoritmo que replica *R* é melhor. Isto porque mesmo que $||S||$ fosse igual a $||R||$, cada nó transmitiria $||R||/2$ tuplas no caso da replicação ao passo que, pela distribuição, cada nó transmitiria $||R||/2 + ||S||/2$ tuplas, o que seria equivalente a transmitir $||R||$ tuplas.

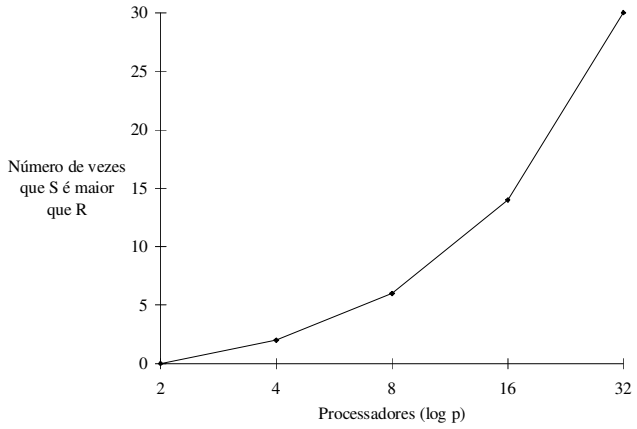


Figura 4 - Gráfico comparativo em relação ao custo de comunicação da 1ª fase dos algoritmos

3.4.3 Divisão de Trabalho entre cada Processador

Uma comparação entre as figuras 2(b) e 3(b) sugere que, como visto no modelo analítico, o número de comparações que cada processador irá realizar é maior no caso do algoritmo paralelo por iterações aninhadas. A fase de junção deste algoritmo requer um número maior de comparações que no caso por *hashing* em paralelo.

Entretanto, é importante ressaltar que nem sempre a distribuição das tuplas de *R* e *S* do algoritmo paralelo por *hashing* é uniforme. Portanto, apesar da fase de junção deste algoritmo

apresentar uma aceleração aproximadamente linear e de, na maioria dos casos, obter melhor performance que o algoritmo por iterações aninhadas, ele pode gerar distribuições não uniformes de trabalho entre os processadores.

4. Implementação dos Algoritmos

Para reportar a forma como foi feita a implementação dos algoritmos, inicialmente iremos descrever o ambiente de processamento utilizado. Em seguida, será explicado como as ferramentas PVM e MPI foram utilizadas na geração de programas que implementassem os algoritmos sendo estudados.

4.1 Ambiente de Processamento

O IBM SP2 [AMM+95] do Laboratório Nacional de Computação Científica (LNCC) foi a plataforma adotada na implementação. O SP2 é um sistema paralelo expansível de uso genérico baseado numa arquitetura com memória distribuída.

O sistema SP2 do LNCC é composto por 16 nós processadores RISC System/6000, cada qual com 256 MB de memória e 2.0 GB de disco privados e sua própria cópia autônoma do sistema operacional AIX. O coração do SP2 é um *switch* de alta velocidade e baixo tempo de latência que interconecta os nós processadores. O *switch* é composto de *links*, cada qual contendo dois canais que possibilitam a comunicação bi-direcional simultânea entre dois dispositivos a ele ligados. A comunicação entre o processador e o *switch* pode se dar de 35 MB/s a 48 MB/s, dependendo da forma como é feita [SSA+95].

A codificação dos algoritmos foi feita em linguagem C e duas interfaces de programação paralela foram utilizadas: o PVM [GBD+94] e o MPI [MPI95]. O PVM (*Parallel Virtual Machine*) é uma interface desenvolvida no *Oak Ridge National Laboratory* (ORNL) que possibilita a criação de processos simultâneos em diferentes processadores de máquinas paralelas ou de diferentes estações de redes, fornecendo facilidades como sincronismo e comunicação entre estes. O MPI (Message Passing Interface) é um esforço conjunto de padronização de uma interface de programação paralela mais recente e sofisticada que o PVM. A implementação do MPI utilizada foi o MPICH [GL96].

4.2 Descrição da Implementação sobre PVM

O primeiro algoritmo implementado foi o da junção paralela por iterações aninhadas. Assim como no modelo de custos deste algoritmo, uma fase de replicação da relação *R* e uma fase de junção local também aparecem na sua implementação. Além destas, existe uma outra fase que faz a distribuição inicial de dados entre os processadores. Esta fase não diz respeito especificamente ao algoritmo, ela apenas se encarrega de simular um ambiente real existente antes da execução de uma junção.

A fase de distribuição inicial de dados é a primeira a ser executada pela implementação, como representado na figura 5. Esta fase tem início assim que se executa o processo dito coordenador. Automaticamente, o processador utilizado para executar este processo passa a ter um status diferente dos outros: é este processador que será o responsável por coordenar as atividades de todos os outros envolvidos na junção. O processo coordenador instancia um processo leitor e um processo junção - que terão suas finalidades descritas adiante - em cada

um dos outros processadores. Uma vez criados estes processos, o coordenador inicia a distribuição de tuplas, enviando a cada processo leitor uma partição distinta de ambas as relações. Ao final desta fase, as relações *R* e *S* estarão particionadas entre todos os processadores, simulando o ambiente de um sistema de banco de dados paralelo desejado.

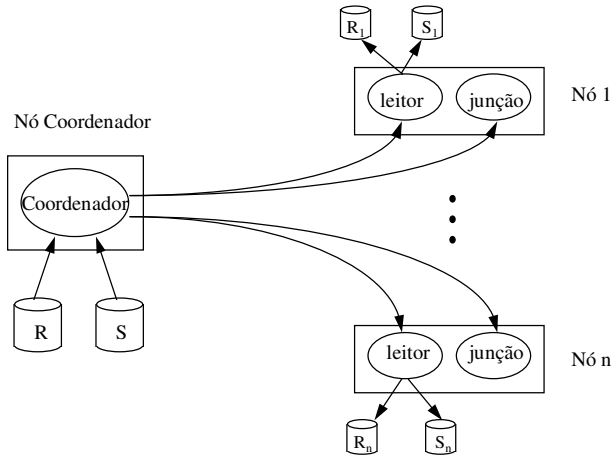


Figura 5 - O processo coordenador instancia os outros processos e distribui os dados

É na segunda fase, ilustrada na figura 6, que a execução paralela da junção propriamente dita começa. Esta fase se inicia quando todos os processos leitor sinalizam ao processo coordenador o recebimento de suas respectivas partições de *R* e de *S*. A partir daí, cada processo leitor envia uma cópia da sua partição de *R* para os processos junção residentes em outros processadores que não o seu.

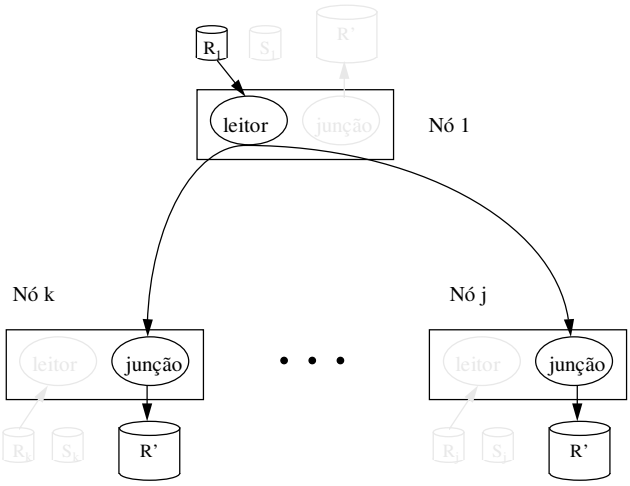


Figura 6 - Cada processo leitor envia sua partição de R para os outros processos junção, que criam réplicas de R em cada nó

Cada processo junção vai recebendo as tuplas enviadas pelos processo leitor dos outros processadores, até que uma réplica da relação *R* esteja totalmente disponível de forma local. Uma vez que todos os processos junção terminem esta tarefa, cada processador terá, além da partição de *S* que inicialmente lhe coube armazenar, uma réplica completa da relação *R*.

A terceira fase é a fase de processamento local, representada na figura 7. Ela tem início quando todos os processos junção sinalizam o recebimento da suas réplicas completas de *R* ao processo coordenador. Nesta fase, cada processo junção executa o algoritmo seqüencial de iterações aninhadas por blocos para fazer a junção da sua partição de *S* com a réplica de *R* recém-adquirida. Quando a condição de junção é satisfeita para um par <*r,s*>, os atributos das duas tuplas são combinados em uma só tupla resultante. Esta tupla é imediatamente enviada ao processo coordenador, que é o responsável por armazenar a relação resultante.

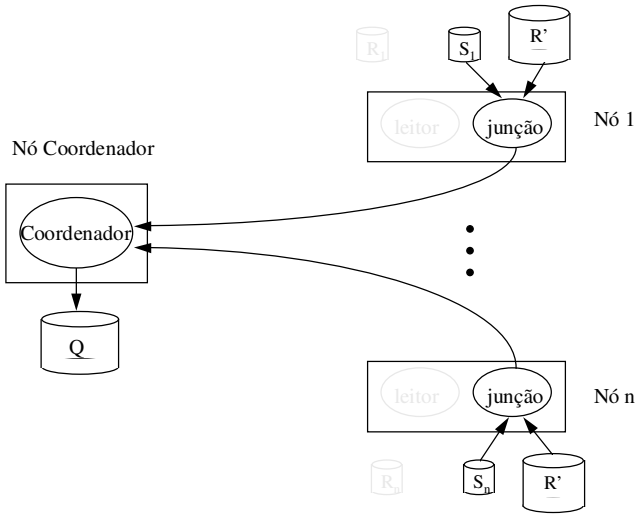


Figura 7 -Na fase de junção local, cada nó faz a junção de sua réplica de R com a partição de S

Decidiu-se não implementar o algoritmo da junção paralela por *hashing* sobre o PVM devido aos resultados obtidos na implementação da junção paralela por iterações aninhadas, discutidos mais adiante.

4.3 Descrição da Implementação sobre MPI

A avaliação da implementação do algoritmo de iterações aninhadas em PVM descrita na seção anterior sugere uma série de melhorias a serem feitas para que fossem alcançados resultados mais próximos dos previstos pelo modelo analítico. Pela comparação a ser vista na seção 4.4, a interface MPI forneceria subsídios para resolver alguns problemas da implementação anterior, notadamente os de velocidade e de interferência característicos de redes com enlace de dados *Ethernet*. A utilização do *switch* em substituição à *Ethernet* como rede de intercomunicação entre os processadores pareceu a principal motivação para uma nova implementação dos algoritmos utilizando a interface MPI.

Os recursos de comunicação envolvendo grupos de processos (comunicação coletiva) da

interface MPI possibilitaram encapsular as operações elementares de movimentação de tuplas entre processadores, eliminando (a) a necessidade de ter dois processos em cada participante da operação e (b) a necessidade de um processo coordenador dedicado.

4.3.1 Operações de Movimentação de Tuplas

As operações de movimentação de tuplas foram utilizadas como "blocos básicos" para a implementação dos algoritmos avaliados no modelo analítico. Foram criadas quatro operações: a fragmentação de uma relação de um processador para vários, a desfragmentação de uma relação fragmentada entre vários processadores para apenas um, a desfragmentação de uma relação fragmentada entre vários processadores para todos estes e o *hashing* de uma relação de vários processadores para vários outros.

Na operação de fragmentação representada na figura 8(a), uma relação armazenada inteiramente por um processador é fragmentada em partições de tamanho equivalente. Os fragmentos são enviados para todos os processadores participantes e ali são armazenados como novas relações. É interessante ressaltar que o processador que distribui a relação também recebe um fragmento, acumulando os papéis de raiz e de participante da operação.

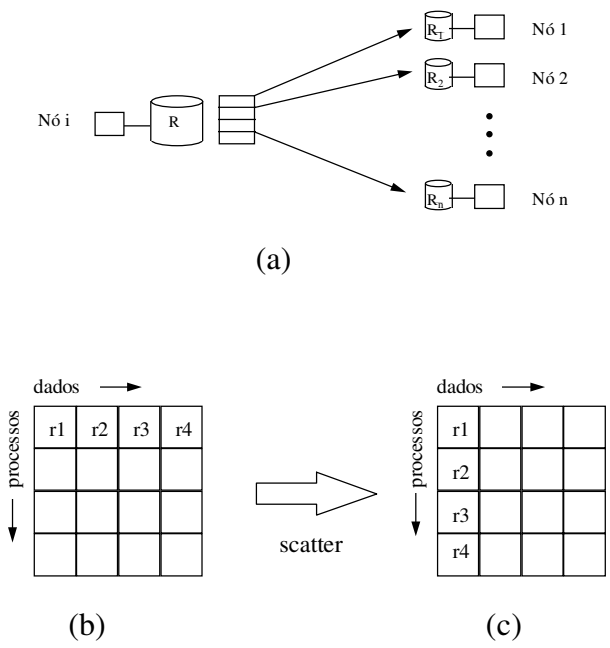


Figura 8 - (a) operação de fragmentação de uma relação, distribuição de páginas (b) antes e (c) depois do *scatter.*

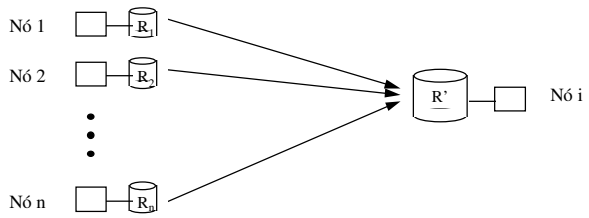
A função de comunicação coletiva que suporta a fragmentação de relações na interface MPI é a *scatter*, ilustrada na figura 8(b) e 8(c). Na figura 8(b), a distribuição de páginas da relação antes de um *scatter* é ilustrada. Cada linha de quadrados representa áreas de dados de um processador. No caso, o primeiro processador tem em memória quatro páginas da relação a ser distribuída. Após a comunicação coletiva cada página foi transferida para um processador

participante distinto, inclusive aquele que inicialmente as continha. A nova situação é representada na figura 8(c).

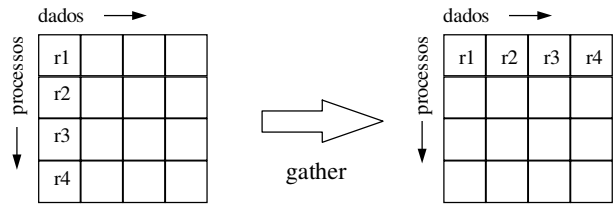
A unidade básica de comunicação entre processadores definida foi de uma página. Sendo assim, o processador raiz deve poder armazenar p páginas de uma relação em memória para cada chamada à uma função de comunicação coletiva onde haja comunicação de p para 1 processador (ou vice-versa).

A função de *scatter* é repetida $\lceil |R|/p \rceil$ vezes para que toda a relação seja fragmentada. Desta forma, a cada iteração são distribuídas p páginas, uma para cada participante. Pode ocorrer que na última iteração, não haja mais páginas da relação a serem distribuídas para todos os participantes. Neste caso, páginas vazias são usadas para poder completar a operação.

A operação de desfragmentação de uma relação é exatamente inversa à operação recém descrita. A figura 9(a) ilustra esta operação. Nela, uma relação inicialmente fragmentada entre um conjunto de processadores é "unificada" como uma nova relação em apenas um dos participantes. Este participante em especial acumula também o papel de raiz da operação.



(a)



(b) (c)

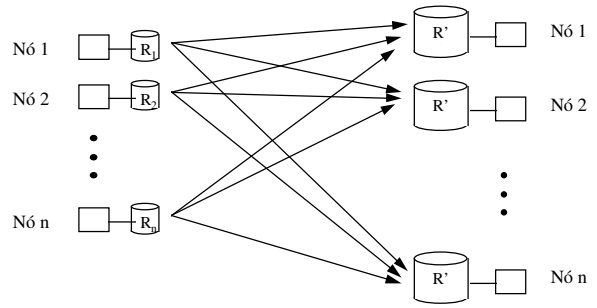
Figura 9 - (a) operação de desfragmentação de uma relação distribuição de páginas (b) antes e (c) depois do *gather*.

A função de comunicação coletiva que suporta a desfragmentação de relações na interface MPI é a *gather*, ilustrada na figura 9(b) e 9(c). Na figura 9(b), a distribuição de páginas da relação antes de um *gather* é ilustrada: cada processador participante tem uma página em memória a ser transmitida. Após a comunicação coletiva, cada página foi transferida para o processador raiz, inclusive aquela que este inicialmente continha. A nova situação é representada na figura 9(c).

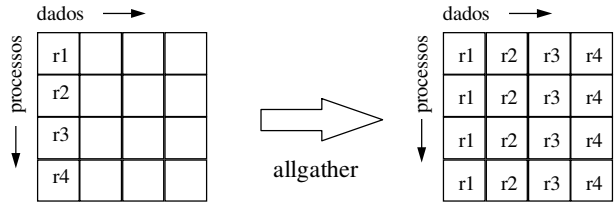
O raiz da operação recebe, a cada passo, *p* páginas da relação, cada qual vinda de um participante diferente. Para isto, ele aloca *p* páginas de memória para efetuar a operação. Não há necessidade dos fragmentos terem o mesmo número de páginas. Porém, a operação completa leva tantas iterações quantas forem o número de páginas do maior fragmento. No caso de um participante já ter enviado todas as suas páginas nas primeiras iterações, ele enviará páginas vazias nas iterações restantes.

A desfragmentação da relação em todos os participantes é uma variação da operação de desfragmentação simples. A diferença reside apenas no fato de não haver um raiz na operação, isto é, todos os participantes assumem o papel de raiz. Assim, uma relação originalmente particionada em *p* fragmentos diferentes, cada qual armazenada por um processador, pode ser replicada por inteiro em cada um destes. Este processo é ilustrado na figura 10(a).

A função de comunicação coletiva que suporta a desfragmentação de relações em todos os participantes é a *allgather*, ilustrada na figura 10(b) e 10(c). Na figura 10(b), a distribuição de páginas da relação antes de um *allgather* é ilustrada: cada processador participante tem uma página em memória a ser transmitida. Após a comunicação coletiva, cada página foi simultaneamente transmitida para todos os processadores envolvidos na operação. A nova situação é representada na figura 10(c).



(a)



(b) (c)

Figura 10 -(a) operação de desfragmentação de uma relação
em todos os processadores e distribuição de páginas (b) antes e (c) depois do *allgather*

Uma vez que, a cada passo, um processador recebe *p* páginas distintas, esta mesma quantidade de memória deve ser alocada por cada processador. Não há necessidade dos fragmentos terem

o mesmo tamanho. Porém, também aqui a operação completa leva tantas iterações quantas forem o número de páginas do maior fragmento. No caso de um participante já ter enviado todas as suas páginas nas primeiras iterações, ele enviará páginas vazias nas restantes.

A operação de *hashing* de uma relação, representada na figura 11(a), difere das três outras descritas até aqui. Nesta operação, uma relação esta inicialmente fragmentada entre um conjunto de processadores. Cada um destes fragmentos passa pela mesma função de *hashing* e dá origem a *p* buckets. Cada bucket *i* gerado num processador é transmitido para *i*-ésimo processador. Desta forma, ao final da operação, cada processador armazenará como uma nova relação os buckets de mesma chave de *hashing* gerados no processo.

A função de comunicação coletiva que suporta o *hashing* de relações é a *alltoall*, ilustrada na figura 11(b) e 11(c). Na figura 11(b), a distribuição de páginas da relação antes de um *alltoall*: cada processador participante tem uma página representando parte de um bucket obtido a partir do seu fragmento. Após a comunicação coletiva, as páginas que compõem um mesmo *bucket* estarão armazenadas num mesmo processador. A nova situação é representada na figura 11(c).

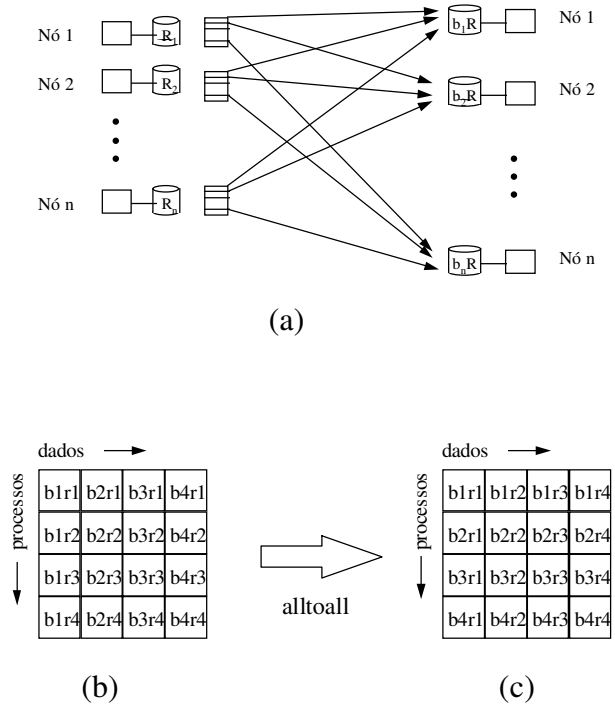


Figura 11 -(a) operação de hashing de uma relação
em todos os processadores e distribuição de páginas (b) antes e (c) depois do *alltoall*.

Também aqui não há necessidade dos fragmentos terem o mesmo número de páginas. Porém, a operação completa leva tantas iterações quantas forem o número de páginas do maior fragmento. No caso de um participante já ter enviado todas as suas páginas nas primeiras iterações, ele envia páginas vazias nas restantes.

Cada participante aloca 2*p* páginas em memória para efetuar a operação: um conjunto de *p* páginas para o envio dos *buckets* e um segundo conjunto para o recebimento. A cada página lida, o *hashing* é aplicado sobre um atributo definido de cada uma das tuplas. A tupla é então movida para a página correspondente àquele valor de *hash*. Ao final deste processo, as *p* páginas são transmitidas, cada qual para um processador, respeitando-se uma ordem definida. Uma página é transmitida mesmo que não contenha tuplas.

4.3.2 Junção Paralela por Iterações Aninhadas

A criação das operações de movimentação de tuplas tornou bastante simples a implementação da junção por iterações aninhadas em paralelo: a implementação se resume a um conjunto destas operações e a um algoritmo de iterações aninhadas seqüencial.

A fase de distribuição inicial dos dados também aparece nesta implementação. O seu único objetivo continua sendo o de simular um ambiente real existente antes da execução de uma junção em um sistema de banco de dados paralelo. Sendo assim, o tempo gasto na sua execução não é computado no tempo total do algoritmo. A distribuição inicial é feita através da operação de fragmentação de relações por vários processadores aplicada à *R* e *S*.

Na fase seguinte, a operação de desfragmentação de uma relação em todos os processadores é utilizada para que réplicas de *R* sejam criadas em todos os processadores a partir do conjunto de fragmentos desta relação. Ao final desta fase, cada processador armazena o seu fragmento inicial de *S* distinto e uma réplica completa da relação *R*.

A terceira fase é onde ocorre a junção local. O algoritmo de iterações aninhadas sequencial é utilizado em cada processador para executar a junção entre um fragmento de *S* e um réplica de *R*. As tuplas resultantes da junção de cada processador são armazenadas localmente sob a forma de uma nova relação. Na última fase, os fragmentos da relação resultante gerados em cada processador são desfragmentados em um único deles, eleito como responsável por tratar desta relação. A operação de desfragmentação de uma relação em um único processador é utilizada para cumprir esta última fase do algoritmo.

4.3.3 Junção Paralela por *Hashing*

Da mesma forma que o algoritmo acima, a implementação da junção por *hashing* paralela foi bastante facilitada pelas operações de movimentação de tuplas. A fase inicial da implementação deste algoritmo é a mesma que a descrita na Seção 4.3.2.

Na segunda fase, as tuplas são redistribuídas entre os processadores de forma que cada um deles armazene apenas tuplas que tenham o mesmo valor de *hash* no seu atributo de junção. Isto é, deseja-se que cada processador, ao final desta fase, armazene um *bucket* de *R* e um de *S* com tuplas de mesmo valor de *hash*. A operação de *hashing* de uma relação de vários para vários processadores atende à esta necessidade perfeitamente. Ela é aplicada primeiramente sobre a relação *R* e depois sobre *S*.

A terceira fase é a fase de junção local, similar à descrita na Seção 4.3.2. Ela realiza a junção por iterações aninhadas entre o *bucket* de *R* e o de *S* armazenados em cada um dos processadores. As tuplas resultantes da junção de cada processador são armazenadas localmente sob a forma de uma nova relação.

Na última fase, os fragmentos da relação resultante gerados em cada processador são

desfragmentados num único processador, eleito como responsável por tratar esta relação. A operação de desfragmentação de uma relação em um único processador é utilizada para cumprir esta última fase do algoritmo.

4.4 Comentários e Comparação entre as Implementações sobre PVM e MPI

Uma característica marcante da implementação PVM é a manutenção de um sincronismo rígido entre os processadores: uma nova fase nunca é iniciada sem que todos os processadores tenham sinalizado o fim da fase atual. Isto significa que todos os processadores iniciam uma dada fase juntos e só passam para uma fase seguinte quando todos terminam. Um sincronismo menos rígido provavelmente levaria à medidas mais favoráveis de tempo. Exceto pela facilidade de depuração do algoritmo, não há prejuízo algum se cada processador avançar na próxima fase independente do ponto de processamento que os outros estejam.

A idéia de sincronismo entre as fases foi abandonada na implementação MPI. Nesta implementação, não existe a fronteira rígida entre as fases como no caso do PVM. Se a situação assim o permitir, um processador continuará executando sua tarefa, independente do ponto onde os outros estejam. Ocorrem, entretanto, sincronismos implícitos forçados pelo MPI devido à algumas chamadas a rotinas de comunicação, porém a implementação não faz qualquer operação presumindo este possível sincronismo.

Existe em ambas as implementações a idéia de operações elementares de movimentação de tuplas. De fato, os algoritmos implementados podem ser vistos como uma combinação destas operações com algoritmos de junção seqüenciais. O que difere um algoritmo do outro seria as operações de movimentação utilizadas. Este conceito está bem melhor encapsulado no MPI do que no PVM. A razão para tanto é a facilidade de programação que cada uma das ferramentas oferece. No MPI existem primitivas de comunicação de alto nível, ao passo que no PVM toda a codificação de uma operação como esta deve ser feita pelo programador.

Este fato também fez com que fosse mais simples, em termos de programação, ter um processador com o status de coordenador na implementação PVM. O MPI permitiu que os processadores trabalhassem de forma mais cooperativa sem a necessidade de um coordenador. Em algumas operações, onde o resultado é armazenado ou vem de apenas um local, um dos participantes é eleito temporariamente como o processador raiz.

Do ponto de vista de performance, o MPI possibilitou melhores medidas de tempo pois foi somente através desta ferramenta que pudemos utilizar o *switch* do sistema SP2. Os programas construídos sobre o PVM utilizaram a rede *Ethernet* para a interconexão entre os processadores, trazendo diferenças de resultados interessantes.

Outro aspecto em relação à comunicação que diferencia as duas implementações é a unidade de comunicação entre processadores. Na implementação sobre PVM, os processadores trocam tuplas entre si. Na implementação MPI, a comunicação é feita página a página. Isto nos pareceu mais razoável, dado ao custo fixo de transmissão em rede, relativamente alto para mensagens de tamanho reduzido, e bastante adequado ao contexto de sistemas de bancos de dados.

Devido à esta característica e à ausência de coordenador, os algoritmos implementados sobre o MPI têm uma fase a mais que os implementados sobre PVM. Ao final, é realizada um operação conjunta para a desfragmentação da relação resultante. Nas implementações sobre PVM, a cada par <r,s> de tupla que atenda à condição de junção, uma tupla resultante é enviada ao

coordenador.

5. Resultados Obtidos

Os algoritmos implementados foram executados sob o ambiente descrito inicialmente com a finalidade de avaliar a variação nos tempos de resposta em decorrência do aumento no número de processadores. Algumas outras execuções também foram feitas para estudar mais especificamente o comportamento do algoritmo quando submetido à concorrência externa e a sensibilidade em relação ao tipo de rede de interconexão entre processadores.

5.1 Geração das Relações

As relações *R* e *S* utilizadas nestas medidas continham 20.000 (vinte mil) e 200.000 (duzentas mil) tuplas respectivamente. Foram utilizados dois pares de relações *R* e *S* diferentes em cada uma das medidas de tempo efetuadas. As tuplas de ambas as relações continham 50 bytes, divididos em um atributo de 5 bytes numérico e um segundo atributo alfanumérico composto de 45 bytes.

As relações foram geradas automaticamente tomando algumas precauções no que diz respeito à distribuição de valores do atributo numérico, uma vez que este seria utilizado como atributo da junção. Esta distribuição visava a geração uniforme de valores repetidos, sem os quais não haveria resultado na junção. Com isto puderam ser obtidos pares de relações *R* e *S* distintos, porém com fatores de seletividade e taxa de repetição de valores bastante aproximados.

5.2 Curvas de Aceleração dos Algoritmos

Os tempos de execução foram medidos assim como percebidos por um usuário do suposto "SGBD paralelo". Neste tempo estão computados gastos com acessos a disco, acessos a rede e trocas de contexto de processos, entre outros, pois assim entendemos ser o tempo total de resposta gasto pela operação.

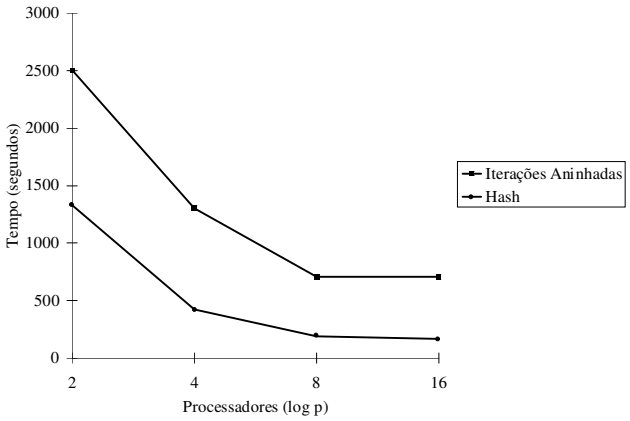


Figura 12 - Curvas de aceleração obtidas na implementação dos algoritmos paralelos por iterações aninhadas e por *hashing* em MPI

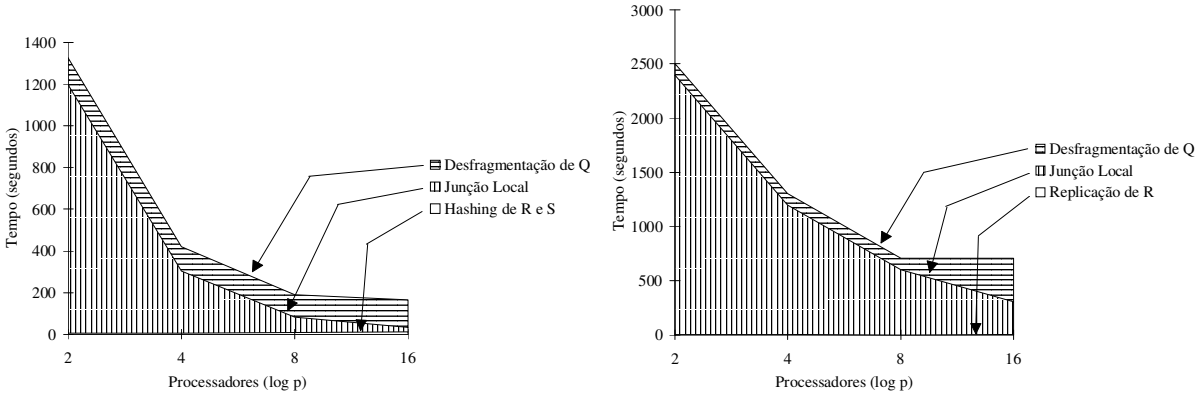
Os algoritmos estudados são particularmente sensíveis à quantidade de memória disponível para a sua execução. Sendo assim, a disponibilidade de memória foi artificialmente reduzida de forma a estudar o comportamento da implementação sob a condição restritiva de que o tamanho das relações sempre é maior do que a quantidade disponível de memória para armazená-las. A quantidade de memória utilizada nesta simulação foi fixada em 10% da quantidade necessária para armazenar a menor das relações.

O gráfico dos tempos obtidos para o processador mais lento utilizando as massas de testes é representado na figura 12. As mesmas massas de dados foram utilizadas em configurações de 2, 4, 8 e 16 processadores.

O gráfico apresenta as curvas de aceleração para a implementação dos algoritmos paralelos de junção por iterações aninhadas (a curva superior) e por *hashing* (a curva inferior) sobre o MPI. Os resultados previstos pelo modelo analítico foram observados de acordo com a exposição de idéias que se segue. A implementação da versão PVM do algoritmo paralelo por iterações aninhadas gerou curvas de aceleração contrárias às expectativas do modelo analítico. Estes resultados são apresentados e comentados na Seção 5.4.

5.3 Comentários sobre as Curvas de Aceleração

Para o algoritmo paralelo por iterações aninhadas, o tempo de resposta decai mais lentamente do que o encontrado no algoritmo por *hashing*, de acordo com o esperado. Através da análise mais detalhada dos custos de cada fase, ilustrados nas figuras 13(a) e 13(b), podemos analisar os fatores que mais contribuíram para estes resultados.



(a) (b)

Figura 13 - Contribuição de cada fase para o tempo total nos algoritmos (a) por *hashing* e (b) por iterações aninhadas

Os custos analíticos sugerem que, conforme comentado na Seção 3.4.2, quando $||S|| >> ||R||$ a replicação de R é menos custosa que a redistribuição de R e S. Este fator pode ser verificado através da figura 13, onde em 13(a) o custo de *hashing* aparece sutilmente e o custo de replicação em 13(b) é praticamente nulo.

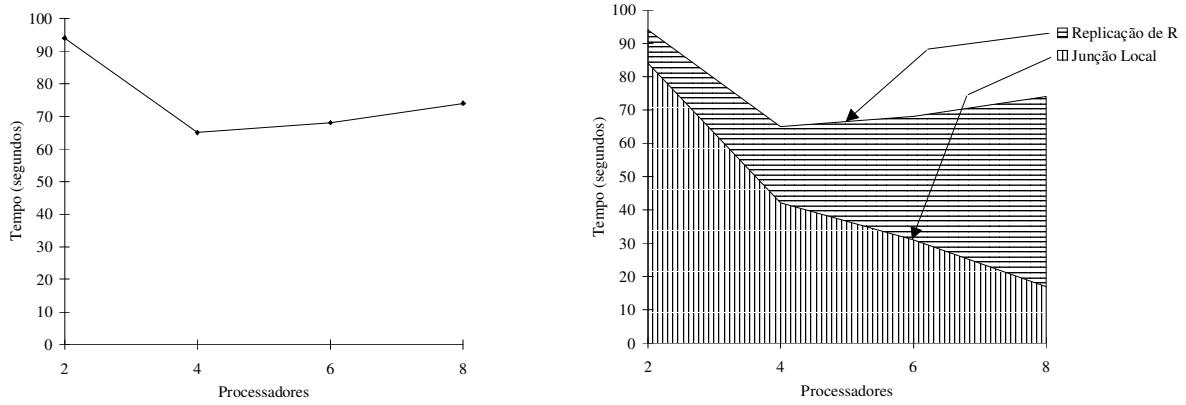
A fase de comunicação se mostrou particularmente sensível ao tipo de rede de interconexão

entre processadores, como será visto na seção 5.4. Os tempos de resposta sinalizaram que o *switch* atendeu à demanda exigida pela troca de informações entre os processadores, apresentando um baixo tempo de latência e alta velocidade de transmissão. Graças a isto, os tempos das fases onde há movimentação de tuplas não contribuíram de forma significativa para o tempo total de execução dos algoritmos.

No que diz respeito à fase de junção local, a adição de processadores trouxe melhorias nos tempos de resposta de ambos os algoritmos. Os tempos totais sempre estiveram fortemente dominados pelo tempo desta fase e, conforme esperado, o algoritmo paralelo por *hashing* apresenta queda mais significativa nos tempos de resposta do que o de iterações aninhadas em paralelo. Além disso, o fato da versão por *hashing* exigir menos comparações entre tuplas, faz com que haja diferença sensível nos tempos de respostas de ambos os algoritmos, seja qual for o número de processadores empregados.

5.4 Curvas Anômalas

Os algoritmos estudados se mostraram sensíveis ao tipo de rede de interconexão entre processadores utilizada na sua execução. A versão do algoritmo paralelo por iterações aninhadas sobre PVM gerou resultados que mereceram especial atenção. Particularmente, esta implementação não teve acesso ao *switch* do SP2, utilizando uma rede *Ethernet* para interligar os processadores.



(a) (b)

Figura 14 - (a) Curva de aceleração obtida na implementação do algoritmo paralelo por iterações aninhadas sobre PVM e (b) a contribuição de cada fase

Na figura 14 são apresentados os gráficos obtidos na tomada de tempo feita para esta implementação. Para este caso em específico foram consideradas relações *R* e *S* com 2.500 (duas mil e quinhentas) e 50.000 (cinqüenta mil) tuplas, respectivamente. Cada tupla continha um total de 10 (dez) *bytes*, sendo um atributo numérico de 4 (quatro) *bytes* e um atributo caracter de 6 *bytes*, gerados conforme descrição anterior.

5.5 Comentários sobre as Curvas Anômalas

Quando utilizando redes tipo *Ethernet*, as fases que envolvem movimentação de tuplas passam a ter o domínio sobre o tempo total do algoritmo. No caso, como representa a figura 14(b), a fase de replicação de R no algoritmo por iterações aninhadas teve forte influência no resultado final obtido. Alguns tempos foram tomados nessas condições para o algoritmo de iterações aninhadas e os resultados obtidos foram todos similares ao exemplo das figuras 14(a) e 14(b). Em tese, não há justificativas para que a fase de replicação apresente aumento de tempo de resposta em função do número de processadores. Isto remete a uma busca das razões que levam ao fenômeno observado.

Quando executada em um número pequeno de processadores, a fase de replicação implica em uma troca de partições significativas da relação *R* entre os processadores. No caso oposto, muitos processadores enviam partições menores a todos os seus pares. Esta última característica requer uma maior capacidade de compartilhamento do meio físico de transmissão, uma vez que num dado instante mais processadores precisam se comunicar. As redes de interconexão de processadores baseadas em *Ethernet* não permitem compartilhamento simultâneo do meio físico, sugerindo um alto tempo de latência devido à contenção na rede. Pelos resultados obtidos e pelas especificações nominais, o *switch* não apresenta este problema, sendo elemento chave para a execução deste tipo de algoritmo.

5.6 Desequilíbrio de Carga

Outro comportamento que se desejava conhecer era a reação dos algoritmos à concorrência externa pelos processadores a eles alocados. O ambiente utilizado no LNCC se mostrou propício também para este tipo de tomada de tempo, pois esta pode ser feita em horários onde a execução de programas parelelos se dava de forma não exclusiva no sistema SP2, quando havia um ambiente real de concorrência pelos processadores.

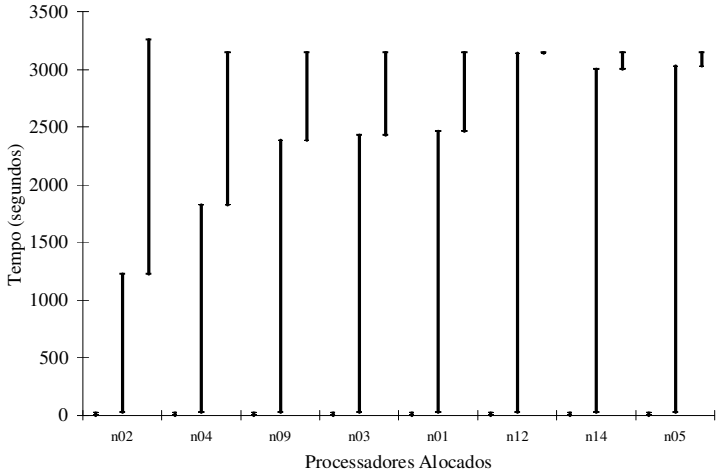


Figura 15 - Execução desbalanceada

Os resultados obtidos são apresentados no gráfico da figura 15. Este gráfico ilustra a cronologia de execução das tarefas para o algoritmo por iterações aninhadas, implementado

sobre o MPI, envolvendo oito processadores num ambiente com concorrência. Os nomes dos processadores alocados para cada execução estão representados na escala do eixo horizontal do gráfico. Para cada processador estão dispostas três barras: a primeira (a mais a esquerda) corresponde à fase de replicação da relação *R*, a segunda (a do centro) à fase de junção local a cada processador e a última (a mais a direita) à fase de desfragmentação da relação resultante.

Este resultado foi cuidadosamente checado para que tivéssemos a certeza de que a origem do desbalanceamento não fosse a distribuição inicial de tuplas ou o fator de seletividade em cada um dos processadores. Tanto os tamanhos das relações temporárias utilizadas no processo de junção quanto os fatores de seletividade destas eram bastante semelhantes de processador para processador.

5.7 Comentários sobre o Desequilíbrio de Carga

A fase de junção foi a fase que se apresentou mais sensível à carga dos processadores envolvidos na operação. No caso da execução da figura 15, a replicação da relação *R* é feita coincidentemente em sincronismo por todos os participantes. O desequilíbrio surge na fase de junção, quando os processadores supostamente mais rápidos (*n02* e *n04*) terminam esta tarefa e iniciam o processo de desfragmentação da relação resultante. Por envolver a participação conjunta de todos, estes processadores são forçados a esperar pelos mais lentos para prosseguir na execução de tarefas.

A figura 15 sugere que os processadores mais rápidos foram prejudicados pelo desempenho dos mais lentos (*n12*, *n14* e *n05*), os quais levaram muito tempo na fase de junção local. Assim que estes processadores terminam a junção, a operação coletiva de desfragmentar a relação resultante é finalizada em pouco tempo, de acordo com a barra que representa a última fase nestes processadores.

O fato da influência que a concorrência exerce ser externa e aleatória torna ineficiente a adoção de estratégias preventivas ao desbalanceamento de carga entre os processadores. Mesmo se, em princípio, pudéssemos avaliar a carga em cada processador, dificilmente poderíamos fazer uma distribuição de trabalho de forma que todos levassem o mesmo tempo. Isto porque, em situações reais, até mesmo esta carga de concorrência irá variar de forma não determinística ao longo do tempo.

6. Equilíbrio Dinâmico de Carga

Como visto até aqui, o tempo de resposta de um algoritmo paralelo é dado pelo tempo do processador que por último encerra a sua tarefa. Portanto, minimizar este tempo de resposta é uma questão associada à distribuição adequada das tarefas entre os processadores, visando o balanceamento de carga e o aproveitamento máximo dos recursos computacionais disponíveis. Entretanto, não basta apenas definir uma distribuição equilibrada de trabalho entre os processadores para que todos terminem as suas tarefas em tempos aproximados pois, em um ambiente paralelo multi-usuário, o nível de concorrência dos diversos processadores não é necessariamente o mesmo.

Suponhamos uma arquitetura com *n* processadores e o problema de calcular a junção entre as relações *R* e *S*, replicadas em todos os nós. Podemos definir *n* partições de *S*, de cardinalidades semelhantes, de forma que cada processador tenha que calcular a junção entre *R* e uma dessas

partições. Devemos portanto esperar que, em um ambiente onde somente aquela aplicação esteja sendo executada, todos os processadores terminem as suas tarefas em tempos aproximados. Porém, na prática, os diferentes níveis de concorrência entre os processadores levam a um desbalanceamento entre os términos de execução destes processadores, prejudicando o tempo total paralelo. Em uma situação hipotética, onde os processadores pares têm o dobro de concorrência dos processadores ímpares, os pares levariam aproximadamente o dobro do tempo que levariam os ímpares. Seria interessante então que os processadores ímpares recebessem mais ou maiores tarefas para que o tempo paralelo seja reduzido e a capacidade dos processadores seja adequadamente aproveitada. Este problema pode ser caracterizado também por uma máquina paralela heterogênea, formada por processadores com diferentes capacidades de processamento.

Desta forma, concluímos que a solução para o problema do balanceamento de carga não se resume à distribuição equilibrada de trabalho pelos processadores. Entendemos que o problema em questão equivale à tentativa de distribuir as tarefas de forma que o tempo de execução de cada processador seja o menor possível e, consequentemente, equilibrado em relação ao tempo dos demais.

6.1 Estratégia para Equilíbrio Dinâmico de Carga

Os algoritmos paralelos de junção analisados neste trabalho, como visto na seção anterior, podem apresentar problemas de balanceamento de carga. Apresentamos, a seguir, duas propostas alternativas para o cálculo da junção baseadas nas idéias apresentadas em [PL95]. Nesta abordagem, não existe uma divisão prévia - estática - das tarefas entre os processadores, evitando assim términos discrepantes dos diversos processadores devido à diferente capacidade de processamento. A solução baseia-se na divisão da tarefa principal em um número grande - maior que o número de processadores - de tarefas menores. Um processador coordenador distribui, em uma primeira fase, uma tarefa para cada processador. Em uma segunda fase, até que não haja mais tarefas, cada processador recebe uma nova tarefa após encerrar a tarefa corrente.

Deste modo é feita uma distribuição dinâmica de tarefas em função do comportamento dos processadores. Os que apresentam um nível maior de concorrência executarão menos tarefas e, por outro lado, os menos carregados compensarão o trabalho executando mais tarefas de forma que os tempos finais se apresentem equilibrados. Observe que, em um ambiente heterogêneo, um determinado processador, excessivamente concorrido, pode receber apenas a tarefa inicial e nenhuma na fase seguinte, enquanto outros processadores, menos concorridos, receberão consequentemente mais tarefas.

A estratégia em questão pode é representada pelo algoritmo 6.1, onde t é o número de tarefas e p o número de processadores:

Para que tal abordagem possa ser adotada, o problema tem que permitir uma quebra da tarefa principal em tarefas menores e o número destas sub-tarefas deve ser maior que o número de processadores envolvidos. Este é o caso do cálculo do fecho transitivo [PL95] e, como veremos mais adiante, o cálculo da junção também permite esta divisão.

Dependendo do problema a ser resolvido, o coordenador pode, além de monitorar a distribuição dinâmica de tarefas, também executar tarefas. Neste caso, deve ser dada prioridade a função de monitoramento para que os processadores não fiquem bloqueados entre o fim de

uma tarefa e o início de outra.

```
COORDENADOR
FASE I:
Envia uma tarefa para cada processador;

FASE II:
Enquanto t > 0 faça:
    Espera o sinal de fim de tarefa de algum processador;
    Envia uma nova tarefa para o processador que sinalizou;
    t ← t – 1;
Fim-Enquanto;
Espera o sinal de fim de tarefa de todos os processadores ainda
trabalhando;
Envia a todos os processadores o sinal de término do algoritmo;


OUTROS PROCESSADORES
FASE I:
Recebe a tarefa inicial;
Executa a tarefa inicial;
Envia ao Coordenador o sinal de fim da tarefa inicial;

FASE II:
Enquanto o Coordenador não indicar fim do algoritmo faça:
    Espera uma nova tarefa;
    Executa a nova tarefa;
    Envia ao Coordenador o sinal de fim de tarefa;
Fim-Enquanto;
```

Algoritmo 6.1

6.2 Aplicação da Estratégia aos Algoritmos Estudados

Apresentaremos a seguir as adaptações dos algoritmos paralelos considerados neste trabalho, baseadas nestas idéias de balanceamento de carga .

6.2.1 Junção Paralela por Iterações Aninhadas com Balanceamento Dinâmico de Carga

Como descrito anteriormente, a junção das relações R e S por iterações aninhadas, em um ambiente com p processadores, é calculada através de p junções entre R (a menor relação) e o i-ésimo fragmento de S, representado por S_i, $1 \leq i \leq p$. Cada partição S_i contém aproximadamente $\|S\|/p$ tuplas. A adaptação das idéias descritas anteriormente requer que o problema possa ser dividido em mais e menores tarefas para serem distribuídas dinamicamente entre os processadores, de acordo com o desempenho de cada um durante o processamento. A principal questão é definir as referidas tarefas. Os algoritmos que devem ser executados pelo coordenador e pelos outros processadores são os apresentados anteriormente.

Cada tarefa será definida como o cálculo da junção entre R e uma partição de S. Esta divisão de S deve ser feita de forma a gerar um número de tarefas maior do que o número de processadores. Suponhamos dispor de p processadores e um número t de tarefas, $t > p$. Neste caso, cada tarefa consiste em calcular a junção entre R e a i-ésima partição de S, $1 \leq i \leq t$, onde a cardinalidade de cada S_i seria aproximadamente $\|S\|/t$. A escolha do número de tarefas t, de

acordo com o que será discutido mais adiante, deve ser feita em função de alguns parâmetros, como o desequilíbrio de concorrência entre os processadores e o custo fixo da tarefa definida. Cabe observar que esta estratégia equivale ao algoritmo paralelo de iterações aninhadas convencional, quando $t = p$.

Podemos então caracterizar cada tarefa como um conjunto de tuplas de S, representado pelo par (tp,n), onde tp é a tupla inicial daquela partição de S, em relação à ordem de armazenamento, e n é a cardinalidade da partição. Pode-se eventualmente evitar a transferência das tuplas que formam as partições de S ao longo do algoritmo, replicando-se não só a relação R mas também S em todos os nós, de forma que cada processador possa, localmente, determinar a partição de S que compõe a tarefa recebida.

O algoritmo, inicialmente, distribui uma tarefa para cada um dos p processadores. Em seguida, serão distribuídas as t-p tarefas restantes de acordo com o desempenho de cada processador: assim que um processador termina uma tarefa, recebe uma nova até que não haja mais tarefas. Desta forma, o balanceamento de carga é feito dinamicamente em função do comportamento dos processadores em tempo de execução.

6.2.2 Junção Paralela por *Hashing* com Balanceamento Dinâmico de Carga

No caso da junção paralela por *hashing*, o i-ésimo processador, $1 \leq i \leq p$, executa uma junção entre uma partição R_i de R e uma outra S_i de S, onde estas partições são definidas a partir de uma função de *hashing* aplicada ao atributo de junção das relações R e S.

Neste caso, diferentemente do algoritmo anterior, a própria distribuição dos dados pode levar a um desbalanceamento de carga, independente dos diferentes níveis de concorrência existente entre os processadores. Vale então, por mais um motivo, definir as tarefas a serem distribuídas dinamicamente entre os processadores para compensar um possível (e provável) desequilíbrio de carga.

A função de *hashing* deve, então, gerar uma quantidade t de partições de R e S maior que o número de processadores p. Cada tarefa continua definida como o cálculo da junção entre R_i e S_i, para $1 \leq i \leq t$. Podemos observar que , de forma análoga à Seção 6.2.1, esta estratégia equivale ao algoritmo paralelo por *hashing*, quando $t = p$.

A i-ésima tarefa, neste algoritmo, será caracterizada por um valor que será comparado com o resultado da avaliação da função de *hashing* aplicada a R e S, de forma a determinar dinamicamente as partições R_i e S_i, $1 \leq i \leq t$. Novamente, pode-se evitar a transferência das tuplas que formam estas partições ao longo do algoritmo por meio de uma replicação das relações R e S em todos os nós.

A partir da definição das t tarefas, a estratégia é semelhante à descrita na seção anterior, sendo que aqui compensamos não só o desbalanceamento devido aos diferentes níveis de concorrência, mas também o desequilíbrio de carga eventualmente causado pela função de *hashing* adotada.

6.3 Comentários sobre a Estratégia para Equilíbrio Dinâmico de Carga

Uma questão ainda sendo estudada é a determinação do número ideal de tarefas a serem distribuídas. A única restrição existente é que seja maior que o número de processadores. Se os processadores tiverem um nível de concorrência equilibrado, pensamos ser interessante definir

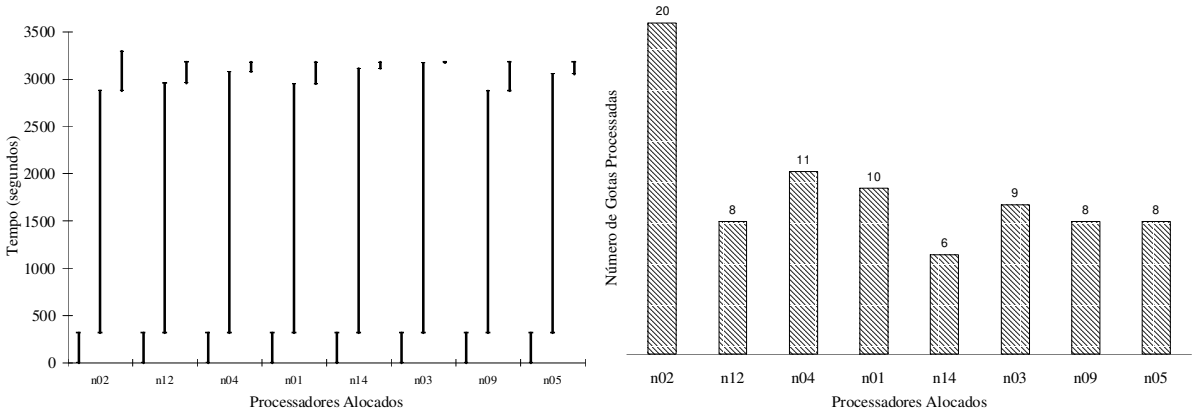
a tarefa inicial da fase I do algoritmo 6.1 maior que as tarefas a serem distribuídas dinamicamente. Desta forma, estas últimas apenas ajustariam algum eventual desbalanceamento. Em contrapartida, níveis heterogêneos de concorrência sugerem tarefas iniciais menores de forma que seja possível identificar e não sobrecarregar os processadores mais concorridos.

Uma outra alternativa a ser estudada seria a variação do tamanho da tarefa ao longo da fase II em função do tempo de resposta dos processadores. Uma alternativa seria fazer com que as últimas tarefas sejam menores para garantir um ajuste fino do balanceamento. Entretanto, dependendo do problema, tarefas muito pequenas têm um custo fixo alto, ou seja, a fração comum e não divisível do algoritmo de todas as tarefas pode se tornar relativamente grande comparada ao tamanho de uma tarefa pequena [PL95]. Neste caso, a repetição desta parte comum um número maior de vezes pode comprometer o desempenho da estratégia.

6.4 Resultados Obtidos

Visando avaliar a estratégia de balanceamento de carga dinâmico proposta, foi feita uma adaptação à implementação do algoritmo de junção paralela por iterações aninhadas sobre MPI, de acordo com o sugerido na Seção 6.2.2. Para que se pudesse ter a noção exata da influência do desequilíbrio de carga entre os diferentes processadores, foram usadas as mesmas relações que as do gráfico da figura 15. Também houve a precaução de executar as duas tomadas de tempo uma em seguida à outra e de utilizar os mesmos processadores em ambas, visando simular um nível relativamente constante de concorrência entre eles.

O resultado obtido é visto através dos gráficos das figura 16(a) e 16(b). Na figura 16(a), é representada a execução utilizando-se a estratégia de equilíbrio dinâmico de carga. Novamente, o eixo horizontal representa cada um dos processadores envolvidos nas tarefas. A cada processador estão representadas três barras simbolizando o tempo de início e de fim de cada uma das fases do algoritmo.



(a) (b)

Figura 16 - (a) Tomada de tempo utilizando a estratégia de equilíbrio dinâmico de carga (b) Número de tarefas alocadas à cada processador durante a execução

Na figura 16(b), um gráfico de barras é utilizado para representar o número de sub-tarefas da

junção que cada processador desempenhou durante a execução do algoritmo. Os tempos representados para a junção de cada processador no gráfico da figura 16(a) são dados pelo somatório gasto por cada uma das sub-tarefas nos diferentes processadores.

Estes gráficos foram gerados para um número de tarefas igual a dez vezes o número de processadores, isto é, o valor de *t* utilizado era igual a 80.

6.5 Comentários sobre os Resultados Obtidos

A diferença no número de sub-tarefas executadas por cada processador demonstra o fato de os mesmos apresentarem disponibilidade de trabalho distintas. Este fato pode tanto ser dado pelo número de tarefas sendo executadas concorrentemente por cada um deles - além do algoritmo em questão - como por heterogeneidade do ambiente de processamento.

Entretanto, esta diferença proporcionou tempos bastante similares na fase de junção local dos processadores envolvidos. Uma rápida análise nos gráficos das figuras 15 e 16(a) mostra que a estratégia de equilíbrio dinâmico de carga surtiu um excelente efeito. No primeiro gráfico, as barras que representam a junção local têm tamanhos diferentes, em oposição a este último gráfico apresentado, onde as barras se apresentam com tamanho mais uniforme. As discrepâncias de tempo ocorridas na fase de desfragmentação da relação-resultado também foram amenizadas.

Os resultados obtidos também nos permitem analisar a eficiência da implementação da estratégia em questão. Pela comparação de tempos dos gráficos das figuras 15 e 16(a) podemos perceber que, se não houveram ganhos de velocidade ao utilizar o balanceamento dinâmico de carga, também não houveram perdas.

Inicialmente, a estratégia aplicada requer uma maior movimentação de tuplas entre os processadores, pois exige, além da replicação da relação *R*, que a relação *S* também seja replicada. Na fase de junção local, ao somarmos o tempo gasto por cada um dos processadores nas diferentes tomadas de tempo, veremos que o custo fixo associado à cada sub-tarefa na estratégia de equilíbrio dinâmico, leva a um tempo de junção local um pouco maior que na primeira abordagem. O ganho significativo está na redução da suposta espera dos processadores para a execução da última fase, constatado pela analise da figura 15. Esta tem seu custo visivelmente menor no gráfico da figura 16(a).

7. Conclusões e Trabalhos Futuros

Este trabalho apresentou um estudo analítico e prático de algoritmos de paralelização da operação de junção. Os resultados práticos obtidos confirmam, na maior parte dos casos, os custos esperados através dos modelos analíticos. Além disso, pode ser estudada a sensibilidade destes algoritmos à fatores como o tipo de redes de interconexão entre processadores sendo utilizadas e a concorrência a qual os processadores que os executam estão expostos numa situação real de processamento.

Quanto ao tipo de redes de interconexão, foram levantadas situações onde o custo de comunicação que os algoritmos impõem podem levar a resultados altamente indesejáveis no que diz respeito à aplicação de paralelismo. Foi visto que dificilmente a classe de algoritmos estudada poderá tirar proveito de redes de interconexão baseadas em *Ethetnet*, ao passo que equipamentos como o *switch* do sistema SP2 da IBM se mostram extremamente propícios à

execução deste tipo de algoritmo.

Quanto ao equilíbrio de carga, constatamos que a definição de uma estratégia para tal não pode ser feita antes da execução dos algoritmos, pois os processadores, em condições reais, estão expostos à cargas aleatórias de trabalho. Sendo assim, aplicamos uma estratégia dinâmica de balanceamento de carga aos algoritmos estudados visando tanto o equilíbrio de carga entre os processadores como a utilização máxima dos recursos disponíveis. Através da aplicação desta estratégia a uma das implementações estudadas, pudemos verificar a sua eficiência em relação às diferenças entre os tempo de execução gastos pelos processadores. Foi também possível constatar que o custo fixo embutido em cada sub-tarefa e a movimentação adicional de tuplas que a utilização da estratégia requer fazem com que não ocorram ganhos no tempo de execução do algoritmo para os níveis de concorrência considerados neste trabalho.

Este estudo tem ainda a possibilidade de se estender ou de se aprofundar em vários aspectos. Para tanto, poderiam ser considerados outros algoritmos para a resolução paralela da operação de junção, como o *hybrid hash*, o *grace* ou o *sort-merge* [ME92].

A estratégia de equilíbrio dinâmico de carga poderia utilizar um número variável de sub-tarefas na resolução da junção. No caso onde os processadores alocados à tarefa não esteja expostos à concorrência, a estratégia poderia identificar este fato e fazer com que o número de tarefas t tendesse ao número de processadores p. No caso oposto, um número variável de tarefas de tamanho dinâmico poderia ser alocado aos processadores de acordo com o seu desempenho no decorrer da execução da operação. Esta idéia conferiria à estratégia um caráter bastante adaptativo às diferentes situações de processamento.

Os resultados obtidos na implementação da estratégia de equilíbrio de carga também sugerem algumas melhorias que poderiam ser feitas para reduzir os custos fixos associados à execução de cada sub-tarefa. Em particular, outras alternativas à replicação de ambas as relações por todos os processadores participantes da operação devem ser estudadas. Por fim, existe também a necessidade de elaboração de um modelo de custos que represente a concorrência e a divisão em sub-tarefas que a estratégia aplicada ao algoritmos aqui discutidos impõem.

Referências Bibliográficas

[AMM+95] T. Agerwala, J.L. Martin, J.H. Mirsa, D.C. Sadler, D.M. Dias e M. Snir , "*SP2 System Architecture*", IBM System Journal, 34(2), pp.152-184, 1995.

[Bra84] K. Bratbergsengen, "*Hashing Methods and Relational Algebra Operations*", Proceedings of the Very Large Data Base Conference, pp. 323-333, 1984.

[DG92] D.J. DeWitt e J. Gray, "*Parallel Database Systems: The Future of High Performance Database Systems*", Communications of the ACM, 35(6), pp. 85-98, 1992.

[DNS+92] D.J. DeWitt, J.F. Naughton, D.A.Schneider e S. Seshadri, *"Practical Skew Handling in Parallel Joins*", Proceedings of the Very Large Data Base Conference, pp 27-39, 1992.

[EN94] R. Elmasri e S.B. Navathe, "*Fundamentals of Database Systems*", The Benjamin/Cummings Publishing Company, 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek e V. Sunderam, "*PVM3 User's Guide and Reference Manual*", Oak Ridge National Laboratory TM-12187, 1994.

[GL96] W. Gropp e E. Lusk, "*User's Guide for mpich, a Portable Implementation of MPI*", Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, 1996.

[HR96] E.P. Harris e K. Ramamohanarao, "*Join Algorithm Costs Revisited*", The VLDB Journal 5, pp 64-84, 1996.

[KGG+94] V. Kumar, A. Grama, A. Gupta, G. Karypis, "*Introduction to Parallel Computing: Design and Analysis of Algorithms*", The Benjamin/Cummings Publishing Company, 1994.

[LOT94] H. Lu, B.C. Ooi e K.L. Tan, editores, "*Query Processing in Parallel Relational Database Systems*", IEEE Computer Society Press, 1994.

[ME92] P. Mishra e M.H. Eich, "*Join Processing in Relational Databases*", ACM Computing Surveys 24, pp 63-113, 1992.

[MPI95] Message-Passing Interface Forum, *"MPI: A Message-Passing Interface Standard*", University of Tennessee, 1995.

[PL95] A. Plastino e S. Lifschitz, "*Uma Nova Estratégia Paralela para o Cálculo do Fecho Transitivo de Relações Binárias*", Anais do II Workshop de Bancos de Dados não Convencionais, pp 33-35, 1995.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu e P. Selinger, *"Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*", Second International Symposium on Databases in Parallel and Distributed Systems, pp. 4-29, 1990.

[RM93] E. Rahm e R. Marek, "*Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems*", Proceedings of the Very Large Data Base Conference, pp 182-193, 1993.

[RM95] E. Rahm e R. Marek, "*Dynamic Multi-Resource Load Balancing in Parallel Database Systems*", Proceedings of the Very Large Data Base Conference, pp 395-406, 1995.

[SSA+95] C.B. Stunkel, D.G. Shea, B. Abali, M.G. Atkins, C.A. Bender, D.G. Grice, P. Hochschild, D.J. Joseph, B.J. Nathanson, R.A. Swetz, R.F. Stucke, M. Tsao e P.R. Varker , "*The SP2 High-Performance Switch*", IBM System Journal, 34(2), pp.185-204, 1995.

[SD89] D.A. Schneider, e D.J. DeWitt, "*A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment*", ACM International Conference, pp. 110-121, 1989.

[Sto86] M. Stonebraker, "*The Case for Shared Nothing*", Database Engineering, 9(1), 1986.