Um Sistema Integrado para Otimização Automática de Paralelismo e de Localidade de Dados *

Edson T. Midorikawa
Nelson T. Minoura
Pedro V. Artigas
João Antônio Zuffo

{emidorik, minoura, artigas, jazuffo}@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

A obtenção de alto desempenho nos modernos computadores paralelos é um problema sério. Para se explorar eficientemente os recursos computacionais disponíveis nesta classe de computadores, o programador deve se preocupar com dois aspectos muito importantes: o paralelismo e a localidade de dados. Este trabalho apresenta um sistema para otimização automática de paralelismo e localidade de dados. Resultados preliminares mostram a funcionalidade do sistema na identificação das transformações de programa que melhoram o desempenho de programas em máquinas multiprocessadoras com memória compartilhada.

ABSTRACT

Obtaining high performance in modern parallel machines is a serious problem. In order to explore efficiently the computational resource available in this class of computers, the programmer must pay attention in two important aspects: parallelism and data locality. This work presents a system for automatic optimization of parallelism and data locality. Preliminary results show that our approach is functional in identifying program transformations that improve performance for shared memory multiprocessors.

* Este trabalho foi parcialmente financiado pelo programa RHAE/CNPq e pela FINEP.

1. INTRODUÇÃO

A acirrada disputa dos fabricantes de computadores para o desenvolvimento de máquinas cada vez mais poderosas vem consolidando uma classe de sistemas caracterizada pelo uso de diversos processadores e um sistema de memória distribuída. Exemplos de máquinas com estas características são o NEC SX-4, Cray T3D e Convex Exemplar. Tais sistemas são conhecidos na literatura como sistemas maciçamente paralelos (MPP).

A obtenção de alto desempenho na execução paralela de programas em tais sistemas multiprocessadores permanece um desafio para os especialistas em processamento paralelo. A pergunta a ser respondida é a seguinte: "como aproveitar eficazmente o desempenho bruto destes computadores ?".

A resposta a esta difícil pergunta se baseia na exploração eficaz dos recursos computacionais disponíveis: paralelismo e acesso aos dados. Um programa obterá grande desempenho em tais máquinas se conseguir fazer uso intenso dos vários processadores e acessar os dados de maneira eficiente.

Este trabalho procura abordar estes assuntos dentro do contexto de um sistema automático para otimização de paralelismo e localidade de dados. O objetivo deste sistema é fornecer subsídios ao usuário de uma máquina MPP para desenvolver um programa que obtenha maior desempenho.

A estrutura deste trabalho é a seguinte: a seção 2 apresenta os conceitos básicos necessários para a compreensão dos assuntos seguintes. Os fundamentos teóricos referentes a otimização de paralelismo e localidade de dados são extensamente descritos na seção 3 e a estratégia adotada para a integração dos dois enfoques é apresentada na seção 4. A próxima seção descreve a situação atual do sistema em desenvolvimento e um estudo de caso é mostrado na seção 6. O trabalho termina na seção 7 com as principais conclusões e indicativos de trabalhos futuros.

2. CONCEITOS BÁSICOS

Assumimos que o leitor é familiar com alguns conceitos utilizados neste trabalho: dependência de dados, transformação de programas, espaço de iterações, paralelização de loops, padrão de acesso a dados, reuso de memória e localidade de dados. Para maiores detalhes sobre estes assuntos, veja, por exemplo, as referências [BANE88][BACO93][EISE90][MCKI92][TAKA95][WOLF92]. Um trabalho anterior que também aborda estes tópicos é [MIDO94].

Este artigo está relacionado com o desenvolvimento de um sistema de reestruturação automática de programas visando a otimização de paralelismo e localidade de dados. É um sistema que é dito um *reestruturador de fonte-para-fonte*; ou seja, dado um programa de entrada escrito em uma linguagem de alto nível, ele executa a análise e as transformações necessárias e, assim, gera um programa de saída também escrito em uma linguagem de alto nível.

3. PARALELISMO E LOCALIDADE DE DADOS

A otimização de paralelismo é um assunto que vem sendo pesquisado a muito tempo. Estas pesquisas têm se concentrado na aplicação de transformações de programas baseado nas restrições de dependências de dados. Para tal, foi desenvolvida um complexo fundamento teórico para os compiladores paralelizadores. Esta teoria teve tanto sucesso que vem sendo aplicada em outras áreas,

como por exemplo, para sistemas de tempo real, análise de "*false sharing*" em máquinas paralelas, consistência de memória em sistemas distribuídos e otimização de acessos a dados.

Com relação a localidade de dados, este assunto vem merecendo esforços de diversos centros devido ao problema da latência de memória existente nos modernos computadores. Assim, servindo-se do suporte fornecido pelos compiladores paralelizadores, novos sistemas vem sendo implantados com o desenvolvimento de novas transformações e métodos de análise.

Os grandes problemas existentes atualmente na área de otimização de programas recaem em duas grandes áreas: primeiro, a determinação da melhor seqüência de transformações em um dado programa. Um passo na solução deste problema são as transformações unimodulares [BANE94]. Fundamentado na teoria das matrizes unimodulares, o objetivo das transformações unimodulares é fornecer uma base consistente para a análise da aplicação de combinações de *loop interchange*, *loop reversal* e *loop skewing*.[1]

A segunda área se refere à quantificação da localidade de dados: de modo a determinarmos seguramente o ganho obtido pela aplicação de uma transformação visando otimizar a localidade, precisamos de uma métrica que quantifique o padrão de acessos a dados de um programa. Uma das métricas mais simples é o vetor distância de dependência. Infelizmente, ele não é conveniente pois descreve apenas o número de elementos que devem ser reaproveitados, sem determiná-los propriamente. Neste sentido, as *janelas de referência* [EISE90] suprem largamente este requisito, fornecendo não somente o número de elementos a serem reutilizados como também os elementos propriamente ditos. Nos ítens a seguir passamos a descrever com algum detalhe estes dois tópicos que são fundamentais para a compreensão do sistema que vem sendo desenvolvido.

3.1 - Transformações Unimodulares

Utilizaremos, de forma a otimizar o programa de entrada em nosso sistema uma classe de transformações conhecidas como transformações unimodulares. Apresentamos a seguir as definições necessárias, e as transformações presentes nesta classe.

Como base para nosso estudo de transformações unimodulares, tomamos um programa **genérico**, que contém K loops perfeitamente alinhados:

```
for(i1=n1; i1<=N1; i1++)
  for(i2=n2; i2<=N2; i2++)
      (.....)
        for(ik=nk; ik<=Nk; ik++)
          {
          (Corpo do loop)
          }
```

Uma das formas de se transformar um conjunto de K loops perfeitamente alinhados é obter novas variáveis de loop a partir de combinações lineares das anteriores. Tal transformação pode ser representada na forma de matriz. Por exemplo, tomando K loops, nas variáveis $I=(i_1, i_2,..., i_K)$ pode-se,

[1] Outros enfoques mais genéricos vem sendo desenvolvidos e que merecerão estudos posteriores, como por exemplo, as transformações lineares (ou inteiras) [LI93], e suas generalizações [KULK94] e os *mappings* [KELL95]. Estes incluem outras transformações não suportadas pelas transformações unimodulares e que são importantes para o processo de otimização, como por exemplo, o *tiling*, *loop fusion* e *loop alignment*.

multiplicando o vetor das variáveis de loop por uma matriz U, obter-se novas variáveis $J=(j_1, j_2,..., j_K)$ de forma que:

$$J = I.U$$

Onde U é a matriz que representa a transformação aplicada ao conjunto de loops. Utilizamos matrizes apenas com elementos inteiros pois as variáveis de loop, supostamente, assumem apenas valores inteiros. Portanto, utilizando uma matriz de transformação inteira, obtemos novas variáveis de loop que também assumem apenas valores inteiros.

Temos, porém, outro problema. No corpo do loop são referenciadas as variáveis $I=(i_1, i_2,..., i_K)$. Devemos, portanto, alterar as referências às variáveis antigas, de forma a obter um programa com a mesma estrutura lógica tomando a transformação inversa:

$$I = J.U^{-1}$$

Embora não de forma tão explícita, assumimos também que o incremento de uma dada variável i_x para uma nova iteração do loop X é positivo e unitário, (pois qualquer loop com variável de loop inteira pode ser facilmente reduzido a esta forma). Se tomarmos uma matriz U, apenas inteira, como matriz de transformação de nosso conjunto de loops, obteríamos um novo conjunto de loops perfeitamente alinhados cujo incremento não deve ser, necessariamente, unitário. Como exemplo tomemos a transformação $U = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$

Aplicando-se ao programa:

```
for(i1=1;i1<=10;i1++)
  for(i2=1;i2<=10;i2++)
    {
    (Corpo do loop)
    }
```

resulta-se o programa:

```
for(j1=2;j1<=20;j1+=2)
  for (j2=1;j2=10;j2++)
    {
    (Novo corpo do loop)
    }
```

Esta transformação, embora muito simples, esclarece porque é conveniente não tomar como matriz de transformação uma matriz apenas inteira. É possível provar que, se tomarmos uma matriz inteira com determinante em módulo igual a um, obtemos um programa transformado tal que o incremento necessário a cada variável, a cada nova iteração do seu respectivo loop é unitário.

Definição 1: Matrizes unimodulares são matrizes compostas apenas de elementos inteiros e cujo determinante tem módulo unitário.

Matrizes unimodulares podem representar transformações já bastante conhecidas. Por exemplo a transformação *loop interchange* pode ser representada, no caso de apenas dois loops, pela matriz $U = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

No caso, as novas variáveis de loop são iguais as anteriores mas o loop mais externo do programa original torna-se o mais interno no programa transformado, e vice-versa. A matriz de transformação U é claramente unimodular.

Uma generalização da transformação *loop interchange* para K loops, conhecida como *loop permutation*, consiste em permutar a ordem de um número qualquer de loops perfeitamente aninhados dentre um conjunto de K loops, pode ser representada através de uma transformação unimodular. Para obter a matriz que representa tal transformação basta permutar as linhas da matriz identidade I_k da mesma forma que se deseja permutar os loops, ou seja, se substituirmos a linha l_x da matriz identidade pela linha l_y, e vice-versa, estaremos representando a troca de posição entre o loop x e o loop y no conjunto de k loops. Repetindo este procedimento o número de vezes necessário, obtemos a matriz unimodular que representa a transformação *loop permutation* desejada.

Outra transformação que também pertence à classe de transformações unimodulares é a transformação *loop reversal*. Esta transformação consiste em inverter a ordem de execução das iterações de um dado loop x contido em um conjunto de k loops perfeitamente alinhados. Por exemplo tomemos a seguinte transformação $U = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$

Aplicada ao programa:

```
for(i1=1;i1<=10;i1++)
      for(i2=1;i2<=10;i2++)
            {
            (Corpo do loop)
            }
```

Que resulta no programa:

```
for(j1=-10;j1<=-1;j1++)
      for (j2=1;j2<=10;j2++)
            {
            (Novo corpo do loop)
            }
```

A transformação aplicada inverteu a ordem de execuções das iterações do loop externo, pois temos que $i_1=-j_1$.

Como terceira transformação presente nesta classe de transformações temos a transformação *inner-loop skewing*, que consiste em se somar à variável de um loop mais interno uma constante multiplicada por uma variável de um loop mais interno. Tal transformação não altera a ordem das iterações quando aplicada individualmente, mas é uma transformação interessante quando aplicada em conjunto com as anteriores. Uma matriz unimodular genérica que representa tal transformação é da forma:

$$U = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 1 & \dots & \lambda & \dots & 0 \\ & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ linhaX \\ \\ \\ \end{matrix}$$

$$\underset{colunaY}{\phantom{U = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \end{bmatrix}}}$$

Esta transformação consiste em se somar a variável de loop I_x uma constante λ multiplicada pela variável de loop $I_{(K-Y)}$. A constante λ é chamada de *skewing factor*.

É interessante limitar a classe de transformações unimodulares as três transformações anteriores pois é intuitivo que, para obter a matriz que representa uma seqüência qualquer de transformações de loops, basta multiplicar as matrizes de cada transformação individual, na mesma ordem com que foram executadas as transformações, e se cada transformação individual for unimodular, a transformação gerada pela composição das transformações executadas é também unimodular (det(A.B)=det(A).det(B), recursivamente aplicada). Outro ponto importante é que, qualquer matriz unimodular pode ser obtida pelo produto das matrizes que representam as transformações "loop permutation", "loop reversal" e "inner-loop skewing" ou seja, **qualquer transformação unimodular pode ser obtida pela aplicação sucessiva de transformações do tipo "loop permutation", "loop reversal" e "inner-loop skewing"**, pode-se portanto dizer que as transformações mencionadas anteriormente são as três únicas presentes na classe de transformações unimodulares pois todas as demais podem ser obtidas a partir destas.

3.1.1. Vetores de Dependência

Definição 2: Dada uma dependência entre comandos de uma iteração $I=(i_1, i_2,..., i_K)$ para outra iteração $J=(j_1, j_2,..., j_K)$, definimos o vetor de dependência $V= (v_1, v_2,..., v_K)$ como sendo V=J-I.

Como uma dependência é sempre em relação a uma iteração anteriormente executada, as definições a seguir tornam-se claras:

Definição 3: Um vetor de dependência é dito positivo quando a sua primeira coordenada não nula é positiva.

Desta forma, podemos dizer que todo vetor de dependência válido deve ser positivo.

Notamos que, como uma dada coluna de um vetor de dependência representa uma dependência em relação a uma iteração ocorrida V_x iterações antes no loop com índice igual ao da respectiva coluna, nenhum vetor de dependência **válido** pode ser negativo. Segundo a definição de vetor de dependência positivo dada acima o vetor (1,-1) é positivo. Tal definição faz sentido pois todas as iterações de qualquer loop mais interno ocorrem dentro de uma única iteração de um loop mais externo. Se o corpo do loop necessita de um valor calculado em uma iteração anterior de qualquer loop mais externo em relação a um dado loop mais interno ela, com certeza, já terá sido processada.

Como os vetores de dependência **representam dependências no espaço vetorial das iterações dos loops**, ao contrário dos mapeamentos π, que relacionam dois espaços vetoriais, temos que:

Teorema 1: Aplicada uma transformação unimodular U a um conjunto de k loops perfeitamente aninhados, os novos vetores de dependência do programa transformado, que representam dependências entre iterações no corpo do loop, são obtidos pós-multiplicando os vetores do programa original pela matriz U. Uma dada transformação é válida se todos os vetores de dependência transformados continuam vetores válidos.

O teorema anterior é de fácil compreensão, pois uma transformação é válida se preserva a semântica do programa, e os vetores de dependência fazem parte desta semântica, no sentido que: se uma iteração depende de resultados obtidos em uma iteração anterior, esta iteração deve, no programa transformado, ocorrer após a iteração da qual ela depende. Isto só ocorre se o vetor de dependência que representa aquela dada dependência continuar positivo após a transformação. O próximo teorema tem implicações mais interessantes.

Teorema 2: Um dado loop x de um conjunto de k loops perfeitamente aninhados pode ser executado em paralelo se alguma das coordenadas anteriores à coordenada x de todos os vetores de dependência for positiva ou se a coordenada x for igual a zero.

Ou seja, se existe uma dependência ou ela foi resolvida por ter sido calculada em iterações anteriores de qualquer loop mais externo, ou ela só existe entre iterações de loops mais internos, supostamente seriais.

Apresentamos, a seguir, um exemplo de transformação de programa válida que permite a obtenção de paralelismo.

Dado o programa, supondo a matriz `Matriz` declarada com as dimensões necessárias:

```
for (i1=0;i1<=10;i1++)
      for(i2=0;i2<=10;i2++)
            Matriz[i1][i2]=Matriz[i1-1][i2-1];
```

Tal programa possui, em seu corpo de loop, a dependência (1,1). Portanto loop externo não pode ser executado em paralelo, pois uma dada iteração depende da iteração anterior. Aplicando-se a transformação $U = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$, o novo vetor de dependência do programa transformado é (0,1). Portanto esta é uma transformação válida, e o novo loop externo, do programa transformado a seguir, pode ser executado em paralelo.

```
for (j1=-10;j1<=10;j1++)
      for(j2=max(-j1,0);j2<=min(10-j1,10);j2++)
            Matriz[j1+j2][j2]=Matriz[j1+j2-1][j2-1];
```

O problema de como calcular os novos limites do programa transformado [BANN94] está fora do escopo deste trabalho.

3.2 - Localidade de Dados e Janela de Referências

A importância de se melhorar a localidade de dados está relacionado ao uso eficiente da hierarquia de memória. O princípio da localidade de referência diz que o dado usado mais recentemente tem grande probabilidade de ser acessado novamente num futuro próximo (localidade temporal). Favorecendo-se o acesso a tais dados, aumenta-se o desempenho do sistema. Então, na medida do possível, os elementos recentemente acessados devem ser mantidos em memórias mais rápidas. Devido ao fato de que

memórias mais rápidas são menores, seu uso como memória de armazenamento dos dados de recente acesso, num nível próximo ao processador, e a utilização de memórias maiores (e mais lentas) em níveis mais distantes do processador, permite que se alcance um melhor compromisso entre custo e desempenho. Este tipo de organização é denominada de hierarquia de memória.

Um gerenciamento adequado da memória local ou da memória cache, associado a um código que faça uso mais intensivo de dados localizados, e portanto, passíveis de serem mantidos na memória local, favorece a melhoria de desempenho na medida em que provoca uma reutilização maior de dados que estão nas memórias rápidas. A otimização da localidade de dados, focalizada aqui, é baseado em transformação de programa direcionada à melhoria da localidade.

Note-se que é prioritário que se estabeleça uma métrica ou índice para se caracterizar quantitativamente essa localidade. Tendo disponibilidade dessa ferramenta, pode-se proceder à manipulação do código fonte através de transformações de programa, de modo a se chegar a uma estrutura com localidade ótima. O parâmetro a ser utilizado para a obtenção quantitativa da localidade de dados é o custo da *janela de referência*, tal como definido em [EISE90].

Neste caso, concentraremos a análise sobre a reutilização de elementos em uma matriz, o que não constitue grave restrição, uma vez que em programação para análise científica, a maior carga computacional está na manipulação de matrizes.

As transformações de programa tratadas aqui ficarão limitadas às transformações unimodulares, particularmente o *loop interchange*.

3.2.1. Quantificação da Localidade de Dados

Diz-se que existe uma dependência de dados entre duas referências em um programa, se existir um fluxo de controle entre elas e ambas se referirem à mesma posição de memória. Neste estudo, devido à análise de reutilização de dados, devemos considerar todas as referências feitas à memória, sejam elas por dependência de fluxo, antidependência, dependência de saída e dependência de entrada.

Seja um conjunto perfeitamente aninhado de loops, cujo interior contém referências atômicas em variáveis estruturadas:

```
for (i1=1; i1<=N1; i1++)
  for (i2=1; i2<=N2; i2++)
      ...
        for (ik=1; ik<=Nk; ik++)
          {
      <S1>          ... A[h(i1,i2,...,ik)] ...
      <S2>          ... A[g(i1,i2,...,ik)] ...
          }
```

definimos:

- A: matriz de dimensão d
- h e g: mapeamento Z^k em Z^d
- $C=\prod_{j=1}^{k}[1, N_j]$, $C \subset Z^k$: espaço de iterações com tamanho igual a $N = \prod_{j=1}^{k} N_j$

- $\vec{I} = (i_1,i_2,...,i_k) \in C$: vetor de iteração que especifica os valores correntes dos índices dos loops

Há um tipo especial de dependência muito comum e que será usado na formulação matemática do resultado do custo da janela:

Definição 4: Dependência Uniformente Gerada é a dependência existente entre dois comandos S_1 e S_2 da forma:

```
<S1> ..... X(h(I) + d1)
<S2> ..... X(h(I) + d2)
```

onde h é um mapeamento de Z^k em Z^d.

A ordem em que as diferentes ocorrências da instrução S são executadas podem ser caracterizadas pela *timing function*, que é um mapeamento um-a-um entre C e {1,...,N}. A *timing function* é formalmente definida por:

$$T: C \rightarrow \{1,...,N\}$$

$$\vec{I} \rightarrow T(\vec{I}) = T(i_1,i_2,...,i_k) = \sum_{j=1}^{k}[(i_j - 1)P_j] + 1$$

onde $P_j = \prod_{q=j+1}^{k} N_q$ e $P_k = 1$. Em casos onde nenhuma ambigüidade é possível, o vetor de iterações $\vec{I}$ e o correspondente passo de tempo $t = T(\vec{I})$ serão identificados.

3.2.2. Janela de Referência

A janela de referência, $W_t(\delta_x)$, para uma dependência δ_x: $S_1 \rightarrow S_2$ sobre uma matriz X, no tempo t, é definido como sendo o conjunto de todos os elementos de X que são referenciados por S_1 até t e que serão também referenciados (de acordo com a dependência) por S_2 após t. [EISE90]

3.2.3. Custo e Benefício de uma Janela de Referência

O custo de uma janela de referência associada à dependência δ_x é definida como:

$$Cost(W(\delta_x)) = \max_t |W_t(\delta_x)|$$

onde $|W_t(\delta_x)|$ denota o número de elementos distintos de $W_t(\delta_x)$.

O benefício $Ben(W(\delta_x))$ de uma janela de referência, associada à uma dependência δ_x, é definida como número máximo de referências à dados realizados por S_2 que podem ser executados da memória local ao invés da memória principal.

3.2.4. Janela Aproximada

Uma janela aproximada associada à janela de referência $W_t(\delta x)$ é uma dupla constituída de um mapeamento $\underline{m}$ de Z^k em Z^d e um subconjunto $\underline{W}$ de Z^d tal que:

$$\forall t, W_{t=\vec{I}}(\delta_x) \subset \{X(\vec{j}) \mid \vec{j} \in m(\vec{I}) + W\}$$

O número de elementos de W é chamado de custo da janela aproximada (denotado Cost(W)).

A idéia de se formular a janela aproximada é que a janela fica enquadrada numa estrutura móvel com forma e tamanho constantes. Deste modo, simplifica-se bastante a avaliação do espaço de memória necessária para conter a janela. Então, o impacto da transformação de programa pode facilmente ser analizada, e a tarefa de selecionar a transformação mais apropriada se torna muito mais fácil.

3.2.5. Resultado

Nesta seção apresentaremos o resultado para o caso em que h: $Z^k \rightarrow Z^d$.

Teorema 3: Consideremos o seguinte aninhamento de k loops:

```
for (i1=1; i1<=N1; i1++)
      for (i2=1; i2<=N2; i2++)
            ...
            for (ik=1; ik<=Nk; ik++)
                  ... a(iπ(1),iπ(2),...,iπ(k))...
```

onde π é um mapeamento um-a-um de {1,...,d} sobre o subconjunto de d-elementos de {1,2,...,k}.

A função h é definida por $h(i_1,i_2,...,i_k) = (i_{\pi(1)},i_{\pi(2)},...,i_{\pi(k)})$

Então, chega-se à seguinte aproximação, provada em [EISE90]:

$$\text{Cost(W)} \leq \min\left(\min_{r=1}^{d}\left(\left[\prod_{q=1,q\neq r}^{d} N_{\pi(q)}\right]\frac{1}{P_{\pi(r)}}\left[\sum_{j=1,j\neq\pi(r)}^{k} P_j N_j\right]\right), \prod_{q=1}^{d} N_{\pi(q)}\right)$$

É este o resultado que utilizaremos para quantificar a localidade de dados no estudo de caso descrito mais adiante.

4. INTEGRAÇÃO DOS ENFOQUES

A integração da otimização de paralelismo e localidade de dados tem sido estudada por diversos pesquisadores [KENN93][LI93][MANJI95][MCKI92][WOLF92]. De uma maneira geral, os trabalhos adotam uma estratégia de otimização onde cada aspecto é tratado em separado, em fases distintas do processo de análise e tranformação. Ou seja, não há um enfoque unificado para tratamento da localidade de dados em conjunto com o paralelismo. Uma contribuição deste trabalho é propor uma alternativa a estas estratégias, onde procuramos adotar um esquema iterativo.

Como mencionado anteriormente, faremos o uso de transformações unimodulares. Esta classe de transformações provoca uma alteração na *timing function* T, que pode ser vista como uma forma diferente de proceder-se à varredura do espaço de iterações.

Com o intuito de facilitar a exposição de conceitos sobre a transformação, particularizaremos a análise para o caso das dependências que existem em um programa de multiplicação de matrizes, o qual será posteriormente objeto do exemplo de implementação do algoritmo completo proposto.

Dado a seguinte estrutura de loops:

```
for (i1=0; i1<N1; i1++)
 for (i2=0; i2<N2; i2++)
  for (i3=0; i3<N3; i3++)
    A[iπA(1)][iπA(2)]=A[iπA(1)][iπA(2)]+B[iπB(1)][iπB(2)]*C[iπC(1)][iπC(2)]
```

onde h: $Z^3 \rightarrow Z^2$, com:

$$h_A(i_1,i_2,i_3) = (i_{\pi A(1)}, i_{\pi A(2)})$$
$$h_B(i_1,i_2,i_3) = (i_{\pi B(1)}, i_{\pi B(2)})$$
$$h_C(i_1,i_2,i_3) = (i_{\pi C(1)}, i_{\pi C(2)})$$

Então, a transformação é definida por:

$$I' = I \times U$$

onde U é a matriz unimodular, I é o vetor com ordenamento original dos índices dos loops e I' é o vetor resultante da transformação aplicada.

Definamos a matriz de mapeamento M como:

$$h = I \times M$$

ou seja, a matriz M é aquela que multiplicada pelo vetor de índices fornece o vetor de mapeamento h.

O processo de transformação do ordenamento dos loops deve manter, contudo, o mapeamento em cada dependência, uma vez que é necessário a manutenção da semântica do programa. Assim, procedido à transformação, o mapeamento h calcula-se por:

$$h = I \times U \times U^{-1} \times M = I' \times M'$$

onde $U^{-1} \times M = M'$ é a nova matriz de mapeamento.

Dispondo-se dessas definições podemos montar um algoritmo de otimização da localidade de dados, tendo como critério, a obtenção do menor custo da janela de referência:

```
Algoritmo: otimização integrada de paralelismo e localidadade de dados
Entrada:  número de loops K
          vetores de dependências D
          mapeamentos π
{
/* determina o custo de cada uma das permutações permitidas dos loops */
for ( cada uma das permutações )
   if ( transformação válida )
      determina custo em termos de acessos à memória
   else
      marca transformação com inválida
ordena as permutações válidas em ordem de custo
for ( cada uma das permutações )  {
   obtém novos vetores de dependências
   if ( é possível paralelizar loop externo )
      break;  /* achou transformação adequada */
   }
if ( não é possível paralelizar loop externo )
   obtém transformação que paraleliza loop interno
obtém novos mapeamentos
re-otimiza loops internos em termos de memória
retorna versão do programa com o menor custo
}
```

Deste modo, teremos ao final, um código com a melhor localidade de dados passível de paralelização. Um procedimento mais genérico onde se levará em conta aspectos relacionados com a paralelização, tais como *overheads* de sincronização e de criação dos processos paralelos, e a incorporação de outras transformações está em desenvolvimento.

5. SISTEMA EM DESENVOLVIMENTO

Dispõe-se de um protótipo de um sistema automático para otimização de localidade de dados e de paralelismo. O escopo de transformações disponíveis é restrito às transformações unimodulares e o cálculo da janela de referências é obtido apenas sobre um espaço de iterações retangulares.

Atualmente, uma interface homem-máquina gráfica está em fase de desenvolvimento, através da qual o usuário deste sistema pode especificar restrições e parâmetros ao processo de otimização e, assim, interagir na busca da melhor solução ao processo de reestruturação.

Maiores informações sobre este sistema estarão brevemente disponíveis publicamente via WWW no endereço http://www.lsi.usp.br.

6. ESTUDO DE CASO

Para estudar a viabilidade de um sistema automático para otimização de localidade de dados e de paralelismo, executamos uma análise detalhada de um problema específico: uma multiplicação de matrizes não quadradas. Então, adotamos o seguinte caso de estudo: "qual a melhor versão paralela para um programa de multiplicação de duas matrizes não quadradas de dimensões 1024×128 e 128×256".

O estudo foi realizado em uma máquina Silicon Graphics Power Series 4D/480 VGX e os programas foram compilados com o **cc** sem opção de otimização. A contagem de tempo foi obtida com o uso da chamada **times**.

Aplicando apenas a transformação de *loop interchange*, obtemos as 6 possíveis permutações dos 3 loops que implementam a multiplicação de matrizes. Todas estas versões podem ter seus loops externos paralelizados.

Aplicando-se as características deste programa no sistema em desenvolvimento, obtivemos a seguinte saída para o valor do custo da janela de referências:

Tabela 1 - Custo de cada uma das versões do programa em estudo.

versão	ijk	ikj	jik	jki	kij	kji
custo	33.155	33.153	263.425	263.429	132.225	132.233

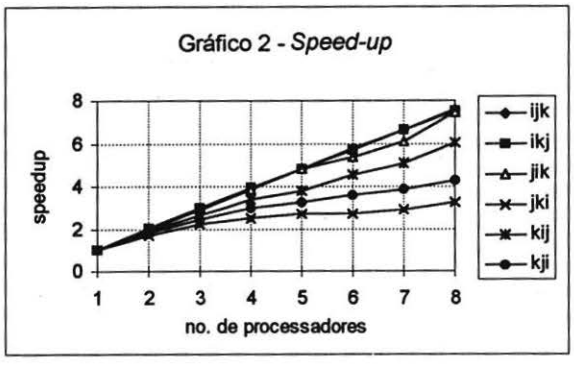
E executando-se cada uma das versões em uma máquina real, obtivemos os seguintes tempos de execução em função do número de processadores utilizados. O gráfico 1 resume estes números.

Tabela 2 - Tempo de execução de cada uma das versões do programa em estudo (em seg).

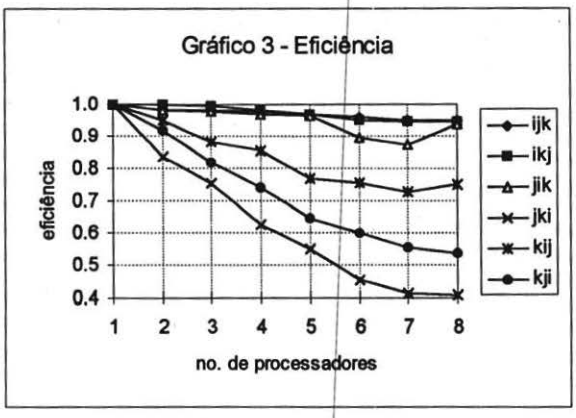
versão x nº processadores	1	2	3	4	5	6	7	8
ijk	65.53	33.32	22.29	16.72	13.57	11.40	9.86	8.62
ikj	36.42	18.21	12.19	9.27	7.52	6.39	5.50	4.82
jik	71.45	36.32	24.36	18.42	14.86	13.30	11.68	9.55
jki	151.85	90.93	67.07	60.50	55.38	55.67	52.18	46.62
kij	148.66	78.15	56.17	43.44	38.81	32.84	29.21	24.73
kji	258.54	140.80	105.25	87.04	80.01	72.07	66.41	60.36

Comparando os resultados das tabelas 1 e 2 vemos que nosso sistema identificou corretamente a versão ikj como a melhor dentre todas. Mas houve um erro em relação às versões jik/jki e kij/kji. A explicação para isto é o fato das versões kij/kji necessitarem usar mecanismos de exclusão mútua para atualização da matriz produto; isto causado por uma série de dependências de dados existentes nestas versões, o que não há nas outras. Isto causou a estimativa de um valor de custo menor.

Analisando-se a curva de *speed-up* correspondente (gráfico 2), vemos que a versão ikj é a que apresenta melhores características de escalabilidade.



O gráfico 3 mostra a curva de eficiência, e mais uma vez, a versão ikj é a que possui um melhor aproveitamento dos processadores.



Assim, podemos dizer que a estratégia é funcional, bastando que sejam propostos novos componentes à função custo que englobem os aspectos relacionados à paralelização do programa. Esforços neste sentidos estão sendo conduzidos e novos resultados são esperados em breve.

7. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma estratégia unificada para a otimização automática de paralelismo e localidade de dados. Embora atualmente o escopo de transformações e de análise de localidade sejam limitados, o estudo de caso apresentado mostrou que nossa estratégia é bastante funcional e, mediante futuros melhoramentos, deve se tornar mais preciso.

Como trabalhos seguintes podemos citar a incorporação imediata de algumas outras transformações, tais como, o *tiling*, o *loop distribuition* e o *loop fusion*, que segundo diversos estudos são muito eficientes quanto a otimização de acessos a dados [CARR94] [KENN93] [MANJ95] [MCKI92] [MIDO95] [WOLF92], e a extensão do cálculo da janela de referências para espaços de iterações não retangulares [WIND92].

Os bons resultados obtidos nos animam a continuar nossas pesquisas. Como outros tópicos que vem sendo estudados e que serão abordados em um futuro próximo estão a análise de programas para sistemas de memória compartilhada distribuída ("*distributed shared memory*"), programas orientados a objeto paralelos e modelos de programação de memória distribuída.

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.

[BANE88] BANERJEE. U. **Dependence analysis for supercomputing**. Kluwer Academic Publishers, 1988.

[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.

[EISE90] EISENBEIS, C. et alii. **A strategy for array management in local memory**. Rapports de Recherche N° 1262. Institut National de Recherche en Informatique et en Automatique (INRIA), France. Juillet 1990.

[KELL95] KELLY, W. & PUGH, W. **A unifying framework for iteration reordering transformations**. Technical Report CS-TR3430. Department of Computer Science, University of Maryland. February, 1995.

[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.

[KULK94] KULKARNI, D. et alii. **A generalized theory of linear loop transformations**. Technical Report CSRI-317. Computer Systems Research Institute, Department of Computer Science, Department of Electrical and Computer Engineering, University of Toronto, Canada. December 1994.

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

[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. & SATO, L. M. **Integrando as otimizações de acessos a dados e de paralelismo**. Submetido ao VII Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho (SBAC-PAD'95), 1995.

[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.

[WIND92] WINDHEISER, D. **Optimisation de la localité de doneés et du parallélism à grain fin**. These de Docteur. Université de Rennes 1, France. 1992.

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