MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-8206-TDI/761

OTIMIZAÇÃO DE DESEMPENHO UTILIZANDO CONTADORES DE HARDWARE

Ricardo Varela Corrêa

Dissertação de Mestrado em Computação Aplicada, orientada pelo Dr. Celso Luiz Mendes, aprovada em 19 de junho de 2000.

INPE
São José dos Campos
2001

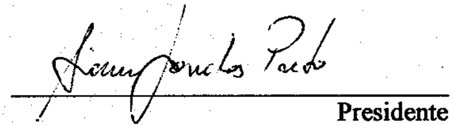
519.863

CORRÊA, R. V.
Otimização de desempenho utilizando contadores de hardware / R. V. Corrêa. – São José dos Campos: INPE, 2000.
193p. – (INPE-8206-TDI/761).

1.Programação paralela. 2.Desempenho. 3.Hardware. 4.Contadores. 5.Otimização. I.Título.

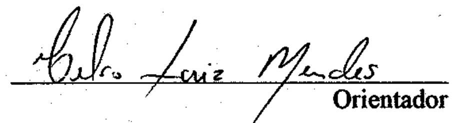
Aprovado pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre** em **Computação Aplicada.**

Dr. Airam Jônatas Preto



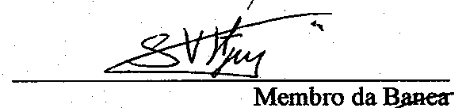
Presidente

Dr. Celso Luiz Mendes



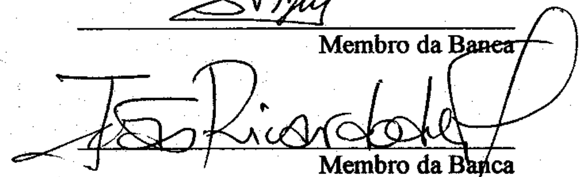
Orientador

Dr. Stephan Stephany



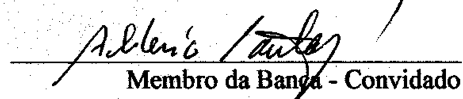
Membro da Banca

Dr. João Ricardo de Freitas Oliveira



Membro da Banca

Dr. Alderico de Paula



Membro da Banca - Convidado

Candidato (a) : Ricardo Varela Corrêa

São José dos Campos, 19 de junho de 2000.

Our continuing mission: to seek out knowledge of C, to explore strange unix commands, and to boldly code where no one has man page 4"

Sven Goldt, 1995.

Dedico este trabalho a Solange, amiga e companheira nos momentos difíceis.

AGRADECIMENTOS

Agradeço a meus pais, Niel e Yedda, pelo incentivo e apoio em todas as fases de minha vida.

A Maria Emília Maia agradeço pelo bom humor e por mostrar que as dificuldades podem ser vencidas.

Ao Dr Celso Mendes agradeço pela orientação segura e, principalmente, por sua paciência e precisa intervenção no conteúdo e texto da dissertação.

RESUMO

A disponibilidade de microcomputadores cada vez mais potentes, com capacidade computacional e área de armazenamento próximos da capacidade de estações de trabalho caríssimas, tem colocado a comunidade científica diante de uma nova ferramenta para o desenvolvimento de aplicações científicas. A utilização dessas novas plataformas no meio científico criou a necessidade para um sistema operacional que fosse estável e confiável, abrindo o caminho para que o Linux assumisse a tarefa de gerenciar essas máquinas. Este aumento da capacidade somado a um ambiente estável, traz a necessidade de se melhorar o desempenho de algumas aplicações científicas, uma vez que o aumento das dimensões dos dados a analisar vem crescendo continuamente. Assim, com o aumento da complexidade dos problemas científicos e principalmente com a necessidade de tratamento de grandes quantidades de dados, tem colocado a otimização de desempenho como fator importante no desenvolvimento de uma aplicação. Especificamente nos processadores fabricados pela Intel, existem recursos internos que podem facilmente auxiliar no estudo do desempenho de um programa. A captura de dados internos que monitorem a atividade do código executando internamente ao processador e em tempo real, pode fornecer informações que mostrem onde estão os pontos problemáticos. O acesso a eventos internos aos processadores pode ser realizado através dos denominados contadores de *hardware*, que trazem informações precisas sobre o estado do processador. A forma de utilização desses contadores, principalmente como acioná-los a partir de uma aplicação de alto nível e estando trabalhando no ambiente Linux, exige alguns conhecimentos que estão discutidos com detalhes neste trabalho. A captura dos dados de desempenho permite assim conhecer como um trecho de programa interage com o processador e através de técnicas clássicas de otimização, pode-se reduzir consideravelmente os pontos de interação adversa com o processador. Uma interface foi desenvolvida para permitir ao programador a captura desses dados. A metodologia aqui descrita foi aplicada inicialmente a simulações de pequenos trechos de código e a uma aplicação seqüencial. Posteriormente, foi aplicada a um exemplo de cálculo de desvio padrão onde contagens de diversos eventos foram realizadas. A otimização de um programa seqüencial real partindo da reestruturação do código original em Fortran 77 foi executada com o código em notação Fortran 90, instrumentado e monitorado em diversos eventos. A paralelização desse código e execução em paralelo, mostra a eficiência do método, disponibilizando aos programadores uma seqüência de etapas para análise de uma aplicação científica, visando a otimização de desempenho. A monitoração de programas paralelos permite assim que o desempenho de uma aplicação paralela seja analisada e através da inspeção e monitoração de seu código, mostrar como torná-lo mais eficiente. Os dados obtidos mostram que o método pode ser utilizado em aplicações seqüenciais ou paralelas fornecendo informações claras do funcionamento de trechos de programa, permitindo ao programador identificar os pontos de interação adversa com a arquitetura.

PERFORMANCE OPTIMIZATION USING HARDWARE COUNTERS

ABSTRACT

The availability of more powerful microcomputers with the computacional and storage capacities similar to expensive workstations, is leading the scientific community upfront to a new tool for the development of scientific applications. The use of these new plataforms created the need of an operational system that was stable and reliable, thus allowing Linux to lead the way to control these machines. Both high computer capacity and stable environment are items needed to boost the search for performance increase. Also, added to this scenario, there is an increasing complexity in scientific problems size of scientific data. With all this information in mind, performance increase is the next thing to do. Specifically Intel processors have built-in resources that can be used to analyze performance. The capture of the processor internal status in real time can provide information to solve software bottlenecks. Access to processor's internal resources to monitor events during execution of a code can be done using *hardware* counters, a set of registers that can monitor internal events occurring in the processor. Capture of performance data provides the necessary information on how an application interacts with the processor and, through the use of optimization procedures, it is possible to reduce the adverse interaction with the processor. An interface including a set of tools to capture performance data is available to the software programmer. The methods described here were applied to a series of simulations of adverse memory access and also to a real application. After its validation, a real scientific application written in Fortran 77 code was implemented and instrumented to monitor memory reference events in the processor internal cache. This application was totally restructured using Fortran 90 notations and again monitored for the same events. The reduction of execution time shows the efficiency of the Fortran 90 version. The reduction of adverse interaction with the cache memory is measured and discussed. This same application is again modified with the insertion of parallel directives in order to distribute the input data using the parallel data programming paradigm. The same events were monitored locally in each processor, showing the feasibility of performance data capture in parallel applications.

SUMÁRIO

Pág.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

CAPÍTULO 1 - INTRODUÇÃO 21

1.1 - Objetivos do Trabalho 22

1.2 - Estrutura da Dissertação 25

CAPÍTULO 2 - INFRA-ESTRUTURA DE CONTADORES DE *HARDWARE* 27

2.1 - Contadores de *Hardware* em Processadores Atuais.. 27

2.2 - Contadores de *Hardware* em Processadores Intel.. 31

2.3 - Arquiteturas de Alto Desempenho 39

2.4 - Gerenciadores de Dispositivos no Linux.. 43

2.5 - Módulos de Acesso aos Contadores.. 47

CAPÍTULO 3 - METODOLOGIA DE MONITORAÇÃO E OTIMIZAÇÃO DE DESEMPENHO................................ 51

3.1 - Captura dos Dados de Desempenho.. 51

3.2 - Interface para Programação em Alto Nível.. 57

3.3 - Utilização de Dados de Desempenho.. 63

CAPÍTULO 4 - TÉCNICAS DE OTIMIZAÇÃO DE PROGRAMAS SEQUENCIAIS.. 67

4.1 - Otimização com Técnicas Clássicas.. 67

4.2 - Otimização de uma Aplicação Real.. 76

CAPÍTULO 5 - MONITORAÇÃO EM PROGRAMAS PARALELOS...... 81

5.1 - Programação Paralela de Dados.. 84

5.2 - Fortran 90.. 87
5.3 - *High Performance Fortran* - HPF.. 89
5.4 - Instrumentação de Programas Paralelos.. 94

CAPÍTULO 6 - EXEMPLO DE OTIMIZAÇÃO EM APLICAÇÕES PARALELAS... 101

6.1 - Exemplo de Otimização por Reestruturação de Código............................ 101
6.2 - Exemplo de Otimização por Mudança de Distribuição............................. 114

CAPÍTULO 7 - CONCLUSÕES E TRABALHOS FUTUROS.................. 121

REFERÊNCIAS BIBLIOGRÁFICAS 125

APÊNDICE A - BIBLIOGRAFIA COMPLEMENTAR........................... 129

APÊNDICE B - Eventos disponíveis para monitoração de CPU's da família INTEL P5 131

APÊNDICE C - Eventos disponíveis para monitoração de CPU's da família INTEL P6... 139

APÊNDICE D - Listagem do módulo de acesso aos MSR......................... 147

APÊNDICE E - Listagens das rotinas de instrumentação.......................... 153

APÊNDICE F - Listagem original do programa "Correlação MASCO"... 165

APÊNDICE G - Listagem HPF do programa "Correlação MASCO"........ 175

APÊNDICE H - Listagem de PDE1... 185

LISTA DE FIGURAS

Pág.

2.1 - Visão gráfica do núcleo do Linux 46

3.1 - Conexão do módulo ao núcleo do Linux 53

3.2 - Fluxograma da subrotina iniciaregistros 59

3.3 - Fluxograma da subrotina leregistros 62

3.4 - Programa para análise de padrão de referência a memória 66

4.1 - Comparação entre técnicas de otimização 73

4.2 - Medidas de *Data cache misses* para a malha do exemplo dado.................... 74

4.3 - *Data cache misses* para várias técnicas de otimização................................ 74

4.4 - Tempo de execução em função das dimensões das matrizes 75

4.5 - *Code cache misses* em função das dimensões das matrizes......................... 76

5.1 - Evitando sincronismo em FORALL com o uso de INDEPENDENT 94

5.2 - Exemplo de programa paralelo instrumentado 98

6.1 - Diagrama de blocos do programa corr_masco 103

6.2 - Comparativo do *Cache Hit Rate* para casos de transposição de linhas....... 109

6.3 - Trecho instrumentado do programa PDE1 115

LISTA DE TABELAS

Pág.

2.1 - Contadores existentes em algums processadores 29

2.2 - Eventos disponíveis em várias plataformas 30

2.3 - Retorno de CPUID em função de EAX 33

2.4 - *bits* de identificação de registros especiais 34

2.5 - Retorno de CPUID com descrição de famílias de CPU 34

2.6 - Resultados da aplicação de CPUID 35

2.7 - Designação de ECX para acesso aos MSR da família P5 36

2.8 - Eventos disponíveis na família P5 37

2.9 - Designação de ECX para acesso aos MSR da família P6 38

2.10 - Descrição de eventos monitorados na CPU P6 39

2.11 - Identificação de dispositivos genéricos 48

3.1 - Endereços dos contadores de *hardware* 63

3.2 - Comparação de acertos ao *cache*.. 64

4.1 - Resultados de aplicação de blocagem 71

4.2 - Perfil de tempo de execução do Hydrolight 77

4.3 - Acessos aos *cache* de instrução e TLB 78

5.1 - Eventos monitorados no exemplo de desvio padrão 97

6.1 - Tempos de execução de subrotinas 102

6.2 - Temporização das malhas internas 104

6.3 - Temporização de trecho para diferentes compiladores 105

6.4 - FLOPs, instruções executadas e referências à memória para o trecho-4....... 106

6.5 - Comparação de eventos monitorados para o trecho-5 108

6.6 - Monitoração do trecho-4 em máquinas paralelas 111

6.7 - Monitoração do trecho-4 em máquina sequêncial 112

6.8 - Valores de tempo da aplicação rodando paralela em duas máquinas 113
6.9 - Comparação de tempos totais de execução antes e depois da otimização..... 113
6.10 - Eventos registrados para a distribuição original do aplicativo PDE1........... 116
6.11 - Eventos registrados para a distribuição original do aplicativo PDE2........... 116
6.12 - Redução de contagens comparando PDE1 com PDE2 117
6.13 - Valores de tempo de PDE1 com 2 processadores(Polaris) 118
6.14 - Valores de tempo de PDE2 com 2 processadores(Polaris) 118
6.15 - Valores de tempo de PDE1 com 4 processadores (Polaris) 118
6.16 - Valores de tempo de PDE2 com 4 processadores (Polaris) 119

LISTA DE SIGLAS E ABREVIATURAS

AT&T	American Telephone and Telegraph Company
CAP	Computação Aplicada
CESR	Control and Event Select Register
CPU	Central Processing Unit
CPUID	CPU Identification
DAS	Divisão de Astrofísica
EAX, EBX, ECX, EDX	Registros de dados do Pentium
FLOP	Floating Point Operation
HPF	High Performance Fortran
INPE	Instituto Nacional de Pesquisas Espaciais
Kb	Kilo bytes = 1,024 bytes
LAC	Laboratório Associado de Computação e Matemática Aplicada
μs	Microsegundos. (1,0 E -6 segundos)
Mb	Mega bytes
ms	Milisegundos (1,0 E -3 segundos)
MSR	Model Specific Registers
MURA	Modified URA
PMC	Performance Monitoring Counters
RDMSR	Read MSR
RDPMC	Read PMC
RDTSC	Read TSC
SIMD	Single Instruction Multiple Data
SLB	Setor de Lançamento de Balões
SPMD	Single Program Multiple Data
MIMD	Multiple Instructions Multiple Data
TLB	Translation Look-aside Buffer
TSC	Time Stamp Counter
URA	Uniformly Redundant Array

WRMSR	Write MSR

CAPÍTULO 1

INTRODUÇÃO

Os microcomputadores atuais são equipados com *Central Processing Units* - CPU's de alto desempenho, com capacidade para oferecer diversos recursos aos usuários, de forma a executar uma aplicação com o máximo de eficiência possível. Assim, contando com um hardware eficiente, passa a existir um interesse em se desenvolver aplicações eficientes, e, portanto, a necessidade de ambientes de programação com características específicas, que permitam ajustes finos entre o aplicativo e o processador de determinadas plataformas.

No caso específico do desenvolvimento de aplicações científicas, muitas vezes não se busca como objetivo primário a eficiência da aplicação. Na maioria dos casos, as aplicações são restritas a determinados laboratórios ou a um número pequeno de usuários. Somando-se a esse quadro, tem-se ainda que diversas aplicações são portadas de outras plataformas ou são provenientes de aplicações mais antigas.

Os programas de aplicações científicas são normalmente elaborados com uma visão direcionada à solução de um determinado problema, sem grande preocupação com o seu desempenho posterior. A rara utilização de técnicas de otimização, acrescida do emprego de chamadas intensivas a subrotinas, produz normalmente uma aplicação com baixo rendimento. Muitas vezes estas aplicações são portadas de outras plataformas e, por esta razão, fazem uso de declarações já obsoletas, impedindo que todos os recursos disponibilizados pela arquitetura sejam utilizados.

De uma forma mais geral, o problema da otimização do desempenho de uma aplicação consiste em encontrar a melhor sintonia entre as estruturas utilizadas no código e o *hardware* no qual ocorre a execução. Tal sintonia, muitas vezes, exige um diagnóstico minucioso sobre o desempenho da implementação original. Na obtenção deste diagnóstico, quanto maior o nível de detalhe das informações de desempenho

disponíveis, mais fácil se torna a tarefa do programador. Em algumas situações, podem ser necessários detalhes que simplesmente não estão disponíveis através das técnicas tradicionais de monitoração por *software*.

1.1- Objetivos do Trabalho

A necessidade de otimização das aplicações científicas justifica a elaboração de uma análise detalhada de seu código fonte. Esta análise deve levar em conta a estrutura do programa, principalmente as chamadas a subrotinas e a forma de utilização de malhas internas. O uso inadequado dessas malhas provoca excessivo consumo de tempo de CPU, contribuindo para a ineficiência da aplicação.

A identificação dos pontos onde existam gargalos de execução é obtida através da temporização das chamadas a subrotinas e de todas as malhas internas. Essa tarefa, normalmente atribuída às ferramentas de *profiling*, realiza um levantamento do perfil dos tempos de execução das subrotinas envolvidas. Tais ferramentas, porém, não fornecem, em geral, informações sobre os tempos de execução das malhas internas; além disso, os resultados gerados não permitem uma análise das causas que expliquem o baixo desempenho da aplicação. A utilização dessas ferramentas existentes, muitas vezes disponíveis como funções de bibliotecas nos compiladores das linguagens C e Fortran, depende de chamadas a *system calls* - rotinas internas do sistema. Portanto, além de intrusivas, muitas vezes não produzem a resolução desejada, ou ainda trazem apenas informações de tempo das chamadas às subrotinas.

Assim, a metodologia aqui apresentada tem como objetivo fornecer informações mais detalhadas dos pontos adversos de interação entre um programa e a arquitetura. A localização de pontos específicos da aplicação onde existam gargalos de consumo excessivo de tempo, bem como as causas destes gargalos, pode ser obtida através da utilização de recursos internos existentes nos processadores atuais.

A maioria dos processadores existentes no mercado possuem registros internos com capacidade para contar a ocorrência de determinados eventos. Embora a documentação seja praticamente inexistente em algumas plataformas e muito pouco elucidativa em sua maioria, os fabricantes admitem sua existência e sua continuidade em versões futuras. Estes recursos estão disponíveis como registros internos e são denominados contadores de hardware; estão disponíveis, por exemplo, nos processadores MIPS R10000, Alpha 21264, IBM Power PC 604 e 604e, Cyrix, K6 e K7 e Intel. A Intel [1], em particular, implementa internamente nos microprocessadores de sua fabricação, desde a família 286 até a arquitetura Pentium atual, vários registros internos com a finalidade de monitorar diversas atividades ocorridas internamente à CPU. A cada novo modelo ou família lançados, novos registros são incorporados com monitoração de eventos mais complexos, podendo fornecer ao desenvolvedor uma poderosa ferramenta para a análise de um código em execução. Por se tratar de registros físicos de *hardware*, sua utilização passa a não depender de compilador ou dos recursos computacionais disponíveis na máquina. Assim, a sua existência em uma certa máquina permite que se analise o código internamente à CPU de forma pouco intrusiva ao código fonte original, disponibilizando informações que estariam inacessíveis de outra maneira.

O preço final das máquinas de uso geral com arquiteturas de microprocessadores do tipo *Complex Instruction Set Computers*(CISC) tem caído consideravelmente, com sua capacidade computacional atingindo o desempenho e armazenamento de dados similares a estações de trabalho do tipo *Reduced Instruction Set Computers*(RISC).

O ambiente Linux, por sua vez, está se expandindo e se firmando como o Sistema Operacional de microcomputadores, devido à sua estabilidade, confiabilidade e disponibilidade de compiladores e aplicativos eficientes, muitas vezes gratuitos. Assim, os microcomputadores com Linux estão sendo adotados como plataforma de desenvolvimento na comunidade científica. Por esta razão, a metodologia de análise aqui apresentada utiliza-se dos recursos internos da CPU dentro do contexto desse Sistema Operacional.

A computação paralela está em constante evolução e, principalmente, com um desenvolvimento acentuado de compiladores específicos nas linguagens C e Fortran ou baseadas nestas. Da mesma forma que o desempenho de uma aplicação seqüencial pode ser otimizado, os mesmos conceitos podem ser aplicados à computação paralela.

O uso dos atuais paradigmas de computação paralela requer que o programador desenvolva um código que contenha diretivas com informações que permitam ao compilador estabelecer o paralelismo da aplicação[30]. Assim, é possível se chegar à solução do problema em menor tempo com diversos processadores trabalhando nesta solução. De forma ideal, uma aplicação seria estruturada tal que a adição de processadores implicaria na redução proporcional do tempo total de execução. Para se escrever aplicações com essa propriedade, um paradigma de computação paralela deve ser adotado pelo programador.

Dentre os paradigmas existentes[14], tais como o modelo *message passing* - troca de mensagens ou programação *data parallel* - paralela de dados, e da mesma forma que na programação seqüencial, as ferramentas de levantamento de perfil de execução são incompletas, pois fornecem poucas informações sobre a eficiência do código ou sobre as razões da sua ineficiência.

A metodologia aqui apresentada procura, assim, fornecer uma solução para atender à programação seqüencial ou à computação paralela, através do uso de contadores de *hardware* para monitoração de desempenho. Este uso em aplicações paralelas permite analisar o código em execução isoladamente em cada processador, obtendo informações de sua interação com o hardware em cada uma das máquinas. Informações obtidas, por exemplo, sobre o total de instruções executadas ou sobre as operações de ponto flutuante efetuadas, permitem analisar a eficiência da execução em paralelo da aplicação. Da mesma forma, detalhes sobre acessos à memória obtidos em cada máquina podem revelar as conseqüências de uma certa distribuição dos dados do programa, permitindo ao programador analisar e procurar uma estrutura mais eficiente.

1.2 - Estrutura da Dissertação

A existência dos contadores de hardware em praticamente todas as plataformas disponíveis no mercado mostra o interesse dos fabricantes em prover recursos para análise do comportamento de um trecho de código executado pelo processador. Assim, este trabalho inicia, no Capítulo 2, com uma breve discussão sobre a disponibilidade desses contadores em diversas plataformas atuais. Em seguida, a disponibilidade desses contadores na arquitetura Intel e, principalmente, sua operação nas famílias Intel Pentium, Pentium MMX, Pentium Pro, Pentium II e Pentium III são discutidas com detalhes. A utilização dos contadores no ambiente Linux exige uma explicação do gerenciamento de dispositivos sob o Linux, especialmente sobre a troca de dados entre um dispositivo e o sistema operacional. Os conceitos, os tipos de dispositivos e como estes são acessados fazem parte desse capítulo.

O entendimento do processo de acesso aos contadores de hardware permite que se passe ao Capítulo 3, que apresenta a metodologia utilizada para otimizar aplicações científicas. Mostra-se o uso de estatísticas geradas pela utilização desses contadores. A captura dos dados de desempenho, os tipos de eventos passíveis de monitoração ou ainda como é feita a interface para programação em alto nível, são discutidos neste capítulo.

O Capítulo 4 descreve algumas técnicas de otimização, consideradas clássicas, e as aplica em trechos de programas seqüenciais. Exemplos práticos são apresentados mostrando a simplicidade de uso dos contadores de hardware. Discute-se ainda as contagens obtidas e como estas podem ser utilizadas na otimização de desempenho.

A utilização da metodologia estudada e aqui descrita pode ser estendida para a programação paralela, assunto discutido no Capítulo 5. O uso do Fortran 90 para explicitar o paralelismo e a posterior utilização de diretivas *High performance Fortran* (HPF) para distribuir dados, mostram o potencial da metodologia discutida nos capítulos anteriores. As operações e os aspectos desejados na programação paralela de dados são apresentados. Uma breve introdução às características do Fortran 90 e do

HPF permite ao programador ter o conhecimento necessário para começar a programar nesse novo paradigma. Os resultados obtidos pela monitoração de um programa paralelo são apresentados e discutidos, mostrando a facilidade de se instrumentar e obter contagens de eventos.

Reescrever uma aplicação seqüencial com diretivas do Fortran 90 de forma a explicitar um paralelismo posterior com a inclusão de diretivas HPF, permitiu analisar a metodologia aqui descrita na programação paralela . O procedimento de identificação e instrumentação de possíveis trechos problemáticos de interação com a CPU é discutido e os resultados comparados com a atualização do código em HPF.

O Capítulo 6 discute dois exemplos de otimização. No primeiro exemplo, o código fonte original, desenvolvido em Fortran 77, é totalmente reestruturado com notação Fortran 90, e posteriormente os dados são distribuídos entre duas máquinas executando em paralelo. No segundo exemplo, analisa-se a influência da distribuição dos dados numa aplicação paralela em HPF.

Considerações gerais sobre algumas das limitações existentes na utilização atual dos contadores de hardware, bem como discussões finais e possíveis caminhos de estudos futuros, são apresentados no Capítulo 7.

CAPÍTULO 2

INFRA-ESTRUTURA DE CONTADORES DE *HARDWARE*

Os contadores de *hardware* estão disponíveis já há algum tempo nas arquiteturas existentes. Para se fazer uso desses contadores, inicia-se este capítulo com um levantamento das plataformas onde eles estão disponíveis para o programador de forma clara e com razoável documentação. Em seguida, descreve-se com detalhes os contadores de *hardware* disponibilizados pela Intel, plataforma escolhida para o desenvolvimento deste trabalho. As diferenças entre famílias de processadores implicam em conjuntos diferentes de eventos passíveis de monitoração. As diferenças de acesso e uso dos contadores são descritas, de forma a permitir ao programador uma visão de seu funcionamento.

Uma rápida discussão sobre o paralelismo interno dos processadores Pentium á apresentada de forma a dar as informações necessárias para a compreensão das técnicas de otimização.

A utilização dos contadores no ambiente Linux exige o entendimento sobre os gerenciadores de dispositivos, responsáveis pela interface entre o programa do usuário e os recursos de contagens de eventos existentes na CPU.

2.1 - Contadores de *Hardware* em Processadores Atuais

A existência de contadores de *hardware* nos processadores é uma história antiga e complicada, pois todos os fabricantes mantinham um forte segredo sobre sua existência e sobre a forma de acesso.

Hoje, praticamente todos os processadores existentes no mercado possuem contadores de *hardware* e com divulgação pelo fabricante[24]. A documentação, porém, ainda parece apontar para o desejo dos fabricantes em mantê-los fora do domínio público,

pois deixa muito a desejar. Aparentemente, existe um pouco de receio dos fabricantes em disponibilizar informações internas de sua CPU.

O estudo sobre os contadores de *hardware* iniciou-se no meio acadêmico e talvez, por essa razão, tenha diminuído a aversão dos fabricantes à liberação de documentação. A proliferação de textos sobre o assunto indica que estes contadores possuem ainda um potencial muito grande na análise e otimização de aplicações, bem como no modelamento e testes de desempenho. Apesar de incompleta e praticamente associada ao meio acadêmico, a documentação liberada por alguns fabricantes ainda gera muitas dúvidas.

A IBM, uma das pioneiras, já em 1991 lançou um cartão externo que monitorava o barramento da CPU para utilização com alguns de seus processadores. Em 1992, foram incluídos nos processadores RISC 6000 POWER2[27] contadores de *hardware* externos à CPU. A IBM sempre demonstrou interesse nos contadores de *hardware*, e em 1993 passou a incluir internamente aos processadores PowerPC 604 [19] contadores exclusivos para a monitoração de eventos ocorridos na CPU. Em 95, o PowerPC 604e foi lançado com o dobro dos contadores de *hardware* que a versão anterior (PowerPC 604), permitindo a monitoração de 111 eventos. A partir de 1997, os processadores RISC System 6000 POWER2SC[26] passaram a disponibilizar dois contadores novos, um contador de ciclos e o outro contando instruções executadas. Os eventos eram monitorados através de cinco outros contadores. O MIPS R10000[25] possui também diversos contadores e com razoável documentação.

A atitude dos grandes fabricantes de processadores em disponibilizar contadores de *hardware* para a monitoração de eventos ocorridos na CPU começa assim a despertar o interesse dos programadores. A cada nova versão de um processador são incluídos novos eventos passíveis de monitoração.

A Tabela 2.1 mostra o total de contadores existentes para algumas plataformas.

TABELA 2.1 - CONTADORES EXISTENTES EM ALGUNS PROCESSADORES

Plataforma	Total de	Dimensão dos registros
MIPS R 10K	2	32 bits
Alpha 21164	3	dois contadores de 16 bits e um de 14 bits
IBM Power PC 604	2	32 bits
IBM Power PC 604e	4	32 bits
Sun Ultra I/ II e III	2	32 bits
Intel Pentium e Pentium MMX	3	dois contadores de 40 bits e um de 64 bits
Intel Pentium II e III	3	dois contadores de 40 bits e um de 64 bits

O conjunto de eventos disponíveis para monitoração depende da plataforma. Porém, alguns eventos são comuns a todos os processadores, como: contagens dos totais de *Floating Point Operations*(FLOPs) ou com inteiros, total de instruções executadas e, principalmente, o registro das atividades ocorridas nas memórias *cache*.

Convém ressaltar que apesar da falta de interesse dos fabricantes em atender ao público externo sobre esse assunto, existe um consenso em manter estes contadores disponíveis, inclusive com evoluções previstas para as gerações futuras de processadores, como é o caso já descrito pela IBM nos processadores RISC 6000 e PowerPC. Em contraste com a IBM, tem-se a SUN, que manteve o total de contadores nos processadores Ultra I, II e Ultra III sem nenhuma modificação nos eventos disponíveis.

A Tabela 2.2 apresenta, dentre as várias arquiteturas, os eventos mais comuns disponíveis.

TABELA 2.2 - EVENTOS DISPONÍVEIS EM VÁRIAS PLATAFORMAS

Evento	*Pentium*	*IBM 604*	*SUN Ultra II*	*Cyrix 6x86MX*	*HP PA RISC*	**Alpha**
Ciclos	x	x	x	x	x	x
Instruções	x	x	x	x	x	x
Loads	x	x	x	x	-	x
Stores	x	x	x	x	-	x
L1 *misses*	x	x	x	x	x	x
L2 *misses*	x	x	x	-	-	x
FLOPs	x	x	-	x	-	x
Condicionais	x	x	-	x	-	x
Condicionais	x	x	-	x	-	x
TLB *misses*	x	x	-	x	x	x
Code cache misses	x	x	x	x	-	x

Dentre as plataformas existentes, as empresas Compaq(Alpha) e IBM(PowerPC) são as que disponibilizam documentação mais detalhada sobre os contadores de *hardware*, como também desenvolvem em seus laboratórios aplicativos que disponibilizam a monitoração de eventos aos seus usuários. A Compaq desenvolve um projeto de monitoração constante de eventos no processador Alpha 21164, denominado "*The Compaq Continuous Profiling Infrastructure-DCPI*"[18]. Este projeto faz uso dos contadores de *hardware* existentes nos processadores Alpha para gerar mapas com as estatísticas mais comuns. O programa instala um dispositivo que gerencia as atividades executadas pelo gerenciador de executáveis ,"exec", permanecendo dormente e disponível para análise de qualquer atividade da CPU, uma vez solicitado. Da mesma forma, a IBM disponibiliza o *"PMAPI - Performance Monitor Application Programming Interface"*[20] para os processadores 604e.

Numa comparação entre os processadores existentes, levando-se em conta a facilidade de uso e a quantidade de eventos monitorados, pode-se afirmar que os processadores Compaq Alpha possuem o melhor conjunto de contadores de *hardware*, seguido dos processadores IBM 640. A Intel também tem procurado colocar as informações sobre os recursos internos de suas CPUs disponíveis. Apesar de poucos documentados e com algumas áreas ainda secretas, já é possível utilizá-los.

2.2 - Contadores de *Hardware* em Processadores Intel

A Intel lança seus produtos com contadores internos desde a versão 286 de sua linha de microprocessadores. Da mesma forma que todos os fabricantes, seus contadores eram secretos. Ao lançar a família de CPUs 486[4], a Intel acrescentou diversos registros com funções que permitiam a monitoração de eventos internos. Estes registros foram tornados públicos ao final de sua vida comercial, basicamente devido à pressão de desenvolvedores de *software* que gostariam de contar com recursos para obtenção de dados de desempenho de seus códigos. A partir de então, esta comunidade passou a contar com informações importantes para a análise de desempenho de aplicações.

Os registros internos nos processadores Intel a partir da família 486 passa a fazer parte de todos os processadores. A complexidade dos eventos passíveis de monitoração evolui consideravelmente a cada novo processador colocado no mercado pela Intel. As gerações mais modernas possuem contadores internos com a capacidade de monitorar diversos eventos. Os modelos denominados P5 (Pentium e Pentium MMX) podem monitorar 41 eventos. As gerações do modelo P6 (Pentium Pro, Pentium II e Pentium III) podem monitorar 68 eventos diferentes.

A Intel disponibiliza diversos registros para testes e verificação de funcionamento da CPU, bem como disponibiliza um conjunto de registros especiais que podem contar eventos ocorridos na CPU. Os registros denominados *Model Specific Registers*(MSR)[3] permitem o controle interno de processos e a monitoração de eventos. Os registros que permitem a contagem de eventos e chamados de contadores de *hardware*, são denominados pela Intel como *Performance Monitoring Counters*(PMC).

O interesse da Intel em manter esses contadores em sua linha pode ser observado pela evolução dos eventos passíveis de monitoração. A cada nova geração, novos contadores e mais eventos foram acrescentados. Para a proposta apresentada neste trabalho, a utilização dos contadores de *hardware* estará limitada às CPU's Intel das famílias P5 e P6.

A família P5 possui 19 registros MSR, onde dois são contadores de *hardware,* permitindo testes aos *caches* de dados, de intrução e TLB. Um dos MSR permite a contagem de pulsos de *clock* da CPU e portanto pode ser utilizado na temporização de trechos de códigos.

A família P6 expandiu consideravelmente os registros MSR, disponibilizando 68 registros de testes com cinco registros PMC. A identificação de sua existência em uma determinada CPU pode ser obtida através da instrução CPUID, que fornece informações sobre os recursos da CPU. Esta instrução permite identificar a família, modelo, tipo e outros dados importantes, como dimensões da memória *cache*.

A identificação de uma CPU Intel [3] pode ser obtida através da leitura de um registro interno onde todas as informações da CPU são registradas. A instrução CPUID executa a leitura desse registro. Seu objetivo é garantir ao desenvolvedor de *software* a compatibilidade entre versões e identificação de gerações de CPU. Uma vez que os contadores de *hardware* são específicos para uma determinada família de CPU, esta instrução é fundamental para o desenvolvimento das rotinas de monitoração de eventos.

A verificação da disponibilidade da instrução CPUID em uma determinada CPU é realizada através de um teste do *bit* 21 do registro EFLAGS, denominado ID *flag*. Sendo aceita a operação de alteração deste *bit*, a CPU permite a utilização de CPUID. As características da CPU são obtidas através da informação de retorno desta instrução, e que depende do valor especificado no registro EAX. A Tabela 2.3 mostra as informações disponibilizadas pela instrução CPUID em função do valor especificado no registro EAX durante a chamada da instrução.

A primeira informação a ser facilmente verificada é a identificação do fabricante, pois a existência dos contadores é específica para as CPU's Intel e sem garantia de estar presente em seus clones. A identificação da família da CPU bem como suas características internas determinam quais eventos podem ser monitorados.

TABELA 2.3 - RETORNO DE CPUID EM FUNÇÃO DE EAX

EAX	*Registros de Retorno*
0	EAX <- Maior valor reconhecido EBX:EDX:ECX <- Identificação do Fabricante
1	EAX <- Tipo do processador EDX <- Características EBX:ECX <- Reservado
2	EAX:EBX:ECX:EDX <- Características da CPU
3 <= EAX <= valor máximo	Reservado
EAX > valor máximo	Não Definido

Adaptada da Nota de aplicação INTEL AP-485(1998, p.3)

A instrução CPUID fornece informações importantes que serão utilizadas pelo módulo de definição dos contadores de *hardware*. O resultado da chamada da instrução com EAX = 1 contém diversas informações sobre a CPU, bem como o tipo de memória *cache* e sua dimensão. Alguns *bits* informam as características da CPU, como os *bits* b0 a b3, que informam a versão (*stepping*) da família atual. Já os *bits* b4 a b7 fornecem o modelo da CPU, e os *bits* b8 a b11 definem a família. Os *bits* b4 e b5 atestam a existência dos contadores de *hardware*. A Tabela 2.4 mostra a informação dada pelos *bits* 4 e 5, que identificam a presença dos registros *Time Stamp* Counter(TSC) e MSR na CPU.

TABELA 2.4 - *BITS* DE IDENTIFICAÇÃO DE REGISTROS ESPECIAIS

bit	*Registro*	*Instrução*
4	TSC *Time Stamp Counter*	RDTSC
5	MSR *Model Specific Registers*	RDMSR e WRMSR

O registro TSC aparece isolado dos registros MSR mas também faz parte do conjunto de registros MSR. Possui 64 *bits* e tem a função de contar os *clocks* gerados na CPU. O *bit* 4 informa que o registro TSC pode ser acessado com a instrução RDTSC e o *bit* 5 informa que os registros MSR podem ser acessados através das instruções RDMSR para leitura e WRMSR para escrita[5].

A identificação do fabricante é extremamente importante na rotina de inicialização, pois a Intel não garante que os registros estão disponíveis em CPUs de outros fabricantes. A identificação de família indica se a CPU é Pentium e o modelo informa qual a geração.

A nota técnica AP - 485 [3], da Intel, lista os códigos que identificam as famílias de vários processadores Pentium, bem como os modelos e versões. A Tabela 2.5 mostra o retorno da instrução CPUID para alguns modelos das famílias P5 e P6.

TABELA 2.5 - RETORNO DE CPUID COM DESCRIÇÃO DE FAMÍLIAS DE CPU

Família	Valor	Descrição
P5	2	Pentium 75,90,100,120,133,150,166,200
P5	5	Pentium com MMX
P6	1	Pentium Pro
P6	3	Pentium II
P6	5	Pentium II Celeron
P6	8	Pentium III

A execução da instrução CPUID também permite a obtenção de dados importantes sobre a memória cache disponível. A Tabela 2.6 mostra os resultados obtidos na análise do microcomputador IBM - Personal Computer 300GL, equipado com um processador P6.

TABELA 2.6 - RESULTADOS DA APLICAÇÃO DE CPUID

Cache	***Dimensão***	***Tipo***
Data TLB	Páginas 4K*bytes*	*4-way set associative*
Instruction TLB	Páginas 4M*bytes*	*Fully associative*
Data	16 K*bytes*	*4-way set associative*
Instruction	16 K*bytes*	*4-way set associative*

A informação da dimensão dos *caches* possibilita uma sintonia bem fina em aplicações onde há necessidade de máxima interação com a arquitetura existente, permitindo explorar ao máximo a localidade da memória *cache*.

A existência do registro TSC permite uma monitoração de tempo independente do compilador utilizado, pois trata-se de uma contagem por *hardware*. Assim, a temporização de um determinado evento pode ser realizada de forma bem precisa.

Os eventos passíveis de registro nas gerações do modelo P5, como Pentium e Pentium MMX, podem ser identificados na nota técnica *Embedded Pentium Processor Family Developer's Manual*[7], da Intel, sendo reproduzidos no Apêndice A. O manual técnico da Intel - *Intel Architecture Optimization Manual*[4] possui uma lista de todos os eventos passíveis de monitoração nos modelos Pentium Pro, Pentium II e Pentium III (gerações do modelo P6), sendo reproduzida no Apêndice B.

O acesso aos contadores de *hardware* difere ligeiramente para as máquinas P5 e P6, porém ambos possuem 40 *bits* e monitoram eventos aos pares. Estes contadores podem contar eventos ou medir a sua duração, dependendo do evento selecionado. Quando

selecionados para contar eventos, ocorre um incremento do contador a cada ocorrência do evento. Quando selecionados para medir duração de um evento, o contador conta o número de ciclos de *clock* enquanto uma condição pré-determinada é verdadeira.

A utilização dos contadores nos modelos P5 inicia-se com a escrita de uma palavra de *Control and Event Select Register*(CESR). O par de eventos que se deseja monitorar é selecionado através desse registro. O acesso é feito através da escrita no registro ECX utilizando a instrução WRMSR, conforme ilustrado na Tabela 2.7. O resultado da contagem de um determinado evento é obtido pela leitura das respectivas posições de memória através da instrução RDMSR.

TABELA 2.7 - DESIGNAÇÃO DE ECX PARA ACESSO AOS MSR DA FAMÍLIA P5

ECX	*Registro*	*Descrição*
0x10	TSC	*Time Stamp Counter*
0x11	CESR	*Control and Event Select Register*
0x12	CTR0	*Counter 0*
0x13	CTR1	*Counter 1*

O registro MSR a ser lido ou escrito é especificado pelo valor atribuído ao registro ECX. A instrução RDMSR lê o valor do registro MSR nos registros EDX:EAX. A instrução WRMSR escreve o valor existente nos registros EDX:EAX para o registro MSR especificado.

O controle da operação dos registros MSR é programado no registro CESR (32 *bits*), selecionando-se os eventos que serão monitorados pelos contadores CTR0 e CTR1. Para cada um dos contadores o CESR reserva um campo de 6 *bits* para a seleção do evento, um *bit* para utilização de pino externo de estado da CPU e três *bits* de controle. O campo de controle do contador permite a habilitação da contagem ou ainda, caso o evento selecionado assim o especifique, contar ocorrências do evento ou sua duração. A

Tabela 2.8 apresenta alguns dos eventos que podem ser monitorados nas máquinas com processadores da família P5.

A coluna "Código" mostra o valor que seleciona o evento descrito na coluna "Evento". A coluna "Contador" indica qual ou quais dos dois contadores podem monitorar o evento selecionado. A indicação “0/1” indica que ambos os contadores permitem a contagem do evento. A coluna "Tipo" informa se o evento é uma contagem de ocorrências ou uma medida de duração do evento.

TABELA 2.8 - EVENTOS DISPONÍVEIS NA FAMÍLIA P5

Código do Evento	***Contador***	***Evento***	***Tipo***
0x0	0/1	*Data Read*	Ocorrência
0x1	0/1	*Data Write*	Ocorrência
0x2	0/1	*Data TLB Miss*	Ocorrência
0x3	0/1	*Data Read Miss*	Ocorrência
0x4	0/1	*Data Write Miss*	Ocorrência
0x14	0/1	Instruções executadas	Ocorrência
0x1A	0/1	Travamento de pipeline	Duração
0x22	0/1	FLOPs	Ocorrência
0x28	0/1	*Data Read ou Data Write*	Ocorrência
0x29	0/1	*Data Read Miss ou Data Write Miss*	Ocorrência

A família P6 também possui dois contadores de *hardware* de 40 *bits*, porém com acesso diferenciado, disponibilizando uma gama maior de eventos . O P6 suporta quatro registros MSR, sendo dois registros diferenciados para seleção de eventos (PerfEvtSel0 e PerfEvtSel1) e dois contadores de eventos(PerfCtr0 e PerfCtr1). O acesso também é realizado através das instruções RDMSR para leitura dos contadores e WRMSR para escrita nos registros de controle.

Nas famílias P5 MMX e P6, a leitura dos contadores de eventos também pode ser realizada com a instrução RDPMC, que carrega a contagem corrente dos contadores 0 e 1 nos registros EDX:EAX.

Os registros de controle da família P6, PerfEvtSel0 e PerfEvtSel1, são um pouco mais complexos que seus equivalentes da família P5. Estes registros permitem que os contadores PMC possam medir não somente a fração de tempo gasto na medida de determinado estado, mas também o tempo gasto na espera do evento. Por exemplo, o tempo de espera para que uma interupção seja atendida pela CPU.

Uma outra característica interessante disponível nas famílias P6 é poder gerar um ciclo de exceção no caso de *overflow* dos contadores, não havendo necessidade de inspeção de *overflow* na rotina. A possibilidade de uso de máscaras é outra evolução da família P6 pois facilita a monitoração de valores pré-determinados para a contagem de eventos. A máscara consiste em carregar um determinado valor de contagem para um evento, de forma que o contador é apenas incrementado a partir desse valor. O controle permite que o contador incremente ao comparar o valor da contagem com esta máscara. A máscara pode ser habilitada para contar eventos abaixo do valor, igual ou superior. A Tabela 2.9 apresenta os endereços dos MSR para a família P6.

TABELA 2.9 DESIGNAÇÃO DE ECX PARA ACESSO AOS MSR DA FAMÍLIA P6

ECX	***Registro***	***Descrição***
0xC1	PerfCtr0	Contador CTR0
0xC2	PerfCtr1	Contador CTR1
0x186	EvntSel0	Seleção de evento do Contador CTR0
0x187	EvntSel1	Seleção de evento do Contador CTR1

Alguns dos eventos monitorados nas CPU's P6 estão apresentados na Tabela 2.10.

TABELA 2.10 - DESCRIÇÃO DE EVENTOS MONITORADOS NA CPU P6

Código	*Evento*	*Descrição*
0x43	DATA_MEM_REFS	Total de referências a memória
0x80	IFU_FETCH	Acessos a *cache* de instrução
0x81	IFU_FETCH_MISS	*Misses* de instrução
0x29	L2_LD	*Loads* ao *cache* L2 de dados
0x2A	L2_ST	*Stores* no *cache* L2 de dados
0x10	FLOP's	Total de operações em ponto flutuante
0xD0	INST_DECODER	Total de instruções decodificadas

2.3 – Arquiteturas de Alto Desempenho

A busca por alto desempenho sempre foi uma preocupação tanto de fabricantes quanto de usuários, utiliza-se desde o paralelismo interno existente nos processadores atuais da arquitetura RISC ou o uso misto de arquiteturas CISC com micro-instruções RISC, como a Intel, e chegando a arquiteturas paralelas, mostrando assim a complexidade das formas existentes para atender a necessidade atual de alto desempenho. Nesse contexto, os contadores de *hardware* podem contribuir de forma incisiva.

Iniciando com a arquitetura RISC, observa-se a busca por técnicas de obtenção do máximo de desempenho e, uma delas, utilizadas hoje nas arquiteturas CISC como os processadores Pentium, consiste no uso de *pipelining*, uma técnica onde o *hardware* do computador executa mais de uma instrução simultâneamente sem necessidade em aguardar o término de uma instrução para executar a próxima.

Da mesma forma que nas máquinas CISC, na concepção RISC, uma instrução passa por quatro estágios: busca, decodificação, execução e escrita, porém a passagem pelos estágios é executada em paralelo. Ao completar um estágio, o resultado é encaminhado para o estágio seguinte e iniciando o estágio atual com outra instrução. Cada instrução consome um ciclo de *clock* permitindo ao processador aceitar uma nova instrução a cada ciclo.

O termo superescalar define uma classe de processadores que permitem operações múltiplas em tempo de execução, através do uso desses *pipelines,* técnica que consiste em conectar estágios formando um "duto"(por essa razão recebe a designação *pipeline).*

Uma instrução entra no "duto" e passo a passo progride dentro dos estágios, onde a cada novo passo, uma nova instrução entra no estágio anterior. O passo para cada estágio é dado pela CPU e denominado ciclo de máquina.

O desempenho de um processador RISC depende basicamente da forma como o código que executa foi escrito e, assim, permite tornar explícita suas características superescalares.

No caso dos processadores Intel, a partir da família 386, diversos estágios foram inseridos, permitindo expor o paralelismo através de seis novas unidades internas. Uma *Bus Interface Unit* - unidade de interface com o barramento passa a ser encarregada do acesso à memória ou a unidades de entrada/saída para as outras unidades Uma *Code Prefetch Unit* - unidade de pré-acesso à instrução recebe o código objeto da unidade anterior e o coloca numa fila para decodificação pela *Instruction Decode Unit* – unidade de decodificação de instrução os tranformando em *micro-ops.* A *Execution Unit* – unidade de execução, executa as micro-instruções. A *Segment Unit* – unidade de segmento traduz o endereço lógico em endereço real sendo também responsável por algumas proteções de acesso, mantém um *cache* com informações sobre as últimas 32 páginas acessadas mais recentemente.

O lançamento da família 486 trouxe mais capacidade de execução paralela através da expansão das unidades de decodificação de instrução de execução existentes na família 386 em cinco estágios para o *pipeline.* Cada estágio passa a operar com os outros em até cinco instruções em diferentes níveis de execução.

A evolução lógica da família 486 seria permitir aos próximos processadores desempenho superescalar. Assim, os processadores da geração seguinte, denominados

Pentium, receberam um segundo *pipeline* de execução para assim permitir desempenho superescalar. Os dois *pipelines* receberam a designação de U e V e juntos podem executar duas instruções simultâneamente.

Nos processadores Pentium da quinta geração(P5), novas características com o objetivo de reduzir os gargalos internos comuns nos processadores da família 486 foram adicionadas. A evolução ocorrida nesses processadores foi considerável nas seguintes áreas:

a) Aumento do barramento de dados para 64 *bits*;
b) *Cache* de Instrução exclusivo;
c) *Cache* de Dados exclusivo;
d) Duas unidades de execução de operações com inteiros;
e) Uma unidade de execução para operações com ponto flutuante.

Com aumento do barramento de dados para 64 *bits* o preenchimento das linhas de *cache* ficou mais eficiente, consumindo menos tempo. O *cache* de instrução dedicado alimenta duas unidades para operações com inteiros e uma unidade independente para operações em ponto flutuante através de dois *pipelines*, tornado os processadores Pentium superescalares, dessa forma permitindo a execução em paralelo de duas instruções com inteiros e uma instrução em ponto flutuante.
A partir da sexta geração de processadores(P6), a Intel passa a utiliza uma arquitetura superescalar de três vias, ou seja, permite executar três instruções a cada ciclo de máquina. Três unidades de decodificação de instrução trabalham em paralelo para decodificar as operações em códigos *micro-ops*, que são executados fora de ordem pelas cinco unidades de execução em paralelo(duas unidades para execução de operações com inteiros, duas unidades para operações com ponto flutuante e uma unidade de interface com memória).

Assim, um programa deve permitir da forma mais eficiente possível a utilização da característica superescalar nas arquiteturas RISC ou mesmo nos processadores Pentium

por permitir desempenho superescalar. A quantidade de operações que podem rodar em paralelo depende assim não apenas do processador, mas também do programa estar escrito de forma a tirar vantagem dessa característica.

Nos casos mais complexos as arquiteturas paralelas permitem uma distribuição de tarefas ou ainda uma distribuição de dados de forma a solucionar no menor tempo possível um determinado problema computacional.

As arquiteturas paralelas podem ser divididas em basicamente três modelos principais definidos por Michel Flynn[13][14]:

a) *Single Instruction, Multiple Data* – SIMD com memória distribuída;
b) *Multiple Instruction, Multiple Data* – MIMD com memória distribuída ;
c) *Multiple Instruction, Multiple Data* – MIMD com memória uniforme compartilhada;
d) *Multiple Instruction, Multiple Data – MIMD* com memória não uniforme compartilhada.

A arquitetura do tipo SIMD consiste na utilização de um conjunto de processadores executando as mesmas operações ao mesmo tempo porém atuando em conjuntos diferentes de dados. Cada processador possui memória local e se comunica com os processadores vizinhos. A memória é distribuída e portanto o programador deve se encarregar da distribuição das variáveis envolvidas. O conjunto de processadores é conectado a uma máquina que fica responsável pela distribuição das instruções a todos os outros processadores.

A distinção entre máquinas MIMD com memória distribuída ou compartilhada é muito importante para o programador. No primeiro caso a memória está particionada e portanto as transferências de dados é explicita entre processadores, caso trabalhem em um mesmo problema. No segundo caso, os processadores vêem a mesma memória como uma área única visível a todos.

Como exemplos de máquinas MIMD com memória uniforme compartilhada pode-se citar a SGI Power Challange com até 36 processadores MIPS-R10000, a estação DEC 8400 com processadores Alpha 21164 e Estação SUN com processadores UltraSparc-2. Os processadores dessas máquinas compartilham a memória igualmente, recebendo a denominação *Uniform Memory Access* –UMA.

Uma outra classe de máquinas paralelas acrescenta uma memória local privada ao processador ou os agrupa em nós, onde acessam uma memória mais lenta através de uma interconexão desses nós, recebendo a denominação de máquinas *Non-Uniform Memory Access* – NUMA. Estações como a SGI-Origin utilizam nós com dois processadores MIPS-R10000 permitindo um máximo de 128 CPUs. A estação AV-20000 da Data General utiliza quatro processadores Pentium por nó, permitindo a interconexão de até 32 processadores.

As técnicas para obtenção de alto desempenho dessas máquinas exige que o programador conheça o tipo de implementação utilizada, por exemplo, as máquinas com memória distribuída exigem mais do programador pois é necessário identificar a localização dos dados e controlar sua movimentação.

2.4 - Gerenciadores de Dispositivos no Linux

No sistema POSIX, base do Linux, diversos processos concorrentes devem atender a diferentes tarefas. Cada processo solicita ao sistema recursos da máquina, como tempo de CPU, memória, conexão com a rede qualquer outro serviço existente, sendo o núcleo do Linux responsável em atender a esses pedidos.

A base de um sistema operacional como o Linux protege o ambiente interno de seu núcleo, identificando-o como espaço reservado, enquanto as aplicações rodam no chamado espaço do usuário. Assim, os modos de execução recebem a denominação de "espaço do núcleo", quando dispõe-se de certos privilégios, e "espaço do usuário" onde impõe-se certas restrições de acesso ao *hardware*.

Assim, um acesso ao *hardware* exige que se trabalhe no espaço do núcleo. A requisição ao sistema operacional para executar uma operação de acesso ao *hardware* ou a uma operação protegida, é realizada por chamadas a *system calls* - rotinas internas do sistema, através de chamadas iniciais a macros que as executam e estão definidas na biblioteca de sistema "libc". Esta biblioteca contém as funções em linguagem “C” utilizadas pelo sistema operacional.

Ao escrever um programa, o código irá rodar no espaço do usuário e recebe a denominação de modo usuário, enquanto os acionadores de dispositivos e o *filesystem* do Linux rodam no espaço do núcleo, denominado de modo do núcleo.

A única forma de um acionador de dispositivo manipular um determinado *hardware* é através das chamadas aos *system calls* que não passam de funções específicas que fazem a interface entre o modo usuário e o modo do núcleo, permitindo assim acesso a um determinado *hardware*. Um *system call* é o modo como uma aplicação rodando no espaço do usuário solicita um determinado serviço que é protegido no espaço do núcleo.

Como exemplo, um *system calls* muito utilizado é a chamada à função “ioctl”, que permite a manipulação direta de periféricos. Todos os *system calls* possuem seus protótipos definidos em /usr/include/asm/unistd.h e a partir do kernel 2.0.34, o Linux passa a contar com 164 *system calls.* Este *include* é inserido nos procedimentos desenvolvidos para acesso aos contadores de *hardware*

As tarefas de gerenciamento desses recursos podem ser divididas em processos, memória, *filesystem* - sistema de arquivos, rede e *device* control - controle de dispositivos. O gerenciamento de dispositivos é a base de construção do trabalho aqui apresentado.

O Linux suporta diferentes sistemas de arquivos, inclusive MS-DOS e Windows. O sistema de arquivos do Linux é denominado “ext2” e tem como objetivo criar regras

para organizar os arquivos dentro do ambiente Linux da mesma forma que todos os outros sistemas operacionais.

O acesso a periféricos no ambiente Linux é realizado através de interfaces de *software* entre o espaço reservado do núcleo e o dispositivo ou periférico, denominados *device drivers* - acionadores de dispositivos, que permitem a conexão dos aplicativos com os periféricos, utilizando estruturas denominadas módulos. Uma boa parte das operações realizadas pelo núcleo eventualmente faz referência a um dispositivo físico.

A própria CPU pode ser considerada como um dispositivo, com funções específicas e constituída por blocos menores que disponibilizam serviços para o núcleo. Assim, como pretende-se utilizar os contadores de *hardware*, deve-se considerá-los como um único dispositivo e dessa forma disponibilizá-los para uso. O acesso aos registros da CPU é feito através de um *device driver* - acionador de dispositivo e a comunicação entre o núcleo e o aplicativo será responsabilidade de um módulo. O Linux distingue três tipos de dispositivos, cada um direcionado a um tipo de periférico. A forma de adicionar um dispositivo ao núcleo é através de um módulo, que tem a função de anexá-lo externamente. A Figura 2.1 fornece uma visão geral do núcleo do Linux, representando o acoplamento dos módulos ao núcleo e assim permitindo a transferência de dados entre o espaço do usuário e o espaço do núcleo.

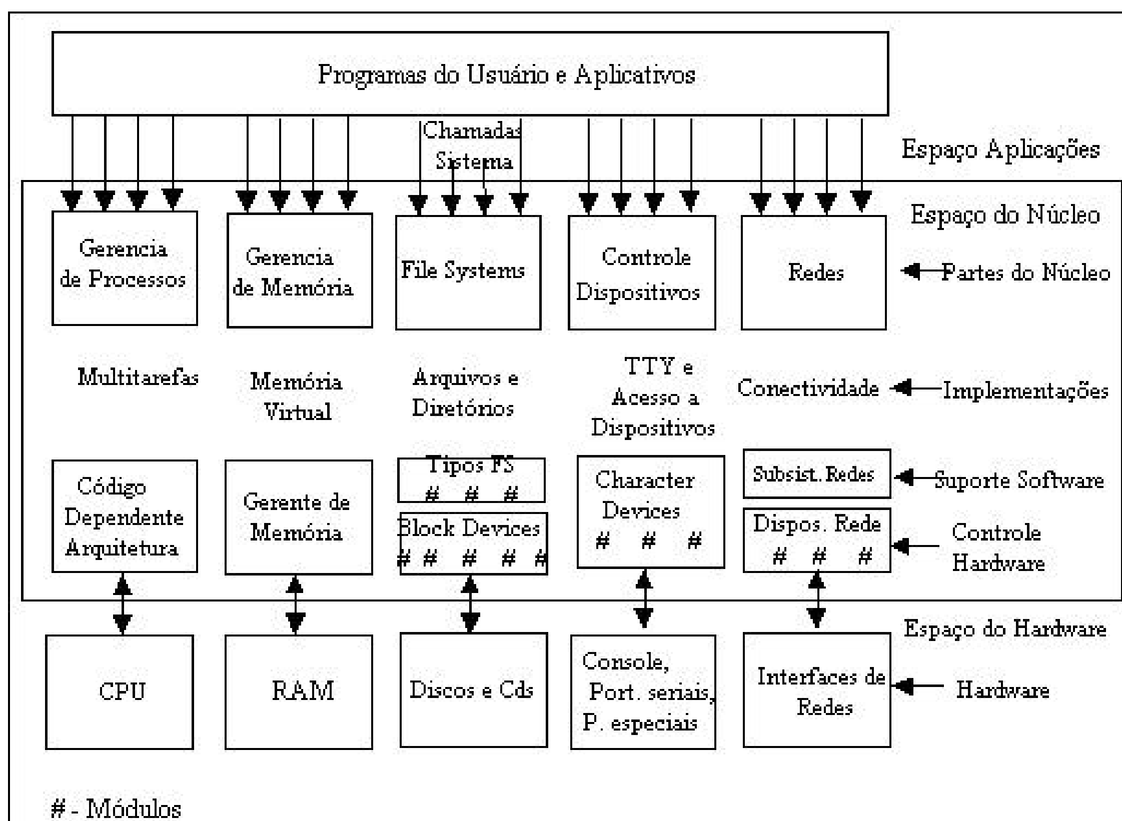


Fig. 2.1 - Visão gráfica do núcleo do Linux.

FONTE: Adaptada de Rubini (1998, p. 4).

O Linux define que cada módulo acessa apenas a um único dispositivo, e os classifica em três tipos: *block devices*, *character devices* e *Network interfaces*, porém existem outros gerenciadores como o próprio *filesystem* do Linux. A diferença básica entre os *block devices* e os *character devices* é simples e não é rígida. Define-se *character devices* como dispositivos que podem ser acessados como arquivos e os *block devices* como passíveis de hospedarem um *filesystem*. Enquanto um *character device* troca dados serialmente (com um número pré-determinado de *bytes*), um *block device* exige a transferência de um bloco de dados com tamanho também pré-determinado.

2.5 - Módulo de Acesso aos Contadores

O acesso aos contadores de *hardware* utilizando o conceito de módulo partiu de Stephan Meyer [11]. A idéia de considerar os contadores como um dispositivo permitiu seu acesso de forma muito simples e eficiente, através de um acionador de dispositivo do tipo *character device*. Com a possibilidade de acesso rápido aos contadores internos, Patrick Goda e Michael Warren[12] desenvolveram o aplicativo "perfmom" utilizando o módulo desenvolvido por Meyer. No Linux as funções do núcleo e os módulos são executados em um único *thread*, permitindo assim que o acesso aos contadores seja imediato.

Um módulo passa a ter sua funcionalidade disponível quando acoplado ao núcleo. Após instalado, um módulo recebe um número de *inode* no *filesystem*, sendo registrado com um nome no diretório */dev*. Este número recebe a denominação *major*, é exclusivo do dispositivo e aponta apenas para o seu módulo designado. A cada nova instância do módulo, o *filesystem* associa um outro número ao processo filho, chamado de *minor*. Assim, a cada novo dispositivo deve-se associar um número de *major* e de *minor*. O valor atribuido ao *minor* é utilizado apenas pelo módulo. A partir do *kernel* 2.2.0 do Linux, a atribuição do valor "10" ao *major*, corresponde a *miscellaneous drivers*, ou seja, acionadores genéricos. O valor *major* igual a "10" informa que o dispositivo é genérico e o *minor* identifica assim os inúmeros dispositivos definidos pelo núcleo. A Tabela 2.11 apresenta alguns dispositivos identificados como genéricos pelo núcleo.

Cada arquivo recebe um número que corresponde à sua designação dentro da estrutura de representação de arquivos do Linux denominado *inode*.

TABELA 2.11 - IDENTIFICAÇÃO DE DISPOSITIVOS GENÉRICOS

Minor	Dispositivo	Identificação
0	/dev/logibm	Logitech bus mouse
1	/dev/psaux	PS/2-style mouse port
134	/dev/apm_bios	Advanced Power Management BIOS
135	/dev/rtc	Real Time Clock
142	/dev/msr	x86 model-specific registers

Assim, qualquer módulo que acesse os contadores internos deve se identificar ao núcleo como:

major = 10 - Dispositivo genérico;
minor = 142 - x86 *model-specific registers*

O módulo é acessado através desses números como um arquivo padrão do *filesystem*, aceitando solicitações de abertura de arquivo, leitura, escrita ou encerramento, como um arquivo comum. Um acionador de dispositivo está necessariamente associado a um módulo, que é dividido em duas partes, uma que registra o dispositivo no núcleo e uma outra parte que faz a interface com o periférico através das funções de acesso a arquivos do *filesystem*. O núcleo mantém uma tabela com os acionadores de dispositivos disponíveis, onde cada um é identificado e registrado pelo número de *major*. Este número informa ao núcleo onde se encontra o módulo.

O registro do número do dispositivo nesta tabela de dispositivos disponíveis é feito através de uma função de registro do módulo, que por sua vez retorna o número de *major* registrado. O acionador de dispositivos necessita de algumas funcionalidades, que serão responsáveis pelas ações nos periféricos e chamadas pelo aplicativo. O núcleo toma conhecimento dessas funções através de uma estrutura de ponteiros que identifica cada função autorizada pelo módulo. No caso específico aqui descrito, o módulo permite que o usuário o acesse com as funções de acesso a arquivo como: *open*, *read*, *write*, *seek* e *close*.

Os contadores de *hardware* são programados através da escrita de palavras de controle em dois registradores, responsáveis pela seleção dos eventos que serão monitorados. O pedido de abertura do módulo registra no núcleo as suas funcionalidades e permite ao usuário que prossiga com a escrita das palavras de controle nos registradores. O módulo também habilita o usuário a realizar a leitura dos registros com os resultados obtidos. As características básicas (funcionalidade) do módulo são implementadas pela estrutura "file_operations". Nesta estrutura, registram-se as funções básicas.

Após instalação do módulo, as funções de leitura e escrita ficam disponíveis para a transferência dos dados obtidos pelos contadores, através das instruções RDMSR para leitura e WRMSR para escrita. A partir do *kernel* 2.2.x, o Linux sofreu alterações nas funções de transferência entre o espaço do *kernel* e do usuário. Assim, o módulo aqui apresentado é compilado apenas para distribuições com *kernel* igual ou superior ao 2.2.0.

Neste capítulo, apresentou-se alguns dados sobre os contadores de *hardware* existentes nas diversas arquiteturas de processadores, bem como a sua disponibilidade nos processadores Intel, especificamente nas famílias P5 e P6. As instruções disponibilizadas pela Intel para acesso aos contadores foram apresentadas, com informações sobre a necessidade de se identificar a família de CPU sob análise. Discutiu-se a forma de acesso aos contadores no ambiente Linux e sua implementação como módulo utilizando o conceito de acionador de dispositivo.

CAPÍTULO 3

METODOLOGIA DE MONITORAÇÃO E OTIMIZAÇÃO DE DESEMPENHO

Conhecendo como é feita a conexão entre o programa do usuário e os contadores de hardware, deve-se instalar o módulo de acesso. Neste capítulo, descreve-se o funcionamento e instalação do módulo MSR, responsável pela conexão entre o programa do usuário e os contadores de hardware. Foram escritas rotinas em C, com funções e procedimentos necessários ao acesso do módulo, que fazem a seleção dos eventos que serão monitorados e a leitura das contagens resultantes. A instrumentação de um trecho de programa, o qual simula um certo padrão de referência à memória, com as rotinas apresentadas, permite a monitoração dos sucessos e falhas nos acessos à memória *cache*. As rotinas desenvolvidas permitem a instrumentação de trechos de programas escritos em linguagem C, Fortran 77 e Fortran 90.

3.1 - Captura dos Dados de Desempenho

A captura dos dados de desempenho implica na leitura dos contadores de hardware, e o Capítulo 2 apresentou como é possível acessar esses contadores através de módulos, responsáveis pela interface com o acionador de dispositivo. Para utilização do módulo de acesso aos contadores, é necessário sua inicialização e registro no núcleo do Linux.

A conexão do módulo ao núcleo do Linux é realizada pela função de inicialização “init_module()”[2], responsável pelo registro das funcionalidades que estarão disponíveis aos usuários, sendo portanto o ponto de entrada do módulo. A instalação do módulo é iniciada com o comando "insmod", que executa a função de inicialização “init_module()”. Assim, é estabelecida a conexão do módulo ao núcleo. Este comando possui a seguinte sintaxe e deve ser executado com permissão de super-usuário:

```
# insmod [nome_do_módulo]
```

O comando é executado no diretório onde se encontra o arquivo-objeto do módulo. Assim, a instalação do módulo MSR exige a execução da seguinte linha de comando:

```
# insmod msr
```

Este comando instala o módulo MSR, arquivo-objeto compilado a partir do arquivo fonte msr.c, cuja listagem pode ser encontrada no Apêndice C. Em seguida, é necessário criar o arquivo do acionador de dispositivo para os contadores. Esta função é realizada pelo comando "mknod" do Linux. Este comando possui a seguinte sintaxe e deve ser executado como super-usuário:

```
# mknod [--mode = MODO] [caminho/nome_do_dispositivo]    \
        [tipo_de_dispositivo] [major] [minor]
```

A opção *mode* designa a permissão para uso do *character device* e deve ser ajustada para que todos os usuários possam ler e escrever no arquivo. Recomenda-se que o diretório /dev seja o local de instalação do *character device*[2]. O tipo do dispositivo indica se o dispositivo é um *character device* ou um *block device*. Os números de *major* e *minor* indentificam o acionador, neste caso do tipo *character device,* para o *filesystem* do núcleo.

Assim, para que o acionador fique disponível a todos os usuários, deve ser executada a seguinte linha de comando:

```
# mknod  --mode = 666  /dev/msr c 10 142
```

A linha de comando acima registra o acionador do tipo *character device*, identificado por msr e instalado no diretório /dev, e o identifica como um dispositivo genérico, pois registra os seguintes valores, reservados para o módulo msr:

major = 10 ;

minor = 142.

A Figura 3.1 exemplifica como funciona a conexão do módulo com o núcleo. O núcleo permite que a funcionalidade do *filesystem* fique disponível ao módulo. O módulo, por sua vez, identifica quais funcionalidades estarão disponíveis para o usuário, sendo definidas pelas funções registradas no módulo.

Como exemplo, a função de leitura (*read*) do *filesystem* do Linux é uma das funcionalidades disponibilizada pelo núcleo. O módulo registra essa funcionalidade como disponível para utilização, e a habilita na estrutura "file_operations", definindo um ponteiro para a função de leitura "msr_read" existente no módulo. Com a funcionalidade de leitura disponível, o módulo implementa esta função para o dispositivo. A função “init_module” aponta para a estrutura “msr_device,” que por sua vez executa a conexão entre o número de *minor* e o módulo msr, apontando para uma outra estrutura onde as funcionalidades do módulo são definidas.

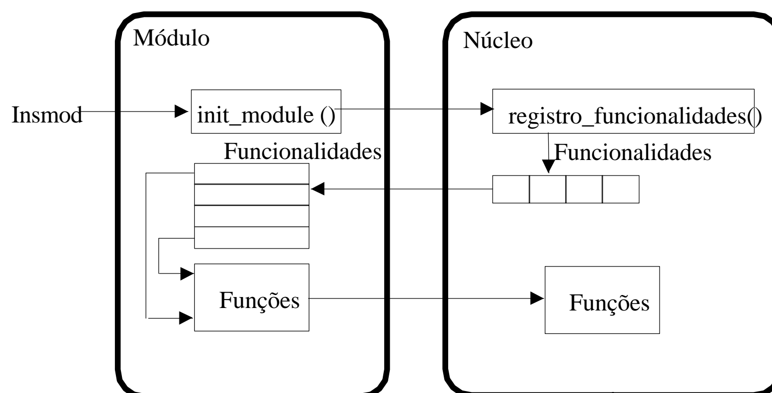


Fig. 3.1 - Conexão do módulo ao núcleo do Linux.

FONTE: adaptada de Rubini(1998, p. 15).

A função "init_module()" utiliza a função "misc_register()" definida na libc do Linux para apontar para a estrutura que registra o *character device* :

```
int init_module(void)
        {
        misc_register( &msr_device);
        return 0;
        }
```

A estrutura "miscdevice" é definida pela libc como estrutura de registro de dispositivos genéricos, definindo o número de *minor*, o nome do módulo e apontando para a estrutura onde estão definidas as funcionalidades do módulo.

```
static struct miscdevice msr_device ={
        MSR_MINOR,
        "msr",
        &msr_fops
        };
```

No módulo msr as funções definidas são as seguintes: abertura do dispositivo (*open*), busca (*seek*), leitura (*read*), escrita (*write*) e encerramento (*close*).

```
static struct file_operations msr_fops = {
        msr_seek,
        msr_read,
        msr_write,
        NULL,           //readdir
        NULL,           //poll
        NULL,           //ioctl
        NULL,           //mmap
        msr_open,
        NULL,           //flush
        msr_close,
        NULL,           //fsync
        NULL,           //fasync
        NULL,           //check_media_change
        NULL,           //revalidate
        NULL            //lock
};
```

A leitura e a escrita aos contadores de hardware são realizadas pelas funçoes "msr_read" e "msr_write," respectivamente, ambas definidas na estrutura de funcionalidade. O módulo completo está listado no Apêndice C. A leitura dos registros MSR é realizada através da instrução RDMSR, conforme discutido anteriormente no Capítulo 2. A instrução em baixo nível deve ser executada a partir de um procedimento em alto nível. Assim, a inserção de diretivas em linguagem assembly no código em linguagem C utiliza a sintaxe desenvolvida pela AT&T/UNIX.

A sintaxe para códigos em linguagem *assembly* define que a instrução deve ser declarada após a diretiva "asm" com a definição dos parâmetros de entrada e saída, conforme apresentado abaixo:

```
__asm__ (
        "instrução"
        : lista de registros de saída
        : lista de registros de entrada
        : lista de nomes  encobertos
        );
```

Assim, o código para leitura dos contadores de hardware terá o seguinte formato:

```
__asm__ __ volatile__ (
                "rdmsr"
                :"=a" (lo), "=d" (hi)
                :C (p)
                :"eax", "ecx", "edx"
                );
```

Os valores lidos são passados para os ponteiros "hi" e "lo". O ponteiro "p" passa valor para ecx. A lista de nomes encobertos especifica ao compilador quais registros serão lidos ou escritos. Assim, associa-se a "a" o registro eax, a "d" associa-se o registro edx e a "c" ao registro ecx.

O mesmo método é utilizado para o código de escrita nos MSR, conforme listado abaixo:

```
__asm__ __volatile__ (
                        "wrmsr"
                        : /* sem valores de saida */
                        : C (p), "a" (lo), "d" (hi)
                        : "eax", "ecx", "edx" );
```

Após leitura dos contadores, é necessário transferir os dados do espaço do kernel para o espaço do usuário. A transferência é realizada pela instrução “copy_to_user”. A instrução deve retornar quatro bytes com os valores dos contadores pois cada registro MSR tem 32 bits :

```
copy_to_user(buf, &lo, 4)
copy_to_user(buf, &hi, 4)
```

Os dados do espaço do kernel são apontados por "lo" e "hi" e transferidos, neste caso, para a área apontada por "buf".

A transferência de dados do espaço do usuário para o espaço do kernel é realizada pela instrução “copy_from_user”:

```
copy_from_user( &lo, buf, 4 )
copy_from_user( &hi, buf + 4, 4 )
```

Os quatro primeiros bytes da área "buf" são apontados por "lo" e os quatro últimos apontados por "hi".

A temporização dos trechos a analisar pode ser executada pela instrução RDTSC, disponível nas famílias Intel P5 MMX e P6. O mesmo formato acima é utilizado para chamada à instrução RDTSC.

```
__asm__ __volatile__ (
                "rdtsc"
                : "=a" (loword), "=d" (hiword)
                : /* sem registros de entrada */
                : "eax", "edx"
                );
```

Os valores lidos do registro TSC são passados para as variáveis "loword" e "hiword" de 32 bits. O valor real é obtido pela composição dos 32 bits inferiores armazenados por "loword" com seus 32 bits superiores armazenados em "hiword".

Os endereços de acesso aos contadores de hardware, bem como a forma de programação dos eventos, diferem entre as famílias de processadores, o mesmo ocorrendo nas formas de escrita/leitura dos contadores de *hardware*. Por essa razão, deve-se executar a instrução CPUID, para identificação da CPU. O procedimento checkcpuid executa a instrução CPUID com o registro eax carregado com o valor zero. O retorno da instrução traz no próprio eax a identificação da família do processador. Um valor eax = 1 indica tratar-se de processador P5 e eax = 2 indica tratar-se de processador P6.

O código utilizando a sintaxe AT&T/UNIX para execução da instrução CPUID é apresentado abaixo:

```
__asm__ __volatile__ (
                "cpuid"
                : "=a" (eax)
                : "a" (0x00)
                : "eax"
                );
```

3.2 - Interface para Programação em Alto Nível

O acesso aos contadores de hardware é realizado através de chamadas diretas às funções habilitadas pelo módulo MSR; assim, as rotinas que executam as funcionalidades do módulo MSR devem permitir o acesso aos contadores a partir de trechos em Fortran ou em C. Para simplificar este acesso, foram desenvolvidas rotinas de interface,

disponíveis para programas nestas duas linguagens. O trecho que se deseja monitorar deve ser antecedido pela chamada ao procedimento “iniciaregistros” e terminado pela chamada à função “leregistros”.

A instrumentação de trechos de código em C ou em Fortran utiliza as mesmas chamadas. O arquivo rotinasPMC.c, listado no Apêndice D, contém todas as subrotinas de monitoração. O protótipo para chamadas a partir de um programa em linguagem C tem a seguinte sintaxe:

```
void iniciaregistros( int r_0, int r_1 )
```

A linguagem C passa valores para o procedimento. Em Fortran, o procedimento passa ponteiros e possui a seguinte sintaxe:

```
void iniciaregistros_( int *r_0, int *r_1 )
```

Em Fortran, as funções externas escritas em outra linguagem devem ser terminadas com o caractere especial *underscore*(sublinhado). Assim, não há necessidade de se dar nomes diferentes para as funções que tratarão chamadas em C e em Fortran. Os eventos que serão monitorados são passados para a função segundo os códigos de evento, listados no Apêndice A para os processadores P5 e no Apêndice B para os processadores P6.

A ordem dos parâmetros corresponde aos contadores. Assim, reg_0 define o evento no contador 0 e reg_1 no contador 1. Deve-se observar que alguns eventos são monitorados exclusivamente no contador 0. O procedimento para inicialização da instrumentação segue o fluxograma da Figura 3.2.

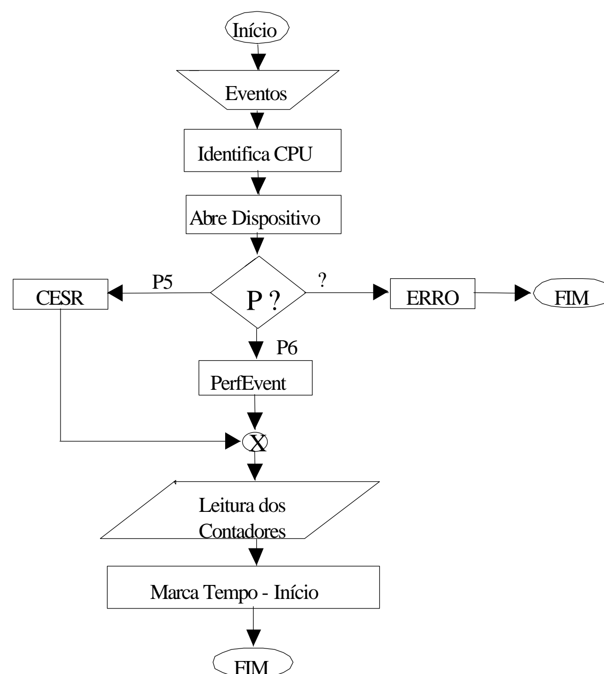


Fig. 3.2 - Fluxograma da subrotina "iniciaregistros".

Os códigos dos eventos monitorados são passados como parâmetros para a subrotina "iniciaregistros". Em seguida, é identificada a CPU (instrução CPUID), garantindo que o programa carregue os registros MSR corretamente. Nos processadores P5, o registro CESR identifica os eventos que serão monitorados nos dois contadores. Nos processadores P6, os registros são carregados individualmente nos registros PerfEvntSel0 e PerfEvntSel1.

Após execução do comando de ativação dos contadores, são chamadas as funções de leitura de MSR, seguido da leitura do registro TSC para a temporização do trecho em análise. Inicialmente, a função de abertura do módulo abre o dispositivo para escrita e leitura:

```
arquivoMSR = open( "/dev/msr", O_RDWR )
```

Para se escrever no registro único CESR dos processadores P5, deve-se identificar o endereço do registro (0x11) e prosseguir com um comando de escrita, conforme descrito abaixo. Deve-se observar que os registros são lidos como um arquivo e assim contados seqüencialmente.

```
lseek( arquivoMSR, 0x11, SEEK_SET );
write( arquivoMSR, &CESR, 8 );
```

O *handler*, identificado por "arquivoMSR", procura o módulo msr, um arquivo, onde cada registro MSR corresponde a um caractere do arquivo. Assim, o acesso ao registro CESR corresponde ao *byte* 0x11 do arquivo. A escrita e leitura do MSR deve garantir que haverá a transferência de oito *bytes*, pois cada um dos dois contadores ocupa 4 *bytes*. A variável CESR aponta para a palavra de comando que aciona a contagem dos eventos selecionados.

Os processadores P6 possuem dois registros MSR de comando, PerfEvtSel0 e PerfEvtSel1. Cada contador de evento é programado individualmente:

```
lseek( arquivoMSR, 0x187, SEEK_SET );
write( arquivoMSR, &PerfEvt_1, 8 );
lseek( arquivoMSR, 0x186, SEEK_SET );
write( arquivoMSR, &PerfEvt_0, 8 );
```

A execução da função de escrita no registro de comando do contador 0 (PerfEvt_0) inicializa a contagem dos eventos selecionados. Na saída do módulo deve ser fechado o dispositivo, através da chamada do procedimento "cleanup_module()".

Ao final do trecho monitorado deve-se executar a função "leregistros", responsável pela leitura final dos contadores e pela subtração dos valores inicialmente encontrados pela

subrotina "iniciaregistros". A diferença entre os valores fornece as contagens dos eventos selecionados e o tempo de execução do trecho.

A Figura 3.3 apresenta o fluxograma da subrotina “leregistros”. A sintaxe da chamada de função a partir de um programa em linguagem C é dada por:

```
long leregistros( int *contador_0, int *contador_1 )
```

Para programas em Fortran, a sintaxe é a seguinte:

```
long leregistros_( int *contador_0, int *contador_1 )
```

A função retorna um valor de tipo *long* e dois ponteiros para os resultados obtidos na monitoração dos eventos.

A função “leregistros” finaliza com a leitura do registro TSC. O valor obtido permite uma temporização precisa do trecho em análise, com valores em microsegundos.

Novamente o dispositivo msr é aberto e passa a identificar a família da CPU. Esta verificação não é realmente necessária, pois o procedimento “iniciaregistros” poderia retornar um parâmetro com a identificação da CPU. Porém, no caso de se trabalhar com programação paralela, esta verificação pode permitir a monitoração de alguns eventos em máquinas com processadores diferentes.

.

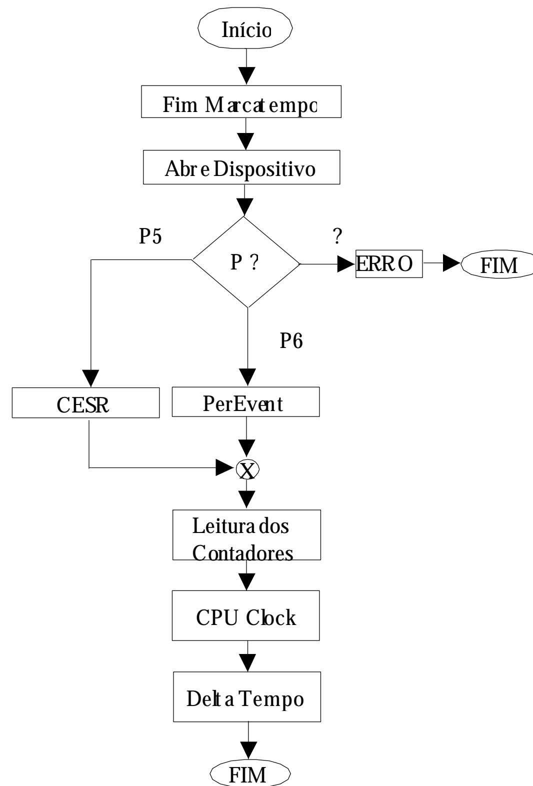


Fig. 3.3 - Fluxograma da subrotina "leregistros".

A temporização do TSC informa as contagens de pulsos de *clock*; assim, a verificação da frequência de *clock* da CPU permite a medida do tempo de execução.

Nos processadores P5 e P6, os resultados da monitoração estão disponíveis nos dois contadores de *hardware*. Os endereços de acesso diferem, mas o processo de leitura é o mesmo. Após abrir o arquivo msr, executa-se a leitura dos contadores:

```
lseek( arquivoMSR, contador_0, SEEK_SET );
read( arquivoMSR, &count_0, 8 );
lseek( arquivoMSR, contador_1, SEEK_SET );
read( arquivoMSR, &count_1, 8 );
```

A Tabela 3.1 mostra os endereços para acesso aos resultados das contagens de eventos para os processadores P5 e P6.

TABELA 3.1 - ENDEREÇOS DOS CONTADORES DE *HARDWARE*

Contador	Endereço P5	Endereço P6
Contador_0	0x12	0xC1
Contador_1	0x13	0xC2

3.3 - Utilização de Dados de Desempenho

Um dos casos clássicos descrito por Dowd[13], onde uma matriz bidimensional tem acesso preferencial por linhas ao invés de por colunas, será aqui analisado. Este caso reflete claramente a potencialidade das ferramentas desenvolvidas na análise detalhada de um trecho de programa.

Um compilador C armazena em posições adjacentes de memória os elementos de uma mesma linha de uma matriz bidimensional, ou seja, uma linha é armazenada seqüencialmente. Em Fortran, tem-se o inverso: as colunas são armazenadas seqüencialmente. No exemplo abaixo, descrevendo uma soma de matrizes utilizando a linguagem C, efetua-se a análise dando preferência ao armazenamento por linhas e, em seguida, por colunas:

a) Operação seqüencial com linhas da matriz:

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

b) Operação seqüencial com colunas da matriz:

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

Como exemplo, implementa-se a monitoração de alguns eventos para os trechos de código acima. Os testes foram realizados em CPU P5 (Pentium MMX 200 MHz). A medida da taxa de acertos ao *cache* pode ser obtida pela monitoração dos acertos e falhas à memória *cache* de dados, eventos disponíveis tanto nas CPU's da família P5 como na família P6. Assim, a taxa de acertos ao *cache* é calculado da seguinte forma:

cache_hit_rate = (1 - data_cache_misses/(total de "loads" + "stores")) * 100   (3.1)

Para a determinação da taxa de acertos ao *cache* de dados, *cache hit rate*, deve-se monitorar três eventos: *data cache misses*, total de *loads* e total de *stores*. Assim, é necessário analisar os acessos à memória monitorando aqueles três eventos. No caso específico da família P5, existem dois eventos que permitem a determinação direta do *cache hit rate*: o evento *data reads or writes* fornece o total de *loads* e *stores* ao *cache* de dados, e o evento *data read or write misses* fornece o total de falhas de acesso ao *cache* de dados. O programa em linguagem C para a captura dos dados de desempenho nos dois casos, de referência à memória, está listado na Figura 3.4. A matriz é monitorada no acesso por linhas e por colunas, para comparação dos acertos ao *cache* de dados através da medida da taxa de acertos ao *cache* (*cache hit rate*). Os tempos de execução dos trechos, bem como os valores do *cache hit rate*, são apresentados na Tabela 3.2.

TABELA 3.2 - COMPARAÇÃO DE ACERTOS AO *CACHE*

Tipo Acesso	Tempo(mseg.)	Cache hit rate(%)
linha	752	96,1
coluna	1640	90,5

Neste exemplo, a redução no tempo de execução preferenciando o acesso por linhas, com conseqüente aumento de acertos em acessos à memória *cache*, é de 96.1 %.

Este capítulo apresentou como se instala o módulo MSR e a forma de acesso ao acionador *character device*, permitindo o acesso aos contadores de hardware. Os procedimentos desenvolvidos para captura dos dados de desempenho foram analisados e testados em um exemplo de código com padrão de referência de acesso à memória.

O próximo capítulo utiliza a metodologia aqui apresentada para a otimização de desempenho de programas seqüenciais, aplicando algumas técnicas clássicas conhecidas. A captura de dados de desempenho é apresentada, com a correspondente instrumentação de alguns trechos de programas.

```
void main(void)

{
int reg_0, reg_1;
int c_0_linhas,c_1_linhas, c_0_colunas, c_1_colunas;
long tempo_exec_linhas, tempo_exec_colunas;
int *count_0, *count_1;
int N = 1000; // definicao das dimensoes das matrizes
float a[N][N], b[N][N], k=10.0;

/* eventos monitorados */
reg_0 = 0x28; // total de loads e stores
reg_1 = 0x29; // total de data cache misses

/* inicializa matrizes */
for( i = 0; i < N; i++)
        for ( j = 0; j < N; j++){
        a[ i  ][ j ] = i*0.1 + j*0.2;
        b[ i  ][ j ] = i*0.2 + j*0.1;
        }

/*-------------------------------*/
/* trecho sob analise           */
/* preferenciando linhas        */
/* -------------------------- ---*/

/* insercao de chamada de monitoracao */
iniciaregistros ( reg_0, reg_1 );

//---------linhas---------------------
for ( i = 0; i < N; i++)
        for ( j = 0; j < N; j++)
        a[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ]*k
//-----------------------------------
tempo_exec_linhas = leregistros( &count_0,  &count_1 );
c_0_linhas = count_0;
c_1_linhas = count_1;

/*---------------------------------*/
/* trecho sob analise              */
/* preferenciando colunas          */
/* -------------------------------- */

/* insercao de chamada de monitoracao */

iniciaregistros ( reg_0, reg_1 );

//---------colunas---------------------
for ( i = 0; i < N; i++)
        for ( j = 0; j < N; j++)
        a[ j ][ i ] = a[ j ][ i ] + b[ j ][ i ]*k
//-----------------------------------
tempo_exec_colunas = leregistros( &count_0,  &count_1 );
c_0_colunas = count_0;
c_1_colunas = count_1;

}
```

Fig. 3.4 - Programa para análise de padrão de referência a memória.

CAPÍTULO 4

TÉCNICAS DE OTIMIZAÇÃO DE PROGRAMAS SEQÜENCIAIS

Neste capítulo, discute-se algumas técnicas de otimização consideradas clássicas. Alguns trechos de programas são instrumentados para verificação das condições adversas de acessos à memória.

O estudo inicial consiste na identificação dos gargalos de tempo de execução de um programa, utilizando os contadores de *hardware*, permitindo assim ao programador estudar a razão das interações adversas entre o programa e a arquitetura. As medidas obtidas através da monitoração do desempenho de um trecho de programa podem dar uma orientação do caminho a seguir para otimizar a aplicação.

Um código otimizado implica em maximizar os acessos às memórias *cache* e minimizar os travamentos nos *pipelines* da CPU. Assim, a alteração de um código deve sempre buscar a diminuição dos erros de acessos às memórias *cache* de dados, de instrução e *Translation Lookaside Buffer* (TLB). Normalmente, um código ajustado para acesso maximizado ao *cache* de dados é quase sempre também ajustado para máximo acesso ao TLB (Dowd, 1995).

4.1 - Otimização com Técnicas Clássicas

As técnicas de otimização clássicas visam, basicamente, diminuir os erros de acesso à memória *cache*. Técnicas conhecidas de otimização, tais como desenrolar malhas, fusão de malhas, colapso de malhas, blocagem de *cache* e remoção de condicionais internos, algumas destas até utilizadas por compiladores, devem ser implementadas nos trechos em análise sempre que possível. O procedimento a ser adotado para a análise de desempenho de um trecho implica, inicialmente, em identificar os pontos onde os tempos de execução estão comprometendo o desempenho do programa.

As ferramentas disponíveis para levantamento do perfil de tempos de execução, ou *profiling*, como por exemplo o “utilitário gprof”, podem identificar as subrotinas consumidoras de tempo de CPU. Após esta filtragem inicial, deve-se inspecionar o código das subrotinas mais críticas, procurando por situações onde possam existir interações adversas com a arquitetura, principalmente trechos que possam provocar *cache misses*. Segundo Downd [13], deve-se ter como objetivo inicial alcançar uma taxa de acertos à memória *cache* de dados superior a 90%.

A inspeção de um código em análise deve iniciar com uma pesquisa dos trechos que possam contribuir para a perda de desempenho, tais como:

a) Chamadas a subrotinas;
b) Referências indiretas à memória;
c) Testes internos a malhas;
d) Testes de entrada de caracteres;
e) Conversão de tipos;
f) Variável não necessária no trecho ou malha.

As chamadas a subrotinas onde o tempo de chamada é muito superior ao tempo de execução da subrotina em si devem ser evitadas. O uso de *inlining*, ou seja, a inserção do código da subrotina substituindo a chamada, pode reduzir consideravelmente as faltas de acesso à memória *cache*. Os testes internos em malhas, tais como condicionais, devem ser retirados da malha, de forma que o condicional seja analisado externamente. A conversão de tipos de dados, como por exemplo referenciar uma variável inteira com um valor de dupla precisão, implica em perda de desempenho. Alguns desses itens também contribuem na restrição ao paralelismo, e portanto devem ser evitados, tais como: chamadas a subrotinas, referências indiretas à memória, testes internos a malhas e ponteiros ambíguos.

Recomenda-se que as malhas internas devem ser estudadas primariamente, visando sempre preferenciar o paralelismo. O padrão de referência à memória deve ser analisado para preferenciar acesso contíguo a posições vizinhas em memória.

Após adaptação do programa com o uso das técnicas de otimização, deve-se inspecionar a validade das otimizações efetuadas. Os valores tratados no trecho não podem sofrer alteração e as medidas de monitoração devem indicar otimização de desempenho.

Algumas operações com matrizes impedem que se otimize e preferencie acessos adjacentes à memória *cache*. Muitas vezes, as matrizes acessam seus elementos com *strides* - passos diferentes. Pode-se exemplificar isto com o caso clássico de blocagem, a partir do trecho listado abaixo, escrito com comandos da linguagem C.

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

A malha interna definida pelo contador “j” faz a leitura dos N elementos adjacentes da matriz “a”: a[0][0]; a[0][1]; a[0][2];.....;a[0][N].

A matriz “b” será lida em colunas, implicando na leitura de elementos não adjacentes: b[0][0]; b[1][0]; b[2][0];.....;b[N][0]. Ao ler o primeiro elemento, b[0][0], o próximo elemento adjacente será o b[0][1] e será lido após N elementos. Por esta razão, a matriz “a” tem um passo (*stride*) unitário, enquanto a matriz “b” tem um passo igual a N.

Neste exemplo, pode-se observar que os elementos da matriz "b" somente serão acessados após cada linha ser lida, e por essa razão, dificilmente disponíveis na memória *cache* de dados.

A otimização clássica proposta por Dowd[13] indica a técnica de blocagem para estes casos. O acesso à memória é realizado em pequenos blocos, ora preferenciando uma matriz, ora preferenciando a outra matriz. O resultado mostra que as faltas de acesso à

memória *cache* são reduzidas e, portanto, ocorre maior taxa de acertos à memória *cache* de dados.

A blocagem no caso acima pode ser implementada, por exemplo, para um bloco dois por dois com N =1.000. Convém observar que não há dependências entre os elementos das matrizes para iterações distintas da malha. A dimensão (N) das matrizes é múltipla do *stride* igual a 2, assim, não há necessidade de uma iteração extra. Caso N não seja divisível por 2, nem todas as iterações originais estariam cobertas, e dessa forma uma iteração extra teria que ser adicionada.

O total de FLOPs e o total de referências à memória fornece uma boa pista para a otimização. Observa-se no trecho apresentado anteriormente que a soma das matrizes corresponde a uma operação de ponto flutuante por iteração. Um elemento de cada matriz é lido, havendo portanto dois acessos de leitura de memória (dois *loads*). O resultado da soma é armazenado, tendo-se assim um *store*. Portanto, nesse exemplo observa-se três acessos à memória para apenas uma operação de ponto flutuante. Esta relação 3:1(3 *load/store* para 1 flop) mostra que o padrão de referência à memória é importante e deve ser cuidadosamente inspecionado.

Abaixo, é apresentada a aplicação da blocagem ao exemplo dado, com blocos 2x2:

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

As primeiras duas linhas preferenciam a matriz "a", pois executam uma leitura de elementos adjacentes a[i][j] e a[i][j+1], enquanto as primeira e terceira linhas preferenciam a matriz "b", lendo os elementos b[j][i] e b[j][i+1]. A Tabela 4.1 apresenta os resultados medidos, para uma comparação entre um trecho não otimizado e a

correspondente otimização pela técnica de blocagem listada acima. Estas medidas foram obtidas utilizando-se uma CPU Pentium MMX, 200MHz.

TABELA 4.1 - RESULTADOS DA APLICAÇÃO DE BLOCAGEM

Trecho	**Cache hit rate(%)**	**Tempo(mseg.)**
Normal	91,5	80
Blocagem	95,2	50

Estes valores mostram que a utilização de blocagem, neste caso, reduz o tempo de execução do trecho consideravelmente. Neste caso, o aumento dos acertos de *cache* em 3,7 pontos percentuais reduz o acesso à memória externa e dessa forma aumenta a eficiência do código executado. A redução do tempo de execução é de 37,5%. Em situações onde há referências à memória com *strides* altos, como no exemplo acima, a técnica de blocagem é uma das mais eficientes para otimizar o trecho.

A técnica de blocagem na realidade consiste na aplicação da técnica de *loop unrolling* nas duas malhas. Esta técnica, parte da idéia de se agrupar elementos adjacentes da matriz, fazendo com que duas ou mais linhas adjacentes sejam lidas simultaneamente, melhorando os acertos ao *cache* de dados. O número de iterações diminui e assim reduzindo o tempo perdido na malha.

No seguinte exemplo, a matriz unidimensional “A” é expandida em quatro linhas adjacentes:

Código não expandido:

```
for ( i = 0; i < N; i++)
A[i] = A[i] * K
```

Com a expansão da malha em quatro linhas(*loop unrolling*), tem-se:

```
for ( i = 0; i < N; i += 4){
A[i] = A[i} * K
A[i+1] = A[i+1] * K
A[i+2] = A[i+2] * K
A[i+3] = A[i+3] * K
}
```

Neste exemplo, pode-se observar a exposição ao paralelismo, permitindo uma utilização mais eficiente da unidade de operações de ponto flutuante. Nesta situação, são lidas quatro linhas de elementos adjacentes, reduzindo o número de iterações da malha.

O acesso contíguo dos elementos de uma matriz pela leitura de toda uma linha, no caso da linguagem C, ou de uma coluna, no caso de Fortran reduz de forma considerável o tempo de execução do trecho uma vez que há um aumento dos acertos ao *cache* de dados. Nos casos de matrizes unidimensionais, *loop unrolling* torna-se essencial como técnica de otimização.

No exemplo dado para blocagem, aplica-se a técnica de *loop unrolling* em três novas situações. Primeiro, aplica-se a expansão da malha externa(*loop unrolling* externo) e da malha interna(*loop unrolling* interno) individualmente. Em seguida, a malha principal é dividida em duas, uma vez que não há dependências, e aplica-se a técnica nas duas malhas resultantes(*loop unrolling* interno e externo); uma das malhas tem a matriz "a" com referência a linhas e a outra preferencia a matriz “b” com referência a colunas.

A Figura 4.1 apresenta a otimização deste trecho de código aplicando-se as quatro técnicas. Os valores estão normalizados para os resultados obtidos no trecho não otimizado. Este gráfico permite uma comparação entre as técnicas de otimização em função das dimensões das matrizes, onde é possível observar que não há otimização no acesso à memória *cache* do trecho sem a utilização de blocagem.

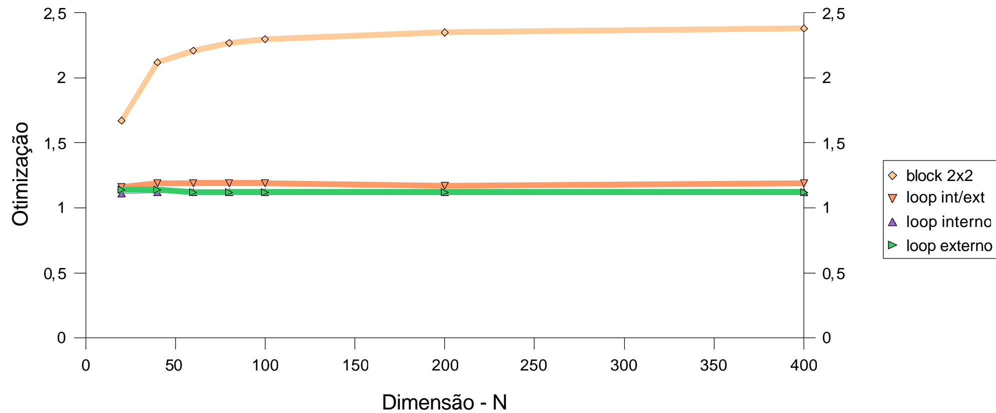


Fig. 4.1 - Comparação entre técnicas de otimização.

Neste caso, a técnica de blocagem é a mais eficiente, otimizando consideravelmente o trecho em análise, mantendo o acesso à memória de forma eficiente praticamente independente das dimensões das matrizes.

A monitoração dos acessos à memória *cache* é de fundamental importância para identificar os pontos problemáticos do código em análise. A Figura 4.2 apresenta o gráfico de faltas ao *cache* de dados, para o trecho em questão na sua forma original, onde é possível observar o aumento das faltas de acesso ao *cache* de dados em função do aumento nas dimensões das matrizes. Observa-se um aumento dos valores de *data cache misses* para dimensões superiores a 100, mantendo-se constante até 500 e tendo posterior explosão a partir deste valor. Os acertos do *cache* L1 podem ser observados pelo primeiro patamar(dimensões próximas a 100). O *cache* externo L2 pode ser observado no segundo patamar onde as matrizes possuem dimensões até 500. Pode-se assim observar a ação dos *caches* L1 e L2 devido à manutenção das falhas de acesso.

.

.

.

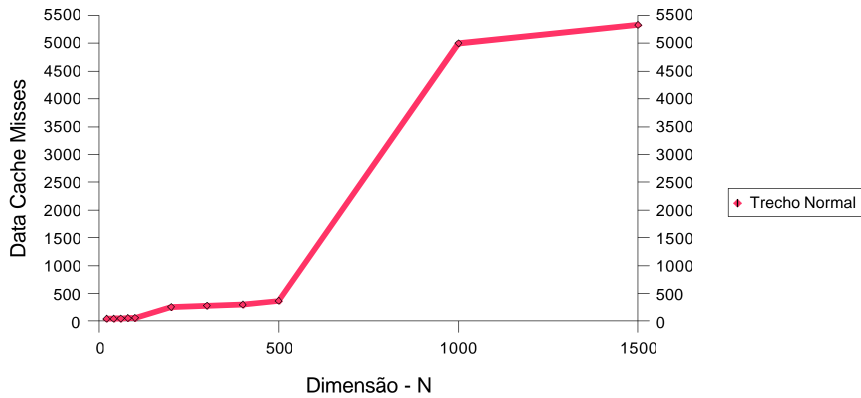


Fig. 4.2 - Medida de *data cache misses* para a malha do exemplo dado.

Uma comparação interessante, apresentada na Figura 4.3, consiste na monitoração das falhas de acesso ao *cache* de dados nos vários casos de otimização. Conforme pode ser observado na Figura 4.1, as técnicas de *loop unrolling* aplicadas ao trecho em análise não são eficientes; a Figura 4.3 mostra a razão. Um aumento das falhas de acesso à memória *cache* devido ao aumento das dimensões das matrizes não permite ganhos no desempenho com estas técnicas. A utilização de blocagem mantém as faltas de acesso reduzidas e praticamente constantes, garantindo assim o patamar constante da otimização observado na Figura 4.1.

.

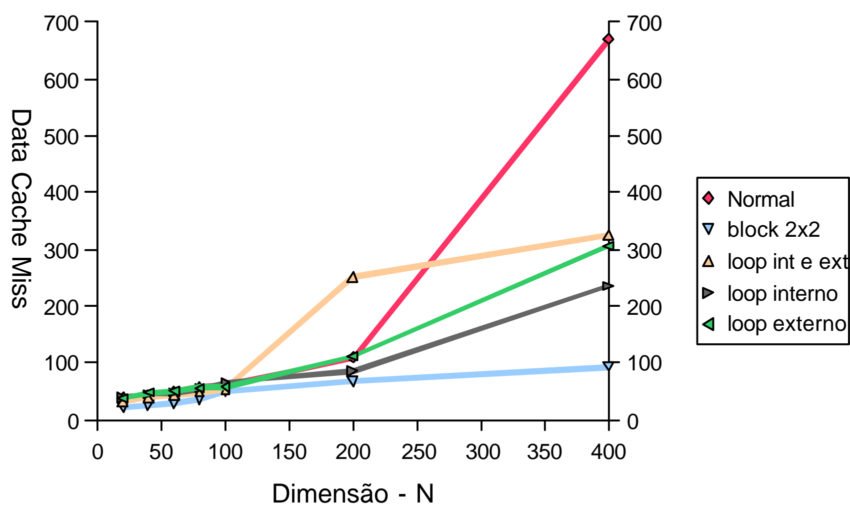


Fig. 4.3 - *Data cache misses* para várias técnicas de otimização.

A Figura 4.4 mostra as curvas de tempo de execução em função das dimensões das matrizes para as diversas técnicas aplicadas. Conforme observou-se na figura anterior, a técnica de blocagem reduz os *data cache misses* e, portanto, espera-se uma redução considerável do tempo de execução do trecho com a sua utilização.

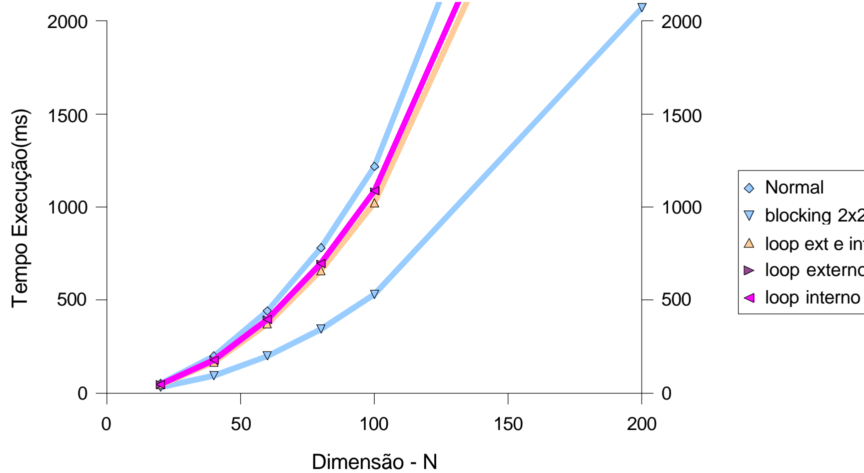


Fig. 4.4 - Tempo de execução em função das dimensões das matrizes.

Ao minimizar as falhas de acesso à memória de dados, as falhas no acesso à memória *cache* de instrução também devem ser analisadas. Sabe-se que, ao expandir as malhas para se implementar a técnica de blocagem, o programa compilado é expandido, aumentando a disputa por mais espaço na memória *cache* de instrução (Dowd,1993).

A Figura 4.5 mostra os *code cache misses* para os quatro casos analisados(*loop unrolling* interno, *loop unrolling* externo, *loop unrolling* externo e interno e blocagem) em função das dimensões das matrizes envolvidas. Observa-se que, para dimensões baixas, a técnica de blocagem não é a mais eficiente para manter os pedidos de instruções dentro do próprio *cache*. Contudo, a técnica de blocagem proporciona os menores valores de *code cache misses* para dimensões maiores, mantendo-se constante.

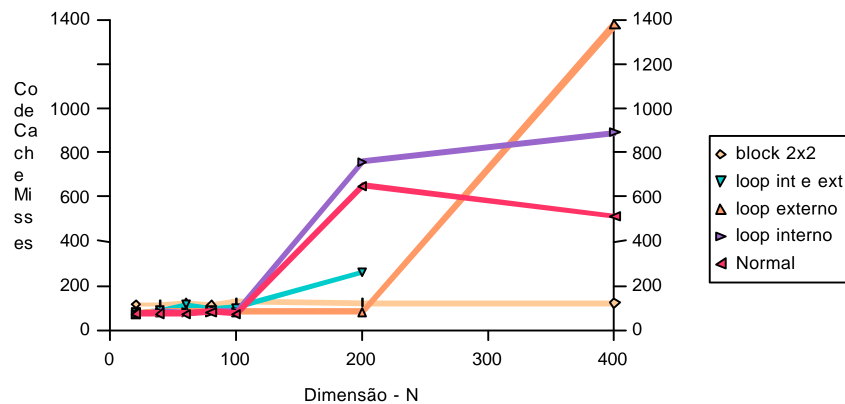


Fig. 4.5 - *Code cache misses* em função das dimensões das matrizes.

4.2 - Otimização de uma Aplicação Real

A instrumentação de um trecho de uma aplicação real e a análise dos dados de desempenho obtidos visam mostrar a simplicidade do método aqui apresentado. Um primeiro teste com uma aplicação científica real foi realizado com o aplicativo Hydrolight [16]. Esta aplicação é utilizada na análise de difusão luminosa em um corpo aquoso [17]. Este programa, desenvolvido em Fortran 77, possui diversas malhas que podem ilustrar algumas das técnicas de otimização.

A obtenção do perfil de tempos de execução das subrotinas que compõem o Hydrolight fornece subsídios para identificar os gargalos de tempo de utilização de CPU. Os resultados obtidos, apresentados na Tabela 4.2, indicam as subrotinas mais problemáticas. Os valores apresentados foram obtidos com o utilitário "gprof," com o aplicativo rodando em estação de trabalho da SUN[21].

TABELA 4.2 - PERFIL DE TEMPO DE EXECUÇÃO DO HYDROLIGHT

Subrotinas	Tempo de CPU(%)	Tempo de Execução(seg.)
sumphas	36,7	57,21
drtdzs	29,88	46,57
rhotau	29,37	45,78
rkck	1,79	2,79
outras	2,26	3,52

O trecho original a seguir, em Fortran 77, faz parte da rotina <sumphas.f> do Hydrolight. O perfil apresentado na Tabela 4.2 identifica esta subrotina como uma das mais custosas em consumo de tempo de CPU.

```
DO I = 1,nmu
        DO J = 1, nmu
        work(I,J) = tauhat(I,J)
                DO K = 1,nmu
                work(I,J) = work(I,J) + rhohat(I,K)*RT(K + (J-1)*nmu)
                END DO
        END DO
END DO
```

As malhas internas existentes no aplicativo Hydrolight são geralmente de baixa dimensão, como no caso acima, onde tem-se a variável "nmu" com valor igual a 10. A taxa de acertos ao *cache* de dados, neste trecho, mostra um valor alto, 96,5 %. O fato das matrizes possuírem dimensões baixas, aumenta a probabilidade da disponibilidade dos elementos no *cache* de dados.

A técnica de *loop unrolling* aplicada na malha "k" mais interna expõe o paralelismo dos pipelines, mantendo-os alimentados e portanto reduzindo as falhas de acesso ao *cache* de instrução.

A substituição de work(I,J) por variáveis temporárias(temp1 e temp2), eliminando o acesso a todos os elementos work(I,J) antes necessários, pode melhorar o desempenho

desse trecho. Esta substituição implica na redução dos *loads/stores* pois é realizado apenas um acesso a cada iteração de “k”.

O trecho passa a ser escrito no seguinte formato:

```
DO I = 1, nmu
        DO J = 1, nmu
        temp1 = tauhat(I,J)
        temp2 = temp1
                DO K = 1, nmu, 2
                temp1 = temp1 + rhohat(I,K)*RT(K+(J-1)*nmu)
                temp2 = temp2 + rhohat(I,K+1)*RT(K+1+(J-1)*nmu)
                END DO
        work(I,J) = temp1 + temp2
        END DO
END DO
```

O trecho acima é monitorado para identificar os acessos às memórias *cache* de instrução e TLB.

A Tabela 4.3 apresenta os valores obtidos para acessos aos *caches* de dados, instrução e TLB. A otimização obtida pela inserção de uma constante e pela utilização da técnica de *loop unrolling* comprova a eficiência do método. O tempo de execução foi reduzido em 31%, com total eficiência dos acessos aos *caches* de instrução e TLB.

TABELA 4.3 - ACESSOS AOS *CACHES* DE INSTRUÇÃO E TLB

Trecho	***Cache hit rate(%)***	***Code TLB misses***	***Code cache misses***	***T.Execução (µseg.)***
Normal	96,5	613	22	600
Otimizado	96,5	0	0	414

A redução a zero das falhas de acesso aos *caches* é devido principalmente às dimensões das matrizes serem baixas. A utilização das variáveis temporárias permitiu uma maior eficiência na alimentação dos *pipelines*.
A aplicação Hydrolight possui muitas subrotinas com tempo de execução inferiores ao tempo de chamada da subrotina, implicando em alto *overhead* para a chamada das funções. Dependendo do modelo analisado, cada subrotina pode ser chamada um milhão de vezes, colocando o tempo de chamada como principal problema na otimização de desempenho do código.

A otimização dos trechos da subrotina "sumphas" permitiu uma redução de até 60 % no tempo de execução do trecho, porém o alto *overhead* de chamada da subrotina não permitiu um ganho na eficiência do código. Apesar dessa redução considerável no tempo de execução obtido na subrotina sumphas, o desempenho total da aplicação praticamente não foi afetado, sendo reduzido em apenas 3%.

A redução desse custo de chamadas das subrotinas pode ser reduzido com o uso de *inlining*. As temporização efetuada nesse caso mostrou uma redução do tempo de execução em 20%, porém amarrou a aplicação para a solução de apenas alguns tipos de problemas.

O aplicativo Hydrolight utiliza o algorítmo de Rkutta de forma intensiva, sendo bastante custoso para a CPU. Uma das alternativas para otimização seria a busca de algorítmos mais eficientes.

Este aplicativo exige um estudo mais detalhado de sua estrutura; um levantamento das variações de configuração deve ser elaborado para identificar as opções mais comuns de uso. Dessa forma, poderiam existir diversas versões otimizadas da aplicação.

Este capítulo apresentou algumas técnicas de otimização de desempenho e mostrou resultados obtidos para o caso específico de blocagem. Um pequeno trecho de uma aplicação real foi analisado, no qual não apenas a monitoração do *cache hit rate*

mostrou-se importante, mas também o efeito dos acessos aos *caches* de instrução e de TLB. O próximo capítulo utiliza a metodologia aqui apresentada e a estende a aplicações paralelas. Uma pequena introdução à linguagem HPF e à monitoração em ambiente paralelo, utilizando as ferramentas desenvolvidas em uma aplicação real, são apresentadas.

CAPÍTULO 5

MONITORAÇÃO EM PROGRAMAS PARALELOS

A metodologia apresentada nos capítulos anteriores pode facilmente ser estendida à programação paralela. Uma aplicação científica pode ter seu desempenho otimizado com a instrumentação seqüencial em uma máquina e, posteriormente, ser executada em várias máquinas em paralelo. Os trechos podem ser novamente monitorados e uma análise de desempenho pode ser realizada em cada máquina, com o objetivo de validar as alterações efetuadas.

Dentre os paradigmas de programação paralela desenvolvidos, dois emergiram com maior aceitação pelos usuários, *message passing* – troca de mensagens e *data parallel* - programação paralela de dados. Ambos possuem compiladores eficientes e atacam um problema através da distribuição do trabalho entre máquinas ou entre processadores.

Basicamente, em máquinas paralelas, o problema a ser resolvido é dividido entre os processadores disponíveis, com o objetivo de produzir a mesma solução que um único processador seqüencial. Em qualquer um dos paradigmas, o objetivo do programador é decompor um problema de forma a prover uma carga balanceada entre as máquinas envolvidas e minimizar a comunicação entre os processadores.

No paradigma *message passing* um programa é carregado em cada processador de um conjunto MIMD. O programa controla a movimentação dos dados entre os processadores através de chamadas a rotinas de comunicação, normalmente disponíveis em uma determinada biblioteca. O programador mantém total controle sobre a distribuição dos dados e a comunicação entre processadores e possui ainda a responsabilidade de organizar os processadores de forma que possam operar coletivamente.

A necessária comunicação entre os processadores passa assim a ser o principal gargalo, pois introduz tempos de comunicação ao tempo de execução. Assim, nesse paradigma, passa a existir uma troca entre o custo da comunicação e a eficiência obtida na execução em paralelo.

O paradigma de *message passing* é ideal para máquinas com memória distribuída, porém, possui algumas desvantagens. Como cada processador realiza uma tarefa isolada e executa códigos diferentes torna o entendimento do programa e sua manutenção muito difícil. Nesse paradigma, a troca de mensagens deve se manter mínima e pode ser uma tarefa exaustiva para o programador. É extremamente importante que as mensagens trocadas tenham destino e retorno, evitando assim que um dos processadores trave enquanto aguarda uma determinada mensagem que não foi enviada. Assim, o paradigma *message passing* fornece controle total ao programador e fica sob sua responsabilidade o eficiente intercâmbio entre os processadores.

O paradigma *parallel data* consiste no suporte a operações com conjuntos executadas em paralelo. Tipicamente, um único programa controla a distribuição e operações com os dados em todos os processadores. Neste paradigma, o compilador é responsável pela distribuição dos dados bem como com a comunicação entre os processadores onde o programador atua apenas como orientador para as operações. Neste paradigma todos os processadores envolvidos executam o mesmo programa objeto, recebendo uma partição dos dados para análise.

O estilo de programação paralela de dados pode ser definido através de suas características como:

a) Um único programa tem o controle de todas as operações;
b) Os detalhes de distribuição dos dados, o acesso à memória e a comunicação são tratados pelo compilador;

c) A execução de cada instrução não é síncrona entre os processadores. Embora todos os processadores executem o mesmo código, cada processador pode estar executando instruções diferentes em um determinado instante;
d) As operações com elementos de conjuntos são executadas simultaneamente em todos os processadores;
e) As diretivas consistem em comentários que apenas auxiliam o compilador na distribuição dos dados.

Assim, a responsabilidade de definição dos detalhes de programação ficam por conta do compilador. A programação paralela de dados trás algumas vantagens sobre a programação por passagem de mensagens pois é bem mais simples de se escrever e manter um código. Porém, há uma redução da flexibilidade permitindo pouco controle para o programador.

O trabalho aqui apresentado foi limitado a programas escritos no modelo de programação paralela de dados, onde o *High Performance Fortran* - HPF tem sua melhor expressão; contudo, a metodologia pode ser aplicada a outros paradigmas de programação paralela.

Neste capítulo, descreve-se como instrumentar um programa que será executado em um ambiente paralelo. Em especial, descreve-se o paradigma de programação paralela de dados, tecendo-se comentários sobre seus conceitos e características. É dada uma breve introdução ao Fortran 90, descrevendo suas características mais marcantes. A distribuição dos dados dentro desse paradigma implica no uso de diretivas da extensão HPF, também descritas neste capítulo. Um exemplo é instrumentado para gerar contagens de vários eventos, com os resultados sendo apresentados e comentados.

O objetivo deste capítulo é mostrar a utilização dos contadores de hardware em ambientes paralelos, permitindo que um determinado código realmente torne o paralelismo efetivo. Assim, deve-se procurar:

a) Minimizar a relação (tempo de comunicação) / (tempo computacional);

b)Balancear a carga computacional entre os processadores envolvidos.

O tempo de comunicação corresponde ao tempo gasto para a inicialização da mensagem que será transmitida, somado ao tempo efetivo para que a mensagem chegue ao destino especificado. Este tempo é crítico, pois pode aumentar consideravelmente o tempo de execução total. Quanto ao balanceamento da carga computacional, é importante garantir que as máquinas envolvidas não tenham falta de atividade.

A busca de maior eficiência de um código-fonte, quando portado a um ambiente paralelo, consiste em seguir, inicialmente, as técnicas de otimização para programas seqüenciais já apresentadas.

5.1 - Programação Paralela de Dados

A programação paralela de dados tem como modelo a execução paralela de operações com vetores e matrizes. Dentre os aspectos mais importantes da programação paralela de dados, pode-se citar:

a) O programa de controle é único;

b) O ambiente de dados é global. O programador, em geral, não se preocupa com detalhes do gerenciamento de memória ou a forma como os dados são distribuídos;

c) Não existe sincronismo. A execução das instruções não é síncrona entre os processadores;

d) As operações com vetores são realizadas em paralelo;

e) As diretivas do compilador, quando presentes, são comentários, fornecendo sugestões ao compilador para a distribuição dos dados e para o estabelecimento de uma estratégia de execução pelos vários processadores.

A programação paralela de dados se encarrega de definir operações coletivas em conjuntos e vetores, ou a seus subconjuntos, de forma a distribuí-los entre um determinado número de processadores. Se um determinado algoritmo pode ser expresso nestes termos, provavelmente a implementação paralela será eficiente.

Este paradigma categoriza um conjunto de operações que forma a base para a implementação de algorítmos paralelos, identificadas a seguir.

a) Diretivas para a distribuição de dados:

Em determinados casos é importante que o programador tenha controle sobre a distribuição dos dados entre os processadores e assim minimize a comunicação entre eles, mantendo todos os processadores ocupados e realizando as operações em paralelo de forma a obter o maior desempenho possível. Essa distribuição é possível através do uso de diretivas disponíveis nos compiladores. A melhor distribuição depende do algoritmo a ser implementado em cada caso.

b) Operações com elementos de matrizes ou vetores:

As operações que usam elementos de matrizes ou vetores e aplicam uma determinada operação a cada elemento, tais como soma, multiplicação ou divisão, são realizadas em paralelo.

c) Seções de matrizes ou vetores:

Partes de uma matriz ou de um vetor são identificados por seus índices. As funções matemáticas podem ser aplicadas a blocos ou subconjuntos de blocos de uma matriz.

d) Operações com condicionais:

Determinadas operações podem ser aplicadas a um subconjunto de uma matriz ou vetor, selecionados por uma máscara condicional ou uma expressão lógica ou aritmética. Assim, o programador pode atuar na determinação de seções críticas ao algoritmo.

e) Operações de redução:

A operação de redução produz um resultado derivado de uma combinação de elementos da matriz. Por exemplo, pode-se estimar os valores máximos e mínimos de uma matriz ou determinar o total de elementos nulos.

f) Operações de deslocamento ou transposição de linhas e colunas:

Alguns algoritmos podem implementar deslocamentos de linhas ou colunas de uma matriz. Determinados deslocamentos podem ser executados em paralelo.

g) Operações de varredura:

A varredura dos elementos ou de um subconjunto de uma matriz são combinados seguindo uma determinada lógica ou expressão, fornecendo um resultado acumulativo.

h) Comunicações generalizadas:

Em determinadas aplicações, é necessário combinar elementos em posições diferentes, ou ainda exige-se movimentos de elementos para outras posições.

Este conjunto de blocos de funcionalidades contém operações que facilitam o paralelismo e, assim, permitem um maior desempenho da aplicação. A maior vantagem

na utilização do paradigma de programação paralela de dados está na simplicidade de seu uso.

Os itens a seguir mostram algumas das características importantes do Fortran 90 e do HPF, fundamentais no paradigma de programação paralela de dados. As especificações das funções intrínsicas do Fortran 90 ou das diretivas HPF não fazem parte dos objetivos desse trabalho.

5.2 - Fortran 90

O Fortran 90 é uma evolução do Fortran 77, desenvolvido com o objetivo de se obter o máximo de eficiência de um código onde novos comandos permitam explorar o paralelismo e estender um conjunto de operações com vetores. Novas características foram acrescentadas e algumas outras do Fortran 77 foram consideradas obsoletas e até substituídas. A base do Fortran 90 é procurar explicitar o paralelismo.

O Fortran 90 traz um novo formato de edição com características interessantes, tais como:

a) Formato livre, sem colunas reservadas;

b) As linhas podem conter até 132 colunas;

c) Possibilidade de declarações múltiplas em uma linha;

d) Possibilidade de comentários na linha da declaração;

e) Não é sensível a letras maiúsculas e minúsculas;

f) Possibilidade de nomes longos com até 31 caracteres;

g) Possibilidade do uso do caracter *underscore* - sublinha.

Na operações com matrizes, o paralelismo pode ser expresso através do uso de novos operadores que permitem o seu seccionamento. A notação do Fortran 90 permite que operações aritméticas possam ser aplicadas diretamente a matrizes e vetores. Estas operações são conceitualmente realizadas em paralelo.
A seguir, descreve-se algumas características importantes do Fortran 90:

a) Inclusão de várias funções intrínsecas, incluindo operações de redução como SUM (soma de todos os elementos de uma matriz/vetor) ou MAX-VAL (executa uma varredura procurando o maior valor da matriz/vetor);

b) O armazenamento de dados no Fortran 77 era um problema, pois não permitia o uso de ponteiros. A introdução do armazenamento dinâmico permitiu que ponteiros e matrizes/vetores com dimensões alocáveis na execução fossem implementados;

c) A disponibilidade da declaração KIND permite maior portabilidade do código, pois torna possível ao usuário definir a precisão das variáveis de forma parametrizada;

d) A possibilidade de se definir tipos de dados facilita a criação de novas variáveis e fornece uma poderosa ferramenta orientada a objetos para o programador;

e) A recursão é permitida, facilitando ao programador solucionar algorítmos complexos;

f) O uso de módulos permite a definição global de tipos, objetos, operadores e subrotinas provendo funcionalidade, pois os detalhes internos ficam mascarados. O uso de prototipação dos procedimentos auxilia a criação e

manutenção de projetos, facilitando principalmente o desenvolvimento de aplicativos, pois os programas e procedimentos podem ser compilados separadamente. Os módulos fornecem mecanismos de proteção e encapsulamento para as subrotinas criadas;

g) Novas estruturas para as malhas DO...ENDDO, reduzindo a necessidade de se usar rótulos nas malhas. O comando EXIT passa a permitir uma saída mais limpa de uma malha. Outro comando importante, o CYCLE, permite que se abandone a iteração corrente de uma malha e se reinicie na próxima iteração;

h) A inclusão do bloco de controle SELECT CASE facilita na elaboração de comparações de forma mais eficiente que o comando IF.

Estas características tornam o Fortran 90 recomendável para o desenvolvimento de aplicações científicas e de engenharia.

5.3 - *High Performance Fortran* - HPF

A utilização de diversas máquinas, distribuindo a busca de solução de um problema, exige um novo paradigma para os compiladores. Dentre as opções existentes, o presente trabalho utiliza o HPF, um conjunto de extensões para o Fortran 90 voltado para o paradigma de programação paralela de dados.

O HPF é resultado da necessidade de se padronizar a linguagen Fortran para o paradigma da programação paralela de dados. Possui um conjunto de construtores e extensões do Fortran 90 que permitem ao programador expressar o paralelismo de uma forma relativamente simples.

O modelo *Single Program Multiple Data* - SPMD é o utilizado para o HPF; cada processador possui uma cópia do mesmo programa, porém cada um deles trabalha com

uma parte dos dados. Existem diretivas HPF que permitem distribuir os dados. Assim, as declarações em Fortran 90, acrescidas das diretivas HPF, definem o paralelismo.

A utilização das diretivas HPF exige, inicialmente, que se defina uma grade conceitual de processadores, que se distribua os dados aos processadores envolvidos e, por fim, que se executem os cálculos necessários. As diretivas apenas dão uma sugestão ao compilador da distribuição desejada. O compilador tem a liberdade de ignorar ou não, ou ainda modificar, qualquer diretiva. As diretivas são comentários para o compilador Fortran 90; assim, um programa em HPF compila sem problemas seqüencialmente.

O uso das diretivas HPF segue o seguinte formato:

```
!HPF$ <diretiva-hpf>
```

No presente caso, a primeira diretiva a ser utilizada deve defrinir uma grade conceitual de processadores; esta grade permite indicar a distribuição dos dados que se deseja efetuar. Por exemplo, a distribuição de um vetor "A" unidimensional de vinte elementos distribuídos entre quatro processadores deve ser definida da seguinte forma:

a) Define-se a grade de distribuição P com quatro processadores:

```
!HPF$ processors, dimension( 4 )  :: P
```

b) Define-se o vetor A com vinte elementos:

```
real, dimension( 20 ) :: A
```

c) Distribui-se o vetor A entre os processadores da grade:

```
!HPF$ distribute (block )  onto  P  :: A
```

Os dados podem ser distribuídos em blocos (BLOCK) ou de forma cíclica (CYCLIC). A forma de distribuição BLOCK indica que os elementos devem ser divididos em blocos entre os 4 processadores. Assim, cada processador recebe blocos de cinco elementos consecutivos do vetor original. A distribuição também pode ser cíclica, ou seja, os dados são distribuídos seqüencialmente entre os processadores. Assim, os vinte elementos divididos entre quatro processadores distribuem cinco elementos para cada um. Portanto, os cinco primeiros elementos(A[1] a A[5]) são propriedade do processador p1, os cinco próximos(A[6] a A[10]) ficam com o processador p2 e assim sucessivamente.

Abaixo, apresenta-se a distribuição em blocos(BLOCK) dos vinte elementos(A[1] a A[20]) entre quatro processadores(p1, p2, p3 e p4):

p1	p2	p3	p4
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

Esta distribuição torna-se útil útil principalmente quando são necessários cálculos entre elementos adjacentes.

Já a distribuição cíclica(CYCLIC), deve ser utilizada onde exista a necessidade de uma distribuição eqüitativa de carga computacional. A distribuição CYCLIC abaixo mostra a distribuição dos mesmos vinte elementos do vetor entre quatro processadores. Os elementos são distribuídos seqüencialmente entre os processadores, o primeiro elemento (A[1]) para o primeiro processador(p1), o segundo elemento(A[2]) para o segundo processador(p2) e assim sucessivamente.

p1	p2	p3	p4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

O programador não precisa se preocupar com as variáveis escalares definidas, pois tais variáveis serão automaticamente distribuídas entre todos os processadores. O compilador é responsável em manter a coerência destes valores.

Para se efetuar a distribuição mais eficiente dos dados, deve-se ter em mente a regra de que o processador que possui o lado esquerdo de uma expressão aritmética (ou o "proprietário" do valor no lado esquerdo da expressão) é o responsável pela realização dos cálculos envolvidos. Esta regra, porém, é seguida por determinação do compilador; embora a maioria dos compiladores atuais a utilizem, isso pode não ser estritamente verdadeiro no futuro.

Ao partir para o uso do paradigma de programação paralela de dados, o programador deve considerar que o HPF envolve uma troca entre paralelismo e comunicação. Assim, ao inserir diretivas HPF, deve-se levar em conta que o aumento do número de processadores pode envolver um aumento da comunicação. Deve-se procurar seguir as seguintes recomendações:

a) Tentar balancear a carga, assumindo a regra de proprietário da expressão;

b) Procurar manter a localidade dos dados;

c) Utilizar a notação de vetores do Fortran 90.

O Fortran 90 possui uma notação para conjuntos que explicita o paralelismo através de: designação de conjuntos, mascaramento de elementos, seccionamento de subconjuntos e transposição de linhas e colunas. O HPF acrescenta a estas características a declaração FORALL , a diretiva INDEPENDENT e a atribuição PURE.

A declaração FORALL é um construtor para a distribuição paralela de dados, pois permite a designação de elementos múltiplos de um conjunto de dados, porém sem forçar a designação individual dos elementos. O FORALL cria um construtor para distribuir em paralelo os dados do conjunto, garantindo o mesmo resultado caso o código seja executado em série ou em paralelo.

A diretiva INDEPENDENT deve ser utilizada em conjunto com a declaração FORALL pois fornece ao compilador informações adicionais sobre a execução do FORALL ou ainda sobre malhas DO-ENDDO. Para uma malha DO-ENDDO a diretiva INDEPENDENT implica que uma iteração não afeta a próxima, enquanto que no FORALL assegura que as operações dos índices são isoladas.

O exemplo a seguir mostra a importância da diretiva INDEPENDENT, conforme descreve Ewing[11]. A expressão do lado direito (*Right Hand Side* - RHS) é executada em paralelo e os resultados obtidos são atribuídos ao lado esquerdo (*Left Hand Side* - LHS). A execução do FORALL sem a diretiva INDEPENDENT provoca um sincronismo das expressões RHS1 e RHS2 antes da atribuição a LHS1 e LHS2. A Figura 5.1 mostra a avaliação das expressões com e sem o uso da diretiva INDEPENDENT.

```
! Sem “INDEPENDENT”
FORALL( i = 1:3 )
        LHS1 ( i ) = RHS1 ( i )
        LHS2 ( i ) = RHS2 ( i )
END FORALL
```

```
!HPF$ INDEPENDENT
FORALL( i = 1:3 )
        LHS1 ( i ) = RHS1 ( i )
        LHS2 ( i ) = RHS2 ( i )
END FORALL
```

.

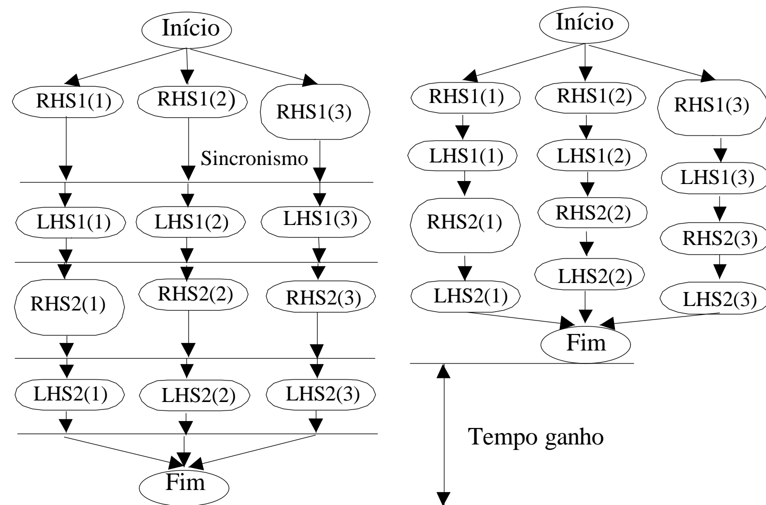


Fig. 5.1 - Evitando sincronismo em FORALL com o uso de INDEPENDENT.
FONTE: Adaptada de Ewing, Hare, Richardson e Simpson (1999, p.53).

O tempo ganho com o uso da diretiva INDEPENDENT resulta do fato de que não há necessidade de se esperar pela finalização das expressões.

O atributo PURE permite identificar funções que não interferem na declaração FORALL. O uso do atributo PURE garante que um procedimento não altera seus dados de entrada. Os índices do FORALL podem ser repassados ao procedimento sem qualquer alteração, permitindo o paralelismo implícito do FORALL.

5.4 - Instrumentação de Programas Paralelos.

A instrumentação de um trecho de programa para utilização no paradigma de programação paralela de dados exige que o módulo MSR esteja instalado nas máquinas que farão parte da grade de processadores a ser definida, conforme anteriormente descrito no Capítulo 3.

A distribuição dos dados em paralelo para as máquinas envolvidas é orientada pela definição da grade de processadores, através da diretiva "processors".

Por exemplo, uma distribuição de uma matriz bidimensional entre "m" processadores poderia ter esta sintaxe:

```
!HPF$ processors, dimension ( m/2, 2 ) :: P
```

A grade definida por P faz uma distribuição de “m” processadores em duas colunas, com “m/2” processadores em cada. Deve-se observar que as dimensões da grade devem ser iguais às dimensões do conjunto que será distribuído. Os dados podem ser distribuídos entre os processadores em blocos ou de forma cíclica, conforme discutido na seção 5.3. Como exemplo, se houvesse duas matrizes "a" e "b" marcadas para distribuição em blocos pela grade "P", seria utilizada a seguinte sintaxe:

```
!HPF$ distribute(block,block) onto P :: a, b
```

A instrumentação de programas paralelos exige que os eventos monitorados sejam atribuídos a variáveis com dimensão igual ao número de processadores, e distribuídas em bloco através de grade unidimensional, da seguinte forma:

```
!HPF$ processors, dimension( n ) :: T
```

Um exemplo de instrumentação em programas paralelos está listado na Figura 5.2. O programa executa o cálculo do desvio padrão de uma multiplicação de duas matrizes bi-dimensionais. A instrumentação monitora a operação de produto das matrizes, inserindo no programa original as definições de tipos das variáveis utilizadas pelas funções de monitoração, "iniciaregistros" e "leregistros". As variáveis cesr_0 e cesr_1 correspondem aos códigos de controle e seleção de eventos. As variáveis count_0 e count_1 são ponteiros para os valores de contagens obtidos. Os valores lidos dos contadores (ct_0 e ct_1) e o tempo de execução medido (t_msr) são armazenados em

variáveis inteiras. Cada máquina lê seus contadores de hardware localmente e os atribui às variáveis distribuídas ct_0 e ct_1. O retorno da função "leregistro" corresponde ao tempo medido entre a chamada ao procedimento "iniciaregistros" e a função "leregistros". O tempo medido é atribuído à variável t_msr. Utilizando a notação do Fortran 90, tem-se:

```
integer  ::  count_0,  count_1, cesr_0, cesr_1

integer, dimension( n ) :: ct_0, ct_1, t_msr

!HPF$ distribute( block ) onto T :: t_msr, ct_0, ct_1
```

As rotinas de acesso aos contadores de *hardware* são inseridas no trecho que se pretende monitorar. A primeira subrotina "iniciaregistros" passa o código dos eventos através dos parâmetros cesr_0 e cesr_1. Assim, o trecho monitorado deve estar contido entre as chamadas às duas rotinas, de acordo com o seguinte esquema:

```
call iniciaregistros( cesr_0, cesr_1 )
...
<trecho monitorado>
...
t_msr = leregistros(count_0, count_1)
ct_0 = count_0
ct_1 = count_1
....
```

O programa exemplo "desvio padrão" está escrito seguindo a notação do Fortran 90 e com as extensões do HPF. Neste exemplo, a operação X = A * B foi monitorada em diversos eventos, com os dados distribuídos em duas máquinas baseadas em processadores P6. Os resultados obtidos estão apresentados na Tabela 5.1, onde se pode observar, principalmente pelo total de operações em ponto flutuante, que a distribuição utilizada foi bem eficiente neste trecho monitorado.

TABELA 5.1 - EVENTOS MONITORADOS NO EXEMPLO DE DESVIO PADRÃO

Eventos Monitorados	***CPU 1****	***CPU 2****
FLOPs	500.000	500.000
Instruções executadas	7.771.662	7.715.026
Referências a memória	7.435.338	7.387.481
Desvios	547.181	536.015
Instruction fetches	15.372.364	15.213.349
Instruction fetch misses	496	119
Tempo(μs)	52.415	51.446

(*) - IBM Celeron 300 MHz.

Os resultados ligeiramente maiores apresentados na Tabela 5.1, mostram que CPU 1 foi a responsável pela comunicação entre as máquinas.

```
! Programa teste para programação paralela de dados
! Nome: std.hpf
! Funcao: Calcula desvio padrao de multiplicacao de matrizes
! Atividade: Monitorar multiplicacao de matrizes

PROGRAM Main
USE meumod
REAL, DIMENSION(N) :: X,A,B
REAL :: stx,sta,stb
INTEGER :: n_elts, istat

!----/ definicoes de variaveis de instrumentacao /----
INTEGER :: cesr_0, cesr_1, cont_0, cont_1
!----/ fim definicoes variaveis de instrumentacao /----

! Definicao da grade de distribuicao e tipo de distribuicao
!HPF$ PROCESSORS, DIMENSION(number_of_processors()) :: P
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: X,A,B

!----/ definicoes de instrumentacao /----
!----/ definicoes dos tempos de execucao /----
integer, dimension(number_of_processors())  :: t_msr

!----/ definicoes dos contadores /----
integer, dimension(number_of_processors())  :: ct_0,ct_1

!----/ Distribuicao das variaveis /----
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: ct_0,ct_1, t_msr

!----/ eventos a monitorar /----
 cesr_0 = z'C1' ! P6 - flops
cesr_1 = z'D0' ! P6 - instrucoes executadas
!----/ Fim das definicoes de instrumentacao /----

!HPF$ INDEPENDENT
DO i=1,N
call dummy(a(i))
call dummy(b(i))
ENDDO
!----/ trecho monitorado /----
call iniciaregistros(cesr_0,cesr_1)
X = A * B
t_msr = leregistros(cont_0,cont_1)
ct_0=cont_0
ct_1=cont_1
!----/ trecho monitorado- fim /----
sta = Std_Dev(A)
stb = Std_Dev(B)
stx = Std_Dev(X)
!resultados
print *,"contador 0 = ", ct_0
print *,"contador 1 = ", ct_1
print *, "stA,stB,stX = ",sta,stb,stx
print *, 'tempo execucao = ',t_msr
```

Fig. 5.2 - Exemplo de programa paralelo instrumentado.

Este capítulo apresentou algumas características das linguagens Fortran 90 e de sua extensão HPF, como paradigmas de programação paralela de dados.

Conforme mostrado em seguida, a instrumentação de códigos paralelos é bem simples, bastando inserir a chamada de inicialização antes do trecho que será monitorado e, ao final do trecho, inserir chamada à função de finalização, a qual retorna o tempo de execução em cada máquina, junto com as contagens locais. Como é possível observar, a instrusão no código original é mínima. Um exemplo simples, onde se calcula o desvio padrão de um produto de matrizes, foi instrumentado para monitoração de diversos eventos.

O capítulo seguinte descreve a instrumentação de dois exemplos reais. No primeiro, uma aplicação escrita originalmente em Fortran 77 é instrumentada para análise, com o objetivo de se identificar os trechos onde existam interações problemáticas com a memória. Posteriormente, o código foi totalmente reescrito em Fortran 90 e novamente monitorado para os mesmos eventos. No segundo exemplo, uma aplicação já com diretivas HPF é instrumentada para monitoração das referências a memória, em duas distribuições diferentes dos dados.

CAPÍTULO 6

EXEMPLOS DE OTIMIZAÇÃO EM APLICAÇÕES PARALELAS

O presente capítulo descreve a captura de dados de desempenho para dois exemplos. O primeiro exemplo, um aplicativo real originalmente escrito em Fortran 77, é totalmente reestruturado com declarações e notação do Fortran 90. Uma análise dos tempos de

de consumo do tempo de CPU. Estes trechos são otimizados e, através da captura de diversos eventos importantes, são comparadas as duas versões. Um segundo exemplo, já escrito em Fortran 90 e com diretivas HPF, é analisado apenas na sua distribuição de dados. A distribuição de uma das matrizes do programa é alterada e os dois casos monitorados. Posteriormente, são executados numa máquina paralela real, variando-se o número de processadores.

6.1 - Exemplo de Otimização por Reestruturação de Código

O programa corr_masco.f é um programa em Fortran 77 que tem por objetivo produzir uma imagem do céu em raios-gama, através da técnica conhecida por máscara codificada. A versão analisada é específica para o projeto MASCO, da Divisão de Astrofísica(DAS) do INPE, de um programa mais geral o qual pode ser utilizado por qualquer sistema imageador que empregue máscaras codificadas com padrões *Uniformly Redundant Arrays* (URA) ou *Modified URAs* (MURA).

O programa, basicamente, calcula a correlação cruzada bidimensional discreta entre a distribuição espacial de contagens (cada contagem corresponde à interação de um fóton de raios-gama) na superfície do detetor do telescópio MASCO com o padrão de aberturas da máscara codificada (formada por elementos de chumbo). O programa lê uma matriz cujos elementos representam o número de contagens registradas pelo detetor em áreas quadradas que cobrem a sua superfície. Essas áreas podem ser de mesmo tamanho do elemento básico da máscara ou qualquer subdivisão inteira dele. O objetivo

é gerar um padrão "suavizado" da máscara, que é então discretizado em uma matriz (normalizada pelo seu coeficiente de auto-correlação), que por sua vez é utilizada na correlação com a distribuição de contagens, aumentando os coeficientes de correlação devido à similaridade das duas distribuições. A matriz-imagem é então armazenada no disco em vários formatos.

Inicialmente, o programa foi executado na máquina de testes, um microcomputador IBM Celeron 300MHz, com processador P6, para verificação dos tempos totais de execução. Os dados de entrada são passados por matrizes com contagens: primeiro, uma matriz de testes denominada MURA_17x17(dimensões 17x17) e depois, numa nova execução, uma matriz com dados reais denominada MURA_43x43 (dimensões 43x43).

A análise de uma aplicação deve iniciar com a elaboração de um perfil de tempos de execução, com o objetivo de identificar os trechos que deverão ser monitorados. O programa corr_masco.f faz chamadas a apenas duas subrotinas (mura_pattern e mura_psf). Inicia-se o processo de análise com a cronometragem dessas chamadas a subrotinas. Os testes de temporização dos trechos de programa e chamadas a subrotinas foram executados em máquina com processador P6 carregando como arquivo de entrada a matriz mura_17x17. Os valores obtidos estão apresentados na Tabela 6.1. Convém ressaltar que as chamadas a estas subrotinas são únicas, ou seja, são executadas uma única vez no decorrer da execução. Assim, os tempos apresentados não implicam em necessidade de otimização, pois são insignificantes quando comparados com o tempo total de execução.

TABELA 6.1 - TEMPOS DE EXECUÇÃO DE SUBROTINAS

Subrotina	***Tempo de Execução(ms)***
mura_pattern	0,357
mura_psf	349

O programa pode ser dividido em dois blocos. O primeiro bloco contém a inicialização, executa a leitura da matriz de entrada e faz chamadas às subrotinas "mura_pattern" e mura_psf". Em seguida, existem algumas malhas que não influem no tempo total de execução, pois são executadas apenas uma vez. O segundo bloco é constituído por cinco malhas que correspondem aos trechos mais custosos em tempo de CPU, seguido de um bloco de finalização. A Figura 6.1 mostra o diagrama com os blocos que representam o programa, onde cada bloco será analisado individualmente.

A Tabela 6.2 mostra os tempos de execução obtidos para os trechos esquematizados na Figura 6.1. O trecho-2 é uma malha que executa "r*res" iterações. O valor "r" corresponde à dimensão da matriz e "res" à resolução.

No caso da matriz MURA_17x17, tem-se :

r = 17 e res = 1, portanto 17 iterações;

No caso da matriz MURA_43x43 tem-se:

r = 43 e res = 5, portanto 215 iterações.

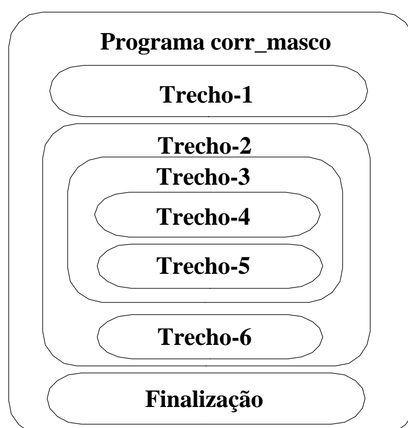


Fig. 6.1 - Diagrama de blocos do programa corr_masco.

Os sete trechos foram instrumentados com chamadas ao procedimento de inicialização "iniciaregistros", que deve ser inserido no início do trecho a ser monitorado. No final do trecho deve-se inserir a função "leregistros". A monitoração dos tempos de execução não exige a passagem de parâmetros. O retorno da função "leregistros" corresponde ao tempo de execução medido em microsegundos. A Tabela 6.2 apresenta os tempos de execução obtidos para cada um dos trechos de programa identificados anteriormente, utilizando a matriz MURA_43x43 como matriz de entrada.

TABELA 6.2 - TEMPOS DE EXECUÇÃO DAS MALHAS INTERNAS

Trecho	*Tempo de Execução(seg)*
1	1,71
2	515,90
3	2,39
4	0,0086
5	0,0024
6	0,0025
Finalização	0,33
Total	526,00

O trecho-2, portanto, é o maior consumidor de tempo de CPU. Este trecho, por sua vez, tem seu tempo de execução definido pelos trechos 3 e 6. O trecho-3 executa as malhas definidas nos trechos 4 e 5. Os trechos 5 e 6 são implementações de transposição de linhas e colunas de matrizes, respectivamente, com tempos de execução iguais.

Os trechos 4 e 5 são executados 215 vezes devido à malha do trecho-3, e mais 215 vezes devido à malha do trecho-2. Assim, os trechos 4 e 5 executarão seu código consumindo 508 segundos (tempo = (0,0086 + 0,0024)*215*215). Portanto, esses trechos são prioritários na análise para otimização.

O primeiro trecho a ser monitorado é o trecho-4, listado abaixo com o código fonte original (Fortran 77). Este trecho executa o produto das matrizes "arr" e "count" e o

armazena na matriz "prod". Cada valor de corr(rperms, cperms) é somado aos valores armazenados na matriz "prod".

```
DO k = 1, res * r
        DO l = 1, res * r
        prod( k, l ) = arr( k, l ) * count( k, l )
        END DO
END DO
DO k = 1, res * r
        DO l = res * r
        corr( rperms, cperms ) = corr( rperms, cperms ) + prod( k, l )
        END DO
END DO
```

Utilizando-se da notação e declarações do Fortran 90, este trecho passa a ser escrito em apenas uma linha, com a seguinte sintaxe:

```
corr( cperms, rperms ) = corr( rperms, cperms ) + SUM( arr * count )
```

As duas versões do trecho-4, em Fortran 77 e Fortran 90, foram instrumentadas para obtenção dos tempos de execução. Os testes foram realizados com as duas matrizes sucessivamente, a matriz MURA_17x17 e a matriz MURA_43x43. Os resultados estão apresentados na Tabela 6.3.

TABELA 6.3 - COMPARATIVO ENTRE TEMPOS DE EXECUÇÃO DO TRECHO-4

Matriz de Entrada	*Tempo Execução Fortran 77 (μs)*	*Tempo Execução Fortran 90 (μs)*	*Redução do tempo de execução(%)*
17 x 17	50	30	40,0
43 x 43	8.602	4.910	42,9

O programa foi compilado utilizando os compiladores pgf77 e pgf90 da Portland Group, disponíveis na máquina Polaris do INPE-LAC (P6 : *quad-processor*) e executado na máquina de testes do INPE-SLB (P6: *mono-processor*). A redução de 40% no tempo de execução, no caso da matriz MURA_17x17, e a redução de 42,9%, no caso da matriz

MURA_43x43, mostram a eficiência da notação Fortran 90. A carga computacional resultante do aumento das dimensões das matrizes é facilmente observada pelo aumento considerável do tempo de execução. Enquanto a matriz 17 x 17 possui 289 elementos , a matriz 43 x 43 possui 1.849 elementos, ou seja, aproximadamente 6,4 vezes mais elementos.

O trecho-4 foi instrumentado para obtenção do total de instruções executadas, total de FLOP's, total de referências à memória e total de falhas de *cache* de dados, com execução para as duas versões do programa (Fortran 77 e Fortran 90) e com as duas matrizes. Os resultados obtidos estão apresentados na Tabela 6.4. Os programas foram executados na máquina IBM Celeron 300 MHz.

TABELA 6.4 - FLOPS, INSTRUÇÕES EXECUTADAS E REFERÊNCIAS À MEMÓRIA PARA O TRECHO-4

Compilador	Matriz	FLOPs	Instruções Executadas	Referências à Memória
F77	17x17	1.864	11.760	8.366
	43x43	278.790	1.264.600	960.673
F90	17x17	1.460	10.370	7.263
	43x43	231.308	1.119.038	775.598

O tempo de execução do trecho-4 para as duas matrizes de entrada apresenta uma redução considerável ao se comparar a versão em Fortran 90 com a versão original em Fortran 77. No caso da matriz MURA_17x17 a redução é de 40% e no caso da matriz MURA_43x43 a redução é de 42,9%. Estes valores se devem à redução do total de operações em ponto flutuante de 21,67% no caso da matriz MURA_17x17 e de 17% no caso da matriz MURA_43x43, e à redução no total de instruções executadas de 11,81% no caso da matriz MURA_17x17 e de 11,51% no caso da matriz MURA_43x43. As referências à memória também são reduzidas em 13,18% para o caso da matriz MURA_17x17 e em 19,27% para o caso da matriz MURA_43x43.

Assim, pode-se observar que nos dois casos apresentados, tanto a matriz 17x17 quanto a matriz 43x43 apresentam redução de FLOPs e de referências à memória, uma clara indicação de que o Fortran 90 fornece mecanismos mais eficientes de acesso à memória.

A redução do total de instruções executadas mostra a eficiência do Fortran 90 em gerar um código mais enxuto.

Os trechos 5 e 6 possuem a mesma estrutura e têm valores de tempo de execução iguais, assim, apenas um deles foi instrumentado. O trecho-5 executa a transposição de uma linha da matriz "arr", enquanto o trecho-6 realiza a transposição de uma coluna. O trecho-5 original, em Fortran 77, é apresentado abaixo.

```
DO i = 1, r * res
tmpx( i ) = arr( r*res, i )
END DO
DO k = r*res, 2, -1
        DO j = 1, r * res
        arr( k, j ) = arr( k - 1, j )
        END DO
END DO
DO i = 1, r * res
arr( l, i ) = tmpx( i )
END DO
```

O mesmo trecho pode ser reescrito com a notação do Fortran 90 de duas formas. A primeira utiliza a notação de vetor para executar a transposição das linhas da matriz:

```
tmpx = arr( lim, 1:lim )
arr( lim : 2 : -1, 1: lim ) = arr( lim - 1 : 1 : -1, 1 : lim )
arr( 1, 1: lim ) = tmpx
```

A segunda forma utiliza o comando CSHIFT, que realiza a transposição de linhas ou colunas de matrizes. O comando CSHIFT tem a seguinte sintaxe:

```
CSHIFT( vetor, qt, dimensão ).
```

O total de transposições é dado por "qt" e o tipo de transposição, linha ou coluna, é definido por "dimensão". Assim, a transposição de uma linha abaixo tem a seguinte sintaxe:

```
t_arr = CSHIFT( arr, 1, 1 )
arr = t_arr
```

A matriz temporária "t_arr" reduz as faltas de acesso ao *cache* pois o valor transladado é armazenado diretamente em outra posição de memória. A temporização dos três casos (original, utilizando notação de vetor e utilizando CSHIFT) apresentou resultados interessantes. A Tabela 6.5 mostra as contagens de FLOPs e instruções executadas neste trecho, bem como o tempo de execução, para os três casos.

TABELA 6.5 - COMPARAÇÃO DE EVENTOS MONITORADOS PARA O TRECHO-5

Evento	*F77*	*F90_array*	*F90_cshift*
FLOP's	352	325	1.180
Instruções Executadas	9.140	9.400	16.500
Tempo (µs)	113	100	150

A notação de vetor em Fortran 90 mostrou-se a mais eficiente por apresentar o menor tempo de execução. Assim, como o esperado era uma redução do tempo de execução com a utilização de CSHIFT, investigou-se os acessos ao *cache* comparando-se os três casos em função da variação das dimensões das matrizes.

Para essa análise, o trecho-5 foi monitorado para obtenção das taxas de acertos ao *cache* de dados (*cache hit rate*), para diversas dimensões de matrizes. A Figura 6.1 mostra os gráficos de *cache hit rate* para os três casos, comparando-os para uma variação das dimensões da matriz.

Pode-se observar que o código em Fortran 77 reduz o *cache hit rate* consideravelmente com o aumento das dimensões da matriz. A utilização da notação de vetor do Fortran 90

para implementar a transposição de linhas da matriz é a mais eficiente, pois o *cache hit rate* mantém-se constante com 98% de acertos em todas as dimensões testadas.

.

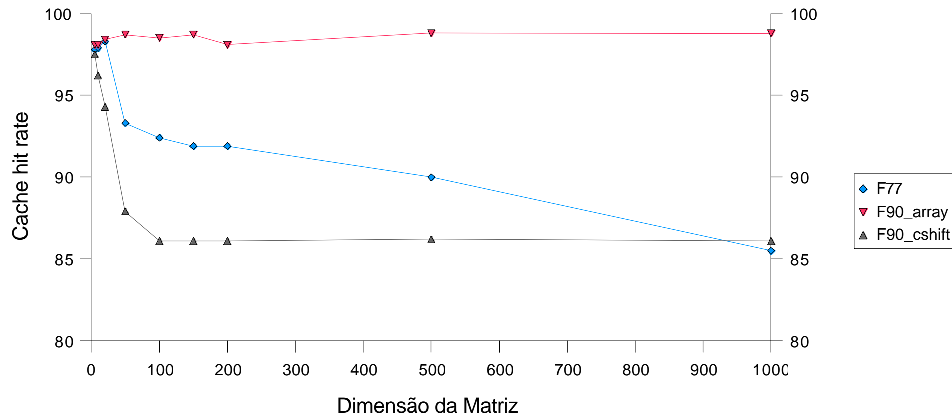


Fig. 6.2 - Comparativo do *cache hit rate* para casos de transposição de linhas.

O uso de CSHIFT em programas seqüenciais não se mostrou eficiente, pelo menos até o limite de dimensão analisado. O aumento do total de instruções executadas mostra que o CSHIFT não consegue gerar uma comunicação eficiente na situação em estudo.

O programa corr_masco não possui matrizes com grandes dimensões, assim, optou-se pelo uso da notação de vetor do Fortran 90, por apresentar o menor tempo de execução. Desta forma, os trechos 4 e 5 foram escritos com esta notação. Diretivas HPF foram inseridas para paralelizar a aplicação e monitorar sua execução em duas máquinas. A inserção de diretivas do HPF já foi discutida no Capítulo 5. Abaixo, descreve-se as diretivas necessárias para distribuir os dados do programa corr_masco entre duas máquinas P6 em rede. A instrumentação de um trecho em HPF é similar ao descrito no Capítulo 5. Deve-se acrescentar a distribuição das variáveis de monitoração entre os processadores envolvidos.

a) Definir grade (2,1) de processadores:

```
!HPF$ processors, dimension ( 2, 1 ) :: P
```

b) Distribuir dados nas dimensões definidas pela grade:

```
!HPF$ distribute(block,block) onto P :: prod, corr
```

c) Definir nova grade de processadores para instrumentação:

```
!HPF$ processors, dimension( 2 ) :: T
```

d) Declarar as variáveis de instrumentação:

```
integer, dimension( 2 ) :: ct_0, ct_1, t_msr
```

e) Distribuir em blocos as variáveis de instrumentação para a grade definida:

```
!HPF$ distribute( block ) onto T :: ct_0, ct_1, t_msr
```

A instrumentação do trecho segue o esquema já mostrado previamente, resultando na seguinte estrutura:

```
call iniciaregistros( cesr_0, cesr_1 )
corr(rperms,cperms) = corr(rperms,cperms) + SUM( arr * count )
tempo_msr = leregistros(count_0, count_1)
ct_0 = count_0
ct_1 = count_1
....
```

O programa é compilado pelo "pghpf" e o arquivo executável deve estar presente nas máquinas envolvidas. O módulo também deve estar instalado em ambas, e com autorização para leitura e escrita. A linha de comando abaixo pode ser executada em qualquer uma das máquinas:

```
corr_masco - pghpf  - np 2 - host <m1> <m2> -stat cpus
```

A identificação das máquinas é dada por <m1> e <m2> . A opção "np" informa que serão utilizadas duas máquinas. A opção "stat cpus" solicita dados dos tempos de execução em cada máquina.

A Tabela 6.6 apresenta as contagens obtidas do total de operações de ponto flutuante e total de instruções executadas para o trecho-4. A matriz MURA_43x43 foi utilizada como matriz de entrada. As contagens são apresentadas para cada uma das máquinas.

TABELA 6.6 - MONITORAÇÃO DO TRECHO-4 EM MÁQUINAS PARALELAS

Evento	*CPU 1**	*CPU 2**
FLOPs	231.427	231.429
Instruções executadas	886.314	885.748
Tempo de execução(μs)	4.220	4.244

(*) - Microcomputador IBM Celeron 300MHz.

O mesmo trecho instrumentado, porém sem as diretivas HPF, foi executado seqüencialmente em uma das máquinas com o objetivo de comparar o código sendo executado em uma máquina seqüencial com a execução em paralelo com duas máquinas, de forma a mostrar a distribuição dos dados entre essas máquinas e a redução do tempo de execução. Os resultados da monitoração sem distribuição dos dados estão apresentados na Tabela 6.7.

TABELA 6.7 - MONITORAÇÃO DO TRECHO-4 EM MÁQUINA SEQÜENCIAL

Evento	***CPU 1**** ***(contagens)***
FLOPs	370.757
Instruções executadas	2.112.221
Tempo de execução(μs)	11.383

A contagem das operações de ponto flutuante e o total de instruções executadas confirmam a distribuição dos dados entre as máquinas. Enquanto ao ser executado seqüencialmente foram registradas 370.757 operações em ponto flutuante, ao executar o código em paralelo, foram registradas 231.427 operações em ponto flutuante em uma máquina e 231.429 na outra. Deve-se observar que nas contagens de FLOPs estão incluídas as operações de acesso aos diversos operandos envolvidos.

O código em Fortran 90 permite reduzir de forma significativa o total de instruções executadas. Enquanto o código para ser executado seqüencialmente gera um pouco mais de dois milhões de instruções executadas, o código para execução em paralelo gera por volta de oitocentos e oitenta mil instruções. Esta redução para 1/3 do total de instruções executadas mostra a eficiência do compilador Fortran 90 e a eficiência da distribuição dos dados. A execução em paralelo desse código permite assim uma redução de 62,3%, enquanto no código seqüencial o tempo de execução é de 11.383 microsegundos, ao executar em paralelo, o tempo reduz-se a 4.220 seg na máquina CPU 1 e a 4.244 seg na máquina CPU 2.

Utilizando como arquivo de entrada a matriz mura_43x43, o compilador pghpf possibilita gerar valores estatísticos de tempos de execução com o uso da opção "stat cpus". Os valores obtidos estão apresentados na Tabela 6.8.

TABELA 6.8 - VALORES DE TEMPO DA APLICAÇÃO EXECUTADA EM PARALELO EM DUAS MÁQUINAS

CPU	***Real(seg.)***	***Usuário(seg.)***	***Sistema(seg.)***
0	382,79	346,02	0,46
1	382,79	348,79	0,37

A comparação entre o código original e o código implementado com diretivas HPF mostra a otimização de desempenho máxima obtida. A Tabela 6.9 apresenta os tempos totais de execução para as matrizes MURA_17 x 17 e MURA_43 x 43, comparando-se o código original, compilado com os compiladores pgf77, pgf90 e com diretivas HPF compilado pelo compilador pghpf e executado em paralelo, em duas máquinas.

TABELA 6.9 - COMPARAÇÃO DOS TEMPOS TOTAIS DE EXECUÇÃO ANTES E DEPOIS DA OTIMIZAÇÃO

Matriz	**F77 Tempo(seg)**	**F90 Tempo(seg)**	**HPF Tempo(seg)**	
			CPU 1	**CPU 2**
17x17	120	105	85	83
43x43	1059	830	382,79	382,79

A Tabela 6.9 permite mostrar a eficiência do código escrito em Fortran 90 e compará-lo com o código original. O código em Fortran 90 reduz o tempo de execução em 12,5% para a matriz 17x17. Utilizando como entrada a matriz 43x43, obtém-se melhores tempos de execução, com redução de 21,6%. A coluna HPF mostra os tempos obtidos com execução do código em paralelo em duas máquinas.

A redução do tempo total de execução da aplicação corr_masco ao se paralelizar o código é de 63,85 % para o caso da matriz de entrada MURA_43x43, comprovando assim que as otimizações implementadas no código original realmente resultaram em aumento de desempenho.

6.2 - Exemplo de Otimização por Mudança de Distribuição

A eficiência de um aplicativo pode ser afetada por uma má distribuição de dados. O uso correto da distribuição é fundamental na otimização de desempenho de um código que utilize o paradigma de programação paralela de dados.

O aplicativo PDE1, listado no Apêndice G, foi desenvolvido por Lemke[22] e resolve a equação de Poissom em três dimensões utilizando aceleração de Chebyshev. A distribuição de uma dimensão em uma das matrizes do programa, denominada RED, foi alterada para melhorar a localidade dos dados. Neste exemplo, faz-se inicialmente a instrumentação no caso da distribuição original (BLOCK,*,*) e, posteriormente, comparam-se os resultados obtidos com uma nova distribuição (*,*,BLOCK). O programa é instrumentado com as funções de monitoração dos eventos, com contagens de operações de ponto flutuante, total de instruções executadas, acessos à memória *cache* e falhas de acesso ao TLB. O programa opera com as matrizes tri-dimensionais U, F, T e RED. A versão HPF apresentada alinha as matrizes U, F e T com a matriz RED, a qual é distribuída para a grade unidimensional FARM, conforme mostrado abaixo.

```
!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN T(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM
```

O trecho identificado por *Relaxation of Red and Black points*, no qual ocorre a maior parte do processamento, foi instrumentado. Os eventos de interesse são monitorados com a distribuição original (BLOCK,*,*), do programa PDE1, e, posteriormente, os

mesmos eventos são monitorados para a distribuição (*,*,BLOCK), do programa PDE2. Os programas são executados para o caso onde as matrizes possuem dimensões I=J=K=128. A Figura 6.3 apresenta o trecho do programa PDE1 instrumentado.

```
Call iniciaregistros( cesr_0, cesr_1)
WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
!    RELAXATION OF THE RED POINTS
U(2:NX-1,2:NY-1,2:NZ-1)                                          &
&  ROMINUS*U(2:NX-1,2:NY-1,2:NZ-1) +                             &
&  RFACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+                         &
&  U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+                &
& U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+                 &
&  U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ)
ELSEWHERE
!     RELAXATION OF THE BLACK POINTS
U(2:NX-1,2:NY-1,2:NZ-1) =                                        &
& BOMINUS*U(2:NX-1,2:NY-1,2:NZ-1) +                              &
& BFACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+                          &
& U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+                 &
& U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+                 &
& U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
END WHERE
t_msr = leregistros( count_0, count_1 )
ct_0 = count_0
ct_1 = count_1
```

Fig. 6.3 - Trecho instrumentado do programa PDE1.

A Tabela 6.10 mostra os resultados obtidos para o trecho monitorado do aplicativo PDE1.

TABELA 6.10 - EVENTOS REGISTRADOS PARA DISTRIBUIÇÃO ORIGINAL DO APLICATIVO PDE1

Eventos	CPU 1 (contagens)	CPU 2 (contagens)
Tempo(μs)	411.954	418.410
FLOPs	1.310.858	1.310.841
Instruções executadas	41.175.264	41.398.247
TLB *misses*	1.115	725
Acessos a memória	29.441.700	29.371.157
L2-*Loads*	566.922	537.263
L2-*Stores*	157.192	146.917
L2-*Requests*	791.662	758.149

Alterando-se a distribuição da matriz RED para (*,*,BLOCK) e mantendo-se a instrumentação para monitoração dos mesmos eventos que na distribuição anterior, obtém-se os valores apresentados na Tabela 6.11, relativos ao novo aplicativo PDE2.

TABELA 6.11 - EVENTOS REGISTRADOS PARA DISTRIBUIÇÃO DO APLICATIVO PDE2

Eventos	CPU 1* (contagens)	CPU 2* (contagens)
Tempo(μs)	358.229	355.121
FLOPs	1.310.890	1.310.853
Instruções executadas	20.986.438	20.922.303
TLB *misses*	1.022	784
Acessos a memória	13.179.861	13.163.032
L2-*Loads*	451.044	446.734
L2-*Stores*	118.483	117.747
L2-*Requests*	636.570	633.065

A comparação dos valores obtidos com as duas distribuições mostra uma redução de contagens em todos os eventos, indicando que a distribuição (*,*,BLOCK) é mais eficiente que a distribuição (BLOCK,*,*). Os valores percentuais de redução para cada

evento estão apresentados na Tabela 6.12. Os dados para estimativa da redução correspondem ao maior valor entre as contagens para cada evento.

TABELA 6.12 - REDUÇÃO DE CONTAGENS COMPARANDO PDE1 COM PDE2

Eventos	Redução(%)
Tempo(μs)	14,38
FLOPs	0,00
Instruções executadas	49,31
TLB *misses*	8,34
Acessos a memória	55,23
L2-*Loads*	20,44
L2-*Stores*	24,63
L2-*Requests*	19,59

A redução de 14,38% no tempo de execução do trecho mostra que a distribuição (*,*,BLOCK) realmente é mais eficiente. As maiores reduções, ocorridas com o total de instruções executadas e acessos à memória, mostram a eficiência do compilador em gerar um código executável mais eficiente para a segunda distribuição.

Observa-se que o total de FLOPs permaneceu constante nas duas versões, e que o total de instruções e de acessos à memória são maiores na versão original comparado com a versão listada em PDE2. Pode-se estimar que há uma maior necessidade de comunicação na distribuição dada em PDE1 (BLOCK,*,*), o que resulta no maior tempo de execução. Assim, esse gargalo representado pela comunicação, para esta versão do programa, deve acentuar-se com o aumento do número de processadores.

As duas versões foram executadas na máquina Polaris do LAC, possuidora de quatro CPUs Intel / Pentium Pro.

A execução do programa original PDE1, não-instrumentado, na máquina Polaris, executando com dois processadores, produz os resultados da Tabela 6.13.

TABELA 6.13 - VALORES DE TEMPO DE PDE1 COM 2 PROCESSADORES(POLARIS)

CPU	T.Real (seg)	T.Usuário (seg)	T.Sistema (seg)
0	206,06	180,64	20,18
1	206,98	181,38	20,10

A Tabela 6.14 mostra os resultados de tempo de execução obtidos com PDE2 rodando na Polaris com dois processadores.

TABELA 6.14 - VALORES DE PDE2 COM 2 PROCESSADORES (POLARIS)

CPU	T.Real (seg)	T.Usuário (seg)	T.Sistema (Seg)
0	170,14	153,94	15,53
1	170,11	155,77	13,90

A redução do tempo total de execução em cada processador, de PDE1 para PDE2, mostra a eficiência da nova distribuição com dois processadores. Os dois casos também foram executados na Polaris com quatro processadores. Os resultados de tempos de execução, em cada processador, estão exibidos na Tabela 6.15, para PDE1, e na Tabela 6.16, para PDE2.

TABELA 6.15 - VALORES DE TEMPO DE PDE1 COM 4 PROCESSADORES (POLARIS)

CPU	T.Real (seg)	T.Usuário (seg)	T.Sistema (seg)
0	242,83	163,54	28,12
1	242,82	179,12	33,52
2	242,81	182,34	34,20
3	242,81	153,16	29,85

TABELA 6.16 - VALORES DE TEMPO DE PDE2 COM 4 PROCESSADORES (POLARIS)

CPU	T.Real (seg)	T.Usuário (seg)	T.Sistema (seg)
0	174,80	147,27	21,67
1	174,79	150,27	22,69
2	174,78	150,37	22,27
3	174,78	137,93	19,36

O tempo de execução de PDE1 com dois processadores é bem menor que o tempo medido com quatro processadores, confirmando a tendência já citada de aumento do tempo de comunicação para este caso, não permitindo assim um ganho de eficiência.

O Capítulo 6 apresentou a utilização dos contadores de hardware no paradigma de programação paralela de dados. Uma aplicação real foi totalmente reestruturada com notação do Fortran 90 e diretivas HPF foram inseridas. Após um levantamento do perfil de tempos de execução do aplicativo, identificando os trechos consumidores de tempo de CPU, vários trechos foram monitorados. O programa foi executado em duas máquinas P6 com a obtenção de contagens de diversos eventos. Em seguida, um outro aplicativo foi instrumentado e apenas fez-se uma análise da distribuição de parte de seus dados, mostrando que uma distribuição ruim pode reduzir o desempenho de uma aplicação. Nos dois casos, os resultados obtidos mostraram a simplicidade de uso e eficiência da metodologia de otimização de desempenho utilizando contadores de hardware.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

O desenvolvimento do presente trabalho buscou disponibilizar um método para analisar o desempenho de aplicações científicas e, em função das informações obtidas, reestruturar o código fonte para se obter o desempenho máximo para um dado sistema.

A preparação do ambiente de trabalho é bem simples, exigindo somente a instalação do módulo de acesso aos contadores de *hardware* e a instrumentação do código-fonte no trecho a ser analisado. A instrumentação de um trecho requer apenas a inserção de poucas linhas de instrumentação, além das definições das variáveis envolvidas. Um processo simples, que permite monitorar diversos eventos internos à CPU, os quais seriam inacessíveis de outra maneira. A monitoração de tais eventos traz ao desenvolvedor resultados confiáveis e rápidos, permitindo identificar os pontos críticos de consumo do tempo de execução.

O método apresentado para programas seqüenciais permite que, em uma primeira etapa, medidas precisas de tempos de execução de qualquer trecho do código original sejam efetuadas, sem perturbar significativamente o trecho em análise. A resolução de microsegundos permite monitorar o tempo de execução de qualquer linha ou trecho de código.

Os testes de monitoração apontam para uma estratégia a ser seguida na análise de um programa, onde os dados de referência a padrões de memória devem ser utilizados para a exploração da localidade nos trechos críticos, identificados pelas medidas de tempo de execução. Nas aplicações seqüenciais a monitoração dos acertos e falhas ao *cache* de dados é fundamental para a análise. Mesmo com a obtenção de redução do tempo de execução de um trecho após melhorar o padrão de referência à memória, deve-se analisar as falhas aos *caches* de instrução e TLB. Em geral, deve-se buscar uma taxa de acertos ao *cache* de dados superior a 90%, procurando alcançar uma taxa de 95%.

Os dados obtidos no exemplo de blocagem mostram a eficiência ao se utilizar técnicas de otimização, bem como a importância da análise do padrão de acesso à memória. Para o caso estudado, um aumento de apenas 3.7 pontos percentuais na taxa de acertos ao *cache* reduz em 37,5% o tempo de execução do trecho.

Os resultados obtidos na análise do aplicativo Hydrolight mostram que a identificação de pontos onde existam problemas de referência à memória pode influir muito pouco no desempenho, pois podem existir outros fatores que se sobrepõem ao problema de referência à memória. Neste caso, deve-se pensar em reestruturar toda a aplicação.

O estudo para efetuar a instrumentação de programas paralelos mostra que se deve preferenciar a monitoração dos acertos e falhas ao *cache* de dados bem como o total de operações de ponto flutuante e de instruções executadas, pois fornecem informações sobre a distribuição dos dados entre os processadores envolvidos.

A estratégia a ser seguida deve procurar a otimização caso a caso. A baixa exploração da localidade dos dados no *cache* é a principal razão dos gargalos, sendo portanto o primeiro ponto de ataque do problema. No caso específico de programação paralela, a localidade deve ser analisada para cada processador. Após aplicação das técnicas de otimização, deve-se confirmar a validade das alterações, através da repetição das medidas.

Nas aplicações paralelas, o objetivo é manter os processadores ocupados com uma distribuição onde a comunicação seja minimizada. O exemplo PDE1 mostrou como uma distribuição errada pode contribuir para a redução de desempenho.

Na implementação apresentada, pode-se ressaltar que o trecho monitorado não pode ser muito grande, ou seja, não pode gerar contagens superiores a 32 *bits* ou assumir um tempo de execução muito alto, pois o Linux irá executar partes de outros aplicativos que contaminariam as medidas efetuadas. Quanto menor o trecho, mais precisas são as

contagens dos eventos. O excesso de processos rodando nas máquinas também implica em contaminação da medida. Uma forma de combater estes efeitos, futuramente, seria modificar o núcleo do Linux de tal maneira que os valores de contagens de cada processo fossem salvos a cada mudança de contexto.

O uso dos contadores de *hardware*, neste trabalho, permite abrir uma única instância do módulo, ou seja, apenas um usuário pode ter acesso ao módulo MSR. Os contadores são utilizados aos pares e assim uma análise detalhada, com monitoração de vários eventos, exige que o programa seja executado várias vezes. Caso as contagens fossem salvas durante as mudanças de contexto, diversos processos poderiam ser monitorados simultaneamente.

Os contadores de *hardware* mostraram ser uma ferramenta de fácil utilização e que gera informações importantes sobre a interação do código executável da aplicação com a arquitetura do sistema. As mesmas técnicas aqui apresentadas poderiam ser aplicadas, no futuro, a muitas das outras arquiteturas atualmente disponíveis, onde também existem recursos de *hardware*, internos ao processador, para contagem de eventos.

A diversidade de eventos que podem ser monitorados nas mais diversas plataformas pode contribuir muito para a computação paralela, pois a evolução dos compiladores para o paradigma da programação paralela não está acompanhando o aumento da complexidade dos problemas científicos; a exigência de mais potência computacional requer a utilização de todas as formas possíveis para maximização da eficiência no uso do *hardware* disponível.

O futuro aponta para o desenvolvimento de um novo paradigma da computação paralela, integrando recursos computacionais e de informação distribuídos geograficamente. Neste novo ambiente, denominado *The Grid*[31], os contadores de *hardware* podem ser de grande utilidade, pois monitorariam e identificariam localmente qualquer adversidade do aplicativo com o processador ou a eficiência de seu acesso à memória.

Futuros desenvolvimentos poderiam contar com aplicativos similares ao desenvolvido pela Compaq[18], disponíveis para as mais diversas plataformas, auxiliando na máxima sintonia do código executado com o processador. Programas locais poderiam monitorar constantemente os *pipelines* do processador ou o acesso à memória, informando sempre que ocorrer um travamento de barramento ou um baixo rendimento do fluxo de instruções. Estas informações circulariam para a máquina mestre que redistribuiria os dados ou códigos, fornecendo informações para otimizar o desempenho do aplicativo. Programas rodando no núcleo poderiam monitorar o total de instruções por ciclo, mantendo o programador informado, em tempo real, das condições de execução de cada máquina, permitindo o estudo dos pontos críticos do aplicativo. Os aplicativos poderiam conter pequenos programas de análise das características internas de cada máquina, tais como tipo de processador ou dimensões das memórias *cache*, e com essas informações ajustar a carga computacional disponível para a correta distribuição dos dados.

A necessidade de se desenvolver aplicativos não restritos a uma plataforma exige que os fabricantes estabeleçam um padrão com definição dos tipos de eventos que poderiam ser considerados básicos. A sua utilização na computação paralela pode ser de grande valia no desenvolvimento de aplicações, pois ofereceria ao programador as condições de contorno para se buscar o máximo de eficiência de um aplicativo nas plataformas disponíveis.

REFERÊNCIAS BIBLIOGRÁFICAS

1 - Intel. **Intel developers**. [online]. <http://developer.intel.com> . February 1998.

2 - Rubini, A. **Linux device drivers.** Los Angeles: O'Reilly & Associates, 1998. 401 p.

3 - Intel developers group. **Intel processor identification and the CPUID instruction – application note – AP485**. Santa Clara: Intel, 1995. 42 p. (Intel-241618-009).

4 - Intel developers group. **Intel architecture software developer's manual**. Santa Clara: Intel, 1998. V.1. 283 p. (Intel-24319001).

5 - Intel developers group. **Intel architecture software developer's manual.** . Santa Clara: Intel, 1998. V.2. 566 p. (Intel-24319101).

6 - Intel developers group. **Intel architecture software developer's manual**. Santa Clara: Intel, 1998. V.3. 584 p. (Intel-24319201).

7 - Intel developers group. **Embedded Pentium processor family developer's manual**.Santa Clara: Intel, 1998. 32 p. (Intel-27320401).

8 - Marshall,A.C. **HPF programming course notes.** [online]. <http://www.liv.ac.uk/HPC/HTMLFrontPageHPF.html>. Sep. 1999.

9 - Ewing, A.K.; Hare, R.J.; Richardson, H.;Simpson, A.D. **Writing data parallel programmes with high performance Fortran.** [online].
<http://www.epcc.ed.ac.uk/epcc-tec/courses.html>. Oct. 1999.

10 - Braga, J. **Correlation program for the MASCO project.** [programa de computador]. INPE, São José dos Campos, mar. 1992.

11 - Meyer, S. **MSR X86.** [online]. <http://62.144.158.47/stephan//msr.html>. Jun. 1999.

12 - Goda, M.P.; Warren, M.S. **Model specific registers and performance monitoring**. [online]. <http://www.scl.ameslab.gov/workshop/PPCworkshop.html> . Jun. 1998.

13 - Dowd, K. **High performance computing.** Sebastopol: O'Reilly & Associates. 1993. 371 p.

14 - Hwang, K.; Briggs, F. A. **Computer architecture and parallel processing.** New York: McGraw-Hill Book. 1984. 846 p.

15 - Ellis, T.M.R. ; Philips, I. R.; Lahey, T. M. **Fortran 90 programming.** Harlow: Addison-Wesley. 1994. 825 p.

16 - Mobley, C. D. **Hydrolight 3.0**. [programa de computador]. Menlo Park: SRI International, 1995.

17 - Stephany, S. **Reconstrução de propriedades óticas e de fontes de bioluminescência em águas naturais.** São José dos Campos. 122 p. (INPE-6968-TDI/656) Tese (Doutorado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, 1997.

18 - Compaq. **The Compaq (Digital) continuous profiling infrastructure(DCPI)** .[online].<http://www.unix.digital.com/dcpi/>. Oct. 1999.

19 - Roth, C.; Levine, F.; Welbon, E**.** Performance monitoring on the PowerPC 604 microprocessor. In: IEEE International Conference on Computer Design, Austin, 1995. **Anais**. Austin: proceedings of the 1995 IEEE International Conference on Computer Design, 1995. pp. 212-215.

20 - Mucci,P.J; Browne,S.;Deane, C.; Ho, G. **PAPI: a portable interface to *hardware* performance counters**. [online]. <http://icl.cs.utk.edu/projects/papi/dodugm99/papi.html>. Apr. 2000.

21 - Stephany, S.; Correa, R.V.; Mendes, C.L.; Preto, A.J. **Identifying performance bottlenecks in a radiative tranfer application**. Sixth International Conference on Application of High-Performance Computers in Engineering. Southampton: WIT Press, 2000. p. 51-60.

22 - Lemke, M. **3-Dimensional poison solver using red-black relaxation SOR with Chebishev accelaration**. [programa de computador]. Germany. May 1993.

23 - Hennessy, J. L.; Patterson, D. A. **Computer architecture: a quantitative approach**. Maddison: Morgan Kauffman Publishers. p.760. 1995.

24 - Bhandarkar, D. RISC versus CISC: A tale of two chips**. Computer Architecture News**, v. 25, n. 1, p. 1-12, mar. 1994.

25 - Zagha, M.; Larson, B.; Turner, S.; Itzkowitz, M.; Yu, J. **Performance analysis using the MIPS R10000 performance counters.** [online]. <http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/INDEX.HTM> Dec. 1999.

26 - Welbon, E. H.; Chan-nui, C. C.; Shippy, D. J. The Power2 performance monitor. **IBM Journal of Research and Development,** v. 38, n. 5, p. 545-554, sep. 1994.

27 - Bakoglu, H. B.; Grohosky, G. F.; Montoye, R. K. The IBM RISC System/6000 processor: hardware overview. **IBM Journal of Research and Development,** v.34, n. 1, p. 12-22, jan. 1990.

28 - Loveman, D. B. High performance Fortran. **IEEE Parallel & Distributed Technology,** v. 1, n. 1, p. 25-42, feb. 1993.

29 - Pacheco, Peter. **Parallel programming with MPI.** Maddison: Morgan Kauffman Publishers. 419 p. 1997.

30 - Gropp, W.; Lusk E.; Skjellum, A. **Using MPI.** Boston: The MIT Press. 371 p. 1994.

31 - Foster, I.; Kesselman, C. **The Grid, blueprint for a new computing infrastructure**. Maddison: Morgan Kauffman Publishers. 676 p. 1999.

APÊNDICE A

BIBLIOGRAFIA COMPLEMENTAR

BIBLIOGRAFIA COMPLEMENTAR

1 - Matthew, N.; Stones, R. **Linux programming**. Birmingham: Wrox Press, 1997. 710 p.

2 - Johnson, M. K.; Troan, E. W. **Linux application development**. Reading: Addison Wesley, 1999. 538 p.

3 - Pitts, D.; Ball, B.. **Red Hat Linux 6 unleashed**. Indianapolis: SAMS, 1999. 1252 p.

4 - Schildt, H. **C - The complete reference**. Berkeley: Osborne McGraw-Hill, 1987. 773 p.

5 - Intel developers group. **Intel architecture optimization manual**. Santa Clara: Intel, 1997. 150 p. (Intel-242816-003) .

6 - Anderson, Don; Shanley,Tom. **Pentium processor system architecture.** Reading: MindShare. 1995. 433 p.

7 - Cameron, K.; Luo, Y.; Lubeck, O. Performance evaluation using *hardware* counters. In: The 26th International Symposium on Computer Architecture. Atlanta, 1999. **Anais**.

8 - Shapiro, J.S. Tuning a fast capability system. In: First Workshop on PC-based System Performance and Analysis. San Jose, 1998. **Anais**.

9 - Intel developers group. **Pentium II processor developer's manual**. Santa Clara: Intel, 1997. 226 p. (Intel-243502-001).

APÊNDICE B

Eventos Disponíveis para Monitoração de CPU's da Família INTEL P5

Eventos Disponíveis para Monitoração de CPU's da Família INTEL P5

Event Num.	Mnemonic Event Name	Description	Comments
00H	DATA_READ	Number of memory data reads (internal data cache hit and miss combined).	Split cycle reads are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
01H	DATA_WRITE	Number of memory data writes (internal data cache hit and miss combined), I/O is not included.	Split cycle writes are counted individually. These events may occur at a maximum of two per clock. I/O is not included.
0H2	DATA_TLB_MISS	Number of misses to the data cache translation look-aside buffer.	
03H	DATA_READ_MISS	Number of memory read accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
04H	DATA WRITE MISS	Number of memory write accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.

Continua

Continuação

Event Num.	Mnemonic Event Name	Description	Comments
05H	WRITE_HIT_TO_ M-_OR_E- STATE_LINES	Number of write hits to exclusive or modified lines in the data cache.	These are the writes that may be held up if EWBE# is inactive. These events may occur a maximum of two per clock.
06H	DATA_CACHE_ LINES_ WRITTEN_BACK	Number of dirty lines (all) that are written back, regardless of the cause.	Replacements and internal and external snoops can all cause writeback and are counted.
07H	EXTERNAL_ SNOOPS	Number of accepted external snoops whether they hit in the code cache or data cache or neither.	Assertions of EADS# outside of the sampling interval are not counted, and no internal snoops are counted.
08H	EXTERNAL_DATA_ CACHE_SNOOP_ HITS	Number of external snoops to the data cache.	Snoop hits to a valid line in either the data cache, the data line fill buffer, or one of the write back buffers are all counted as hits.
09H	MEMORY ACCESSES IN BOTH PIPES	Number of data memory reads or writes that are paired in both pipes of the pipeline.	These accesses are not necessarily run in parallel due to cache misses, bank conflicts, etc.
0AH	BANK CONFLICTS	Number of actual bank conflicts.	
0BH	MISALIGNED DATA MEMORY OR I/O REFERENCES	Number of memory or I/O reads or writes that are misaligned.	A 2- or 4-byte access is misaligned when it crosses a 4-byte boundary; an 8-byte access is misaligned when it crosses an 8-byte boundary. Ten byte accesses are treated as two separate accesses of 8 and 2 bytes each.
0CH	CODE READ	Number of instruction reads whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0DH	CODE TLB MISS	Number of instruction reads that miss the code TLB whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0EH	CODE CACHE MISS	Number of instruction reads that miss the internal code cache whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.

Continua

Continuação

Event Num.	Mnemonic Event Name	Description	Comments
0FH	ANY SEGMENT REGISTER LOADED	Number of writes into any segment register in real or protected mode including the LDTR, GDTR, IDTR, and TR.	Segment loads are caused by explicit segment register load instructions, far control transfers, and task switches. Far control transfers and task switches causing a privilege level change will signal this event twice. Note that interrupts and exceptions may initiate a far control transfer.
10H	Reserved		
11H	Reserved		
12H	Branches	Number of taken and not taken branches, including conditional branches, jumps, calls, returns, software interrupts, and interrupt returns.	Also counted as taken branches are serializing instructions, VERR and VERW instructions, some segment descriptor loads, hardware interrupts (including FLUSH#), and programmatic exceptions that invoke a trap or fault handler. The pipe is not necessarily flushed. The number of branches actually executed is measured, not the number of predicted branches.
13H	BTB_HITS	Number of BTB hits that occur.	Hits are counted only for those instructions that are actually executed.
14H	TAKEN_BRANCH_ OR_BTB_HIT	Number of taken branches or BTB hits that occur.	This event type is a logical OR of taken branches and BTB hits. It represents an event that may cause a hit in the BTB. Specifically, it is either a candidate for a space in the BTB or it is already in the BTB.
15H	PIPELINE FLUSHES	Number of pipeline flushes that occur. Pipeline flushes are caused by BTB misses on taken branches, mis-predictions, exceptions, interrupts, and some segment descriptor loads.	The counter will not be incremented for serializing instructions (serializing instructions cause the prefetch queue to be flushed but will not trigger the Pipeline Flushed event counter) and software interrupts (software interrupts do not flush the pipeline).

Continua

Continuação

Event Num.	Mnemonic Event Name	Description	Comments
16H	INSTRUCTIONS_ EXECUTED	Number of instructions executed (up to two per clock).	Invocations of a fault handler are considered instructions. All hardware and software interrupts and exceptions will also cause the count to be incremented. Repeat prefixed string instructions will only increment this counter once despite the fact that the repeat loop executes the same instruction multiple times until the loop criteria is satisfied. This applies to all the Repeat string instruction prefixes (i.e., REP, REPE, REPZ, REPNE, and REPNZ). This counter will also only increment once per each HLT instruction executed regardless of how many cycles the processor remains in the HALT state.
17H	INSTRUCTIONS_ EXECUTED_ V PIPE	Number of instructions executed in the V_pipe. It indicates the number of instructions that were paired.	This event is the same as the 16H event except it only counts the number of instructions actually executed in the V-pipe.
18H	BUS_CYCLE_ DURATION	Number of clocks while a bus cycle is in progress. This event measures bus use.	The count includes HLDA, AHOLD, and BOFF# clocks.
19H	WRITE_BUFFER_ FULL_STALL_ DURATION	Number of clocks while the pipeline is stalled due to full write buffers.	Full write buffers stall data memory read misses, data memory write misses, and data memory write hits to S-state lines. Stalls on I/O accesses are not included.
1AH	WAITING_FOR_ DATA_MEMORY_ READ_STALL_ DURATION	Number of clocks while the pipeline is stalled while waiting for data memory reads.	Data TLB Miss processing is also included in the count. The pipeline stalls while a data memory read is in progress including attempts to read that are not bypassed while a line is being filled.
1BH	STALL ON WRITE TO AN E- OR M-STATE LINE	Number of stalls on writes to E- or M-state lines	
1CH	LOCKED BUS CYCLE	Number of locked bus cycles that occur as the result of the LOCK prefix or LOCK instruction, page-table updates, and descriptor table updates.	Only the read portion of the locked read-modify-write is counted. Split locked cycles (SCYC active) count as two separate accesses. Cycles restarted due to BOFF# are not re-counted.

Continua

Continuação

Event Num.	Mnemonic Event Name	Description	Comments
1DH	I/O READ OR WRITE CYCLE	Number of bus cycles directed to I/O space.	Misaligned I/O accesses will generate two bus cycles. Bus cycles restarted due to BOFF# are not re-counted.
1EH	NONCACHEABLE_ MEMORY_READS	Number of noncacheable instruction or data memory read bus cycles. Count includes read cycles caused by TLB misses, but does not include read cycles to I/O space.	Cycles restarted due to BOFF# are not re-counted.
1FH	PIPELINE_AGI_ STALLS	Number of address generation interlock (AGI) stalls. An AGI occurring in both the U- and V- pipelines in the same clock signals this event twice.	An AGI occurs when the instruction in the execute stage of either of U- or V- pipelines is writing to either the index or base address register of an instruction in the D2 (address generation) stage of either the U- or V- pipelines.
20H	Reserved		
21H	Reserved		
22H	FLOPS	Number of floating-point operations that occur.	Number of floating-point adds, subtracts, multiplies, divides, remainders, and square roots are counted. The transcendental instructions consist of multiple adds and multiplies and will signal this event multiple times. Instructions generating the divide by zero, negative square root, special operand, or stack exceptions will not be counted. Instructions generating all other floating-point exceptions will be counted. The integer multiply instructions and other instructions which use the FPU will be counted.

Continua

Continuação

Event Num.	Mnemonic Event Name	Description	Comments
23H	BREAKPOINT MATCH ON DR0 REGISTER	Number of matches on register DR0 breakpoint.	The counters is incremented regardless if the breakpoints are enabled or not. However, if breakpoints are not enabled, code breakpoint matches will not be checked for instructions executed in the V-pipe and will not cause this counter to be incremented. (They are checked on instruction executed in the U-pipe only when breakpoints are not enabled.) These events correspond to the signals driven on the BP[3:0] pins. Refer to Chapter 14, *Debugging and Performance Monitoring*, for more information.
24H	BREAKPOINT MATCH ON DR1 REGISTER	Number of matches on register DR1 breakpoint.	See comment for 23H event.
25H	BREAKPOINT MATCH ON DR2 REGISTER	Number of matches on register DR2 breakpoint.	See comment for 23H event.
26H	BREAKPOINT MATCH ON DR3 REGISTER	Number of matches on register DR3 breakpoint.	See comment for 23H event.
27H	HARDWARE INTERRUPTS	Number of taken INTR and NMI interrupts.	
28H	DATA_READ_OR_ WRITE	Number of memory data reads and/or writes (internal data cache hit and miss combined).	Split cycle reads and writes are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
29H	DATA_READ_MISS OR_WRITE MISS	Number of memory read and/or write accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.

Conclusão

APÊNDICE C

Eventos Disponíveis para Monitoração de CPU's da Família P6

Eventos Disponíveis para Monitoração de CPU's da Família P6

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit(DCU)	43H	DATA_MEM_ REFS	00H	All memory References, both Cachable and noncachable.	
	45H	DCU_LINES_IN	00H	Total lines allocated in The DCU.	
	46H	DCU_M_LINES_IN	00H	Number of M state Lines allocated in the DCU.	
	D5H	SEG_REG_ RENAMES		Number of Segment Register Renames:	Available in Pentium II Processor only.
			01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register GS	
			OFH	Segment register ES + DS + FS + GS	
	D6H	RET_SEG_ RENAMES	00H	Number of segment Register rename Events retired	Available in Pentium II Processor only.

Continua.

Continuação.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_INM	00H	Number of modified lines allocated in the L2.	
	27H	L2_M_LINES_ OUTM	00H	Number of modified lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Number of L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the data bus was busy.	
	23H	L2_DBUS_BUSY_ RD	00H	Number of cycles during which the data bus was busy transferring data from L2 to the processor.	
External Bus Logic (EBL)[2]	62H	BUS_DRDY_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY is asserted.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY. Unit Mask = 20H counts in processor clocks when any agent is driving DRDY.
	63H	BUS_LOCK_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK is asserted.	Always counts in processor clocks.
	60H	BUS_REQ_ OUTSTANDING	00H (Self)	Number of bus requests outstanding.	Counts only DCU full-line cacheable reads, not RFOs, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of burst read transactions.	

Continua.

Continuação.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of write back transactions.	
	68H	BUS_TRAN_ IFETCH	00H (Self) 20H (Any)	Number of instruction fetch transactions.	
	69H	BUS_TRAN_ INVAL	00H (Self) 20H (Any)	Number of invalidate transactions.	
	6AH	BUS_TRAN_ PWR	00H (Self) 20H (Any)	Number of partial write transactions.	
	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of deferred transactions.	
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all transactions.	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	Number of memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR pin.	
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT pin.	Includes cycles due to snoop stalls.
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM pin.	Includes cycles due to snoop stalls.

Continua.

Continuação.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired.	Counter 0 only
	10H	FP_COMP_OPS_ EXE	00H	Number of computational floating-point operations executed.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only.
	12H	MUL	00H	Number of multiplies.	Counter 1 only.
	13H	DIV	00H	Number of divides.	Counter 1 only.
	14H	CYCLES_DIV_ BUSY	00H	Number of cycles during which the divider is busy.	Counter 0 only.
Memory Ordering	03H	LD_BLOCKS	00H	Number of store buffer blocks.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles.	
	05H	MISALIGN_ MEM_REF	00H	Number of misaligned data memory references.	
Instruction Decoding and Retirement	C0H	INST_RETIRED	OOH	Number of instructions retired.	
	C2H	UOPS_RETIRED	00H	Number of UOPs retired.	
	D0H	INST_DECODER	00H	Number of instructions decoded.	
Interrupts	C8H	HW_INT_RX	00H	Number of hardware interrupts received.	
	C6H	CYCLES_INT_ MASKED	00H	Number of processor cycles for which interrupts are disabled.	

Continua.

Continuação.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	C7H	CYCLES_INT_ PENDING_ AND_MASKED	00H	Number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_ RETIRED	00H	Number of branch instructions retired.	
	C5H	BR_MISS_PRED_ RETIRED	00H	Number of mispredicted branches retired.	
	C9H	BR_TAKEN_ RETIRED	00H	Number of taken branches retired.	
	CAH	BR_MISS_PRED_ TAKEN_RET	00H	Number of taken mispredictions branches retired.	
	E0H	BR_INST_ DECODED	00H	Number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Number of branches that miss the BTB.	
	E4H	BR_BOGUS	00H	Number of bogus branches.	
	E6H	BACLEARS	00H	Number of time BACLEAR is asserted.	
Stalls	A2	RESOURCE_ STALLS	00H	Number of cycles during which there are resource related stalls.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls.	
Segment Register Loads	06H	SEGMENT_REG_ LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX™ Unit	B0H	MMX_INSTR_ EXEC	00H	Number of MMX Instructions Executed.	Available in Pentium® II processor only.
	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX Saturating Instructions Executed.	Available in Pentium II processor only.

Continua.

Continuação.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	B2H	MMX_UOPS_EXEC	0FH	Number of MMX UOPS Executed.	Available in Pentium II processor only.
	B3H	MMX_INSTR_ TYPE_EXEC	01H 02H 04H 08H 10H 20H	MMX packed multiply instructions executed. MMX packed shift instructions executed. MMX pack operation instructions executed. MMX unpack operation instructions executed. MMX packed logical instructions executed. MMX packed arithmetic instructions executed.	Available in Pentium II processor only.
	CCH	FP_MMX_TRANS	00H 01H	Transitions from MMX instruction to floating-point instructions. Transitions from floating-point instructions to MMX instructions.	Available in Pentium II processor only.
	CDH	MMX_ASSIST	00H	Number of MMX Assists (that is, the number of EMMS instructions executed).	Available in Pentium II processor only.
	CEH	MMX_INSTR_RET	00H	Number of MMX Instructions Retired.	Available in Pentium II processor only.
Segment Register Renaming	D4H	SEG_RENAME_ STALLS	 01H 02H 04H 08H 0FH	Number of Segment Register Renaming Stalls: Segment register ES Segment register DS Segment register FS Segment register FS Segment registers ES + DS + FS + GS	Available in Pentium II processor only.

Conclusão

APÊNDICE D

Listagem do módulo de acesso aos MSR

```
#ifndef __KERNEL__
#define __KERNEL__
#endif


#include <linux/config.h>
#include <linux/module.h>
#include <asm/uaccess.h>
#include <linux/miscdevice.h>
#include <linux/smp.h>
#include <asm/system.h>
#include <linux/kernel.h>
#include <asm/processor.h>



#define MSR_MINOR 142

/* Permite apenas uma instancia */

static int is_open = 0;

static int msr_open(struct inode *finode, struct file *file)
{
	if(is_open)
		return -EBUSY;
#ifdef  MODULE
	MOD_INC_USE_COUNT;
#endif
	is_open = !0;
	return  0;
}

static int msr_close(struct inode *inode, struct file *file)
{
#ifdef  MODULE
	MOD_DEC_USE_COUNT;
#endif
	return is_open = 0;
}
```

```
/* leitura de 8 bytes */
static ssize_t msr_read(struct file *file, char *buf, size_t count,
                                        loff_t *dummy)
{
        unsigned long int p=file->f_pos,hi,lo;//,pn;

        if(count<8)
                return  0;
        __asm__ __volatile__ (
                "rdmsr"
                :"=a" (lo), "=d" (hi)
                :C (p)
                :"eax","ecx","edx");

        if(__copy_to_user(buf,&lo,4))
                return -EFAULT;
        if(__copy_to_user(buf+4,&hi,4))
                return -EFAULT;

}

/* escreve 8 bytes */

static ssize_t msr_write(struct file *file, const char *buf, size_t count,
                                        loff_t *dummy)
{
        unsigned long p=file->f_pos,hi,lo;

        if(count<8)
                return  0;

        if(__copy_from_user(&lo,buf,4))
                return -EFAULT;
        if(__copy_from_user(&hi,buf+4,4))
                return -EFAULT;

        __asm__ __volatile__ (
                "wrmsr"
                :
                :C (p), "a" (lo), "d" (hi)
                :"eax","ecx","edx");
        return 8;
}
```

```
static loff_t msr_seek(struct file *file, loff_t offset, int origin)
{
	switch (origin) {
		case 0:  /* SEEK_SET */
			file->f_pos=offset;
			return file->f_pos;
		case 1:  /* SEEK_CUR */
			file->f_pos+=offset;
			return file->f_pos;
		default:
			return -EINVAL;
	}

}

static struct file_operations msr_fops = {
	msr_seek,
	msr_read,
	msr_write,
	NULL,		//readdir
	NULL,		//poll
	NULL,		//ioctl
	NULL,		//mmap
	msr_open,
	NULL,		//flush
	msr_close,
	NULL,		//fsync
	NULL,		//fasync
	NULL,		//check_media_change
	NULL,		//revalidate
	NULL		//lock
};

static struct miscdevice msr_device = {
	MSR_MINOR,
	"msr",
	&msr_fops
};

#ifdef MODULE

MODULE_AUTHOR("Stephan Meyer <Stephan.Meyer@pobox.com>");
MODULE_DESCRIPTION("Acesso aos MSR");
```

```
int init_module(void)
{
	misc_register(&msr_device);
	printk("<0>MSR: Instalado!\n");

	return 0;
}

void cleanup_module(void)
{
	misc_deregister(&msr_device);
	printk("<0>MSR: Retirado!\n");
}

#else

void msr_init (void)
{
	if ((!have_cpuid) || (!(x86_capability & 32))) {
		printk("<0>MSR: Nao possui MSR !\n");
		return;
	}
	printk("<0>MSR: instalado!.\n");
	misc_register(&msr_device);
}

#endif
```

APÊNDICE E

Listagem das Rotinas de Instrumentação

```
/****************************************/
/*          Mestrado CAP                */
/*   Rotinas para medidas de Performance */
/*   Utilizando contadores de hardware   */
/*                                      */
/*          10-12-99                    */
/*                                      */
/*  Chamadas em C e Fortran :           */
/*          iniciaregistros()           */
/*          leregistros()               */
/*                                      */
/****************************************/


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>


        void zerarcontadores(void);
        long long marcatempo(void);
        int  cpuclock(void);
        void iniciaregistros( int , int);
        void iniciaregistros_( int * , int*);
        long leregistros(int *, int *);
        long leregistros_(int *, int *);
        int  checkcpuid(void);

        static int arquivoMSR, tipocpu;
        static long long tempo_0,tempo_1;
        static unsigned long contador_0, contador_1;
        static unsigned int cesr_0, cesr_1;

        char *driver = {"/dev/msr"};
```

```
//------------------------------------------
// Nome: marcatempo
// Funcao: Executa leitura do TSC
// Descricao: Marca posicao do clock no TSC
// Entra: Nada
// Retorna: retorna valor 64 bits do TSC
// Data: 3-8-99
//------------------------------------------

long long marcatempo(void)
{
        unsigned int loword, hiword;
        long long contador;

        __asm__ __volatile__(        "rdtsc"
                                : "=a" (loword), "=d" (hiword)
                                :/* sem entrada */
                                : "eax", "edx");

        contador = hiword;
        contador <<= 32;
        contador ^= loword;

        return contador;
}


//-----------------------------------------------------------------
// Nome: zerarcontadores.c
// Funcao: zerar MSR counter_0 e counter_1
// Descricao: chama device e escreve zero nos MSR de contagem eventos
// Entra: nada
// Retorna: nada
// Data: 6-8-99
//-----------------------------------------------------------------

void zerarcontadores(void)
{
  int f;
  long zero = 0;

  f=open(driver,O_RDWR);//abre o device para leitura e escrita

  lseek(f,0x12,SEEK_SET);// seleciona contador 0
  write(f,&zero,8);
  lseek(f,0x13,SEEK_SET);//seleciona contador 1
```

```
   write(f,&zero,8);

   close(f);

}

//------------------------------------------
// Nome: cpuClock
// Funcao: verifica o clock da CPU
// Descricao: utiliza TSC para calcular o clock da cpu
// Entra: Nada
// Retorna: valor do clock da CPU
// Data: 3-8-99
//------------------------------------------

int cpuclock(void)
{
        unsigned long  clock_cpu;
        int f,SECS=1;
        long long t_0,t_1;

        t_0 = marcatempo();
        sleep(SECS);
        t_1 = marcatempo();


        clock_cpu=(t_1 - t_0 + 500000)/(SECS*1000000);// em MHZ
     return clock_cpu;

}

//------------------------------------------
// Nome: iniciaregistros chamada em C
// Funcao: inicializa a aquisicao de eventos
// Descricao: chama as funcoes para setar MSR
// Entra: eventos a monitorar
// Retorna: nada
// Data: 23-09-99
//------------------------------------------

void iniciaregistros( int r_0, int r_1)
{


   long CESR,base;
   int hab_contar=0x3;
```

```
  unsigned int selc_0, selc_1, ctr0, ctr1;

  tipocpu = checkcpuid();

  arquivoMSR=open(driver,O_RDWR);//abre o device para leitura e escrita

  if (tipocpu == 5){

   CESR = hab_contar << 22 | r_1 << 16 | hab_contar << 6 |r_0;
   cesr_0 = r_0;
   cesr_1 = r_1;
   ctr0 = 0x12;
   ctr1 = 0x13;
   zerarcontadores();

   lseek(arquivoMSR,0x11,SEEK_SET);//seliciona CESR
   write(arquivoMSR,&CESR,8);//escreve em CESR
  }

  if (tipocpu == 6){
   selc_0 = r_0;
   selc_1 = r_1;
   ctr0 = 0xC1;
   ctr1 = 0xC2;
   base = 0x43000;
   cesr_0 = (base << 4)^selc_0;
   cesr_1 = (base << 4)^selc_1;

   lseek(arquivoMSR,0x187,SEEK_SET);
   write(arquivoMSR,&cesr_1,8);
   lseek(arquivoMSR,0x186,SEEK_SET);
   write(arquivoMSR,&cesr_0,8);
  }

  lseek(arquivoMSR,ctr0,SEEK_SET);// seleciona contador 0
  read(arquivoMSR,&contador_0,8);//le contador 0

  lseek(arquivoMSR,ctr1,SEEK_SET);//seleciona contador 1
  read(arquivoMSR,&contador_1,8);//le contador 1


  close(arquivoMSR);

  tempo_0 = marcatempo();

}
```

```
//------------------------------------------
// Nome: iniciaregistros chamada em Fortran
// Funcao: inicializa a aquisicao de eventos
// Descricao: chama as funcoes para setar MSR
// Entra: eventos a monitorar
// Retorna: nada
// Data: 23-09-99
//------------------------------------------

void iniciaregistros_( int *r_0, int *r_1)
{
  long CESR,base;
  int hab_contar=0x3, tp_r0, tp_r1;
  unsigned int selc_0, selc_1, ctr0, ctr1;

  tp_r0 = *r_0;
  tp_r1 = *r_1;
  tipocpu = checkcpuid();

  arquivoMSR=open(driver,O_RDWR);//abre o device para leitura e escrita

  if (tipocpu == 5){

   CESR = hab_contar << 22 | tp_r1 << 16 | hab_contar << 6 |tp_r0;
   cesr_0 = *r_0;
   cesr_1 = *r_1;
   ctr0 = 0x12;
   ctr1 = 0x13;
   zerarcontadores();

   lseek(arquivoMSR,0x11,SEEK_SET);//seliciona CESR
   write(arquivoMSR,&CESR,8);//escreve em CESR
  }

  if (tipocpu == 6){
   selc_0 = *r_0;
   selc_1 = *r_1;
   ctr0 = 0xC1;
   ctr1 = 0xC2;
   base = 0x43000;
   cesr_0 = (base << 4)^selc_0;
   cesr_1 = (base << 4)^selc_1;

   lseek(arquivoMSR,0x187,SEEK_SET);
   write(arquivoMSR,&cesr_1,8);
   lseek(arquivoMSR,0x186,SEEK_SET);
```

```
    write(arquivoMSR,&cesr_0,8);
   }

   lseek(arquivoMSR,ctr0,SEEK_SET);// seleciona contador 0
   read(arquivoMSR,&contador_0,8);//le contador 0

   lseek(arquivoMSR,ctr1,SEEK_SET);//seleciona contador 1
   read(arquivoMSR,&contador_1,8);//le contador 1

   close(arquivoMSR);

   tempo_0 = marcatempo();


}


//-------------------------------------------
// Nome: leregistros chamada em C
// Funcao: leitura final dos contadores
// Descricao: ler contadores MSR MSR
// Entra: nada
// Retorna: valores dos contadores MSR
//            e      tempo de execucao
// Data: 23-09-99
//-------------------------------------------

long leregistros(int *counter_0, int *counter_1)
{
   int cpu;
   long deltatempo;
   long CESR,base;
   int hab_contar=0x3;
   unsigned long c_0_i, c_1_i;
   unsigned int dif_0,dif_1;
   unsigned int selc_0, selc_1, ctr0, ctr1;

   tempo_1 = marcatempo();


   c_0_i = contador_0;
   c_1_i = contador_1;
   arquivoMSR=open(driver,O_RDWR);//abre o device para leitura e escrita

   if (tipocpu == 5){
```

```
   CESR = hab_contar << 22 | cesr_1 << 16 | hab_contar << 6 |cesr_0;
   ctr0 = 0x12;
   ctr1 = 0x13;
   zerarcontadores();

   lseek(arquivoMSR,0x11,SEEK_SET);//seliciona CESR
   write(arquivoMSR,&CESR,8);//escreve em CESR
  }

  if (tipocpu == 6){
   selc_0 = cesr_0;
   selc_1 = cesr_1;
   ctr0 = 0xC1;
   ctr1 = 0xC2;
   base = 0x43000;
   cesr_0 = (base << 4)^selc_0;
   cesr_1 = (base << 4)^selc_1;

   lseek(arquivoMSR,0x187,SEEK_SET);
   write(arquivoMSR,&cesr_1,8);
   lseek(arquivoMSR,0x186,SEEK_SET);
   write(arquivoMSR,&cesr_0,8);
  }

  lseek(arquivoMSR,ctr0,SEEK_SET);// seleciona contador 0
  read(arquivoMSR,&contador_0,8);//le contador 0

  lseek(arquivoMSR,ctr1,SEEK_SET);//seleciona contador 1
  read(arquivoMSR,&contador_1,8);//le contador 1

  close(arquivoMSR);



 dif_0 = contador_0 - c_0_i;
 dif_1 = contador_1 - c_1_i;

 cpu = cpuclock();
 deltatempo = (tempo_1 - tempo_0)/cpu; //microsec
 *counter_0 = dif_0;
 *counter_1 = dif_1;

 return deltatempo;
}

//------------------------------------------
```

```
// Nome: leregistros chamada em Fortran
// Funcao: leitura final dos contadores
// Descricao: ler contadores MSR MSR
// Entra: nada
// Retorna: valores dos contadores MSR
//            e      tempo de execucao
// Data: 23-09-99
//------------------------------------------


long leregistros_(int *counter_0, int *counter_1)
{

  int cpu;
  long deltatempo;
  long CESR,base;
  int hab_contar=0x3;
  unsigned long c_0_i, c_1_i;
  unsigned int dif_0,dif_1;
  unsigned int selc_0, selc_1, ctr0, ctr1;

  tempo_1 = marcatempo();


  c_0_i = contador_0;
  c_1_i = contador_1;


  arquivoMSR=open(driver,O_RDWR);//abre o device para leitura e escrita

  if (tipocpu == 5){

   CESR = hab_contar << 22 | cesr_1 << 16 | hab_contar << 6 |cesr_0;
   ctr0 = 0x12;
   ctr1 = 0x13;
   zerarcontadores();

   lseek(arquivoMSR,0x11,SEEK_SET);//seliciona CESR
   write(arquivoMSR,&CESR,8);//escreve em CESR
  }

  if (tipocpu == 6){
   selc_0 = cesr_0;
   selc_1 = cesr_1;
   ctr0 = 0xC1;
   ctr1 = 0xC2;
```

```
    base = 0x43000;
    cesr_0 = (base << 4)^selc_0;
    cesr_1 = (base << 4)^selc_1;

    lseek(arquivoMSR,0x187,SEEK_SET);
    write(arquivoMSR,&cesr_1,8);
    lseek(arquivoMSR,0x186,SEEK_SET);
    write(arquivoMSR,&cesr_0,8);
   }

   lseek(arquivoMSR,ctr0,SEEK_SET);// seleciona contador 0
   read(arquivoMSR,&contador_0,8);//le contador 0

   lseek(arquivoMSR,ctr1,SEEK_SET);//seleciona contador 1
   read(arquivoMSR,&contador_1,8);//le contador 1

   close(arquivoMSR);



  dif_0 = contador_0 - c_0_i;
  dif_1 = contador_1 - c_1_i;

  cpu = cpuclock();
  deltatempo = (tempo_1 - tempo_0)/cpu; //microsec
  *counter_0 = dif_0;
  *counter_1 = dif_1;

  return deltatempo;

}

//-------------------------------------------
// Nome: cpuid
// Funcao: identifica modelo da cpu
// Descricao: executa instrucao cpuid
// Entra: nada
// Retorna: id da cpu - 5 ou 6
//
// Data: 23-09-99
//-------------------------------------------

int checkcpuid(void)
{
    int ptype, eax;
    __asm__ ( "cpuid"
```

```
            : "=a" (eax)
            : "a" (0x00)
            : "eax");

   if (eax == 1 || eax == 2)
   {
     if (eax == 1) {ptype=5;}
     if (eax == 2) {ptype=6;}
   }
   else

     ptype=0;


   return ptype;
}
```

APÊNDICE F

Listagem Original do Programa “Correlação MASCO”

```
c           CORRELATION PROGRAM FOR FOR THE MASCO PROJECT
c                Joao Braga - 3-18-92
c     version that includes built-in psf-convolved MURA patterns
c                (8-5-95 in Boulder)
c     modified on 2-7-97 to include decoding function for
c        mask-antimask subtracted distribution (Byard, Exp. Astr. 1992)


c     input file :  file with binned positions of events on detector
c     output files :  corr_masco.out (correlated ASCII array)
c                   corr_masco_b.out (correlated binary array)
c     subroutines :  mura_pattern (generates G function for MURA pattern)
c                   mura_psf (generates psf-convolved G MURA pattern)

      program corr_masco

c  >> declarations
      external leclock
      character*60 arg(7), hora*8
      integer g(200,200),count(500,500)
      integer r,res,psf,p,an,bn,rperms,cperms,ntempo,mtempo
      real energy, elapsed, t0, t1, t2, t3, t4, tempo1
      real tempx(500),tempy(500),tmpx(500),tmpy(500)
      real arr(500,500),prod(500,500),corr(0:500,0:500)

      double precision t_ini, t_fim

c  >> initialize time

      an=0
      bn=0
      t0=0.0
      t1=secnds(t0)
      tempo1=t1
c**** INPUT ***********************************************************

c  >> get number of arguments

      narg=Iargc()

      if (narg.lt.7) then
        print *,'Usage: corr_masco fl dim res mk psf energy exfl'
        print *,'fl -> name of file with array to be correlated'
        print *,'dim -> dimension of MURA pattern used'
        print *,'res -> binning factor (1,2,3...)'
        print *,'mk -> observation with mask(0) or mask-antimask(1)'
```

```
        print *,'psf -> normal(0), psf-convolved(1) or ',
     c          'external(2) mask pattern'
        print *,'energy -> mean energy of photons ',
     c          '(0 for normal or external mask)'
        print *,'exfl -> name of file with external G-function array',
     c          ' (0 for none)'
        stop
      endif


c  >> read arguments
      do i=1,narg
        call getarg(i,arg(i))
      end do

c  >> MURA dimension
      read(arg(2),*) r

c  >> division factor of detector bin relative to mask cell size
      read (arg(3),*) res

c  >> mask or antimask
      read (arg(4),*) mk

c  >> normal mask, PSF-convolved mask or external mask
      read(arg(5),*) psf

c  >> energy to determine detector spatial resolution
      read(arg(6),*) energy

      if (psf.ne.2) then
         print *,'CREATING MURA DECODING FUCTION G...'
c  >> calling MURA decoding function (G)

         call mura_pattern(r,g)

         print *,' -> done.'
      end if

c  >> if it's a mask-antimask observation, change one G value

      if (mk.eq.1) g(1,1)=0

c  >> reading in the file with the BINNED positions
      open (1,file=arg(1))
      print *, 'READING POSITION-BINNED EVENTS...'
```

```
      do i=1,r*res
          read(1,*) (count(i,j),j=1,r*res)
      end do
      close(1)

      print *,' -> done.'

c  >> creating expanded mask (actually G-function) array

      if(psf.ne.2) then
        if(res.gt.1) print *,'EXPANDING (SUBDIVIDING) DECODING ',
     c                   'FUNCTION ARRAY...'

        do i=res,res*r,res
          do j=res,res*r,res
            do k=0,res-1
              do l=0,res-1
                arr(i-k,j-l)=g(i/res,j/res)
              end do
            end do
          end do
        end do
        if (res.gt.1) print *,' -> done.'

c         open(8,file='array_bin.out',form='unformatted')


        open(8,file='array.out')
        do i=1,r*res
c           write(*,20) (arr(i,j),j=1,r*res)
c           write(8) (arr(i,j),j=1,r*res)
          write(8,*) (arr(i,j),j=1,r*res)
        end do
        close(8)


      end if
c20   format(40(f6.3,1x))

c  >> creating new expanded mask pattern (actually G function)
c  >> convolved with detector point spread function
      if (psf.eq.1) then
c------------------------
c      call leclock(t_ini)
          call mura_psf(r,res,arr,energy)
c      call leclock(t_fim)
```

```
c      print *,'t_fim - t_ini = ',t_fim - t_ini
c-------------------------
       open(9,file='psf_mask.out')
       do i=1,r*res
         write(9,*) (arr(i,j),j=1,r*res)
c          write(*,10) (arr(i,j),j=1,r*res)
       end do
       close(9)
       print *,'psf-convolved MURA pattern stored on psf_mask.out'
      end if
c10   format(40(f6.3,1x))

c  >> reading in external file with G function to be used

      if (psf.eq.2) then
        open (10,file=arg(7))
        print *, 'READING EXTERNAL MASK PATTERN (ACTUALLY ',
     c           'G DECODING FUNCTION)...'
        do i=1,r*res
          read(10,*) (arr(i,j),j=1,r*res)
        end do
        close(10)
        print *,' -> done.'
      end if

c  >> adding int((r*res+1)/2) permutations in the x and y directions
c  >> so that the image will be centered for a central source
      an=an+(r*res+1)/2
      bn=bn+(r*res+1)/2

c    >> permutation of the array by one row
c    >> n-th row to (n+1)-th row
c    >> (done an times)

      print *,'PERMUTING DECODING ARRAY TO CENTER IMAGE...'


      do m=1,an
         do i=1,r*res
           tmpx(i)=arr(r*res,i)
         end do
         do k=r*res,2,-1
           do j=1,r*res
             arr(k,j)=arr(k-1,j)
           end do
         end do
```

```
          do i=1,r*res
            arr(1,i)=tmpx(i)
          end do
      end do


c    >> permutation of the array by one column
c    >> n-th column to (n+1)-th column
c    >> (done bn times)

      do m=1,bn
          do i=1,r*res
            tmpy(i)=arr(i,r*res)
          end do
          do k=r*res,2,-1
            do j=1,r*res
              arr(j,k)=arr(j,k-1)
            end do
          end do
          do i=1,r*res
            arr(i,1)=tmpy(i)
          end do
      end do

      print *,' -> done.'

c****  CORRELATING DETECTOR ARRAY WITH MASK (G-FUNCTION)
PATTERN ******

c    >> calculating the sum of all the elements of each product
c    >>  array through every possible permutation.
c    >> product = (extended) decoding function (G) value (1 or -1)
c    >> times number of photon counts on the detector square bin

      print *,'PERFORMING CORRELATION WITH DETECTOR DISTRIBUTION ',
     c         'OF COUNTS:'

      call time(hora)
      write(*,"(' The time now is : ', A8 )" ) hora

      print *,'CORRELATION IN PROGRESS...'

      t3=secnds(t0)
      do 680 j=0,r*res-1
          cperms=j
        if (j.eq.1) then
```

```
          t4=secnds(t3)
          print *,'each step takes ', t4,' seconds'
          print *,'expected correlation time : ', t4*r*res,
     c           ' sec (', t4*r*res/3600.,' hours )'
        end if
        print *,'STEP #',j+1,' OF ',r*res
        t2=secnds(t0)
        if (t2.le.tempo1) then
            if (ntempo.eq.0) mtempo=mtempo+1
            ntempo=1
          else
            ntempo=0
        end if

          do 650 i=0,r*res-1
          rperms=i
             corr(rperms,cperms)=0.

             do k=1,res*r
               do l=1,res*r
              prod(k,l)=arr(k,l)*count(k,l)
               end do
             end do

          do k=1,res*r
            do l=1,res*r
              corr(rperms,cperms)=corr(rperms,cperms)+prod(k,l)
            end do
          end do

c         if (cperms.eq.0) print *,'corr = ', corr(rperms,cperms)

c    >> permutation of the mask array by one row
c    >> n-th row to (n+1)-th row

             do 622 p=1,res*r
           tempx(p)=arr(res*r,p)
 622         continue
             do 630 m=res*r,2,-1
               do 628 n=1,res*r
                 arr(m,n)=arr(m-1,n)
 628           continue
 630         continue
             do 636 p=1,res*r
               arr(1,p)=tempx(p)
 636         continue
```

```
 650    continue

c    >> permutation of the mask array by one column
c    >> n-th column to (n+1)-th colum

        do 652 p=1,res*r
          tempy(p)=arr(p,res*r)
 652    continue
        do 660 n=r*res,2,-1
          do 658 m=1,res*r
            arr(m,n)=arr(m,n-1)
 658      continue
 660    continue
        do 666 p=1,res*r
          arr(p,1)=tempy(p)
 666    continue

 680  continue

      print *,' -> done.'

c****************** OUTPUT **********************************

c     >> storing the correlated image

      open (2,file='corr_masco.out')
      open (3,file='corr_masco_bin.out',form='unformatted')

      do i=0,r*res-1
c         write (2,*) (idnint(corr(i,j)), j=0,r*res-1)
          write (2,*) (nint(corr(i,j)), j=0,r*res-1)
c         write (3) (idnint(corr(i,j)), j=0,r*res-1)
          write (3) (nint(corr(i,j)), j=0,r*res-1)
      end do

      close(2)
      close(3)

      if(psf.ne.2) then
        if (res.gt.1) then
          print *,'expanded MURA decoding array stored in ',
c    c              'array_bin.out (unformatted)'
     c              'array.out'
         else
          print *,'MURA decoding array stored in array.out'
```

```
        end if
      end if

      print *,'correlated ASCII image stored in file',
     c         ' corr_masco.out'
      print *,'correlated binary image stored in file',
     c         ' corr_masco_bin.out'

c     >> calculate the running time

      print *,'number of midnight crossings = ', mtempo
      call time(hora)
      write(*,"(' The time now is : ', A8 )" ) hora
      elapsed = secnds(tempo1)+86400.0*mtempo
      print *,'running time : ', elapsed, ' seconds  ( ',
     c          elapsed/3600.,'  hours )'

      end
```

APÊNDICE G

Listagem HPF do Programa “Correlação MASCO”

```
!------------------------------------------------------------------------
! Versao HPF - Ricardo 26/2/00
!------------------------------------------------------------------------
!          CORRELATION PROGRAM FOR FOR THE MASCO PROJECT
!              Joao Braga - 3-18-92
!    version that includes built-in psf-convolved MURA patterns
!              (8-5-95 in Boulder)
!    modified on 2-7-97 to include decoding function for
!      mask-antimask subtracted distribution (Byard, Exp. Astr. 1992)


!    input file :  file with binned positions of events on detector
!    output files :  corr_masco.out (correlated ASCII array)
!                corr_masco_b.out (correlated binary array)
!    subroutines :  mura_pattern (generates G function for MURA pattern)
!                mura_psf (generates psf-convolved G MURA pattern)
!-------------------------------------------------------------------------

PROGRAM corr_masco

!  >> declarations

     IMPLICIT NONE

!---- prototipos ------------------------------

     INTERFACE
      SUBROUTINE mura_psf(r,res,arr,energy)
        INTEGER,INTENT(IN)                      :: r, res
        REAL,DIMENSION(r*res,r*res),INTENT(INOUT)  :: arr
        REAL,INTENT(IN)                        :: energy
      END SUBROUTINE mura_psf

      SUBROUTINE mura_pattern(r,g)
        INTEGER,INTENT(IN)                 :: r
        INTEGER, DIMENSION(r,r),INTENT(OUT) :: g
      END SUBROUTINE mura_pattern
    END INTERFACE

!---- fim prototipos -------------------------

!---- declaracao variaveis --------------------


     INTEGER, PARAMETER                    ::  narg = 7
     CHARACTER(LEN=60), DIMENSION(narg)     ::  arg
```

```
CHARACTER(LEN=8)                  ::  hora

INTEGER,ALLOCATABLE, DIMENSION(:,:) ::  g
INTEGER, ALLOCATABLE,DIMENSION(:,:) ::  count

INTEGER               ::          r,      &
                                  res,    &
                                   psf,   &
                                  p,      &
                                  an,     &
                                  bn,     &
                                  rperms, &
                                  cperms

REAL                  ::          energy

REAL(KIND=8)          ::          t0,     &
                                  t1,     &
                                  t2,     &
                                  t3,     &
                                  t4,     &
                                  elapsed


REAL,ALLOCATABLE, DIMENSION(:)   ::  tempx,  &
                                     tempy,  &
                                     tmpx,   &
                                     tmpy
REAL,ALLOCATABLE,DIMENSION(:,:)  ::  t_arr
REAL,ALLOCATABLE, DIMENSION(:,:) ::          arr,    &
                                     prod,   &
                                     corr

INTEGER    ::    i, j, k, l, &
                 mk, m, n

INTEGER    ::    lim

!---monitoracao MSR ------

INTEGER  ::  cesr_0, cesr_1, cont_0, cont_1

!----- fim declaracao variaveis ---------------------
```

```
!--- diretivas hpf ---------------------

!HPF$                                                    PROCESSORS,
DIMENSION(number_of_processors()/2,number_of_processors()/2) :: P
!HPF$ PROCESSORS NUM_PROC(number_of_processors())
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: prod,corr


!HPF$ PROCESSORS,DIMENSION(number_of_processors()) :: T
   REAL(KIND=8), DIMENSION(number_of_processors()) :: tempo,tempo_step
   INTEGER,DIMENSION(number_of_processors()) :: ct_0,ct_1,t_msr
!HPF$ DISTRIBUTE(BLOCK) ONTO T :: tempo,t_msr,tempo_step,ct_0,ct_1


!---fim diretivas hpf ------------------

!  >> initialize time

    call leclock(t1)

!**** INPUT ********************************************************

    arg(1) = 'mura_43_res5.out'
    arg(2) = '43'
    arg(3) = '5'

!     arg(1) = 'mura_17x17'
!     arg(2) = '17'
!     arg(3) = '1'

    arg(4) = '0'
    arg(5) = '0'
    arg(6) = '0'
    arg(7) = '0'

    PRINT *,'input file: ', arg(1)

    read(arg(2),*) r
    PRINT *,'MURA dimension: r = ', r

!  >> division factor of detector bin relative to mask cell size : res

    read (arg(3),*) res
    PRINT *,'energy to determine detector spatial resolution: energy = ', res

     read (arg(4),*) mk
```

```
      PRINT *,' mask or antimask : mk = ', mk

!  >> normal mask, PSF-convolved mask or external mask : psf

       read(arg(5),*) psf
      PRINT *,' normal mask, PSF-convolved mask or external mask : psf = ',psf

!  >> energy to determine detector spatial resolution: energy

       read(arg(6),*) energy
      PRINT *,'energy to determine detector spatial resolution: energy = ', energy


!---- alocacao matrizes ----------

      ALLOCATE( g(r,r),count(r*res,r*res) )
      ALLOCATE( tempx(r*res),tempy(r*res),tmpx(r*res),tmpy(r*res))
      ALLOCATE( arr(r*res,r*res), prod(r*res,r*res),corr(0:r*res,0:r*res))
      ALLOCATE( t_arr(r*res,r*res))

!---- fim alocacao matrizes ------

      if (psf.ne.2) then
        print *,'CREATING MURA DECODING FUCTION G...'
!  >> calling MURA decoding function (G)
        call mura_pattern(r,g)
        print *,' -> done.'
      end if


!  >> if it's a mask-antimask observation, change one G value

      if (mk.eq.1) g(1,1)=0

!  >> reading in the file with the BINNED positions

      open (UNIT=1,file=arg(1),FORM="formatted")
      print *, 'READING POSITION-BINNED EVENTS...'
      read (UNIT=1,FMT=*) count(1:r*res,1:r*res)
      close(UNIT=1)

      print *,' -> done.'

!  >> creating expanded mask (actually G-function) array

      if(psf.ne.2) then
```

```
       if(res.gt.1) print *,'EXPANDING (SUBDIVIDING) DECODING ',&
                     'FUNCTION ARRAY...'

!-----------------------------------------------------------------
      FORALL ( i= res:res*r:res,j=res:res*r:res,k=0:res-1,l=0:res-1)
      arr(i-k,j-l)=g(i/res,j/res)
      END FORALL
!-----------------------------------------------------------------
       if (res.gt.1) print *,' -> done.'

       open(UNIT=8,file="array.out",FORM="unformatted")
         write(UNIT=8) (arr(1:r*res,j),j=1,r*res)
       close(UNIT=8)
     end if


!  >> creating new expanded mask pattern (actually G function)
!  >> convolved with detector point spread function

     if (psf.eq.1) then
         call mura_psf(r,res,arr,energy)
      open(UNIT=9,FILE='psf_mask.out')
      do i=1,r*res
        write(9,*) (arr(i,j),j=1,r*res)
      end do
      close(UNIT=9)

      print *,'psf-convolved MURA pattern stored on psf_mask.out'
     end if

!  >> reading in external file with G function to be used

     if (psf.eq.2) then
       open (UNIT=10,file=arg(7))
       print *, 'READING EXTERNAL MASK PATTERN (ACTUALLY ', &
              'G DECODING FUNCTION)...'
       do i=1,r*res
         read(10,*) (arr(i,j),j=1,r*res)
       end do
       close(UNIT=10)
       print *,' -> done.'
     end if

!  >> adding int((r*res+1)/2) permutations in the x and y directions
!  >> so that the image will be centered for a central source
     an=0
```

```
      bn=0
      an=an+(r*res+1)/2
      bn=bn+(r*res+1)/2

!    >> permutation of the array by one row
!    >> n-th row to (n+1)-th row
!    >> (done an times)

      print *,'PERMUTING DECODING ARRAY TO CENTER IMAGE...'

!---------------------------------

      !HPF$ INDEPENDENT
      DO  m=1, an
         arr = CSHIFT(arr,1,1)
      END DO

!------------------------------------

!    >> permutation of the array by one column
!    >> n-th column to (n+1)-th column
!    >> (done bn times)

      !HPF$ INDEPENDENT
      DO m=1,bn
         arr = CSHIFT(arr,1,2)
      END DO

      print *,' -> done.'

!**** CORRELATING DETECTOR ARRAY WITH MASK (G-FUNCTION)
PATTERN ******

!    >> calculating the sum of all the elements of each product
!    >>  array through every possible permutation.
!    >> product = (extended) decoding function (G) value (1 or -1)
!    >> times number of photon counts on the detector square bin

      print *,'PERFORMING CORRELATION WITH DETECTOR DISTRIBUTION ',
&
            'OF COUNTS:'
      call time(hora)
      write(*,"(' The time now is : ', A8 )" ) hora

      print *,'CORRELATION IN PROGRESS...'
!---------------------
```

```
      call leclock(t3)
!---------------------
      lim = r*res

      DO j=0,lim-1
           cperms=j
        if (j.eq.1) then
!          -----------------
           call leclock(t4)
!          -----------------
           tempo_step = t4-t3
           print *,'each   step takes = ', tempo_step
           print *,'correlation time = ', tempo_step*(lim-1)*(lim-1)
        end if
        print *,'STEP #',j+1,' OF ',r*res

!------------
cesr_0 = z'10'
cesr_1 = z'D0'
!-----------


        !HPF$ INDEPENDENT
           DO i=0,lim-1
!--------------------------------
call iniciaregistros(cesr_0,cesr_1)
!--------------------------------
          rperms=i
!          !HPF INDEPENDENT
             corr(rperms,cperms)=0.
             !HPF$ INDEPENDENT
          corr(rperms,cperms)=corr(rperms,cperms)+SUM(
arr(1:lim,1:lim)*count(1:lim,1:lim)   )
!         if (cperms.eq.0) print *,'corr = ', corr(rperms,cperms)

!    >> permutation of the mask array by one row
!    >>  n-th row to (n+1)-th row

!---------------------------------
t_msr=leregistros(cont_0,cont_1)
ct_0 = cont_0
ct_1 = cont_1
!---------------------------------


!HPF$ INDEPENDENT
```

```
        tempx(1:lim) = arr(lim,1:lim)
!HPF$ INDEPENDENT
        arr(lim:2:-1,1:lim)=arr(lim-1:1:-1,1:lim)
!HPF$ INDEPENDENT
        arr(1,1:lim)=tempx(1:lim)

!         t_arr = CSHIFT(arr,1,1)
!         arr = t_arr

          END DO

!    >> permutation of the mask array by one column
!    >>  n-th column to (n+1)-th colum

       tempy(1:lim) = arr(1:lim,lim)
       arr(1:lim,lim:2:-1)=arr(1:lim,lim-1:1:-1)
       arr(1:lim,1)=tempy(1:lim)
!        arr = CSHIFT(arr,1,2)

print *,'ct_0 = ',ct_0
print *,'ct_1 = ',ct_1
print *,'tmsr = ', t_msr

      END DO

!call leclock(t4)
!print *,'t=',t4-t3

      print *,' -> done.'

!******************* OUTPUT ************************************

!     >> storing the correlated image

      open (2,file='corr_masco.out')
      open (3,file='corr_masco_bin.out',form='unformatted')

      do i=0,r*res-1
!          write (2,*) (idnint(corr(i,j)), j=0,r*res-1)
           write (2,*) (nint(corr(i,j)), j=0,r*res-1)
!          write (3) (idnint(corr(i,j)), j=0,r*res-1)
           write (3) (nint(corr(i,j)), j=0,r*res-1)
      end do

      close(2)
      close(3)
```

```
      if(psf.ne.2) then
        if (res.gt.1) then
          print *,'expanded MURA decoding array stored in ',&
                'array_bin.out (unformatted)',&
                 'array.out'
        else
          print *,'MURA decoding array stored in array.out'
        end if
      end if

      print *,'correlated ASCII image stored in file',&
             ' corr_masco.out'
      print *,'correlated binary image stored in file',&
             ' corr_masco_bin.out'

!     >> calculate the running time

      call time(hora)
!--------------------
      call leclock(t2)
!---------------------
      write(*,"(' The time now is : ', A8 )" ) hora
!      elapsed = t2-t1
      tempo = t2 - t1
      print *,'tempo : ',tempo

END PROGRAM corr_masco
```

APÊNDICE H

Listagem de PDE1

```
      PROGRAM PDE1
!
================================================================
!  ===                                                        ===
!  ===          GENESIS Distributed Memory Benchmarks         ===
!  ===                                                        ===
!  ===   3-Dimensional Poisson Solver Using Red-Black Relaxation  ===
!  ===             SOR with Chebyshev Acceleration            ===
!  ===                    Subset HPF version                  ===
!  ===                                                        ===
!  ===    Original author:  Max Lemke (1989, PALLAS GmbH)     ===
!  ===            PALLAS GmbH                                 ===
!  ===            Hermulheimer Str. 10                        ===
!  ===            5040 Bruhl, GERMANY                         ===
!  ===    tel.:+49-2232-18960   e-mail:karls@pallas-gmbh.de   ===
!  ===                                                        ===
!  ===         Subset HPF:  Vladimir Getov                    ===
!  ===                   (1993, University of Southampton)    ===
!  ===                                                        ===
!  ===         Last update: May 1993; Release: 2.2            ===
!  ===                                                        ===
!
================================================================
!
!    FOR NUMERICAL VERIFICATION:
!         THE  AVERAGE  CONVERGENCE  RATE  SHOULD  BE  SOMEWHERE
BETWEEN
!         0.7 AND 0.98
!
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!
!      THE  TIMER  (DOUBLE  PRECISION  IN  SECONDS)  IS  CALLED  IN  MAIN
PROGRAM.
!    THE MACHINE DEPENDENT TIMING ROUTINE HAS TO BE INCLUDED.
!
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!
      IMPLICIT NONE
      INTEGER N, NITER, NX, NY, NZ, NPTS, NOP, NW1, NW2
      INTEGER LOGP, LNX, LNY, LNZ, NXPROC, NYPROC, NZPROC
      DOUBLE PRECISION H, RES, RES1, RES2, CONVR, RMFLOP, SEC1, SEC2
!
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

      INCLUDE 'pde1.inc'
      PARAMETER (NX=2**N+2)
```

```
      PARAMETER (NY=NX)
      PARAMETER (NZ=NX)

      PARAMETER (LNX=LOGP/3, NXPROC=2**LNX)
      PARAMETER (LNY=(LOGP-LNX)/2, NYPROC=2**LNY)
      PARAMETER (LNZ=LOGP-LNX-LNY, NZPROC=2**LNZ)

      PARAMETER(NW1=11)
      PARAMETER(NW2=12)

      DOUBLE PRECISION U(NX,NY,NZ),F(NX,NY,NZ),T(NX,NY,NZ)
      DOUBLE PRECISION L2NORM
      LOGICAL RED(NX,NY,NZ)
      INTEGER I,J,K

!     .. Data Distribution ..
!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN T(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM

      H=1.0D0/(NX-1)

      OPEN(NW1,FILE='result')
      CALL HEADER(NW1)
      WRITE(NW1,91000)
91000 FORMAT(' ***** 3D RED BLACK RELAXATION BENCHMARK
************* '/ &
     &   )
!
      CALL INIT(U,F,RED,NX,NY,NZ,H)
C
C The first calculation of RES2 matches the 2.3 F77 version, the 2nd
C matches the 2.3 PARMACs version (which I think is correct - dsm)
C
cpgi  RES1=RES(U,F,RED,NX,NY,NZ,H)
      RES1=L2NORM(U,F,T,RED,NX,NY,NZ,H)
      RES1=RES1*RES1
C
!----------------> TIMER IN SECONDS
      CALL TIMER(SEC1)
      CALL RELAX(U,F,RED,NX,NY,NZ,H,NITER)
      CALL TIMER(SEC2)
!----------------> TIMER IN SECONDS
C
```

```
C The first calculation of RES2 matches the 2.3 F77 version, the 2nd
C matches the 2.3 PARMACs version (which I think is correct - dsm)
C
cpgi  RES2=RES(U,F,RED,NX,NY,NZ,H)
      RES2=L2NORM(U,F,T,RED,NX,NY,NZ,H)
      RES2=RES2*RES2
C
C The first calculation of CONVR matches the 2.3 F77 version, the 2nd
C matches the 2.3 PARMACs version (which I think is correct - dsm)
C
cpgi  CONVR=(RES2/RES1)**(0.1)
      CONVR=(SQRT(RES2/RES1))**(1.0D0/DBLE(NITER))
C
      NPTS = (NX-2)*(NY-2)*(NZ-2)
      WRITE(NW1,9100) NX-2, NY-2, NZ-2, NPTS, NITER, CONVR
      CALL PRTRES(NW2,NX-2,NY-2,NZ-2,NPTS,NITER,CONVR)
!
      NOP = NPTS*10
      RMFLOP = (NOP/1000000.)/(SEC2-SEC1)*NITER

      WRITE(NW1,900) (SEC2-SEC1)/NITER,NOP, RMFLOP
      WRITE(NW1,'(////)')
 9100 FORMAT (/' NUMBER OF INTERNAL POINTS PER DIRECTION = ', 3I12 /
&
   &       ' INTERNAL GRIDSIZE OF CUBIC  GRID =        ', I12/       &
   &       ' AVERAGE CONV. RATE (', I6, ' ITERATIONS)  = ', D12.4/)
  900 FORMAT(/' TIME FOR 1 RELAX (SEC.):', E15.8/                    &
   &         ' NUMBER OF OPERATIONS:   ',I15/                    &
   &         ' PERFORMANCE IN MFLOP/S: ',F15.5/)
!
      CLOSE(NW1)

      STOP
      END
!
!
      SUBROUTINE RELAX(U,F,RED,NX,NY,NZ,H,ITER)
!-------------------------------------------------------C
!    3D RED BLACK RELAXATION STANDARD IMPLEMENTATION    C
!-------------------------------------------------------C

      IMPLICIT NONE
      INTEGER K,J,I
      INTEGER NX, NY, NZ, ITER, NREL
      DOUBLE PRECISION H,HSQ,FACTOR
      DOUBLE PRECISION U(NX,NY,NZ), F(NX,NY,NZ)
```

```
      LOGICAL RED(NX,NY,NZ)
      DOUBLE PRECISION PYE, RADIUS
      DOUBLE PRECISION ROMEGA, RFACTOR, ROMINUS
      DOUBLE PRECISION BOMEGA, BFACTOR, BOMINUS

!     .. Data Distribution ..

!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM

      PYE = 2.0D0 * ACOS(0.0D0)
      RADIUS = COS(PYE/(NX-1))**2
      HSQ = H*H

      DO NREL=1,ITER

       IF (NREL .EQ. 1) THEN
        ROMEGA = 1.0D0
        RFACTOR = ROMEGA/6.0D0
        ROMINUS = 1.0D0 - ROMEGA
        BOMEGA = 1.0D0/(1.0D0 - RADIUS/2.0D0)
        BFACTOR = BOMEGA/6.0D0
        BOMINUS = 1.0D0 - BOMEGA
       ELSE
        ROMEGA = 1.0D0/(1.0D0 - RADIUS*BOMEGA/4.0D0)
        RFACTOR = ROMEGA/6.0D0
        ROMINUS = 1.0D0 - ROMEGA
        BOMEGA = 1.0D0/(1.0D0 - RADIUS*ROMEGA/4.0D0)
        BFACTOR = BOMEGA/6.0D0
        BOMINUS = 1.0D0 - BOMEGA
       ENDIF

!     .. Array Constructs ..

       WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
!
!      RELAXATION OF THE RED POINTS
!
        U(2:NX-1,2:NY-1,2:NZ-1) =                          &
     &         ROMINUS*U(2:NX-1,2:NY-1,2:NZ-1) +            &
     &         RFACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+         &
     &   U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+      &
     &   U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+      &
     &   U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
```

```
      ELSEWHERE
!
!      RELAXATION OF THE BLACK POINTS
!
        U(2:NX-1,2:NY-1,2:NZ-1) =                              &
   &            BOMINUS*U(2:NX-1,2:NY-1,2:NZ-1) +                 &
   &            BFACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+             &
   &   U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+              &
   &   U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+              &
   &   U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

      END WHERE

    ENDDO

    END

    DOUBLE PRECISION FUNCTION RES(U,F,RED,NX,NY,NZ,H)
!
!        COMPUTES THE MAXIMUM NORM OF THE DEFECT
!
    IMPLICIT NONE
    INTEGER NX, NY, NZ
    DOUBLE PRECISION H, FAC
    DOUBLE PRECISION U(NX,NY,NZ), F(NX,NY,NZ)
    LOGICAL RED(NX,NY,NZ)
    INTEGER I,J,K

!    .. Data Distribution ..

!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM

!
    FAC=1/(H*H)

    RES = MAXVAL(ABS(F(2:NX-1,2:NY-1,2:NZ-1)+                  &
   &     FAC*(-6.0*U(2:NX-1,2:NY-1,2:NZ-1)+                  &
   &     U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+          &
   &     U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+          &
   &     U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))))

    END
```

```
      DOUBLE PRECISION FUNCTION L2NORM (U,F,T,RED,NX,NY,NZ,H)
!
!   L2NORM COMPUTES THE L2-NORM OF THE DIFFERENCES (EXACT
SOLUTION -
!  APPROXIMATE SOLUTION). THE EXACT SOLUTION IS GIVEN BY THE
FUNCTION
!  G.
!
      IMPLICIT NONE
      INTEGER NX, NY, NZ, I, J, K
      DOUBLE PRECISION H, FAC, D, G
      DOUBLE PRECISION U(NX,NY,NZ), F(NX,NY,NZ), T(NX,NY,NZ)
      LOGICAL RED(NX,NY,NZ)

!     .. Data Distribution ..

!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN T(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM

!
      D = 0.0D0
CC    DO K = 2, NZ-1
CC      DO J = 2, NY-1
CC        DO I = 2, NX-1
CC          D = D + (U(I,J,K) - G(I*H,J*H,K*H)) ** 2
CC        ENDDO
CC      ENDDO
CC    ENDDO

CC    FORALL (I=2:NX-1,J=2:NY-1,K=2:NZ-1) T(I,J,K) = G(I*H,J*H,K*H)
      FORALL (I=2:NX-1,J=2:NY-1,K=2:NZ-1) T(I,J,K) =
     &      (I*H)**2 + (J*H)**2 + (K*H)**2
      T(2:NX-1,2:NY-1,2:NZ-1) = (U(2:NX-1,2:NY-1,2:NZ-1) -
     &                      T(2:NX-1,2:NY-1,2:NZ-1)) ** 2
      D = SUM(T(2:NX-1,2:NY-1,2:NZ-1))

      L2NORM = SQRT(D)

      END
!
!.......................................................................
!
```

```
      SUBROUTINE INIT(U,F,RED,NX,NY,NZ,H)

      IMPLICIT NONE
      INTEGER NX, NY, NZ, I, J, K, NRED
      DOUBLE PRECISION H, X, Y, Z, G
      DOUBLE PRECISION U(NX,NY,NZ),F(NX,NY,NZ)
      LOGICAL RED(NX,NY,NZ)

!     .. Data Distribution ..

!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN U(I,J,K) WITH RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH RED(I,J,K)
!HPF$ DISTRIBUTE RED(BLOCK,*,*) ONTO FARM

!     INITIALISATION OF BOUNDARY POINTS (X = 0.0 & 1.0)
      X = (NX-1) * H
      DO K = 1, NZ
         Z = (K-1) * H
         DO J = 1, NY
            Y = (J-1) * H
            U(1,J,K) = G(0.0D0,Y,Z)
            U(NX,J,K) = G(X,Y,Z)
         ENDDO
      ENDDO


!     INITIALISATION OF BOUNDARY POINTS (Y = 0.0 & 1.0)
      Y = (NY-1) * H
      DO K = 1, NZ
         Z = (K-1) * H
         DO I = 1, NX
            X = (I-1) * H
            U(I,1,K) = G(X,0.0D0,Z)
            U(I,NY,K) = G(X,Y,Z)
         ENDDO
      ENDDO

!     INITIALISATION OF BOUNDARY POINTS (Z = 0.0 & 1.0)
      Z = (NZ-1) * H
      DO J = 1, NY
         Y = (J-1) * H
         DO I = 1, NX
            X = (I-1) * H
            U(I,J,1) = G(X,Y,0.0D0)
            U(I,J,NZ) = G(X,Y,Z)
```

```
        ENDDO
      ENDDO

!     INITIALISATION OF INNER POINTS

      U(2:NX-1,2:NY-1,2:NZ-1) = 0.
C
C The first definition matches the 2.3 F77 version, the 2nd matches
C the 2.3 PARMACs version
C
cpgi  F(2:NX-1,2:NY-1,2:NZ-1) = 6.
      F(2:NX-1,2:NY-1,2:NZ-1) = -6.0D0
!

!     INITIALISATION OF THE RED-BLACK LOGICAL ARRAY

      RED = .FALSE.
C
C The following matches the s77 version in Genesis 2.3
C
cpgi  RED(2:NX-1:2,2:NY-1:2,2:NZ-1) = .TRUE.
cpgi  RED(3:NX-1:2,3:NY-1:2,2:NZ-1) = .TRUE.
C
C The following matches the d77 version in Genesis 2.3 (I think this
C one's correct - dsm)
C
      RED(2:NX-1:2,2:NY-1:2,2:NZ-1:2) = .TRUE.
      RED(3:NX-1:2,3:NY-1:2,2:NZ-1:2) = .TRUE.
      RED(3:NX-1:2,2:NY-1:2,3:NZ-1:2) = .TRUE.
      RED(2:NX-1:2,3:NY-1:2,3:NZ-1:2) = .TRUE.

      END

      DOUBLE PRECISION FUNCTION G(X, Y, Z)
!
!    G IS THE BOUNDARY VALUE FUNCTION FROM THE POISSON EQUATION
!
      DOUBLE PRECISION X, Y, Z
```