Análise da Otimização de Acessos à Memória

Edson Toshimi Midorikawa

Laboratório de Sistemas Integráveis
Departamento de Engenharia Eletrônica
Escola Politécnica da USP
Av. Prof. Luciano Gualberto, trav.3, nº158
05508-900 - São Paulo - SP
tel/fax: (011) 211-4574
e-mail: emidorik@lsi.usp.br

RESUMO

Projetos de processadores modernos têm enfocado o aumento do desempenho computacional, levando a um desbalanceamento entre a velocidade de computação e a velocidade da memória. De forma a aliviar esta diferença de velocidades, hierarquias de memória complexas têm sido elaboradas. Este trabalho apresenta um estudo onde se mostra a aplicabilidade de técnicas para melhorar o padrão de acessos à memória de forma a aumentar a localidade de referências, e assim, utilizar mais eficientemente todos os níveis da hierarquia de memória, desde os registradores até os módulos de memória remota. Os resultados obtidos revelam que o uso de um compilador para a otimização de acessos é bastante promissor.

ABSTRACT

Modern processor design strategies have focused on increasing the computational power. The result is an imbalance between computational speed and memory speed. In order to aliviate this problem, complex memory hierarchies have been proposed. This paper presents a study on application of some techniques to enhance memory access patterns of the programs in order to make them more localized and to use the memory hierarchy efficiently. The results reveal that the use of a compiler to optimize memory accesses promises very good results.

1. Introdução

O que se tem notado nos últimos anos é o crescente aumento de desempenho computacional dos processadores comerciais. Os novos processadores, como o Pentium da Intel ou o Alpha da DEC, são aproximadamente uma ordem de grandeza mais poderosos que os processadores da geração anterior. Esta nova geração possui uma freqüência de operação de 150 a 270 MHz, diversas unidades funcionais com recursos de pipeline e suporte a execução de múltiplas instruções.

O resultado deste desenvolvimento é o aumento no número de ciclos para o acesso à memória. É comum termos uma latência de 10 a 20 ciclos, o que causa um desbalanceamento muito grande entre a velocidade na qual o processamento pode ser executado e a velocidade na qual os operandos podem ser enviados aos processadores. Além disso, as novas arquiteturas superescalares demandam uma banda de acesso à memória ainda maior, com a execução simultânea de diversas instruções de máquina. Por exemplo, os processadores POWER-2 da IBM têm capacidade de executar até 6 instruções simultaneamente.

Para resolver este problema, os fabricantes têm optado pela adoção de complexas hierarquias de memória. Por exemplo, o KSR-1 tem uma hierarquia de memória principal composta de três níveis. Já os processadores MIPS R4000 apresentam dois níveis de memória cache. Embora o uso de memórias cache atenue este desbalanceamento, verifica-se que aplicações científicas têm um desempenho muito baixo quando seu "working set" é maior que o tamanho do cache [6], ou seja, se as aplicações não forem estruturadas de forma a aproveitarem a estrutura da hierarquia de memória da máquina alvo. Desta maneira, é vital que os programas façam uso adequado dos diversos níveis da hierarquia.

Para aproveitar melhor estas complexas hierarquias de memória, muitos programadores tiveram de re-estruturar seus códigos manualmente de forma a extraírem o maior desempenho possível [8][9]. Contudo este mecanismo é muito tedioso e demorado, além de criar uma versão específica para um determinado sistema. Assim, quando uma nova geração for lançada, todo este trabalho deve ser refeito para a nova arquitetura.

Este trabalho mostra como utilizar de maneira eficiente hierarquias de memória complexas, com o emprego da tecnologia desenvolvida para a paralelização automática de programas [16]. Aplicando-se conceitos desenvolvidos há muito tempo nesta área, mostra-se como efetuar a reestruturação de programas, visando um melhor desempenho em relação aos acessos à memória. A seção 2 inicia o trabalho definindo o problema da latência de memória. A seguir, métodos para solucionar este problema são analisados e a técnica abordada neste trabalho é apresentada. A seção 4 descreve o estudo efetuado, onde se mostra a efetividade da técnica empregada, obtendo-se um

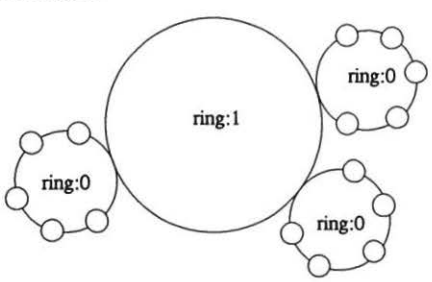
grande aumento de desempenho em uma rotina com a aplicação de transformações. Finalmente, são apresentados alguns trabalhos relacionados e conclusões gerais dos resultados obtidos.

2. O Problema da Latência da Memória

De forma a atender a crescente demanda de processamento, máquinas paralelas com centenas ou milhares de processadores estão chegando ao mercado. Em tais máquinas, a memória é distribuída fisicamente pelos processadores, que se comunicam entre si via uma rede de interconexão (por exemplo, rede omega, *fat tree*, *butterfly*). Algumas destas arquiteturas fornecem um paradigma de programação com memória compartilhada, surgindo assim o problema da latência da memória.

De forma a realizar um acesso a uma posição de memória remota [1], o processador precisa enviar uma mensagem através da rede de interconexão. Esta mensagem trafega pela rede até alcançar o módulo de memória desejado. O módulo lê o valor requisitado e envia de volta uma mensagem contendo o dado para o processador. O intervalo de tempo entre o envio da mensagem com a requisição e o retorno da mensagem resultante é chamado *latência de acesso à memória*. Tipicamente, este tempo é algumas ordens de grandeza maior que o tempo de acesso a uma memória local. Desta forma, é conveniente evitar acessos remotos à memória.[2]

Os grandes gargalos nas máquinas modernas são o tempo de acesso dos módulos de memória e a banda da rede de interconexão, que apresentam uma velocidade bem baixa em relação à velocidade do processador. A figura 1 abaixo mostra os tempos de acesso à memória do KSR-1 da Kendall Square, uma máquina paralela com memória compartilhada com cache coerente. Observando os tempos de acesso de cada nível da hierarquia, constata-se que o tempo de acesso a uma posição de memória remota presente em outro anel é da ordem de 30 vezes maior que o de uma posição de memória local.



Latências da Memória		
memória	ciclos	capacidade
cache	2	256KB
memória local	18	32MB
ring0	126	1GB
ring1	600	34GB

Fig.1 - Latências de memória na KSR-1.

1. ou seja, em uma posição de memória pertencente ao módulo de memória de outro processador.
2. este problema também ocorre em máquinas com o paradigma de programação de memória distribuída (por exemplo, que usam a linguagem HPF), onde os dados são trocados entre os processadores via troca de mensagens.

3. Otimização dos Acessos à Memória

Diversas soluções têm sido propostas para o problema da latência da memória para máquinas paralelas escaláveis. Estas soluções variam na forma como eles abordam o problema e como eles são implementados. Descreve-se a seguir as diversas estratégias propostas na literatura, que de alguma forma procuram minimizar a latência da memória.

3.1. Estratégias Possíveis

A figura 2 apresenta um resumo das diversas alternativas propostas na literatura (veja por exemplo [5]). O uso de memória cache ("*caching*") procura eliminar acessos redundantes à memória. *Multithreading* emprega um conceito semelhante à multiprogramação, permitindo que o processador passe a executar um outro thread enquanto espera o término do acesso a uma posição de memória remota. *Weak consistency* é um modelo de consistência de memória, onde é permitida a sobreposição de algumas requisições de escrita a uma posição de memória sem que haja um restrição de ordem. A antecipação do uso de um dado por parte do programa é explorada pela técnica de *prefetching*. O principal objetivo da técnica de *layout* é rearranjar os dados na memória de forma a mantê-los bem próximos ao processador que irá usá-los [20]. E agregação ("*aggregation*") procura "empacotar" grandes volumes de dados e distribuí-los pelos processadores da máquina.

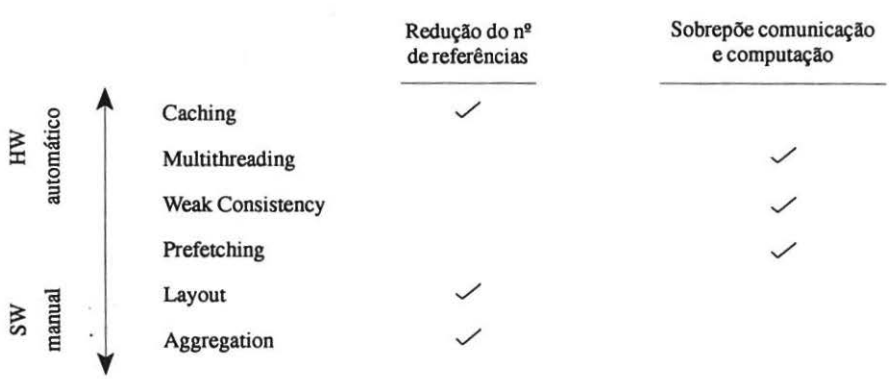


Fig.2 - Mecanismos para reduzir o impacto da latência da memória.

Os mecanismos próximos ao topo da figura acima são normalmente automáticos, ao passo que aqueles presentes na parte inferior são normalmente explorados pelo programador ou por um compilador otimizador. Todas estas alternativas têm suas limitações. Por exemplo, as memórias

cache precisam ser mantidas coerentes entre si, multithreading requer um hardware muito complexo e a escolha de uma boa distribuição de dados pelos processadores é limitada a poucas classes de aplicações.

Neste trabalho, procura-se abordar uma outra alternativa, complementar àquelas apresentadas acima, onde se busca uma otimização no acesso à memória através de um padrão de acesso aos dados mais localizado, ou seja, procura-se melhorar a localidade de referências do programa de forma a aproveitar melhor toda a hierarquia de memória presente nas modernas máquinas. Para tal utiliza-se de conceitos e métodos desenvolvidos para a paralelização automática de programas.

3.2. Conceito de Reuso de Memória

O conceito de dependência de dados[3] quando aplicado para a otimização de acessos e à gerência da hierarquia de memória pode ser considerado como um instrumento poderoso para verificar possibilidades de reuso de posições de memória e uma medida da localidade das referências.

A reutilização pode ser de dois tipos: temporal e espacial [7]. Uma *reutilização temporal* ocorre quando um comando do programa acessa uma posição de memória referenciada anteriormente. A *reutilização espacial* ocorre quando um acesso se refere a uma posição em endereço próximo a outro acesso anterior. Ela pode ser aplicada sobre memórias cache (onde a localidade espacial se refere às linhas do cache) e sobre as páginas (onde ela se refere às posições consecutivas de uma página física de memória).

Seja o seguinte exemplo:

```
for (i=0; i<N; i++) {
    a[i] = a[i-5] + b[i];
}
```

Tem-se no loop acima que o valor definido pela referência `a[i]` é reutilizada pela referência `a[i-5]` cinco iterações[4] mais tarde (reultilização temporal). E as referências a `b[i]` têm uma reutilização espacial com o acesso a elementos consecutivos da mesma linha de cache (ou da página de memória).

Notamos acima então que temos dois tipos de reuso de posições de memória. A primeira é aquela onde uma mesma posição de memória é acessada novamente em uma diferente iteração. A

3. para uma explicação sobre dependência de dados, transformações e compiladores paralelizantes consulte as referências [4][16][22][24]

4. Aqui pode-se usar o conceito de distância de dependência como uma medida da localidade temporal. Uma distância pequena pode ser considerada como um forte indício que aquela dependência de dados deve causar acessos "cacheáveis", por exemplo.

outra é aquela onde uma referência causa o encontro de outra referência, presente na mesma linha de cache, por exemplo.

Este tipo de informação pode ser utilizada para melhorar a gerência da hierarquia de memória, sendo possível explorar sua aplicação em diversos níveis como o reaproveitamento de registradores, melhora na taxa de acerto do cache e diminuição da taxa de faltas de página.

3.3. Uso de Transformações

Baseado na análise de dependência de dados, diversas transformações podem ser aplicadas de forma a melhorar o desempenho de programas "memory-bound".[5] Introduz-se aqui algumas destas transformações. Embora algumas destas transformações também tenham uso na paralelização de programas, como o *loop interchange*, elas são analisadas aqui sob o ponto de vista dos acessos à memória.

3.3.1. Scalar Replacement

Scalar replacement visa auxiliar o compilador na análise de fluxo para a alocação de variáveis em registradores. Compiladores atuais não conseguem manipular variáveis multidimensionais[6]. No seguinte trecho de código

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i] = a[i] + b[j];
```

a referência `a[i]` poderia ser mantida num registrador durante a execução do loop em `j`. Para facilitar sua otimização, o código acima pode ser reescrito como

```
for (i=0; i<N; i++) {
    T = a[i];
    for (j=0; j<N; j++)
        T = T + b[j];
    a[i] = T;
}
```

onde mesmo os compiladores mais simples podem alocar `T` em um registrador durante a execução do loop interno.

3.3.2. Unroll-and-jam

O principal objetivo desta transformação é aumentar a quantidade de computação efetuada no corpo do loop mais interno de um trecho de código, sem aumentar proporcionalmente o número de referências à memória. Por exemplo

5. consulte [3] para uma lista da maioria das transformações existentes.
6. mesmo vetores unidimensionais com expressões de índices simples.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i] = a[i] + b[j];
```

pode ser transformado em

```
for (i=0; i<N; i+=2)
    for (j=0; j<N; j++)  {
        a[i] = a[i] + b[j];
        a[i+1] = a[i+1] + b[j];
    }
```

O trecho de código anterior apresenta um acesso à memória por operação de ponto flutuante. Já o código otimizado executa duas operações em ponto flutuante por acesso à memória principal.[7]

3.3.3. Loop interchange

Esta transformação muito utilizada em paralelizadores tem também sua aplicação na otimização da memória. Considere o seguinte exemplo

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a = a + b[j][i];
```

A execução deste trecho de código causará a carga da coluna `j` da matriz `b` na memória cache. Isto torna muito difícil a reutilização das linhas do cache, pois o elemento seguinte só será referenciado na próxima iteração do loop `i` [8]. Desta forma a simples reordenação dos loops

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a = a + b[j][i];
```

fará com que elementos contíguos da matriz `b` sejam acessados sucessivamente.

3.3.4. Tiling

No caso geral, tiling transforma um trecho de programa com n loops aninhados em um trecho equivalente com 2n loops, onde os n loops mais internos executam um número determinado de iterações. Ele divide o espaço de iterações em múltiplos blocos (*tiles*)[9] [23] que podem aproveitar melhor a hierarquia de memória. No seguinte exemplo,

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

7. considerando-se que o acesso a a `[i]` proporcione a busca de a `[i+1]` da memória para a mesma linha da memória cache.

8. lembre-se que a linguagem C armazena matrizes por linha, ao contrário de outras linguagem como Fortran onde o armazenamento é organizado pelas colunas.

podemos dividir o espaço de iterações NxN em pequenos blocos de tamanho SxS, resultando em

```
for (jj=0; jj<N; jj+=S)
  for (kk=0; kk<N; kk+=S)
    for (i=0; i<N; i++)
      for (j=jj; j<min(jj+S,N); j++)
        for (k=kk; k<min(kk+S,N); k++)
          c[i][j] += a[i][k]*b[k][j];
```

Vemos que com o espaço de iterações dividido em pequenos blocos, torna-se possível melhorar a reutilização de dados através da aplicação conjunta de outras transformações.

4. Estudo e Análise dos Resultados

Nesta seção apresenta-se um estudo de caso, onde é feita uma análise da aplicação de algumas das transformações descritas na seção anterior, visando otimizar o acesso à hierarquia de memória. Para efetuar este estudo foi escolhido um programa que realiza uma multiplicação de matrizes.

4.1. Caso de Estudo Escolhido

Escolheu-se o problema da multiplicação de matrizes por diversas razões: uma delas foi devido à sua grande utilização nas aplicações científicas [9], mas a maior delas foi a sua estrutura simples, tornando-se fácil ilustrar a aplicação das técnicas a seguir.

O problema estudado consistiu em inicializar duas matrizes 512 x 512, e multiplicá-las.

4.2. Ambiente de Estudo

Os testes foram realizados em três diferentes máquinas: uma servidora Silicon Graphics Power Series 4D/480 VGX com 8 processadores MIPS R3000A (40 MHz), uma estação de trabalho Silicon Graphics Indigo 4000 XS24 com um processador MIPS R4000 (100 MHz) e uma estação de trabalho Sun SPARCstation 10 Modelo 50 com um processador SuperSPARC (45 MHz).

Na SGI 4D/480, cada processador possui um cache de instruções de 64 Kbytes, um cache de dados de primeiro nível de 64 Kbytes e um cache de dados secundário de 1 Mbyte. A memória principal do sistema é de 64 Mbytes. A arquitetura do sistema é esquematizada na fig.3 abaixo.

9. daí o fato de ser conhecido também como blocagem ("blocking").

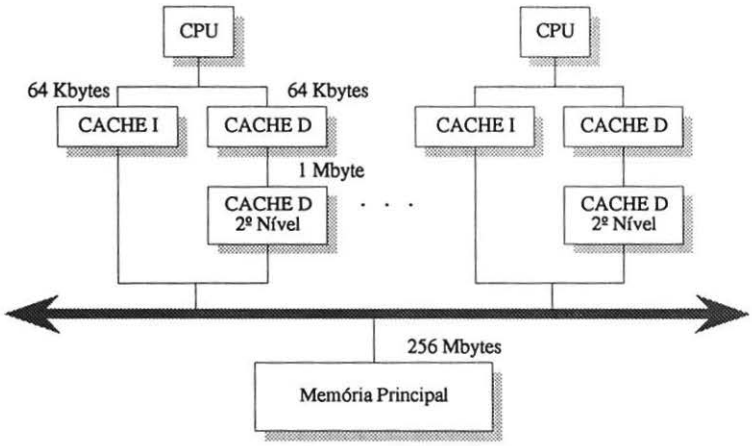


Fig.3 - Esquema simplificado da arquitetura da Power Series 4D/480 VGX.

A SGI Indigo apresenta uma hierarquia de memória semelhante [12], mas com um cache de instruções e um cache de dados com 8K bytes cada e o cache secundário de 1M bytes sendo utilizado para armazenar dados e instruções. A estação Sun SPARCstation 10 Modelo 50 possui um cache de instruções de 16K bytes e um cache de dados de 20K bytes [19].

Os programas foram codificados em linguagem C e foi utilizado o sistema de programação dos sistemas IRIX e SunOS. O tempo de execução dos programas foi medido com o utilitário *time* e os programas foram compilados sem o uso do otimizador de código[10] de forma a evidenciar o efeito das transformações aplicadas. Embora se dispusesse de uma máquina paralela, os estudos foram conduzidos com os programas seqüenciais de cada versão analisada.

4.3. Estudo da Aplicação das Transformações e Resultados Obtidos

Apresenta-se aqui uma análise do estudo realizado e os principais resultados obtidos. A partir do algoritmo tradicional de multiplicação de matrizes, aplicaram-se manualmente diversas transformações e analisaram-se seus impactos no tempo de execução total do programa.

De forma a avaliarmos melhor a otimização aos acessos à hierarquia de memória, analisaram-se os seguintes casos:

- efeito da blocagem (*tiling*) do espaço de iterações somente
- uso conjunto da blocagem com outras transformações existentes

10. utilizando-se a opção de compilação -g.

Este trabalho visa elaborar um estudo onde se avalia o emprego da transformação de blocagem para a melhora do padrão de acessos à memória.

A definição do produto da multiplicação de duas matrizes C = A × B, onde A = $(A_{i,j})$, B=$(B_{i,j})$ e C=$(C_{i,j})$ pode ser dada por: [10]

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j} \qquad \text{para } 1 \leq i,j \leq n.$$

Normalmente, a implementação tradicional utiliza três loops aninhados, que são a tradução direta da definição acima.

```
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                for (k=0; k<n; k++)
S:                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

e denominaremos esta versão original com a designação ijk.

4.3.1. Estudo do efeito da blocagem:

Assume-se que os elementos da matriz c são iniciados com zero. O comando S dentro do loop mais interno forma o produto interno da i-ésima linha de a e a j-ésima coluna de b. Desta forma, o cálculo dos elementos de c envolve n^2 produtos internos. A fig.4 mostra como é este processo de multiplicação (versão ijk).

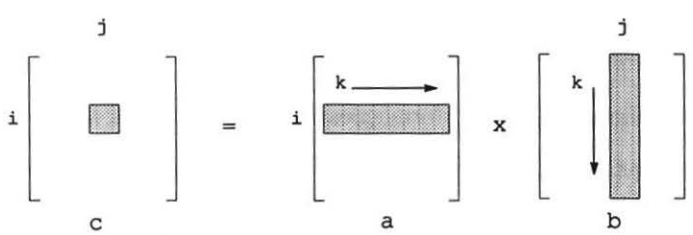


Fig.4 - Multiplicação de matrizes tradicional.

Uma análise do padrão de acesso mostra que as matrizes a e b são acessadas por linha e por coluna, respectivamente. Devido ao esquema de alocação dos elementos na memória, conclui-se que o acesso à matriz b não está correto sob o ponto de vista da localidade espacial. A primeira tentativa efetuada foi aplicar uma blocagem "ingênua" do algoritmo acima (versão tiling jik), onde

os loops `j` e `i` foram trocados de ordem e divididos em blocos. [11] Notou-se uma melhora sensível de desempenho da ordem de 60% em todas as máquinas. A tabela 1 abaixo mostra o tempo de execução de ambas as versões nas três máquinas analisadas.

Tabela 1 - Aplicação "ingênua" do tiling.

Máquina	SGI 4D/480		SGI Indigo		SPARCstation 10 / 50	
Versão	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhòra
ijk	473,5	63,7%	296,1	66,6%	203,2	69,6%
tiling-jik	171,8	2,8x	98,8	3x	62,3	3,3x

Esta melhora de desempenho pode ser explicada observando-se uma maior localidade de referências devido ao padrão de acessos às matrizes (fig.5). Note que mesmo um acesso aparentemente "errado" das matrizes `b` e `c` (por coluna), houve uma melhora significativa no tempo de execução.

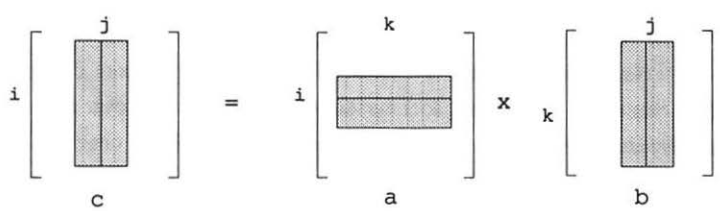


Fig.5 - Multiplicação de matrizes com loop interchange aplicado nos loops i e j com blocagem do espaço de iterações de 2x2.

Analisando melhor o padrão de acessos da versão acima, partimos para uma aplicação mais eficiente da transformação. Conciliando a organização dos elementos na memória, os padrões de acesso e características de hardware (como o tamanho da linha do cache), dividiu-se o espaço de iterações de forma diferente, com os loops na ordem original, obtendo a versão tiling-ijk. Esta versão se mostrou superior a anterior, com uma melhora de desempenho da ordem de 75% em relação à versão original. Os tempos de execução estão mostrados na tabela 2. Esta melhora pode ser bem representada considerando que esta última é cerca de 4,5 vezes mais rápida que a versão original.

11. Convém ressaltar que aqui foi aplicada a transformação de tiling visando um melhor uso dos registradores dos processadores. Um estudo semelhante visando a memória cache foi efetuada também, mas por falta de espaço não será abordado neste trabalho.

Tabela 2 - Aplicação mais eficiente do tiling.

Máquina	SGI 4D/480		SGI Indigo		SPARCstation 10 / 50	
Versão	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora
ijk	473,5	76,1%	296,1	78,3%	203,2	79,7%
tiling-ijk	113,4	4,2x	64,1	4,3x	41,7	4,9x

Analisando o padrão de acessos vemos porque esta versão é muito melhor. A figura abaixo ilustra como as matrizes são acessadas.

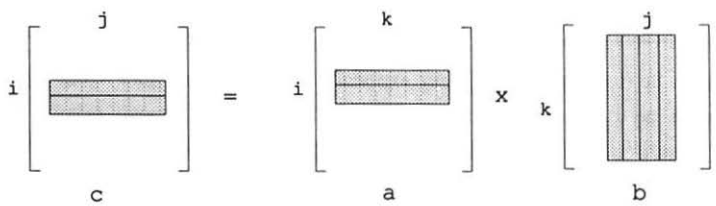


Fig.6 - Multiplicação de matrizes da versão ijk com blocagem do espaço de iterações.

Daí, face a este resultado promissor resolveu-se efetuar uma compilação com a chave de otimização ativada[12] (versão tiling-ijk.ot). O resultado obtido foi surpreendente, com uma melhora de desempenho de cerca de 84% em relação à versão original. Para comprovar a efetividade das transformações aplicadas, compilou-se a versão original também com a chave de otimização habilitada (ijk.ot) e obteve-se uma melhora bem menor (de 15 a 35%). Os tempos de execução são mostrados na tabela 3. A diferença entre as versões otimizadas é da ordem de 80 %.

Tabela 3 - Comparação com compilação otimizada.

Máquina	SGI 4D/480		SGI Indigo		SPARCstation 10 / 50	
Versão	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora
ijk	473,5		296,1		203,2	
ijk.ot	342,1	27,7%	250,5	14,41%	132,8	35,38%
tiling-ijk.ot	64,2	86,4%	46	84,5%	33,8	83,5%

4.3.2. Estudo do efeito conjunto da blocagem com outras transformações:

A seguir realizou-se um estudo do comportamento do tempo de execução com a aplicação conjunta da transformação de blocagem com outras, como loop interchange, scalar replacement (sr)

12. utilizando-se a opção de compilação -O3.

e unroll-and-jam (uj). A tabela 4 abaixo mostra a evolução dos tempos de execução à medida que as transformações foram sendo aplicadas. A nomenclatura segue o seguinte esquema: as primeiras três letras descrevem a ordem de aninhamento dos loops, e as seguintes apresentam as transformações aplicadas.

Tabela 4 - Tempos de execução das diversas versões de multiplicação de matrizes.

Máquina	SGI 4D/480		SGI Indigo		SPARCstation 10 / 50	
Versão	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora
ijk	473,5		296,1		203,2	
ikj	194,4	58,9%	110,6	62,6%	123,4	39,3%
ikj-sr	161,7	65,9%	90,1	69,6%	95,8	52,9%
ikj-uj	141,5	70,1%	96,9	67,3%	112,2	44,8%
ikj-uj-sr	123,4	73,9%	85,3	71,2%	83,8	58,8%
ikj-uj-sr-bl	92,2	80,5%	49	83,5%	62.9	69,1%

Analisando os valores acima, vemos que a versão final (ikj-uj-sr-bl) apresenta uma melhora de desempenho de cerca de 75% (de 69,1% a 83,5%), representando uma velocidade de execução cerca de 4 vezes maior (de 3,3 a 6,1). Convém notar também que a aplicação final da transformação de blocagem resultou em uma melhora adicional de desempenho de pelo menos 25 % em relação à versão anterior (ikj-uj-sr), onde foram aplicadas as outras transformações.

Uma análise com compilação otimizada, aumenta mais ainda a diferença em relação à versão original, resultando numa melhora de um fator de aproximadamente 6,3. Aqui, a melhora adicional da blocagem foi de aproximadamente 10%. Os valores dos tempos de execução estão na tabela 5 abaixo.

Tabela 5 - Tempos de execução das diversas versões otimizadas.

Máquina	SGI 4D/480		SGI Indigo		SPARCstation 10 / 50	
Versão	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora	Tempo total (em segundos)	Melhora
ijk.ot	342,1		250,5		132,8	
ikj.ot	77,2	77,4%	40	84%	48,6	63,4%
ikj-sr.ot	-	-	-	-	42,5	68%
ikj-uj.ot	-	-	35	86%	47,7	64,1%
ikj-uj-sr.ot	59	82,8%	35	86%	40,7	69,4%
ikj-uj-sr-bl.ot	54,3	84,1%	30,3	87,9%	30,2	77,3%

É interessante notar a diferença entre as versões compiladas com a chave de otimização habilitada: a aplicação das transformações possibilitou ao compilador a geração de uma versão 8.7 vezes mais rápida que a versão original (ijk) e 6,3 vezes mais rápida que a versão tradicional otimizada (ijk.ot).

Embora a aplicação destas transformações tenha sido feita manualmente neste estudo, sua automatização não apresenta dificuldade, visto que ela se baseia numa extensão das técnicas utilizadas atualmente pelos compiladores paralelizantes.

5. Trabalhos Relacionados

Abu-Sufah em seu trabalho de doutorado [1] foi o pioneiro na utilização da teoria de compiladores paralelizantes para a gerência de memória, mas seu enfoque se concentra na diminuição das operações de paginação entre a memória principal e o disco. Já Steve Carr [7] possui um enfoque semelhante a este trabalho, mas ele concentra suas análises para otimizações visando um melhor aproveitamento da memória cache. Este dois trabalhos enfocam apenas sistemas monoprocessadores tradicionais.

Outros grupos de pesquisa trabalham em outros tópicos como prefetching de dados entre a memória global e local, alocação de registradores vetoriais e cache prefetching. Granston [11] propõe um sistema híbrido para gerência de memórias cache controlado por hardware e software, o Priority Data Cache. Rogers [18] estuda a localidade de referência em máquinas com memória distribuída. Li e Pingali [13] estudam técnicas específicas para máquinas NUMA, ao passo que Anderson e outros [2] trabalham em técnicas para máquinas com memória distribuída. Outros trabalhos similares são [6][21].

Este trabalho é um estudo preliminar para a aplicação da tecnologia de compiladores paralelizadores para uma otimização da hierarquia completa de memória de sistemas paralelos. Trabalhos anteriores fazem outras análises similares a esta [14][15].

6. Conclusão

Este trabalho apresentou um estudo de técnicas para melhoria da localidade de referências de um programa. Para tal foram aplicados conceitos da área dos compiladores paralelizantes que tem quase 25 anos de desenvolvimento. Descrevendo-se como os conceitos relacionados podem ser aplicados neste novo contexto, mostrou-se como estas técnicas podem levar a uma melhora significativa de desempenho.

Isto prova que estas técnicas podem ser facilmente incorporadas em qualquer compilador paralelizante existente atualmente, como o Parafrase-2 [17]. Trabalhos neste sentido vêm sendo desenvolvidos.

Outro assunto que vem sendo objeto de pesquisa é a análise do acesso à memória de programas paralelos, onde são importantes aspectos como escalonamento de processos, coerência e poluição de memórias cache e efeito da multiprogramação sobre a velocidade de execução, entre outros.

Agradecimentos

Gostaria de agradecer a todos os colegas, que de alguma forma ajudaram na realização deste trabalho. Em especial, à Dra. *Liria Matsumoto Sato* e ao Prof. *Sérgio Takeo Kofuji* pelas críticas, sugestões e comentários das versões preliminares. Agradeço também ao LSI-EPUSP, na pessoa do Prof. Dr. *João Antonio Zuffo*, pelo suporte à pesquisa e pelo uso das máquinas (servidor Silicon 4D/480 VGX, Silicon Indigo 4000 XS24 e Sun SPARCstation 10 Modelo 50).

Referências Bibliográficas

[1] ABU-SUFAH, W. **Improving the performance of virtual memory computers**. PhD Thesis. University of Illinois at Urbana-Champaign, 1978.

[2] ANDERSON, J. M. et al. Global optimizations for parallelism and locality on scalable parallel machines. In: SIGPLAN Conference on Programming Language Design and Implementation, 1993. **Proceedings**. p.112-25.

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

[4] BANERJEE, U. **Dependence analysis for supercomputing**. Kluwer Academic Publ., 1988.

[5] BOOTHE, R. F. **Evaluation of multithreading and caching in large shared memory parallel computers**. PhD Thesis, University of California at Berkeley, 1993.

[6] CARR, S.; KENNEDY, K. Compiler blockability of numerical algorithms. In: International Conference in Supercomputing (ICS 92), 1992. **Proceedings**. p.114-24.

[7] CARR, S. **Memory-hierarchy management**. PhD. Thesis. Rice University, 1993.

[8] DONGARRA, J.J. et al. A set of level 3 basic linear algebra subprograms. **ACM Transactions on Mathematical Software**, v.16, n.1, p1-17. 1990.

[9] DONGARRA, J. J. Linear algebra library for high-performance computers: a personal perspective. **IEEE Parallel & Distributed Technology**, v.1, n1., p.17-24. Feb. 1993.

[10] GOLUB, G. H.; VAN LOAN, C.F. **Matrix computations**. 2.ed. John Hopkins University Press, 1989.

[11] GRANSTON, E. D. **Reducing memory access delays in large-scale, shared-memory multiprocessors**. PhD Thesis. University of Illinois at Urbana-Champaign, 1992.

[12] KANE, G. & HEINRICH, J. **MIPS RISC architecture**. Prentice-Hall, 1992.

[13] LI, W. & PINGALI, K. Access normalization: loop restructuring for NUMA computers. **ACM Trans. on Computer Systems**, v.11. n.4, p.353-75, Nov. 1993.

[14] MIDORIKAWA, E. T. **Gerência de memória para um sistema de computação de alto desempenho**. Dissertação de Mestrado, Escola Politécnica, Universidade de São Paulo, 1991.

[15] MIDORIKAWA, E. T. Aplicação da tecnologia de compiladores paralelizantes para gerência de memória: um estudo de caso. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 5, Florianópolis, 1993. **Anais**. Vol.1, p.259-74.

[16] POLYCHRONOPOULOS, C.D. **Parallel programming and compilers**. Kluwer Academic Publ., 1988.

[17] POLYCRONOPOULOS, C. D. et al. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In: GELERNTER, D. et al., eds. **Languages and compilers for parallel computing**. MIT Press, 1990. p.423-53.

[18] ROGERS, A. M. **Compiling for locality of reference**. PhD Thesis, Cornell University, 1990.

[19] SUN. **SPARCstation 10 system architecture**. Technical White Paper. Sun Microsystems, 1992.

[20] TSENG, C-W. **An optimizing Fortran D compiler for MIMD distributed-memory machines**. PhD Thesis. Rice University. 1993

[21] WOLF, M. E. & LAM, M. S. A data locality optimizing algorithm. In: SIGPLAN Conference on Programming Language Design and Implementation, 1991. **Proceedings**. p.30-44.

[22] WOLFE, M. **Optimizing supercompilers for supercomputers**. MIT Press, 1989.

[23] WOLFE, M. **More iteration space tiling**. Technical Report No. CS/E 89-003, Oregon Graduate Center, 1989.

[24] ZIMA, H.; CHAPMAN, B. **Supercompilers for parallel and vector computers**. Addison-Wesley, 1991.