Aplicação da Tecnologia de Compiladores Paralelizantes para Gerência de Memória: um Estudo de Caso

Edson Toshimi Midorikawa

Laboratório de Sistemas Integráveis
Departamento de Engenharia Eletrônica
Escola Politénica 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
emidorik@fox.cce.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. Este trabalho apresenta um estudo de caso onde se mostra a aplicabilidade da tecnologia desenvolvida para os compiladores paralelizadores para uma gerência eficiente da hierarquia de memória. Os resultados obtidos revelam que o uso do compilador para a otimização de programas "memory-bound" é bastante promissor.

ABSTRACT

Modern processor design strategies have focused on increasing the computacional power. The result is an imbalance between computational speed and memory speed. This paper presents a case study on application of parallelizing compiler technology to memory-hierarchy management. The results reveal that the use of compiler optimizations to memory-bound programs 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 microprocessadores, como o Pentium da Intel ou o Alpha da DEC, são aproximadamente uma ordem de grandeza mais poderosos que os processadores da geração anterior. Isto se deve ao desenvolvimento de novas tecnologias em microeletrônica e de arquitetura. Esta nova geração possui uma freqüência de operação de 150 a 200 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. É um problema similar ao do acesso aos dispositivos de armazenamento secundário conhecido como "I/O Bottleneck".

Para resolver este problema, os fabricantes têm optado pela adoção de complexas hierarquias de memória. Por exemplo, os processadores MIPS R3000/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 [4], ou seja, se as aplicações não forem estruturadas de forma a aproveitarem a estrutura da hierarquia de memória da máquina alvo.

Isto tem levado muitos programadores a re-estruturarem seus códigos manualmente de forma a extraírem o maior desempenho possível [7][8]. 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.

Em paralelo, problemas muito semelhantes na área de otimização de programas para máquinas vetoriais e paralelas foram resolvidos, automatizando-se o processo de reestruturação através do compiladores vetorizadores / paralelizantes [13] [16] [17]. Anteriormente, os programas eram otimizados manualmente, aplicando-se uma série de técnicas específicas a uma determinada arquitetura. Com o desenvolvimento das técnicas de paralelismo, tornou-se possível automatizar o processo e, hoje, a paralelização pode ser feita automaticamente [6][14][15].

Este trabalho mostra como esta tecnologia desenvolvida para a paralelização automática de programas pode ser aproveitada para a reestruturação de programas visando um melhor desempenho em relação aos acessos à memória. A seção 2 inicia o trabalho presentando uma visão geral da tecnologia atual dos compiladores paralelizantes, com a

introdução dos conceitos de dependência de dados e de transformação de programas. A seguir, algumas transformações aplicáveis para a melhora de desempenho da memória são descritas. A seção 4 apresenta um estudo de caso, onde se mostram as diferenças de desempenho de uma rotina com a aplicação destas transformações. Finalmente, são apresentados alguns trabalhos relacionados e conclusões gerais dos resultados obtidos.

2. Background

O uso de otimizações pelo compilador é feita há muito tempo, de forma a se alcançar independência de máquina no desenvolvimento de programas. Nos primórdios das linguagens de "alto-nível", o compilador Fortran I tornava possível aos programadores da época abandonarem a linguagem da máquina sem se preocuparem em aproveitar bem os recursos de hardware. E atualmente, programa-se computadores dos mais diversos tipos de forma independente de máquina em linguagem largamente difundidas como C, Fortran e Pascal.

Mais recentemente, os vetorizadores automáticos tornaram possível programar os supercomputadores vetoriais de forma independente de arquitetura. Utilizando-se um subconjunto do Fortran-77 ou o Fortran-90, na maioria dos casos, as aplicações escritas nestas linguagens não precisam ser reescritas[1] ao serem transportadas de um Cray C90 para um NEC SX-3R, por exemplo. O mesmo está sendo desenvolvido para as arquiteturas paralelas das mais diversas tendências [17].

E assim, por que o mesmo não pode acontecer quanto a otimização dos acessos à memória ? Esta seção apresenta uma breve revisão dos principais conceitos relacionados com estes *supercompiladores otimizadores*: a dependência de dados e as transformações de programas.

2.1. Dependência de Dados

De forma ao compilador poder analisar as referências à memória automaticamente e daí aplicar as otimizações relevantes, ele deve primeiro fazer uma *análise de dependência 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 [11]. Por exemplo, no seguinte trecho de código

```
S1:  a = b + 1;
S2:  c = a + 7;
```

1. Infelizmente, para ser preciso, pode ainda ser necessário fazer algumas adaptações. Isto devido a pequenas idiossincrasias existentes entre os sistemas (p.ex. extensões diferentes ao Fortran-77).

os comandos S_1 e S_2 fazem acesso à variável a. Diz-se então que existe uma dependência de dados entre S_1 e S_2 e isto é representado por $S_1 \delta S_2$. Os tipos existentes de dependência de dados são os seguintes: [2]

- *dependência verdadeira* ou *dependência de fluxo*: se a primeira referência escreve na posição de memória e a segunda efetua uma leitura da mesma posição. Ela é representada por $S_1 \delta^f S_2$.
- *antidependência*: se a primeira referência for uma leitura e a segunda for uma escrita. É normalmente representada por $S_1 \delta^a S_2$.
- *dependência de saída*: se ambas as referências escrevem na posição de memória. É representada por $S_1 \delta^o S_2$.
- *dependência de entrada*: se ambas as referências lêem da mesma posição de memória. É representada por $S_1 \delta^i S_2$.

Define-se também o conceito de *distância de dependência* de dados quando se considera o caso em que as referências ocorrem no corpo de um loop. De uma forma resumida, a distância pode ser definida como o número de iterações que separam a fonte e o sumidouro da dependência de dados. No exemplo seguinte

```
for (i=0; i<N, i++)  {
    a[i] = a[i-2] * b;
}
```

temos uma dependência entre as referências `a[i]` e `a[i-2]` com uma distância de dependência igual a 2, visto que a posição de memória acessada por `a[i]` é referenciada por `a[i-2]` duas iterações mais tarde.

2.2. Transformações de Programas

A otimização de programas é efetuada aplicando-se uma série de transformações aos seus comandos. Uma transformação pode ser entendida como uma manipulação sintática dos comandos de um programa com o objetivo de melhorar seu desempenho. Diversas transformações têm sido divulgadas na literatura. Entre as otimizações clássicas podemos citar: [2]

- propagação de constante
- eliminação de "dead-code"
- eliminação de subexpressões comuns
- movimentação de código
- eliminação de variáveis de indução

2. Em vetorizadores e paralelizadores automáticos, apenas os três primeiros tipos são relevantes. [3]

Da mesma forma diversas outras transformações foram propostas para os vetorizadores e paralelizadores: [13][15]

- loop distribution
- loop fusion
- loop interchange
- loop vectorization
- iteration reordering
- loop alignment
- message coalescing
- message vectorization

Dada uma linguagem $\mathcal{L}$, estas transformações operam sobre um programa $p \in \mathcal{L}$. Uma transformação pode ser vista como uma função $\mathcal{T} : \mathcal{L} \rightarrow \mathcal{L}$, ou seja, $\mathcal{T}(p) = p'$. Qual a relação entre p e p'? Diversas métricas da corretude de uma transformação podem ser aplicadas. A mais comum é a saída produzida pelo programa. Assim, p e p' são considerados equivalentes se produzirem os mesmos resultados para a mesma entrada.[3] Ou seja, a semântica do programa deve ser preservada em cada uma das transformações aplicadas.

Por exemplo, a figura 1 abaixo mostra a aplicação de uma transformação em um trecho de código de um programa. O trecho de código apresenta um loop com três comandos. Existe dependência de dados entre os comandos devido ao cálculo e uso de a e b. Desta forma, este loop não pode ser paralelizado. Se este loop for quebrado[4], deixando-se o comando S_3 no corpo de um segundo loop, é possível paralelizá-lo conforme indicado em fig.1.b.

```
for (i=1; i<N; i++)  {
    a[i+1] = b[i-1] + c[i];  S1
    b[i] = a[i] * k;         S2
    c[i] = b[i] - 1;         S3
}
```

a) trecho de código antes.

```
for (i=1; i<N; i++)  {
    a[i+1] = b[i-1] + c[i];  S1
    b[i] = a[i] * k;         S2
}
forall i=1 to N-1 {
    c[i] = b[i] - 1;         S3
}
```

b) trecho de código otimizado.

Fig.1 – Exemplo de aplicação de uma transformação de programa.

3. Outras métricas possíveis são o consumo de memória e o tempo de execução.

4. Aplicando-se a transformação loop distribution.

3. Gerência da Hierarquia de Memória

Nesta seção procura-se empregar alguns dos conceitos apresentados na seção anterior para uma possível otimização automática dos acessos à hierarquia de memória existente nos sistemas atuais de alto desempenho. Inicia-se com a aplicação do conceito de dependência de dados neste novo contexto e depois com a introdução das transformações relevantes.

3.1. Reutilização de Posições de Memória e Localidade de Referências

O conceito de dependência de dados quando aplicado para a otimização de acessos e à gerência da hierarquia de memória pode ser considerado como um instrumento 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 [5]. 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]` 5 iterações[5] mais tarde (reultilização temporal). E as referências a `b[i]` tem uma reutilização espacial com o acesso a elementos consecutivos da mesma linha de cache (ou da página de memória).

3.2. Transformações

Baseado na análise de dependência de dados, diversas transformações podem ser aplicadas de forma a melhorar o desempenho de programas "memory-bound". 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.

5. 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.

3.2.1. Fission by name

Esta transformação visa quebrar um loop em diversos loops, de modo que dois comandos estarão no mesmo loop se eles referenciarem a mesma variável escalar ou vetor. Isto faz com que cada loop apresente um working set menor. Por exemplo, o loop seguinte

```
for (i=10; i<N; i++) {
    c = a[i-1] + 6*b[i-8];
    x[i] = y[i+1] * z[i];
    a[i] = b[i] * 2;
    b[i] = b[i+4] * c / N;
    y[i] = x[i-1] - 1;
}
```

pode ser transformado no seguinte

```
for (i=10; i<N; i++) {
    c = a[i-1] + 6*b[i-8];
    a[i] = b[i] * 2;
    b[i] = b[i+4] * c / N;
}
for (i=10; i<N; i++) {
    x[i] = y[i+1] * z[i];
    y[i] = x[i-1] - 1;
}
```

3.2.2. Strip mining

Transforma um loop em dois loops aninhados. O loop externo percorre blocos de valores de índice. Considere o seguinte exemplo

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

após a aplicação da transformação tem-se

```
for (j=1; j<N; j=+64)
    for (i=j; i<min(j+63,N); i++) {
      a[i] = b[i] + 1;
      d[i] = b[i] - 1;
    }
```

O tamanho do bloco pode ser determinado por alguma característica da máquina alvo, como por exemplo, o comprimento dos registradores vetoriais ou o tamanho da memória cache.

3.2.3. Loop collapsing

Transforma dois loops aninhados em um único loop. Por exemplo, no caso do sistema não suportar loops paralelos aninhados, seja o seguinte trecho de código

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

que pode ser transformado em

```
forall k=0 to (N*M)-1 {
    i = ceil(k/M);              /* i = ⌈k/M⌉; */
    j = (k-1) % M + 1;
    a[i][j] = b[i][j] + 2;
}
```

3.2.4. 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.

3.2.5. 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

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

pode ser transformado em

6. mesmo vetores unidimensionais com expressões de índices simples.

```
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 uma acesso a memória por operação de ponto flutuante. Já o código otimizado executa duas operações em ponto flutuante por acesso à memória principal.[7]

3.2.6. 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.

4. Multiplicação de Matrizes

Nesta seção apresenta-se um estudo de caso, onde é feita uma análise da aplicação de algumas das transformações descritas na seção anterior para um programa que realiza uma multiplicação de matrizes.

4.1. Escolha do Caso

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

O problema estudado consiste em inicializar duas matrizes 500 x 500, e multiplicá-las.

4.2. Ambiente de Estudo

Os testes foram realizados em uma servidora Silicon Graphics Power Series 4D/480 VGX com 8 processadores MIPS R3000A (40MHz). Cada processador possui uma cache de

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.

instruções de 64 Kbytes, um cache de dados de primeiro nível de 64 Kbytes e um cache de dados de segundo nível de 1 Mbyte. A memória principal do sistema é de 64 Mbytes. A arquitetura do sistema é esquematizado na fig.2 abaixo.

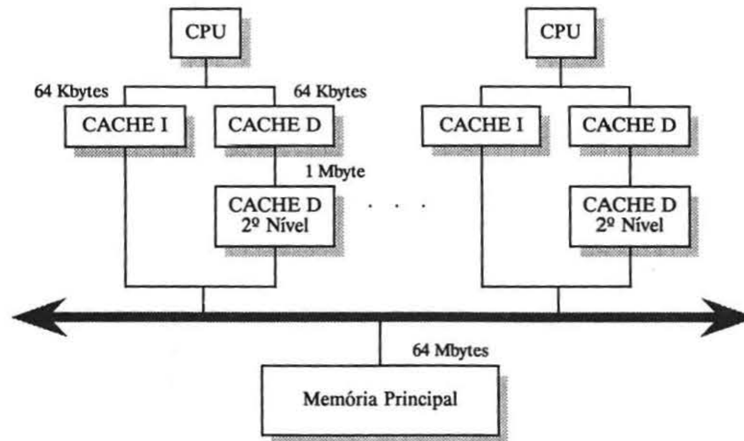

Fig.2 - Esquema simplificado da arquitetura da Power Series 4D/480 VGX.

Os programas foram codificados em linguagem C e foi utilizado o sistema de programação do sistema IRIX. O tempo de execução dos programas foi medido com o utilitário *time* e os programas foram compilados sem o uso do otimizador de código[9] de forma a evidenciar o efeito das transformações aplicadas. Embora se dispusesse de uma máquina paralela, o estudo foi feito com os programas sequenciais 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 seu impacto no tempo de execução total do programa.

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: [9]

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

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

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

```
    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];
```

Assumindo-se que os elementos da matriz c é iniciada com zeros. 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.3 mostra como é este processo de multiplicação (versão ijk).

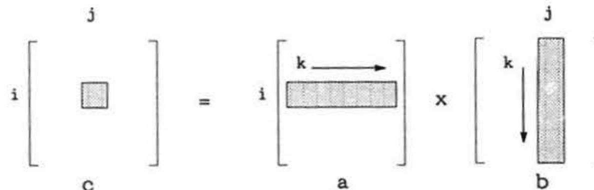

Fig.3 - Multiplicação de matrizes tradicional.

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 correta sob o ponto de vista da localidade espacial. A primeira otimização efetuada foi aplicar a transformação loop interchange nos loops em j e k (versão ikj). Notou-se uma melhora de desempenho de 37,5%. A tabela 1 abaixo mostra o tempo de execução total das duas versões.

Tabela 1 - Aplicação de loop interchange.

Versão	Tempo total (em segundos)	Melhora
ijk	287,1	
ikj	179,3	37,5%

Esta melhora de desempenho pode ser explicada observando-se o novo padrão de acessos às matrizes (fig.4). Tem-se então que as matrizes b e c são acessadas por linha.

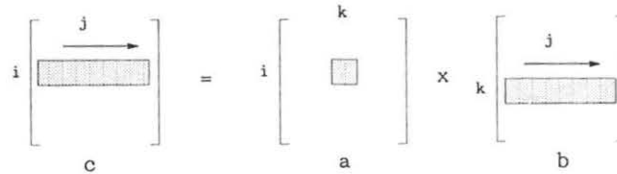

Fig.4 - Multiplicação de matrizes com loop interchange aplicado nos loops j e k.

O próximo passo foi tentar aumentar a quantidade de computação realizada no loop mais interno. Desta forma aplicou-se a transformação unroll-and-jam. Executou-se a transformação de "unrolling" 8 vezes. Esta versão (ikj-uj) obteve um desempenho 51% melhor que o algoritmo inicial e 21,5% melhor que a versão anterior (ikj). Os tempos de execução são mostrados na tabela 2.

Tabela 2 - Aplicação de unroll-and-jam.

Versão	Tempo total (em segundos)	Melhora
ijk	287,1	
ikj-uj	140,7	51 %

Outra estratégia possível é aplicar a transformação scalar replacement sobre a versão ikj. Notou-se que o acesso ao elemento $a_{i,k}$ no loop mais interno poderia ser associado a um registrador (versão ukj-sr). Desta forma, obteve-se uma melhora de desempenho de 52,6% em relação à versão original (ijk) e de 24,2% em relação à versão ikj. Os tempos de execução são mostrados na tabela 3.

Tabela 3 - Aplicação de scalar replacement.

Versão	Tempo total (em segundos)	Melhora
ijk	287,1	
ikj-sr	136	52,6 %

A seguir, aplicou-se as transformações unroll-and-jam e scalar replacement sobre a versão ikj. Obteve-se assim uma melhora final de 60% no tempo de execução. A tabela 4 mostra os tempos de execução medidos.

Tabela 4 - Aplicação de unroll-and-jam e scalar replacement.

Versão	Tempo total (em segundos)	Melhora
ijk	287,1	
ikj-uj-sr	114,2	60 %

Daí, face a este resultado promissor resolveu-se efetuar uma compilação com a chave de otimização ativada[10] (versão ikj-uj-sr.ot). O resultado obtido foi surpreendente, com uma melhora de desempenho de 87% 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 de apenas 36%. Os tempos de execução são mostrados na tabela 5. A diferença entre as versões otimizadas é de 79,7 %.

Tabela 5 - Utilização de compilação otimizada.

Versão	Tempo total (em segundos)	Melhora
ijk	287,1	
ikj.ot	183,8	36 %
ijk-uj-sr.ot	37,3	87 %

A figura 5 abaixo resume resultados obtidos, apresentando os tempos de execução medidos para todas as versões do caso estudado.

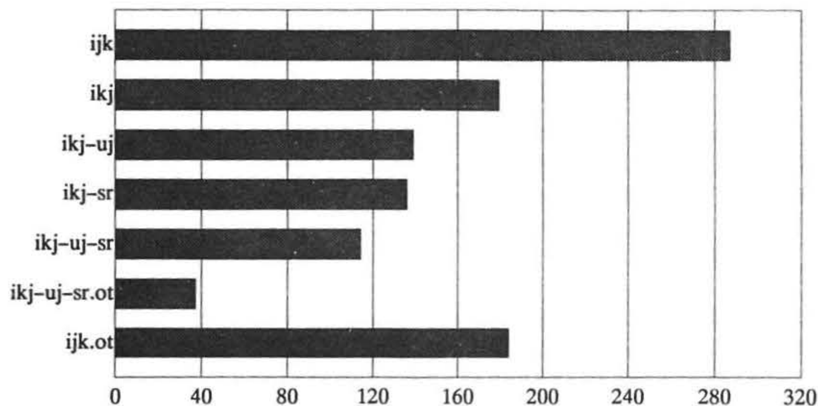

Fig.5 - Resultados obtidos na Power Series 4D/480 VGX (tempo em segundos).

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

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 7,7 vezes mais rápida que a versão original (ijk) e 4,9 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.

Análises semelhantes foram efetuadas em um servidor Sparc Server 330, obtendo-se uma melhora de desempenho final da ordem de 72%.

5. Trabalhos Relacionados

Diversos grupos de pesquisa vem trabalhando na área de paralelização automática. Os pioneiros foram os grupos do Prof. David Kuck da Universidade de Illinois [11] e Prof. Ken Kennedy da Rice University.

O enfoque de gerência de memória pelo compilador foi iniciado por Abu-Sufah em seu trabalho de doutorado [1], mas seu enfoque se concentra na diminuição das operações de paginação entre a memória principal e o disco. Já Steve Carr [5] possui um enfoque muito 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 [10] propõe um sistema híbrido para gerência de memórias cache controlado por hardware e software, o Priority Data Cache.

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. Um trabalho anterior faz outras análises similares a este [12].

6. Conclusão

Este trabalho apresentou um estudo de caso de aplicação da tecnologia desenvolvida pelos compiladores paralelizantes para a otimização do acesso à memória. 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.

Embora este estudo tenha envolvido apenas um caso, outros estudos estão em andamento com a análise de outros casos como a resolução de sistemas de equações lineares (decomposição LU), criptografia e ray-tracing volumétrico.

Agradecimentos

Gostaria de agradecer a todos os colegas que de alguma forma ajudaram na realização deste trabalho. Em especial, a Dr. *Liria Matsumoto Sato* e *Hsueh Tsung Hsiang* 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 do servidor Silicon 4D/480 VGX. Finalmente, agradeço ao Prof. Dr. *Claudio Kirner* (UFSCar) pelo apoio durante o desenvolvimento das pesquisas que culminaram neste trabalho.

Referências Bibliográficas

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

[2] AHO, A.V.; SETHI, R.; ULLMAN, J. D. **Compilers - principles, techniques and tools**. Addison-Wesley, 1986.

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

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

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

[6] CHOW, J.-H. **Compile-time analysis of explicitly parallel programs**. PhD Thesis. University of Illinois at Urbana-Champaign, 1993.

[7] 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..

[8] 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.

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

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

[11] KUCK, D. **The structure of computers and computations**. Vol.1. John Wiley and Sons, 1978.

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

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

[14] 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.

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

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

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