Integrando as Otimizações de Acessos a Dados e de Paralelismo *

Edson T. Midorikawa
Liria Matsumoto Sato
{emidorik, liria}@lsi.usp.br

Laboratório de Sistemas Integráveis
Escola Politécnica da USP
Av. Prof. Luciano Gualberto, travessa 3, nº158
05508-900 - São Paulo - SP - Brasil

RESUMO

Pesquisas anteriores têm adotado o uso de transformações de programa para aproveitar o paralelismo e explorar a localidade de dados. Infelizmente, ambos os objetivos têm sido considerados de maneira independente. Este trabalho apresenta meios para utilizar eficazmente o paralelismo e a hierarquia de memória dos modernos processadores paralelos. Os resultados experimentais utilizando alguns estudos de caso mostram que nossa estratégia resulta em ganhos significativos de desempenho.

ABSTRACT

Previous research has used program transformation to introduce parallelism and to exploit data locality. Unfortunately, these two objectives has usually been considered independently. This work explores the tradeoffs between effectively utilizing parallelism and memory hierarchy on modern parallel processors. The experimental results using some case studies show that our approach improve performance significantly.

* Este trabalho foi parcialmente financiado pela FINEP.

1. INTRODUÇÃO

Estudos efetuados mostram que os programas apresentam, em supercomputadores vetoriais, como o Cray C90 e o NEC SX-3R, um desempenho médio da ordem de apenas 15% do desempenho máximo possível. Isto é causado pelo não aproveitamento de todos os recursos computacionais disponíveis. Para auxiliar os programadores na difícil tarefa de otimizar seus programas neste tipo de computador, foram desenvolvidos os compiladores vetorizadores automáticos.

Os vetorizadores são atualmente uma tecnologia bem madura; de modo que seus fundamentos básicos estão hoje sendo aplicados em máquinas maciçamente paralelas. Esta nova classe de computadores, representados pelo Cray T3D, NEC SX-4 e Convex Exemplar, adotam um número muito grande de processadores e uma complexa hierarquia de memória principal. Estas características levam a uma grande complexidade na programação de aplicações que consigam obter altos desempenhos. Embora muitas vezes seja satisfatório usar estas máquinas para resolver múltiplas tarefas simultaneamente, em outras ocasiões é necessário aproveitar todos os recursos na solução de um único problema.

De forma a conseguirmos obter alto desempenho nesta classe de computadores, dois aspectos são fundamentais: a exploração eficaz do *paralelismo* presente nas aplicações e um uso adequado do complexo sistema de memória principal, através de um correto padrão de *acesso aos dados*.[1]

Este trabalho apresenta uma estratégia integrada para a otimização de paralelismo e de acesso aos dados. A seção seguinte apresenta como ambos os aspectos estão sendo abordados atualmente e introduz alguns conceitos que serão necessários posteriormente. A seção 3 indica algumas situações em que há conflitos em um enfoque integrado. Mas a seção 4 descreve dois estudos que mostram a viabilidade de se otimizar de forma conjunta o paralelismo e o acesso aos dados. As conclusões e sugestões de trabalhos futuros são apresentados na seção 5.

2. PARALELISMO E LOCALIDADE DE DADOS

Bacon apresenta em [BACO93] um catálogo bem completo das técnicas existentes para otimização de programas visando obter alto desempenho. Constata-se que a grande maioria das transformações de programa descritas tem como objetivo a otimização do paralelismo. Apenas uma subseção é devotada para as técnicas para otimizar os acessos à memória.

De uma maneira geral, os trabalhos nas áreas de paralelismo e de acesso aos dados vem sendo desenvolvidos de forma independente. Embora algumas transformações possam ser aplicadas para ambos os objetivos[2], a grande maioria foi desenvolvida tendo objetivos bem distintos.

---

[1] um bom padrão de acessos pode ser obtido através de um processo de otimização da localidade de referências do programa: a exploração da localidade temporal e espacial pode levar a um uso efetivo de todos os níveis da hierarquia de memória (memória cache, memória local e os diversos níveis de memória principal). [MIDO94]

Trabalhos recentes têm proposto estratégias unificação das técnicas. Podemos citar como exemplo os trabalhos desenvolvidos nas teses de doutoramento de Michael Wolf emStanford [WOLF92], de Kathryn McKinley na Rice University [MCKI92] e de Wei Li em Cornell [LI93]. Mais recentemente, tem-se estudado o efeito de algumas transformações para a otimização de paralelismo e de localidade de dados [KENN93][MANJ95]. Mas analisando-se com cuidado, estes trabalhos ainda efetuam um processo de otimização em duas fases distintas, onde se processam os dois aspectos em separado.

2.1 - Otimização de Paralelismo

Existem três passos básicos para a utilização do paralelismo presente em um programa em um sistema multiprocessador: [MORE95]

- *extração:* o paralelismo intrínsico do programa, ou do algoritmo, deve ser identificado;
- *especificação:* uma vez identificado, o paralelismo deve ser especificado explicitamente para os programas de sistema ou para o hardware;
- *exploração:* o passo final consiste em realmente usar o paralelismo para manter ocupados os processadores da máquina, executando o máximo de trabalho útil possível.

As técnicas tradicionais de análise para extração de paralelismo têm se limitado ao paralelismo de loops, ou paralelismo estruturado, de forma a maximizar a eficiência dos programas[3]. Para isto, diversas transformações de loops foram desenvolvidas [BACO93][BANN94]. O objetivo principal das transformações de loops é definir uma nova ordem de execução das iterações de uma estrutura de loops de forma a permitir a sua execução paralela. Exemplos de transformações existentes são *loop interchange*, *loop skewing*, *loop distribution*, *strip mining* e *tiling*.[4]

De forma a avaliarmos o processo de otimização de paralelismo, diversas métricas são propostas na literatura: (veja por exemplo [HWAN93])

- *custo*: representado pelo custo total do processamento, envolvido pelo uso de hardware e software;
- *balanceamento de carga*: medida da uniformidade da carga de processamento de cada um dos processadores envolvidos na execução do programa;
- *tempo de processamento*: tempo em segundos gasto para o processamento de um dado programa;
- *speed-up*: medida do ganho obtido pela execução paralela de um programa em p processadores. Seja, T_1 o tempo de execução em um único processador e T_p, em p processadores, o speed-up é

[2] o exemplo mais conhecido é a transformação de *loop interchange*. Ele pode se empregado tanto para posicionar um loop sem dependências na parte mais externa de um aninhamento de loops visando a paralelização, como para alterar o padrão de acesso aos elementos de uma matriz.

[3] somente trabalhos recentes têm começado a apresentar resultados muito interessantes com respeito à exploração de paralelismo não estruturado, ou paralelismo funcional [MORE95].

[4] devido a limitações de espaço não detalharemos estas transformações neste trabalho. Sugerimos que sejam consultadas as referências para uma completa descrição das mesmas.

definido por $S(p) = \frac{T_1}{T_p}$. Idealmente, o *speed-up* deve ser igual ao número de processadores empregado na execução do programa.

- *eficiência*: medida da taxa de ocupação dos p processadores utilizados durante a execução paralela. A eficiência é definida por $E(p) = \frac{S(p)}{p}$. A situação ideal é aquela onde a eficiência é igual a 1 (ou 100%), mas na prática, espera-se que a eficiência seja menor e diminua à medida que aumentamos o número de processadores.

As análises deste trabalho adotam as últimas três métricas para a comparação dos programa paralelos.

2.2 - Otimização de Localidade de Dados

A análise de localidade de dados para um dados programa é um dos aspectos mais importantes no processo de otimização de aplicações para máquina maciçamente paralelas. Existem três subproblemas que devem ser levadas em conta: [EISE90]

- *deteção e estimativa da localidade de dados*: este problema esta relacionado com a quantificação das propriedades de localidade de um código;
- *exploração da localidade de dados*: esta questão é específica aos níveis onde as transferências são gerenciadas pelo software (registradores, memória local). Neste casos, aspectos relacionados com coerência e sincronização devem ser resolvidos;
- *melhoramento da localidade de dados*: este ponto é abordado pelas transformações de programa aplicados de modo a aumentar as propriedades de localidade de acesso aos dados.

Estes aspectos são muito complexos e são atualmente alvo de intensas pesquisas. Convém ressaltar que é muito importante a análise da organização dos dados na memória; neste sentido, além das transformações de loop, ou *transformações de controle*, uma outra classe de transformações vem sendo desenvolvida. Estas transformações, chamadas *transformações de dados*, visam mudar o padrão de acesso aos dados modificando a ordem como são armazenados os elementos na memória. Estudos mostram que seu efeito é similar às transformações de controle e podem ser aplicadas de modo distinto [CIER94][TAKA95].

Exemplos de transformações de controle para a otimização de localidade de dados são o *scalar replacement*, *unroll-and-jam*, *tiling*, *scalar expansion* e *loop fusion*. Podemos citar também como exemplos de transformações de dados, o *memory copying* e *data alignment*.

3. PROBLEMAS COM A INTEGRAÇÃO

Apesar dos avanços alcançados na otimização de paralelismo e de localidade de dados, as técnicas não foram desenvolvidas tendo-se em mente uma aplicação conjunta. De uma maneira simples, podemos

dizer que uma análise visando a otimização de um dos aspectos analisados aqui pode resultar em um efeito contrário em relação ao outro. Por exemplo, a aplicação de uma transformação de programa visando melhorar a localidade de dados pode impedir a paralelização do programa. Apresentamos nesta seção alguns aspectos que devem ser levados em conta quando for necessário otimizar em conjunto a localidade de dados e o paralelismo.

Descrevemos a seguir dois exemplos que ilustram algumas situações onde podem haver conflitos entre os dois enfoques: o primeiro quando se deseja otimizar o paralelismo e o segundo, a localidade de dados. Consideremos o seguinte trecho de código

```
for (i=0; i<N1; i++)
  for (j=0; j<N2; j++)
    vxo[i][j] = vxo[i-1][j] + dty[i]*po[j+2];
```

Uma análise levando-se em conta a paralelização detecta uma dependência verdadeira no acesso à matriz `vxo`, impossibilitando assim a paralelização do loop externo (loop i). Uma solução simples para este caso é a aplicação da transformação de *loop interchange*, colocando o loop j como o loop externo, que não apresenta nenhuma dependência e, portanto, é paralelizável. O código resultante é o seguinte

```
for (j=0; j<N2; j++)  /* paralelizável */
  for (i=0; i<N1; i++)
    vxo[i][j] = vxo[i-1][j] + dty[i]*po[j+2];
```

Agora uma análise do padrão de acessos do exemplo acima mostra que o programa resultante realiza um acesso ineficiente à matriz `vxo`. O programa original tinha um padrão de acesso por linha e passou a ter um padrão de acessos por coluna, o que é muito ruim em termos de localidade de dados[5]. Concluímos que neste caso, a transformação de *loop interchange* otimizou o paralelismo, mas piorou a localidade de dados no programa-exemplo acima.

Vejamos um outro exemplo. A transformação de *loop fusion* tem demonstrado ser muito eficiente na otimização de localidade de dados [KENN93][MANJ95]. A ação básica desta transformação de programa é combinar o corpo de dois loops em um único e reunir os respectivos espaços de iteração em um espaço combinado. O efeito resultante é a diminuição do número de iterações que separam referências à mesma matriz de dados. Consideremos o seguinte exemplo

```
for (i=2; i<N; i++)
  xdt[i] = rs[i+4] * K;

for (i=2; i<N; i++)
  rl[i+1] = work[i] - (xdt[i-1] + xdt[i-2]) / 2;
```

A aplicação de *loop fusion* para diminuir a distância dos acessos ao vetor `xdt` resulta no seguinte trecho de programa

[5] convém lembrar que a linguagem C organiza os elementos de uma matriz por linha. Deste modo, o melhor padrão de acessos a dados em programas escritos nesta linguagem é acessar sequencialmente os elementos pertencentes à mesma linha da matriz, antes de passar para uma outra coluna.

```
for (i=2; i<N; i++) {
  xdt[i] = rs[i+4] * K;
  rl[i+1] = work[i] - (xdt[i-1] + xdt[i-2]) / 2;
}
```

onde temos um aumento considerável no reuso dos elementos de `xdt`.

Considerando-se agora o ponto de vista da otimização de paralelismo, vemos que o programa original acima apresenta os dois loops sem dependências e, portanto, paralelizáveis. Contudo, o programa transformado contém duas dependências de fluxo que impedem a paralelização do loop, impedindo assim a sua execução em múltiplos processadores.

Estes dois exemplos descritos acima descrevem situações possíveis que podem ser encontrados em diversos programas. Apesar disto parecer apresentar um cenário pessimista em relação à otimização conjunta de paralelismo e de localidade de dados, diversas outras situações permitem a aplicação destas técnicas, resultando em excelentes ganhos de desempenho. A próximo seção mostra dois estudos de caso, onde são mostrados situações reais onde ambos os enfoques são integrados de modo a obter maiores desempenhos.

4. ESTUDOS DE CASO

De modo a estudarmos a viabilidade da aplicação conjunta de técnicas de otimização de paralelismo e de localidade de dados, realizou-se dois estudos de caso. Para isto, tomou-se dois programas e, após uma série de análises e transformações, executou-se as diversas versões resultantes destes programas em uma máquina real para a medição de seus respectivos desempenhos.

4.1 - Ambiente de Execução e Biblioteca de Paralelismo

A execução dos programas foi realizada em uma maquina Silicon Graphics Power Series 4D/480 VGX, que é composto por 8 processadores MIPS R3000A. Cada processador possui uma memória cache de instruções de 64Kbytes, uma cache de dados de primeiro nível de 64Kbytes e uma cache de dados secundário de 1Mbytes. A memória principal do sistema é de 256 Mbytes.

A medição de tempo foi efetuada instrumentando-se os programas com a chamada times. O sistema operacional é o IRIX 5.2. A paralelização dos programas foi efetuada utilizando-se a biblioteca de paralelismo disponível no sistema IRIX [SILI89]. A tabela 1 mostra as principais rotinas empregadas.

4.2 - Descrição dos Casos Escolhidos

Passamos agora a fazer uma rápida descrição dos casos estudados. Apresentamos para cada um dos programas a forma de paralelização adotada e algumas dificuldades enfrentadas na execução dos experimentos.

Tabela 1 - Algumas rotinas da biblioteca de paralelismo do IRIX.

rotina	descrição
m_fork	cria processos paralelos
m_get_myid	retorna identificador do processo paralelo
m_kill_procs	termina execução dos processos paralelos
m_next	atualiza contador global do sistema utilizado para sincronização dos processos
m_sync	sincroniza todos os processos paralelos em algum ponto do código do programa
m_set_procs	determina quantos processadores serão utilizados na execução paralela do programa
m_lock	trava semáforo global
m_unlock	destrava semáforo global

4.2.1. Multiplicação de Matrizes

O programa de multiplicação de matrizes consta do produto de duas matrizes quadradas de ordem 512, implementado segundo o algoritmo tradicional baseado no cálculo do produto interno das linhas da primeira matriz com as colunas da segunda matriz, segundo

$$c_{i,j} = \sum_{k=1}^{512} a_{i,k} \times b_{k,j}$$

Denominaremos `mm`, a versão original do programa, e, a medida que formos aplicando as transformações, acrescentaremos sufixos ao nome, como por exemplo, `mm-li`, na aplicação de *loop interchange*.

A estratégia de paralelização adotada foi paralelizar o loop mais externo distribuindo as iterações pelos processadores de forma estática (*pre-scheduling*). Desta forma, a matriz `b` tem seus elementos acessados por todos os processadores e as matrizes `a` e `c` têm seus elementos distribuídos pelos processadores segundo as linhas. A figura 1 abaixo ilustra o esquema adotado.

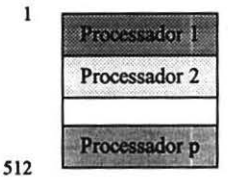


Figura 1 - Divisão dos acessos aos elementos das matrizes a e c do programa de multiplicação de matrizes.

O programa não necessitou de nenhum mecanismo de exclusão mútua para a atualização da matriz produto, visto que seus elementos foram distribuídos pelos processadores, que tinham acesso exclusivo às suas respectivas linhas. Durante a tomada de medidas de desempenho não houve maiores problemas e todos os processadores puderam ser utilizados.

4.2.2. Decomposição LU

O programa de decomposição LU executa a computação de duas matrizes triangulares inferior e superior, L e U respectivamente, a partir de uma matriz quadrada A de ordem 512, tal que $A = L \times U$. a implementação segue o algoritmo tradicional sem pivotamento.

A versão original do programa será identificada como `lu`, e as sucessivas versões transformadas terão a identificação da transformação acrescentada ao final, como por exemplo, `lu-sr` para a versão onde a transformação *scalar replacement* for aplicada.

A estratégia de paralelização adotada seguiu a seguinte estratégia: para cada passo do algoritmo quando uma nova linha é processada, a atualização da respectiva submatriz foi executada pelos diversos processadores da máquina. A divisão de processamento para a computação desta submatriz é a mesma empregada para a multiplicação de matrizes. A figura 2 ilustra o esquema adotado.

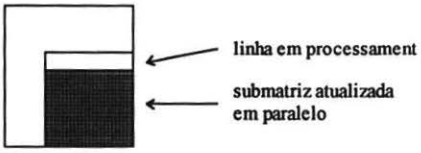


Figura 2 - Esquema de paralelização adotado para o programa de decomposição LU.

O procedimento experimental foi executado em somente até 7 processadores. Não foi possível utilizar todos os processadores da máquina devido a alta carga de utilização no período de avalidação do caso de estudo.

4.3 - Análise dos Resultados Obtidos

Analisamos aqui o efeito da aplicação de uma série de transformações visando a localidade de dados em programas paralelos. Foram aplicadas as transformações de *loop interchange* (li), *scalar replacement* (sr), *unroll-and-jam* (uj) e *tiling* (t). Apresentamos além dos tempos de execução, uma análise do *speed-up* obtido para cada uma das versões e das respectivas eficiências.

4.3.1. Decomposição LU

A versão inicial do programa, denominado `lu`, foi implementada usando o algoritmo tradicional formado pelo aninhamento de loops seguinte:

```
for (k=0; k<N; k++) {
  U[k][k] = A[k][k];
  for (i=k+1; i<N; i++) {          /* parte I */
    L[i][k] = A[i][k] / U[k][k];
    U[k][i] = A[k][i];
  }
  for (i=k+1; i<N; i++)        /* parte II */
    for (j=k+1; j<N; j++)
      A[i][j] -= L[i][k] * A[k][j];
}
```

Uma análise no padrão de acessos à memória [TAKA95] verifica que, na parte I do algoritmo acima, as matrizes L e A são acessadas por coluna e a matriz U, por linha e, na parte II, ambas as matrizes L e A são acessadas por linha. Verifica-se também que o elemento L[i][k] é constante durante a execução do loop mais interno; o que permite a aplicação da transformação de *scalar replacement* (levando à versão `lu-sr`).

De forma a obtermos mais operações de ponto flutuante por acessos à memória, o loop mais interno (loop j) da parte II do algoritmo foi modificadado com a transformação de *unroll-and-jam* (`lu-uj`). A versão final de nosso estudo avaliou a aplicação conjunta de ambas as transformações (versão `lu-uj-sr`).

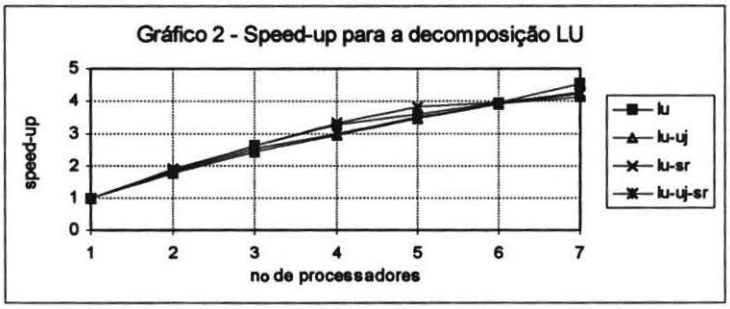
Cada uma das versões do programa de decomposição LU foi executada em uma máquina real, e os tempos de execução medidos são mostrados na tabela 2 abaixo.

Tabela 2 - Tempos de execução das versões do programa de decomposição LU (em seg).

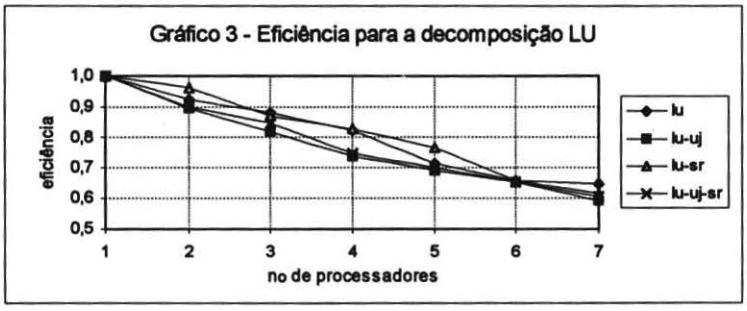
Nº processadores	lu	lu-uj	lu-sr	lu-uj-sr
1	47.20	40.70	39.0	31.80
2	25.50	22.70	20.30	17.70
3	17.90	16.60	14.90	12.50
4	14.35	13.80	11.80	10.60
5	13.20	11.80	10.20	9.10
6	12.00	10.40	9.90	8.10
7	10.40	9.88	9.10	7.50

Observando-se os tempos de execução notamos uma melhora de desempenho em média de 1,45 vezes entre as versões `lu` e `lu-uj-sr`. Isto comprova a validade de nossa estratégia. O gráfico 1 abaixo resume os tempos de execução, onde visualizamos os ganhos de desempenho obtidos com a seqüência de aplicação das tranformações.

Analisando-se também o efeito das transformações sobre o *speed-up*, ilustrado no gráfico 2, comprovamos que, para este caso, as curvas praticamente se coincidem.



O mesmo efeito é observado com relação à eficiência. O gráfico 3 mostra que de uma maneira geral. todas as versões do programa apresentam o mesmo comportamento.



Resumindo, observamos que devido à estratégia de paralelização adotada, as curvas de *speed-up* e de eficiência não se alteraram com a aplicação das transformações estudadas neste caso. Devido à escolha da paralelização do loop intermediário, o *overhead* da criação das múltiplas execuções em paralelo para a atualização das submatrizes acarretou em ganho apenas no tempo de execução da versões transformadas em relação à versão original. Neste sentido novos estudos estão planejados tendo-se em mente a paralelização do loop mais externo.

4.3.2. Multiplicação de Matrizes

A versão inicial do programa de multiplicação de matrizes, denominada mm, empregou a implementação canônica do produto interno implementado por um aninhamento de três loops:

```
for (i=0; i<N; i++) /* loop paralelizado */
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Observa-se através de uma análise do padrão de acessos que a matriz b é acessada por coluna. A aplicação de *loop interchange* soluciona esta questão (`mm-li`). Visando um melhor aproveitamento dos registradores disponíveis e um aumento na razão de execução de instruções de ponto-flutante em relação às instruções de controle, foram aplicadas as transformações de *scalar replacement* (`mm-li-sr`) e *unroll-and-jam* (`mm-li-uj`), respectivamente. Verificou-se também o efeito conjunto de ambas as transformações em `mm-li-uj-sr`. E finalmente, de forma a obtermos uma maior localidade de dados aplicou-se a transformação de `tiling` (mm-li-uj-sr-t).

Os tempos de execução obtidos para cada uma destas versões é mostrada na tabela 3 e resumidos no gráfico 4.

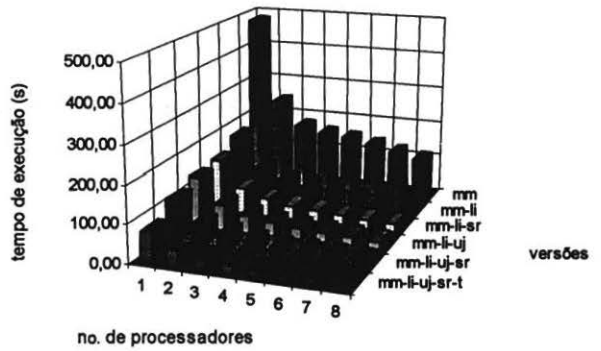
Tabela 3 - Tempos de execução das versões do programa de multiplicação de matrizes (em seg).

Nº processadores	mm	mm-li	mm-li-sr	mm-li-uj	mm-li-uj-sr	mm-li-sr-uj-t
1	480.0	171.6	140.3	133	117.5	79.3
2	245.0	86.2	71.1	68.1	60.3	40.2
3	178.0	58.2	48.3	46.2	41.2	27.7
4	161.9	44.8	37.4	35.4	31.2	20.8
5	154.5	36.3	30.4	29.0	26.1	17.2
6	140.6	31.6	26.0	25.8	23.6	14.4
7	124.5	27.3	22.0	22.1	20.6	12.4
8	111.9	24.1	19.7	19.8	18.1	11.2

Constatamos uma elevada melhora de desempenho ao compararmos a versão original (mm) com a última (`mm-li-uj-sr-t`). E esta melhora é maior em função do número de processadores. Partindo de um

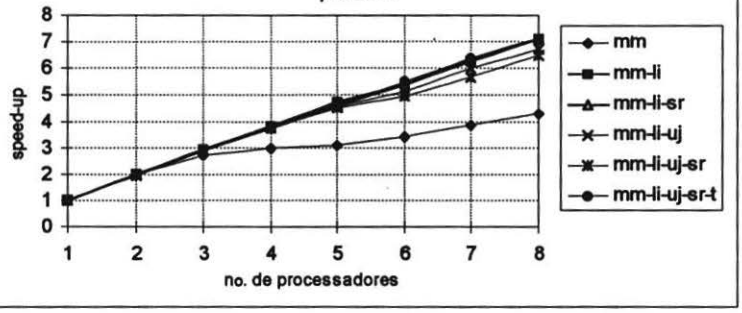
aumento de desempenho de 6 vezes para a versão para um único processador, o ganho chega a 10 vezes para as versões do programa para 7 e 8 processadores. Isto comprova uma boa sintonia entre as transformações de paralelismo e de localidade de dados, onde nota-se que seus efeitos foram somados.

Gráfico 4 - Tempos de execução para a multiplicação de matrizes paralela



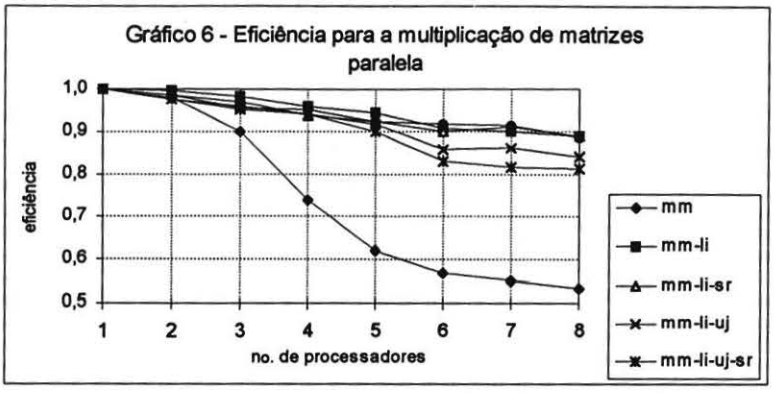
E o efeito das transformações não está restrito somente no tempo de execução. Do gráfico 5, onde está apresentado a curva de *speed-up* para cada uma das versões, vemos a grande diferença entre a versão original e as versões transformadas.

Gráfico 5 - *Speed-up* para a multiplicação de matrizes paralela



O speed-up máximo da versão mm é 4,23 para 8 processadores, ao passo que as versões transformadas atingem um speed-up de até 7 vezes para o mesmo número de processadores. O mesmo efeito pode ser observado através da curva da eficiência, apresentada no gráfico 6. Verificamos um sensível aumento na eficiência devido à aplicação das transformações. A eficiência do programa original chega a diminuir

até 0,54 para 8 processadores, ao passo que para as versões transformadas, a menor eficiência para o mesmo número de processadores é 0,81 (para o `mm-li-uj-sr`).



Concluímos então com os efeitos favoráveis das transformações sobre o comportamento do programa de multiplicação de matrizes, tanto com respeito ao tempo de execução, como no *speed-up* e na eficiência obtidos.

5. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou alguns aspectos importantes no processo de otimização de programas nos modernos computadores maciçamente paralelos. Apesar de existirem algumas situações onde o paralelismo e a localidade de dados levem a pequenos conflitos, os casos estudados mostraram que é possível conciliar ambos os aspectos num processo de otimização conjunto.

Os resultados obtidos com os programas de decomposição LU e de multiplicação de matrizes mostraram que a integração da otimização de paralelismo e de acesso a dados é possível, levando a bons aumentos de desempenho. E no caso da multiplicação de matrizes, as transformações aplicadas levaram também a uma melhora nas curvas de *speed-up* e da eficiência.

Como trabalhos futuros, planejamos seguir dois caminhos: o primeiro, com a implementação destas técnicas aqui descritas em um sistema automático para otimização de paralelismo e localidade de dados que se encontra em desenvolvimento [MIDO95], e o segundo, com a aplicação destas e outras técnicas[6] em programas aplicativos completos, como por exemplo, na área de processamento de imagens médicas [LOPE95].

[6] os próximos trabalhos deverão verificar a influência das transformações de *loop fusion* e *loop distribution* [KENN93][MANJ95], e novas transformações de dados, como por exemplo, *data alignment* e *data copying*. Pretende-se também averiguar qual o efeito obtido com a aplicação conjunta de transformações de controle e de transformações de dados.

Esperamos que em breve estas técnicas aqui descritas estejam incorporados nos programas de sistema das máquinas paralelas, de forma que mesmo usuários não especializados possam tirar proveito da grande capacidade de processamento que estes computadores possuem.

AGRADECIMENTOS

Os autores gostariam de agradecer a todos que de uma maneira ou outra contribuíram para a realização deste trabalho e pelos comentários e sugestões na elaboração da versão preliminar deste artigo. Em especial gostaríamos de agradecer a *Sérgio Takahashi* pela execução dos estudos sobre o programa de decomposição LU. Agradecemos também ao Laboratório de Sistemas Integráveis, nas pessoas dos *Prof. Dr. João Antônio Zuffo* e *Prof. Dr. Sérgio Takeo Kofuji*, pelo apoio e pela concessão do uso da máquina Silicon Graphics Power Series 4D/480 VGX para a execução dos procedimentos experimentais deste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[BACO93] BACON, D. F. et alii. **Compiler transformations for high-performance computing**. Technical Report Nº UCB/CSD-93-781. Computer Science Division, University of California at Berkeley. 1993.

[BANE94] BANERJEE, U. **Loop parallelization**. Kluwer Academic Publishers, 1994.

[CARR94] CARR, S.; McKINLEY, K. S.; TSENG, C.-W. Compiler optimizations for improving data locality. In: Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 6, San Jose, CA. **Proceedings**. p.??-??. October, 1994.

[CIER94] CIERNIAK, M. & LI, W. **Unifying data and control transformations for distributed shared memory machines**. Technical Report 542. Department of Computer Science, University of Rochester. November 1994.

[EISE90] EISENBEIS, C. et alii. **A strategy for array management in local memory**. Rapports de Recherche Nº 1262. Institut National de Recherche en Informatique et en Automatique (INRIA), France. Juillet 1990.

[HWAN93] HWANG, K. **Advanced computer architecture**: parallelism, scalability, programability. McGraw-Hill, 1993.

[KENN93] KENNEDY, K. & McKINLEY, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Workshop on Languages and Compilers for Parallel Processing, 6, Portland, OR. **Proceedings**. p.301-20. August 1993.

[LI93] LI, W. **Compiling for NUMA parallel machines**. PhD Thesis. Cornelll University, 1993.

[LI94] LI, W. **Compiler optimizations for cache locality and coherence**. Technical Report 504. Department of Computer Science, University of Rochester. April 1994.

[LOPE95] LOPES, R. D. & RANGAYYAN, R. N. **3D region-based filters for noise removal**. Submitted in Proc. IEEE ICIP-95 Int. Conf. Image Processing. IEEE, Washington, D.C., October 1995.

[MANJ95] MANJIKIAN, N. & ABDELRAHMAN, T. **Fusion of loops for parallelism and locality**. Technical Report CSRI-315. Computer Systems Research Institute, Department

of Computer Science, Department of Electrical and Computer Engineering, University of Toronto, Canada. February 1995.

[MCKI92] McKINLEY, K. S. **Automatic and interactive parallelization**. PhD. Thesis. Department of Computer Science, Rice University. April 1992.

[MIDO94] MIDORIKAWA, E. T. Análise da otimização de acessos à memória. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 6, Caxambu, MG. **Anais**. p. 37-52. Agosto de 1994.

[MIDO95] MIDORIKAWA, E. T. et alii. **Um sistema integrado para otimização automática de paralelismo e localidade de dados**. Submetido ao VII Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho (SBAC-PAD'95), 1995.

[MORE95] MOREIRA, J. E. **On the implementation and effectiveness of autoscheduling for shared-memory multiprocessors**. PhD. Thesis. Deparment of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. 1995.

[SILI89] SILICON GRAPHICS. **Parallel programming on Silicon Graphics systems**. Document number 007-0770-010, 1989.

[TAKA95] TAKAHASHI, S.; MIDORIKAWA, E. T.; ZUFFO, J. A. **Análise do padrão de acessos e otimização de localidade em sistemas de computação de alto desempenho**. Submetido ao XII Concurso de Trabalhos de Iniciação Científica (CTIC'95). 1995.

[WOLF92] WOLF, E. **Improving locality and parallelism in nested loops**. PhD. Thesis. Department of Computer Science, Stanford University. August 1992.