

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**ANÁLISE DA ESCALABILIDADE DE
APLICAÇÕES EM COMPUTADORES
MULTICORE**

Samuel Reghim Silva

Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos – SP

Junho/2013

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**ANÁLISE DA ESCALABILIDADE DE
APLICAÇÕES EM COMPUTADORES
MULTICORE**

Samuel Reghim Silva

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Sistemas de Computação.

Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos – SP

Junho/2013

**Ficha catalográfica elaborada pelo DePT da
Biblioteca Comunitária da UFSCar**

S586ae

Silva, Samuel Reghim.

Análise da escalabilidade de aplicações em computadores multicore / Samuel Reghim Silva. -- São Carlos : UFSCar, 2013.
136 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2013.

1. Computação. 2. Escalabilidade. 3. Multithread. 4. Processamento paralelo. I. Título.

CDD: 004 (20^a)

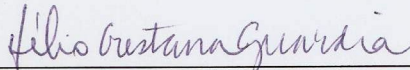
Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

**“Análise da Escalabilidade de Aplicações em
Computadores Multicore”**

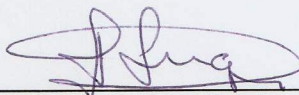
Samuel Reghim Silva

Dissertação de Mestrado apresentada ao
Programa de Pós-Graduação em Ciência da
Computação da Universidade Federal de São
Carlos, como parte dos requisitos para a
obtenção do título de Mestre em Ciência da
Computação

Membros da Banca:



Prof. Dr. Hélio Crestana Guardia
(Orientador - DC/UFSCar)



Prof. Dr. Hermes Senger
(DC/UFSCar)



Prof. Dr. Alfredo Goldman vel Lejbman
(IME/USP)

São Carlos
Junho/2013

À minha família.

Agradecimentos

À minha família, pelo apoio e paciência que ajudaram a tornar este projeto possível.

Ao meu orientador, prof. Dr. Hélio Guardia, pela oportunidade de pesquisa, pela confiança em mim depositada e pelo aprendizado técnico e humano.

Ao amigo Dr. Marcos Lourenzoni, pelas valiosas dicas e pelo acesso às máquinas.

Aos colegas de laboratório, pela inestimável companhia durante a pesquisa.

À gerência, equipe e instalações do Intel Manycore Testing Lab.

Ao CNPq e à CAPES, pela ajuda financeira durante este trabalho.

Resumo

Processadores *multicore* permitem que aplicações explorem paralelismo no nível de *threads* para habilitar melhorias no tempo de conclusão da execução. O compartilhamento do subsistema de memória e a disparidade entre as velocidades dos processadores e das operações de acesso à memória, contudo, podem implicar em limitações na escalabilidade causadas pela competição das *threads* pelos recursos. A determinação da quantidade apropriada de *threads* que garanta execuções eficientes para uma aplicação é um problema não trivial cuja obtenção automatizada é amplamente desejada.

Neste trabalho, buscou-se avaliar os fatores limitantes para a escalabilidade de aplicações paralelas com OpenMP relacionados à contenção pelos recursos compartilhados em processadores *multicore*, com o objetivo de identificar características das aplicações que limitem sua escalabilidade.

Constatou-se que os acessos à memória são a principal limitação aos ganhos de desempenho com o paralelismo. A granularidade, que indica a proporção de acessos à memória em relação ao processamento, foi verificada como sendo um indicativo importante do desempenho das execuções paralelas.

Estimativas de granularidade podem ser obtidas a partir do código-fonte das aplicações. Diferentes modos de acessos aos dados apontam, todavia, para a necessidade de combinação da estimativa de granularidade com informações sobre a localidade dos acessos aos dados para determinar corretamente a escalabilidade das aplicações.

Palavras-chave: escalabilidade, paralelismo, *multicore*, *threads*

Abstract

Multicore processors allow applications to explore thread-level parallelism in order to enable improvements on the elapsed time. The sharing of the memory subsystem and the discrepancy between the speeds of processors and memory access operations, however, may entail limitations to the scalability caused by thread competition for the resources. The automatic determination of the appropriate number of threads for an application that ensure efficient executions, although widely desired, is a non-trivial problem.

This work aimed to evaluate the factors limiting the scalability of OpenMP parallel applications related to the contention for shared resources in multicore processors, with the goal of identifying the characteristics of applications that limit their scalability.

It was found that memory accesses are a major limitation to the performance gains with parallelism. The granularity, indicating the ratio of memory accesses to processing, has been verified as being an important performance factor of parallel executions.

Estimates of granularity can be obtained from the applications' source code. Different data access modes, however, point to the need to estimate the combination of granularity with information about the data access locality to properly determine the scalability of applications.

Keywords: scalability, parallelism, multicore, threads

Lista de Figuras

1.1	Curva de <i>speedup</i> para diferentes frações sequenciais do tempo de execução (WILKINSON; ALLEN, 2004).	18
1.2	Esquema de um processador Intel com QuickPath Interconnect: os <i>cores</i> são interligados por um componente <i>crossbar</i> e o controlador de memória é integrado (INTEL CORPORATION, 2009).	20
1.3	Diagrama de blocos do processador Tiler TILE64 mostrando a rede de interligação entre os <i>cores</i> e os diversos componentes que o classificam como um SoC (BELL et al., 2008).	21
1.4	Sistema NUMA com quatro processadores interligados por QuickPath Interconnect (ZIAKAS et al., 2010).	22
1.5	Comparação entre aumento da velocidade do processador e aumento da largura de banda de memória (MCCALPIN, 1991-2007).	24
4.1	Curvas de <i>speedup</i> para NPB classe A na máquina SGI.	44
4.2	Curvas de eficiência para NPB classe A na máquina SGI.	44
4.3	Curvas de <i>speedup</i> para NPB classe B na máquina SGI.	45
4.4	Curvas de eficiência para NPB classe B na máquina SGI.	45
4.5	Curvas de <i>speedup</i> para NPB classe C na máquina SGI.	46
4.6	Curvas de eficiência para NPB classe C na máquina SGI.	46
4.7	Taxas de uso de barramento de memória para NPB classe A na máquina SGI.	48
4.8	Taxas de uso de barramento de memória para NPB classe B na máquina SGI.	49
4.9	Taxa de uso de barramento de memória para NPB classe C na máquina SGI.	50

4.10	Tempos de CPU para NPB classe A na máquina SGI (em relação à versão sequencial).	52
4.11	Tempos de CPU para NPB classe B na máquina SGI (em relação à versão sequencial).	52
4.12	Tempos de CPU para NPB classe C na máquina SGI (em relação à versão sequencial).	53
4.13	Taxas de faltas no cache de dados de nível 1 para NPB classe A em SGI. .	54
4.14	Taxas de faltas no cache de dados de nível 1 para NPB classe B em SGI. .	54
4.15	Taxas de faltas no cache de dados de nível 1 para NPB classe C em SGI. .	55
4.16	Taxas de faltas no cache de nível 2 para NPB classe A em SGI.	56
4.17	Taxas de faltas no cache de nível 2 para NPB classe B em SGI.	56
4.18	Taxas de faltas no cache de nível 2 para NPB classe C em SGI.	57
4.19	Curvas de <i>speedup</i> para NPB na máquina HP.	58
4.20	Curvas de eficiência para NPB na máquina HP.	58
4.21	Curva de <i>speedup</i> para NPB na máquina AMD.	59
4.22	Curva de eficiência para NPB na máquina AMD.	60
4.23	Curvas de <i>speedup</i> para NPB na máquina MTL sem afinidade de CPU. . .	62
4.24	Curvas de <i>speedup</i> para NPB na máquina MTL com afinidade de CPU ativa. .	62
4.25	Curvas de eficiência para NPB na máquina MTL com afinidade de CPU ativa.	63
4.26	Curvas de <i>speedup</i> para STREAM nos quatro sistemas testados.	64
4.27	Curvas de eficiência para STREAM nos quatro sistemas testados.	65
4.28	Tempo de CPU para STREAM nos quatro sistemas testados (em relação à versão sequencial).	65
4.29	Tempo de CPU e tempo decorrido de STREAM- <i>base</i> para cada carga de trabalho adicional em SGI, em relação à execução sem perturbação.	67
4.30	Curvas de <i>speedup</i> para MultMat nos três sistemas testados.	67
4.31	Curvas de eficiência para MultMat nos três sistemas testados.	68

5.1	Tempo de execução das cinco funções mais demoradas de cada aplicação de NPB classe B em SGI.	73
5.2	Granularidade efetiva de NPB classe A na máquina SGI.	75
5.3	Granularidade efetiva de NPB classe B na máquina SGI.	75
5.4	Granularidade efetiva de NPB classe C na máquina SGI.	76
5.5	Granularidade efetiva de MultMat e de suas variações na máquina SGI. . .	77
5.6	Granularidade efetiva de STREAM na máquina SGI.	77
5.7	Curvas de <i>speedup</i> para as versões originais e com granularidade modificada de EP, FT e MG na máquina SGI.	82
5.8	Curvas de <i>speedup</i> para as versões originais e com granularidade modificada de FT e MG na máquina HP.	83
5.9	Curvas de <i>speedup</i> para as versões originais e com granularidade modificada de FT e MG na máquina AMD.	84
5.10	Curvas de <i>speedup</i> para as versões originais e com granularidade modificada de EP, FT e MG na máquina MTL.	85
5.11	Taxas de uso de barramento de memória para as versões originais e com granularidade modificada de EP, FT e MG na máquina SGI.	86
5.12	Granularidade efetiva para as versões original e com granularidade modificada de MultMat na máquina SGI.	87
5.13	Taxas de uso de barramento de memória para as versões original e com granularidade modificada de MultMat na máquina SGI.	87
5.14	Curvas de <i>speedup</i> para as versões de MultMat com granularidade alterada nas máquinas MTL, SGI e AMD.	88
5.15	Curvas de <i>speedup</i> para o componente COPY de STREAM com variações no volume de processamento na máquina SGI.	89
5.16	Curvas de <i>speedup</i> para o componente SCALE de STREAM com variações no volume de processamento na máquina SGI.	90
5.17	Curvas de <i>speedup</i> para o componente ADD de STREAM com variações no volume de processamento na máquina SGI.	90

5.18	Curvas de <i>speedup</i> para o componente TRIAD de STREAM com variações no volume de processamento na máquina SGI.	91
5.19	Taxas de uso de barramento de memória para o componente COPY de STREAM com variações no volume de processamento na máquina SGI. . .	92
5.20	Taxas de uso de barramento de memória para o componente SCALE de STREAM com variações no volume de processamento na máquina SGI. . .	92
5.21	Taxas de uso de barramento de memória para o componente ADD de STREAM com variações no volume de processamento na máquina SGI. . .	93
5.22	Taxas de uso de barramento de memória para o componente TRIAD de STREAM com variações no volume de processamento na máquina SGI. . .	93
5.23	Curvas de <i>speedup</i> para as versões originais e as com <i>loops</i> invertidos de FT e MG na máquina SGI.	94
5.24	Curvas de <i>speedup</i> para as versões originais e as com <i>loops</i> invertidos de FT e MG na máquina MTL.	95
5.25	Curvas de <i>speedup</i> para a versão original e a com <i>loops</i> invertidos de MultMat nas máquinas SGI e AMD.	96
5.26	Curvas de <i>speedup</i> para a versão original e a com <i>loops</i> invertidos de MultMat na máquina MTL.	97
5.27	Taxas de uso do barramento de memória para a versão original e a com <i>loops</i> invertidos de MultMat na máquina SGI.	97
5.28	Granularidade efetiva para a versão original e a com <i>loops</i> invertidos de MultMat na máquina SGI.	98

Lista de Tabelas

2.1	Características de aplicações usadas no aprendizado de máquina (TOUR-NAVITIS et al., 2009)	29
2.2	Características extraídas de aplicações (WANG; O'BOYLE, 2009)	29
3.1	As quatro operações realizadas por STREAM.	37
4.1	Eventos de hardware observados com a ferramenta VTune para execuções no sistema SGI.	42
4.2	Aumento médio de <i>speedup</i> máximo para NPB em MTL com o uso de afinidade de CPU.	61
4.3	Taxas de uso de barramento e de faltas no cache para STREAM em SGI.	66
4.4	Taxas de uso de barramento e de faltas no cache para MultMat em SGI.	68
5.1	Velocidade de processamento de um <i>core</i> e hipotética total, largura de banda e β_M dos sistemas computacionais testados.	71
5.2	Granularidade de <i>loop</i> das aplicações de NPB, STREAM e MultMat.	72
5.3	Granularidade de <i>loop</i> de aplicações de NPB e MultMat após alterações.	81
5.4	Valores das variáveis da equação 5.1 para o componente TRIAD de STREAM na máquina SGI.	100
A.1	Desvio padrão do tempo decorrido de NPB classe A na máquina SGI.	108
A.2	Desvio padrão do tempo decorrido de NPB classe B na máquina SGI.	108
A.3	Desvio padrão do tempo decorrido de NPB classe C na máquina SGI.	109
A.4	Desvio padrão da taxa de uso de barramento de NPB na máquina SGI.	109
A.5	Desvio padrão da taxa de faltas no cache de nível 1 de NPB na máquina SGI.	110

A.6	Desvio padrão da taxa de faltas no cache de nível 2 de NPB na máquina SGI.	110
A.7	Desvio padrão do tempo decorrido de NPB na máquina HP.	111
A.8	Desvio padrão do tempo decorrido de NPB na máquina AMD.	111
A.9	Desvio padrão do tempo decorrido de NPB na máquina MTL sem afinidade de CPU.	112
A.10	Desvio padrão do tempo decorrido de NPB na máquina MTL com afinidade de CPU ativa.	113
A.11	Desvio padrão de γ_E de NPB na máquina SGI.	114
A.12	Desvio padrão do tempo decorrido para as versões com granularidade alterada de EP, FT e MG na máquina SGI.	114
A.13	Desvio padrão do tempo decorrido para as versões com granularidade alterada de FT e MG na máquina HP.	114
A.14	Desvio padrão do tempo decorrido para as versões com granularidade alterada de FT e MG na máquina AMD.	115
A.15	Desvio padrão do tempo decorrido para as versões com granularidade alterada de EP, FT e MG na máquina MTL.	115
A.16	Desvio padrão da taxa de uso de barramento para as versões com granularidade alterada de EP, FT e MG na máquina SGI.	115
A.17	Desvio padrão de γ_E para as versões com granularidade alterada de EP, FT e MG na máquina SGI.	116
A.18	Desvio padrão do tempo decorrido para as versões com <i>loops</i> invertidos de FT e MG na máquina SGI.	116
A.19	Desvio padrão do tempo decorrido para as versões com <i>loops</i> invertidos de FT e MG na máquina HP.	116
A.20	Desvio padrão do tempo decorrido para as versões com <i>loops</i> invertidos de FT e MG na máquina AMD.	117
A.21	Desvio padrão do tempo decorrido para as versões com <i>loops</i> invertidos de FT e MG na máquina MTL.	117

A.22 Desvio padrão do tempo decorrido para as versões de MultMat na máquina SGI.	118
A.23 Desvio padrão do tempo decorrido para as versões de MultMat na máquina AMD.	118
A.24 Desvio padrão do tempo decorrido para as versões de MultMat na máquina MTL.	119
A.25 Desvio padrão do tempo decorrido para STREAM nas máquinas SGI, AMD e HP.	119
A.26 Desvio padrão do tempo decorrido para STREAM na máquina MTL. . . .	120
A.27 Desvio padrão de γ_E para STREAM e seus componentes na máquina SGI.	120
A.28 Desvio padrão do tempo decorrido para STREAM com diferentes volumes de processamento extra na máquina SGI.	121
A.29 Desvio padrão do tempo decorrido para STREAM-COPY com diferentes volumes de processamento extra na máquina SGI.	121
A.30 Desvio padrão do tempo decorrido para STREAM-SCALE com diferentes volumes de processamento extra na máquina SGI.	122
A.31 Desvio padrão do tempo decorrido para STREAM-ADD com diferentes volumes de processamento extra na máquina SGI.	122
A.32 Desvio padrão do tempo decorrido para STREAM-TRIAD com diferentes volumes de processamento extra na máquina SGI.	123
A.33 Desvio padrão da taxa de uso de barramento para STREAM com diferentes volumes de processamento extra na máquina SGI.	123
A.34 Desvio padrão da taxa de uso de barramento para STREAM-COPY com diferentes volumes de processamento extra na máquina SGI.	124
A.35 Desvio padrão da taxa de uso de barramento para STREAM-SCALE com diferentes volumes de processamento extra na máquina SGI.	124
A.36 Desvio padrão da taxa de uso de barramento para STREAM-ADD com diferentes volumes de processamento extra na máquina SGI.	125
A.37 Desvio padrão da taxa de uso de barramento para STREAM-TRIAD com diferentes volumes de processamento extra na máquina SGI.	125

Sumário

CAPÍTULO 1 –INTRODUÇÃO	16
1.1 Componentes de um sistema multicore	19
1.1.1 Processador	19
1.1.2 Memória	21
1.2 Memory wall	23
1.3 Características de aplicações paralelas	23
1.3.1 Localidade de referência	23
1.3.2 Granularidade dos acessos	25
1.4 Escopo deste trabalho	25
1.5 Principais contribuições	26
1.6 Organização desta dissertação	26
CAPÍTULO 2 –GRAU DE PARALELISMO	27
2.1 Decisão direta	27
2.2 Técnicas de aprendizado de máquina	28
2.3 Modelagem analítica	30
2.4 Análise de dependência de dados	32
CAPÍTULO 3 –GRANULARIDADE DOS ACESSOS À MEMÓRIA	33
3.1 Objetivos	33
3.2 Metodologia de investigação	34

CAPÍTULO 4 –ESCALABILIDADE DE APLICAÇÕES OPENMP COM PARALELISMO DE DADOS	40
4.1 Hardware utilizado	40
4.2 Métricas e ferramentas	41
4.3 Número máximo de threads	42
4.4 NPB	42
4.4.1 Speedup	42
4.4.2 Taxa de uso de barramento	47
4.4.3 Total de instruções executadas	49
4.4.4 Tempo de CPU	50
4.4.5 Taxa de faltas no cache	51
4.5 STREAM	61
4.5.1 Speedup	61
4.5.2 Taxa de uso de barramento e taxas de faltas no cache	63
4.6 Multiplicação de matrizes	66
4.7 Conclusões	66
CAPÍTULO 5 –INFLUÊNCIA DA GRANULARIDADE DOS ACESSOS NA ESCALABILIDADE DE APLICAÇÕES PARALELAS	70
5.1 Caracterização dos sistemas computacionais	70
5.2 Granularidade de loop	71
5.3 Granularidade efetiva	73
5.4 Relação com o desempenho	78
5.5 Variações na granularidade	81
5.5.1 NPB	81
5.5.2 MultMat	85
5.5.3 STREAM	86

5.6	Variações na localidade	91
5.7	Conclusões	98
5.7.1	Considerações sobre modelagem analítica	99
CAPÍTULO 6 –CONCLUSÕES		103
6.1	Trabalhos Futuros	105
APÊNDICE A – DESVIO PADRÃO DOS EXPERIMENTOS		107
APÊNDICE B – CÓDIGO-FONTE DAS ALTERAÇÕES DE GRANULARIDADE		126
REFERÊNCIAS		132
GLOSSÁRIO		137

Capítulo 1

Introdução

O desenvolvimento dos computadores modernos permitiu grandes avanços nas áreas de ciência e tecnologia. Com o suporte provido pelos computadores, grandes problemas científicos são resolvidos e novos são propostos. Novas aplicações apresentam demandas computacionais sempre crescentes.

Tradicionalmente, a oferta de aumento da capacidade de computação se deu pela construção de processadores cada vez mais velozes, possibilitada pela redução do tamanho dos componentes básicos, pela exploração de paralelismo no nível das instruções e pelo aumento da frequência de *clock*. Apesar das tecnologias de produção de circuitos evoluírem continuamente, limitações físicas se tornaram uma questão fundamental no projeto de computadores baseados em eletrônica de semicondutores, restringindo a evolução de cada processador individualmente (SUTTER, 2005).

Por outro lado, a grande capacidade de integração, combinada com a redução dos custos de fabricação, torna o uso de múltiplos processadores independentes atraente, dando início à era dos chips *multicore*. Tais processadores apresentam desde uns poucos até várias dezenas de *cores* (núcleos de processamento) homogêneos ou heterogêneos. Embora sua disponibilidade beneficie imediatamente ambientes multiprogramados, processadores *multicore* exigem que desenvolvedores de aplicações explorem paralelismo no nível de *threads* para habilitar melhorias no tempo de execução (OLUKOTUN et al., 1996). Tanenbaum (2005) define *threads* como “entidades escalonadas para a execução sobre a CPU”. São também chamadas de *processos leves*, pois *threads* de um mesmo processo compartilham alguns recursos, como o espaço de endereçamento e arquivos abertos, e ainda assim suas execuções apresentam elevado grau de independência entre si.

A avaliação de desempenho de aplicações paralelas é realizada com o uso de um

conjunto de métricas, escolhidas de acordo com o resultado desejado da análise (GRAMA et al., 2003). Além do tempo de execução, duas outras métricas se destacam. A primeira, *speedup*, é calculada, conforme a equação 1.1, pela razão entre o tempo de execução da versão sequencial da aplicação, t_s , e o tempo de execução da versão paralela em um computador com p processadores, t_p . *Speedup* indica o quão mais rápida foi a execução paralela em comparação à versão sequencial. Já a *eficiência*, segundo Grama et al. (2003, p. 202), aponta a fração de tempo em que as unidades de processamento são usadas. É calculada pela relação entre o *speedup*, S , e o número de processadores, p , como mostra a equação 1.2.

$$S(p) = \frac{t_s}{t_p} \quad (1.1)$$

$$E(p) = \frac{S}{p} \quad (1.2)$$

Assim, a execução eficiente de aplicações paralelas envolve a determinação do grau de paralelismo, seja em multicomputadores, em multiprocessadores ou em ambientes *multicore*. Em processadores *multicore*, significa a escolha da quantidade de *threads* a usar. Intuitivamente, pode-se considerar que o número adequado de *threads* é fixado pela quantidade de *cores* no sistema. Idealmente, o *speedup* seria p e a eficiência seria 1 para qualquer quantidade de processadores. Amdahl (1967) observou, contudo, que há uma fração das operações de uma aplicação paralela que precisa ser realizada de forma sequencial, usualmente relacionada ao gerenciamento dos dados manipulados, e notou que os ganhos com a paralelização estão limitados a este trecho sequencial. Supondo uma aplicação com fração sequencial igual a f e fração paralelizável igual a $(1 - f)$, a Lei de Amdahl (WILKINSON; ALLEN, 2004) diz que o *speedup* esperado é (equação 1.3):

$$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \quad (1.3)$$

Mesmo com uma quantidade ilimitada de processadores, o *speedup* máximo é de (equação 1.4):

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{f} \quad (1.4)$$

A figura 1.1 mostra este resultado para diferentes valores de f e indica que uma fração

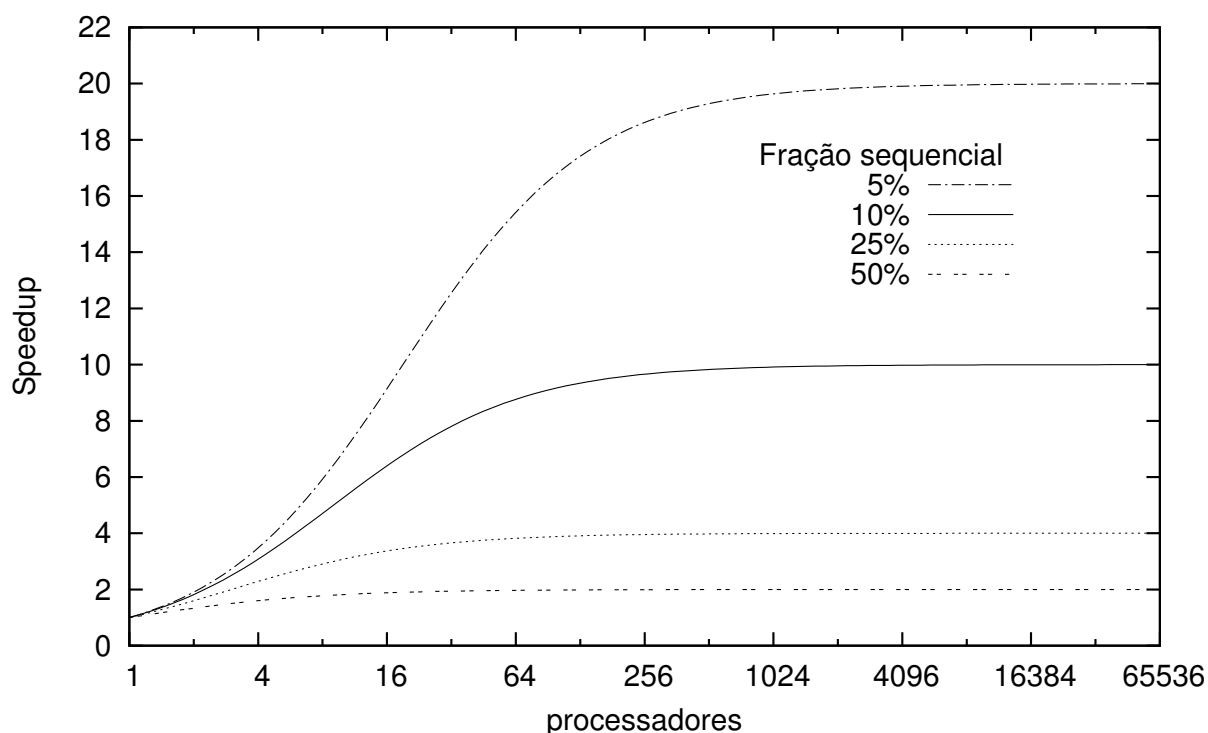


Figura 1.1: Curva de *speedup* para diferentes frações sequenciais do tempo de execução (WILKINSON; ALLEN, 2004).

sequencial equivalente a apenas 5% do tempo total de execução limita o *speedup* a no máximo 20. Este resultado, ainda que assuma que o trecho paralelo da aplicação pode ser dividido indefinidamente em operações simultâneas sem causar sobrecarga, foi usado para estimular o uso de sistemas monoprocesados. Gustafson (1988) argumenta, no entanto, que o sucesso do paralelismo está na resolução de problemas maiores, que acompanhem o incremento dos recursos computacionais, e aponta *speedups* muito maiores com esta abordagem. A maior quantidade de operações realizadas simultaneamente diminui o valor de f e habilita maiores ganhos de desempenho. De fato, os supercomputadores mais poderosos do planeta, reunidos em uma lista semestral pelo projeto TOP500 (MEUER et al., 2013), são máquinas com milhares de processadores.

Para aplicações embaraçosamente paralelas com fração sequencial f pequena, a abordagem de utilização de todos os recursos disponíveis pode ser viável. No entanto, a competição pelo uso dos recursos compartilhados pode limitar os ganhos de desempenho com o paralelismo (MALLADI, 2009), fazendo com que a escolha da quantidade de *threads* dependa, então, do comportamento de cada aplicação e de características do sistema

computacional utilizado (SUN; BYNA; HOLMGREN, 2009). A determinação do grau de paralelismo adequado para uma aplicação torna-se um problema não trivial cuja obtenção automatizada é amplamente desejada. Em função disto, a seção 1.1 descreve os principais componentes de um sistema *multicore*, ao passo que a seção 1.2 apresenta um fator importante que influencia a execução em sistemas *multicore* e portanto o grau de paralelismo. Finalmente, a seção 1.3 descreve características importantes de aplicações com relação a seu desempenho.

1.1 Componentes de um sistema *multicore*

1.1.1 Processador

A abordagem *multicore* segue o modelo de multiprocessadores, em que diversos processadores independentes acessam um espaço comum de endereçamento de memória (GRAMA et al., 2003). Um sistema *multicore* consiste na aglomeração de múltiplas unidades funcionais, os *cores* (núcleos), em uma única pastilha de processador (doravante referido apenas como processador). Cada *core* é uma unidade de processamento completa, com seu próprio contador de programa, registradores e lógica de controle. Em relação à arquitetura do conjunto de instruções, os *cores* podem ser idênticos (ou homogêneos), como AMD Opteron, Intel Xeon e Oracle SPARC, ou heterogêneos, como STI Cell BE. *Cores* homogêneos permitem uma programação mais simplificada e tipicamente incluem lógica de controle avançada, como paralelismo no nível de instruções, previsão de desvios e busca antecipada de dados (CONWAY; HUGHES, 2007; CASAZZA, 2010). *Cores* heterogêneos, por sua vez, focam em simplicidade e desempenho para a otimização de cargas de trabalho típicas ao custo de maior esforço de programação; nestes casos, alguns dos *cores* podem ter uma lógica de controle reduzida, a fim de minimizar sincronizações e custos de movimentação de dados e permitir mais *cores* no *chip* (GSCHWIND et al., 2006). Alguns fabricantes empregam a técnica de *simultaneous multithreading* (SMT), que permite a “várias *threads* independentes expedir instruções para múltiplas unidades funcionais superescalares em um único ciclo” (TULLSEN; EGGERS; LEVY, 1998), com o objetivo de esconder latências no acesso à memória. A quantidade de vias de uma implementação SMT indica o máximo de *threads de hardware* em cada *core*. Intel utiliza SMT de duas vias, o *Hyper-Threading* (BARKER et al., 2008), em alguns de seus processadores, ao passo que Oracle usa SMT de 8 vias (SHIN et al., 2010).

Comumente são reunidos de 4 até 16 *cores* em cada processador, mas também há

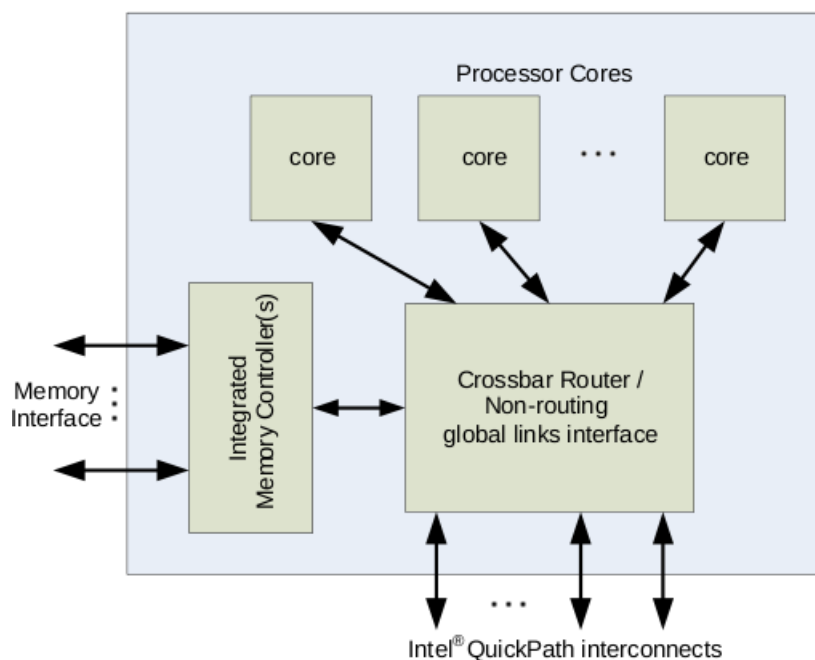


Figura 1.2: Esquema de um processador Intel com QuickPath Interconnect: os cores são interligados por um componente *crossbar* e o controlador de memória é integrado (INTEL CORPORATION, 2009).

produtos com até 72 *cores*, como os da família Tileria (TILERA CORPORATION, 2013). Os *cores* podem ser ligados entre si por um mecanismo do tipo *crossbar*, que permite comunicação de cada *core* com todos os demais, como mostrado na figura 1.2, ou através de uma rede do tipo malha, como na figura 1.3, que liga cada *core* a seus vizinhos imediatos. Alguns processadores são ainda do tipo *system-on-a-chip* (ou *SoC*), em que muitos componentes tradicionalmente externos são trazidos para dentro do processador, tais como controladores de rede, interfaces seriais e interfaces de componentes periféricos; é o caso dos processadores SPARC (SHIN et al., 2010) e Tileria (TILERA CORPORATION, 2013). Uma tendência comum aos principais fabricantes, mesmo em abordagens não *SoC*, é a integração do controlador de memória ao processador, tal como exemplificado pela figura 1.2. As implicações desta mudança são mostradas na subseção 1.1.2.

Por muito tempo, os processadores foram ligados às demais partes do sistema computacional pelo barramento frontal, FSB (*front-side bus*). A necessidade de melhores topologias e canais de comunicação mais rápidos estimulou a introdução de canais de comunicação entre *chips*. As tecnologias *HyperTransport* (HT), da AMD, e *QuickPath Interconnect* (QPI), da Intel, possuem ambas canais bidirecionais (dois *links* unidirecionais) ponto-a-ponto de baixa latência que substituem o FSB e permitem a conexão de processadores entre si e com circuitos de entrada/saída.

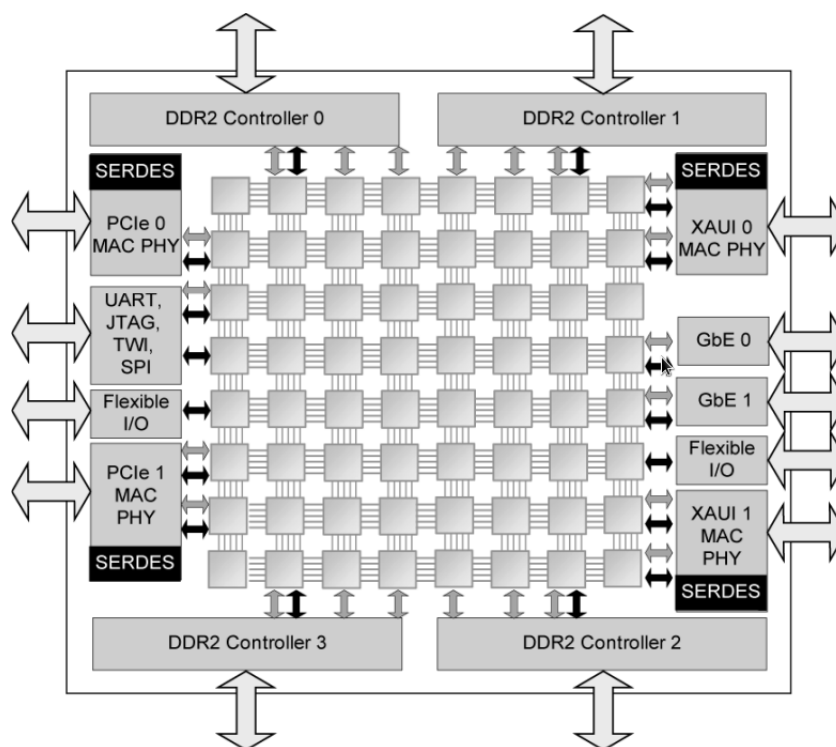


Figura 1.3: Diagrama de blocos do processador Tiler TILE64 mostrando a rede de interligação entre os *cores* e os diversos componentes que o classificam como um SoC (BELL et al., 2008).

1.1.2 Memória

Em um sistema multicore, a memória é do tipo compartilhada, com todos os *cores* acessando um mesmo espaço de endereçamento. Conforme exposto na subseção 1.1.1, a ligação de um processador à memória pode ser feita através do barramento frontal, caso em que o controlador de memória é externo ao processador, ou, se for o caso, pelo controlador de memória embutido no processador. Quando o sistema possui apenas um processador, as duas abordagens são equivalentes. O tempo de acesso a qualquer posição de memória é sempre o mesmo (desconsiderando os efeitos de cache, discutidos a seguir), configurando um sistema UMA (*Uniform Memory Access*, acesso uniforme à memória). A inclusão de canais de comunicação, como HT e QPI, e do controlador de memória no processador permite a disposição de vários *chips* em uma configuração NUMA (*Non-Uniform Memory Access*, acesso não uniforme à memória), conforme ilustrado pela figura 1.4 para um sistema com 4 processadores. Em uma arquitetura NUMA, cada processador tem sua memória local e pode acessar a memória dos demais processadores pelo canal de comunicação. Esta distância adicional torna o tempo de acessos remotos maior do que de acessos locais.

Entre os *cores* e a memória principal existe o cache, uma memória temporária pequena

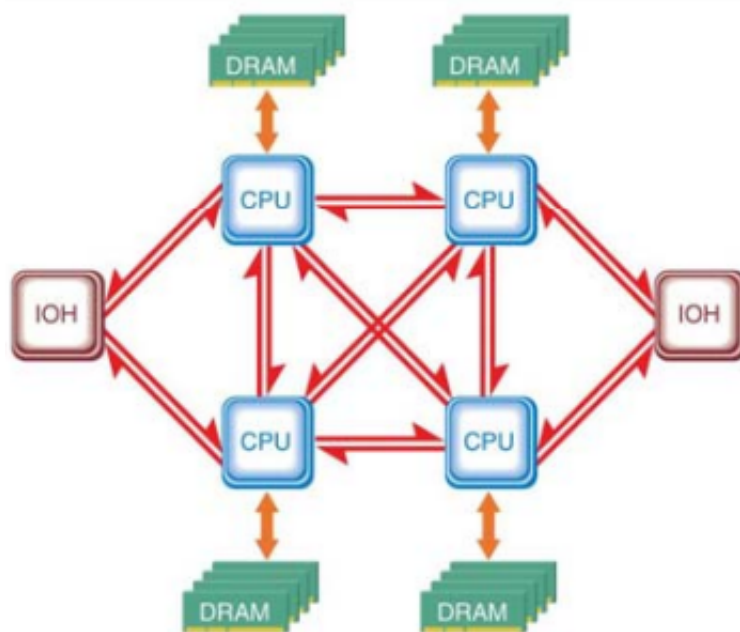


Figura 1.4: Sistema NUMA com quatro processadores interligados por QuickPath Interconnect (ZIAKAS et al., 2010).

e de alta velocidade usada para armazenar dados da memória principal que são acessados frequentemente com a intenção de reduzir seu tempo de acesso (SMITH, 1982). Cache é usado há décadas e aos poucos foi incorporado ao processador. O aumento da diferença de velocidade entre processador e memória estimulou a introdução de níveis de cache, com os níveis mais rápidos mais próximos do processador. Os processadores *multicore* contemporâneos apresentam de dois a três níveis de cache, com compartilhamento variado. É comum que os níveis 1 e 2, mais rápidos, sejam particulares a cada *core*, e o último nível seja compartilhado por todos ou por grupos de *cores*. O processador AMD Opteron 8439 SE, por exemplo, tem três níveis de cache, com 128 KB para o nível 1, privativo a cada *core*, 512 KB no nível dois, também privativo, e 6 MB no nível três, compartilhado por todos os *cores* (ADVANCED MICRO DEVICES, INC., 2013), enquanto o processador Intel Xeon E5420 conta com dois níveis de cache, sendo o primeiro privativo a cada *core* e com 32KB de capacidade e o segundo compartilhado por dois *cores*, com 6 MB de capacidade. Um mecanismo de coerência é usado para garantir a consistência de dados compartilhados quando posicionados nos caches particulares. Finalmente, também pode ser usado o mecanismo de *prefetching* (busca antecipada) de dados, em que um preditor monitora os acessos à memória, tenta prever os acessos futuros e traz para o cache as posições de memória correspondentes (MARS; TANG; SOFFA, 2011).

1.2 Memory wall

Historicamente, os circuitos que compõem as memórias não têm acompanhado o mesmo ritmo de evolução dos processadores. A figura 1.5 exibe dados reunidos ao longo de desesseis anos (MCCALPIN, 1991-2007) desta evolução e exibe uma grande disparidade entre as velocidades de processador e de memória. A escala logarítmica do eixo vertical mostra que o aumento da taxa com que os dados são movimentados da memória para o processador, em milhões de palavras de 64 bits por segundo (MWords/s), é consideravelmente menor do que a taxa com que tais dados podem ser processados (MFLOPS). A existência de múltiplos cores, a partir do ano de 2006, prejudica ainda mais essa condição. Técnicas têm sido usadas na tentativa de mascarar essa discrepância e diminuir o tempo de acesso aos dados, como memórias cache e busca antecipada dos dados. Wulf e Mc-Kee (1995) observaram que a diferença de melhoria de desempenho entre processador e memória resultaria em uma barreira, a *memory wall*, que impediria o aumento da velocidade dos sistemas computacionais; uma nova tecnologia de memória seria necessária para se manter o ritmo de evolução. Já Rogers et al. (2009) argumentaram que, sem técnicas de conservação de largura de banda de memória, um processador com 8 *cores* escalaria para apenas 24 *cores* após 4 gerações tecnológicas, ao contrário dos 128 *cores* esperados considerando uma duplicação na capacidade de integração a cada dezoito meses. Blagodurov et al. (2010), Hood et al. (2010) e Diamond et al. (2011) apontaram a largura de banda de memória como uma importante limitação para o desempenho em sistemas *multicore*. Com as tecnologias atuais, o tráfego de memória mostra-se, então, como uma possível restrição para a escalabilidade de futuros sistemas *multicore*.

1.3 Características de aplicações paralelas

Aplicações de interesse prático precisam de dados sobre os quais operar, que inicialmente se encontram na memória principal. Dois conceitos indicam sua forma de acesso aos dados: *localidade* e *granularidade*.

1.3.1 Localidade de referência

Localidade é um fenômeno bem conhecido e alvo de atenção pelos desenvolvedores de aplicações e projetistas de computadores. Sua importância estimulou a introdução de *caches* ao subsistema de memória (SMITH, 1982). A *localidade espacial* indica a tendência

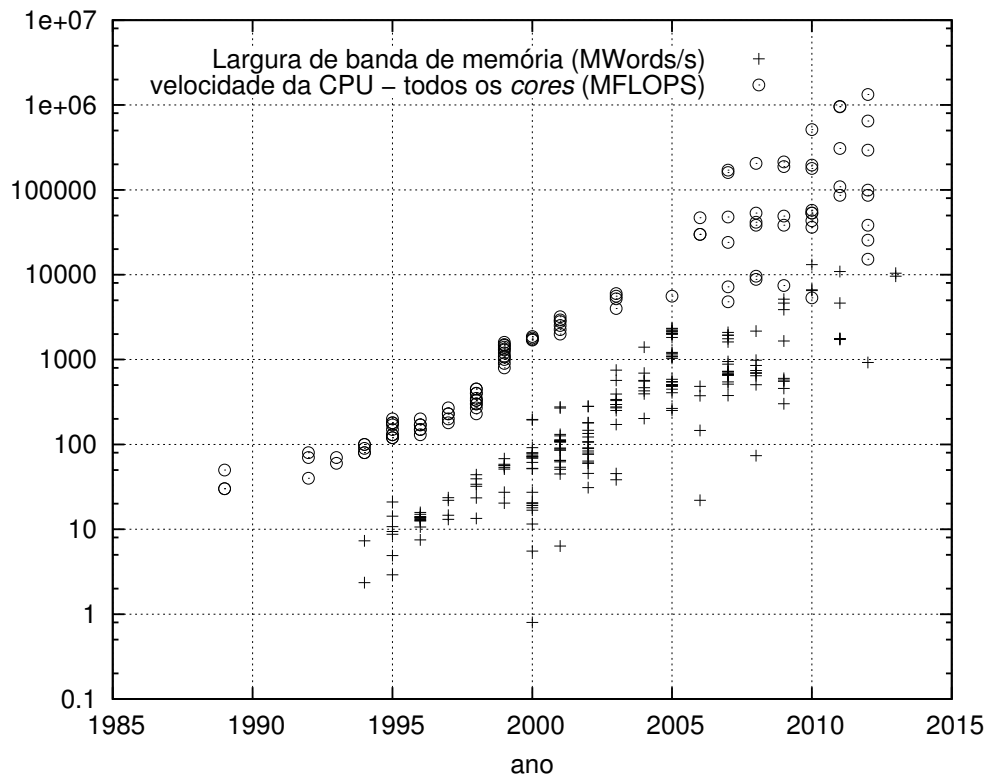


Figura 1.5: Comparação entre aumento da velocidade do processador e aumento da largura de banda de memória (MCCALPIN, 1991-2007).

de uma aplicação de acessar dados na memória em regiões próximas entre si. Já a *localidade temporal* indica sua tendência de acessar as mesmas posições de memória que foram referenciadas recentemente (PATTERSON; HENNESSY, 1996). Bunt e Murphy (1984) consideram que “localidade parece ser uma consequência natural da forma com que a maioria dos desenvolvedores de aplicações escrevem programas”.

A aferição da localidade de uma aplicação pode guiar melhorias nos padrões de acesso aos dados, permitindo explorar melhor o mecanismo de cache e conseqüentemente obter ganhos de desempenho. Usualmente, a localidade de acesso aos dados de uma aplicação é medida a partir da série de endereços de memória referenciados pelas aplicações, obtida tipicamente com o uso de instrumentação binária da aplicação (WEINBERG et al., 2005) ou com o uso de simuladores de *hardware* (GUPTA et al., 2012). A modelagem analítica da localidade é uma alternativa à sua medição direta. Embora dispense a necessidade de execução da aplicação, não é uma atividade simples (GRAMA et al., 2003, p.61) (LI et al., 2010) (SUN; BYNA; HOLMGREN, 2009).

A existência de caches de grande capacidade e com diversos níveis indica que a localidade também é relevante na execução em sistemas *multicore*.

1.3.2 Granularidade dos acessos

Granularidade indica a proporção de acessos à memória em relação ao processamento. Callahan, Cocke e Kennedy (1988) exploraram este conceito para estimar o desempenho de *loops* sequenciais em processadores com *pipeline*, preocupando-se com bloqueios devido a dependências de dados entre as instruções. Sua métrica de granularidade, ou *balanceamento de loop*, β_L , indica a proporção de palavras de dados do tipo ponto flutuante acessadas (M_L), em relação às operações de cálculos em ponto flutuante (*flops*, *floating-point operations*) dentro de um *loop* (F_L), conforme a equação 1.5.

$$\beta_L = \frac{\text{número de palavras acessadas}}{\text{número de } \textit{flops} \text{ realizadas}} = \frac{M_L}{F_L} \quad (1.5)$$

Já o *balanceamento da máquina*, β_M na equação 1.6, indica a razão entre a taxa com que dados de ponto flutuante são trazidos da memória (m_M) e a taxa em que operações de aritmética de ponto flutuante são realizadas (f_M).

$$\beta_M = \frac{\text{max palavras / ciclo}}{\text{max } \textit{flops}/\text{ciclo}} = \frac{m_M}{f_M} \quad (1.6)$$

Ainda na análise de Callahan, Cocke e Kennedy (1988), quando β_L é maior do que β_M , o *loop* precisa de mais dados do que é possível fornecer, e a aplicação é do tipo *memory-bound* (limitada pela memória). Se β_L for menor do que β_M , há uma folga na capacidade de trazer dados da memória e a aplicação é classificada como *compute-bound* (limitada pela capacidade de processamento da CPU). Estes conceitos podem ser utilizados na análise de contenção por recursos compartilhados em ambientes *multicore*, auxiliando a compreensão sobre o desempenho de aplicações.

1.4 Escopo deste trabalho

O fenômeno de *memory wall* sugere limitações preocupantes para o desempenho. Em um ambiente *multicore*, em que a memória é usualmente compartilhada por múltiplas unidades funcionais, este fenômeno pode ser mais notado.

Este trabalho procura avaliar os fatores limitantes para a escalabilidade de aplicações paralelas com OpenMP relacionados à contenção pelos recursos compartilhados em processadores *multicore*, com o objetivo de identificar características das aplicações que correspondam a sua escalabilidade e possam ser usadas em uma determinação automática

da quantidade de *threads*, ou ofereçam dicas ao programador sobre limitações do código.

1.5 Principais contribuições

A busca por fatores que permitam estimar a escalabilidade de aplicações paralelas sem necessidade de execução prévia produziu as seguintes contribuições principais:

- A confirmação de que o acesso à memória é uma limitação para a execução paralela em processadores *multicore*;
- A constatação de que a granularidade dos acessos à memória é um indicativo importante do desempenho paralelo;
- A avaliação de uma métrica, obtida do código-fonte da aplicação, que estima a granularidade dos acessos como determinante do desempenho de execuções paralelas.

1.6 Organização desta dissertação

O capítulo 2 apresenta aspectos da determinação da quantidade de *threads* para execução paralela em processadores *multicore*. A metodologia adotada na análise feita neste trabalho é abordada no capítulo 3. A análise da escalabilidade de aplicações OpenMP com paralelismo de dados é desenvolvida no capítulo 4, ao passo que a análise da influência da granularidade dos acessos na escalabilidade de aplicações paralelas é apresentada no capítulo 5. O capítulo 6 apresenta as conclusões.

Capítulo 2

Grau de paralelismo

Diversas metodologias têm sido propostas para o ajuste do grau de paralelismo de aplicações em sistemas *multicore*. Este capítulo apresenta algumas dessas técnicas e relaciona as informações que são empregadas nas deliberações. Decisões sobre quantidade de *threads* que não consideram o comportamento da aplicação ou que são feitas pelo desenvolvedor são apresentadas na seção 2.1. A seção 2.2 discute propostas automáticas baseadas em técnicas de inteligência artificial, ao passo que a seção 2.3 expõe uma modelagem analítica para o problema da quantidade de *threads*. Finalmente, a seção 2.4 cita um exemplo com análise de dependência de dados.

2.1 Decisão direta

Aplicações paralelas parametrizadas podem usar informações sobre o número de processadores locais para explicitamente decidir como particionar suas atividades. Compiladores paralelizantes podem, por sua vez, usar diretivas de compilação para determinar automaticamente o número de *threads* para uma aplicação paralela em função do número de processadores disponíveis. A API (*Application Programming Interface*, interface de programação de aplicações) OpenMP indica que o número de *threads* pode ser solicitado pela aplicação através de diretivas de compilação *pragma* (NOVILLO, 2006), de chamadas à rotina *omp_set_num_threads* ou ainda com o uso da variável de ambiente OMP_NUM_THREADS. A quantidade de *threads* criada é decidida durante a execução com base em um conjunto de variáveis, como o número de *threads* já ativas, o número de regiões paralelas e o número máximo permitido de *threads*, e pode recorrer a um comportamento definido pelo implementador da API (OPENMP ARCHITECTURE REVIEW BOARD, 2008, p. 36). No compilador GCC, por exemplo, a ação padrão é criar uma *thread* para

cada CPU¹(FREE SOFTWARE FOUNDATION, INC., 2006c).

2.2 Técnicas de aprendizado de máquina

De modo geral, as abordagens de paralelização apresentadas nesta seção se baseiam na similaridade entre aplicações para sugestão de número de *threads*. Técnicas de ML (*Machine Learning*, aprendizado de máquina) são usadas para prever o desempenho da aplicação de interesse pela comparação com aplicações previamente executadas. *Machine Learning* é uma área da Inteligência Artificial que tenta mimetizar o aprendizado humano criando uma base de conhecimento de exemplos que é consultada para a tomada de decisões (RUSSEL; NORVIG, 1995).

Tournavitis et al. (2009) apresentaram uma técnica baseada em ML para identificar paralelismo e determinar a política de escalonamento para aplicações sequenciais. Sua base de conhecimento foi populada por exemplos provenientes de características do código-fonte e de execuções monitoradas de aplicações de teste. Deste modo, foi criado um preditor, reunindo características de aplicações e resultados de mapeamentos de paralelismo, para estimar o comportamento de uma nova aplicação e aplicar o melhor mapeamento considerado. Os aspectos relevantes das aplicações foram divididos em características estáticas, coletadas da representação interna do compilador para a aplicação sendo compilada, e características dinâmicas, obtidas de execuções instrumentadas, apresentadas na tabela 2.1. O treinamento do componente de ML consistiu da confrontação de pares de características de aplicações e decisões de mapeamento desejadas, obtidas através de registros de desempenho de execuções repetidas e cronometradas de versões sequenciais e paralelas de aplicações com *loops* sabidamente paralelizáveis, com diferentes opções de política de escalonamento. Tais políticas foram escolhidas como sendo as quatro disponíveis no OpenMP: CYCLIC, DYNAMIC, GUIDED e STATIC. Concluída esta fase, treinamentos posteriores não foram realizados. Dada uma nova aplicação, suas características (mostradas na tabela 1) são coletadas e apresentadas ao componente de ML, que indica, para cada *loop* (iniciando pelos mais externos, por apresentarem granularidade mais grossa), uma classificação indicativa de benefícios da execução paralela e uma sugestão de política de escalonamento. Finalmente, código paralelo é gerado com um compilador OpenMP nativo. Intervenção do usuário pode ser exigida se uma paralelização, embora aparentemente possível e vantajosa, não puder ser provada como correta pela

¹Conforme informado pelo sistema operacional. Devido ao SMT, *threads* de *hardware*, ao invés de *cores*, podem ser consideradas como CPUs.

Tabela 2.1: Características de aplicações usadas no aprendizado de máquina (TOUR-NAVITIS et al., 2009)

Características estáticas	número de instruções número de operações de <i>Load/Store</i> número de saltos/desvios número de iterações de <i>loop</i>
Características dinâmicas	número de acessos a dados número de instruções número de saltos/desvios realizados

Tabela 2.2: Características extraídas de aplicações (WANG; O'BOYLE, 2009)

Características de código	ciclos por instrução número de saltos/desvios número de instruções de <i>Load/Store</i> computações por instrução
Características dinâmicas	contagem de interações de <i>loops</i> taxa de acertos no cache L1
Característica de tempo de execução dinâmicas	tempo da execução paralela

análise estática, inibindo a operação automática da proposta.

Wang e O'Boyle (2009) estenderam essa abordagem para o mapeamento automático de aplicações em processadores *multicore*. Além da política de escalonamento, seu mecanismo sugere também o número de *threads* a serem usadas. Outras características de aplicações foram consideradas, mostradas na tabela 2.2. O modelo de ML, alimentado com tais características, determina a escalabilidade da aplicação e indica um número de threads e a política de escalonamento. Dada uma nova aplicação, suas características de código e de dados são coletadas a partir da execução instrumentada da versão sequencial; em seguida, a aplicação é paralelizada usando políticas padrão e as características de tempo de execução são obtidas. Os preditores, então, tomam estas características e sugerem um esquema de paralelização. Embora o processo todo seja automatizado, são exigidas várias execuções prévias da aplicação de interesse.

Grewe, Wang e O'Boyle (2011) também aplicaram ideias de ML no mapeamento de aplicações com paralelismo de dados na presença de cargas de trabalho adicionais. As características de aplicações coletadas e usadas para treinar o componente de ML foram os *speedups* de aplicações teste com diferentes quantidades de *threads* em uma máquina sem carga de trabalho extra. As características usadas para representar a carga de trabalho foram o total de programas da carga de trabalho e o total de *threads* pertencentes a estes. Após o modelo preditivo ter sido criado, treinamentos posteriores só são necessários se

for desejado alterar o sistema computacional usado. O preditor de desempenho, para ser capaz de obter as características da carga de trabalho adicional e indicar a quantidade de *threads*, é incorporado a cada aplicação que se deseja executar com a técnica, o que pode gerar uma sobrecarga adicional.

2.3 Modelagem analítica

Uma modelagem analítica tenta estabelecer uma relação entre as variáveis de um sistema através de uma função matemática. Na determinação do grau de paralelismo, informações sobre a aplicação são fornecidas a um modelo para a sugestão de número de *threads* a usar.

O modelo analítico proposto por Sun, Byna e Holmgren (2009) se dispõe a prever a latência no acesso aos dados, a determinar a quantidade de *cores* a usar e a quantificar o impacto de otimizações no acesso aos dados, considerando que a contenção pelos recursos compartilhados é uma característica dependente de cada aplicação. A arquitetura de hardware assumida, formada por um único processador, apresenta dois níveis de cache privados e barramento entre L2 e memória principal compartilhado pelos *cores*. A aplicação modelada segue o esquema de divisão e conquista, sem dependência de dados entre suas *threads*, e com no máximo uma *thread* executando em cada *core*. Se cada *core* tiver um desempenho mantido de $P_{sustained}$ MFLOPS (milhões de instruções de ponto flutuante por segundo), n *cores* forem usados para se executar o total de I instruções, e R for a proporção de instruções que acessam a memória, então o tempo de computação T_{comp} é dado pela equação 2.1:

$$T_{comp} = \frac{I \times (1 - R)}{P_{sustained} \times n} \quad (2.1)$$

Uma vez que cada instrução de memória acessa L_{word} bytes e cada linha de cache armazena L_{cache} bytes, e dadas as taxas de acerto aos caches H_{L1} e H_{L2} , obtidas via execução instrumentada da versão sequencial, o total de bytes acessados por uma instrução de memória é dado por $F \times L_{word} + (1 - F) \times L_{cache}$. F é o fator de reuso espacial da aplicação e varia de 0, quando blocos acessados da memória são maiores do que L_{cache} , até 1, quando os acessos são contíguos e toda a linha de cache é usada. O total de bytes transferidos entre o cache de nível 2 e memória é então determinado pela equação 2.2.

$$L = I \times R \times (1 - H_{L1}) \times (1 - H_{L2}) \times [F \times L_{word} + (1 - F) \times L_{cache}] \quad (2.2)$$

Sendo $B_{sustained}$ a largura de banda do barramento em *bytes/s*, o tempo de comunicação T_{comm} é modelado como (eq. 2.3):

$$T_{comm} = \frac{I \times R \times (1 - H_{L1}) \times (1 - H_{L2}) \times [F \times L_{word} + (1 - F) \times L_{cache}]}{B_{sustained}} \quad (2.3)$$

Finalmente, o tempo total de execução é a soma dos tempos de execução e de comunicação, conforme eq. 2.4.

$$T = T_{comp} + T_{comm} \quad (2.4)$$

Para a estimativa de quantidade de *threads* sem gerar contenção, o modelo assume que as taxas de acerto de cache da aplicação não se alteram com o número de *cores* usados. O tempo de acesso é, portanto, independente da quantidade de *threads* e o tempo de execução diminui com a inclusão de *threads*. Desejando-se, por exemplo, que o tempo de computação seja ao menos 90% do tempo total, a quantidade n de *threads* a usar é dada pela eq. 2.5.

$$n < \frac{I \times (1 - R)}{9P_{sustained} \times T_{comm}} \quad (2.5)$$

Por outro lado, desejando-se limitar o tempo total de execução a uma constante C ($T < C$), tem-se a eq. 2.6. Neste caso, é necessário que $C > T_{comm}$.

$$n > \frac{I \times (1 - R)}{P_{sustained} \times (C - T_{comm})} \quad (2.6)$$

O modelo também foi expandido para considerar o efeito de busca antecipada dos dados. Embora consiga apresentar uma fórmula fechada para a determinação do número de *threads*, o modelo falha em supor que o tempo de acesso aos dados independe da quantidade de *cores* usados na execução, como discutido no capítulo 4.

2.4 Análise de dependência de dados

Nicolau e Kejariwal (2011) propuseram a determinação da quantidade de *threads* para *loops* considerando as dependências de dados para evitar vazios no *pipeline*. As dependências de dados, tanto em uma mesma iteração quanto entre iterações do *loop*, são representadas por um grafo acíclico direcionado, denotado por $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. Cada vértice $v \in V$ simboliza uma operação dentro do *loop*, e cada aresta $(i, j) \in E$ indica uma dependência de dados entre as operações v_i e v_j . As operações são reorganizadas e distribuídas entre as *threads*, de modo a evitar sincronizações. Esta técnica ignora as características de desempenho da memória e do processador, bem como eventual sobrecarga de trocas de contexto, tornando sua aplicação em diferentes sistemas computacionais questionável.

Capítulo 3

Granularidade dos acessos à memória

É sabido que o desempenho do subsistema de memória é um importante limitante para a quantidade de *threads* em um sistema *multicore*. A princípio, pode-se pensar que a restrição à paralelização, provocada pela competição pelo acesso à memória, se dê devido apenas à quantidade de *threads*. Contudo, diferenças de desempenho entre aplicações indicam uma oportunidade de melhor compreensão da escalabilidade através do estudo da forma como os dados são acessados, em especial da frequência dos acessos à memória. Este capítulo introduz, então, o estudo da granularidade e, em menor grau, da localidade dos acessos, com as metas da pesquisa listadas na seção 3.1 e a metodologia de investigação adotada indicada na seção 3.2.

3.1 Objetivos

A intenção de verificar a viabilidade de fornecer automaticamente dicas ao desenvolvedor de aplicações sobre possíveis limitações à paralelização guiou a definição de três objetivos para este trabalho:

Primeiro Avaliar o efeito da granularidade dos acessos à memória na escalabilidade de aplicações paralelizadas pelo modelo de paralelismo de dados em processadores *multicore*.

Segundo Verificar a possibilidade de uso de granularidade como um fator na sugestão automática de quantidade de *threads*.

Terceiro Avaliar o efeito da localidade na granularidade.

3.2 Metodologia de investigação

Esta pesquisa guiou-se pelas seguintes etapas: determinação de escalabilidade de aplicações, cálculo da granularidade dos acessos, e associação de granularidade com escalabilidade. Como objeto de estudo foram selecionados programas paralelizados com OpenMP no modelo de paralelismo de dados. Neste modelo de programação, as regiões paralelizáveis do programa, em especial os *loops*, são marcadas como tal no código-fonte através da inclusão de diretivas de compilação (*pragmas* na linguagem de programação C e comentários especiais em FORTRAN (OPENMP ARCHITECTURE REVIEW BOARD, 2008)), que instruem o compilador a gerar código paralelo com o uso de *threads*. Os programas escolhidos foram o conjunto *NAS Parallel Benchmarks*, ou NPB, na versão 3.3.1 OpenMP (JIN; FRUMKIN; YAN, 1999; FENG et al., 2004), largamente reconhecido como uma medida padrão de desempenho. *Benchmarks* são programas usados para se determinar desempenho de sistemas computacionais. Podem ser escritos do zero, especialmente adaptados ao aspecto que se deseja avaliar, ou então ser oriundos de aplicações reais. As aplicações de NPB foram criadas para

ajudar a avaliar o desempenho de supercomputadores paralelos. Os *benchmarks* são derivados de aplicações de dinâmica de fluidos computacional e consistem de cinco *kernels* e três pseudo-aplicações na especificação original “papel-e-lápis” (NPB 1). O conjunto foi estendido para incluir novos *benchmarks* para malhas adaptativas desestruturadas, entrada/saída paralela, aplicações com múltiplas zonas e grades computacionais. Tamanhos de problema em NPB são predefinidos e indicados como diferentes classes. Implementações de referência de NPB estão disponíveis em modelos de programação comumente usados, como MPI e OpenMP (NPB 2 e NPB 3) (NASA, 2013).

As aplicações de NPB têm seu código-fonte disponível gratuitamente com uma licença de uso permissiva. O conjunto é formado por dez programas:

BT (Block Tridiagonal): Calcula equações tridimensionais de Navier-Stokes. A solução por diferenças finitas do problema é baseada numa fatorização aproximada que desacopla as três dimensões e aplica a resolução sequencialmente em cada uma.

CG (Conjugate Gradient): Calcula uma aproximação do menor autovalor de uma matriz grande e esparsa, testando computações e comunicações não estruturadas com posições de entradas da matriz geradas aleatoriamente.

DC (Data Cube): Testa a capacidade de manipular grandes conjuntos de dados distribuídos em ambiente de *grid*.

- EP (Embarrassingly Parallel):** Gera pares de desvios gaussianos aleatórios. Referência para máximo desempenho, requer pouca comunicação entre *threads*.
- FT (Fourier Transform):** Solução para uma equação diferencial parcial 3D através da transformada rápida de Fourier.
- LU (Lower-Upper):** Soluciona um sistema de equações resultante da discretização de equações de Navier-Stokes pela sua quebra em sistemas triangulares com o método de Gauss-Seidel.
- IS (Integer Sort):** Ordenação de números inteiros.
- MG (Multigrid):** Soluciona equações de Poisson tridimensionais escalares. Atua sobre um conjunto de grades alternantes entre grossa e fina e testa movimentações de dados de curta e de longa distâncias.
- SP (Scalar Pentadiagonal):** Similar a BT, porém usa outra fatoração, resultando em sistemas de equações escalares pentadiagonais, também resolvidas em cada dimensão.
- UA (Unstructured Adaptive):** Modela transferências de calor e apresenta padrões de acesso à memória irregulares e continuamente se alterando.

Os tamanhos de problema são chamados de classes. Neste trabalho, foram utilizados os tamanhos padrão de teste, representados pelas classes A, B e C. Nestes casos, o problema aumenta cerca de quatro vezes entre uma classe e a seguinte. Os tamanhos de problema maiores, das classes D, E e F, têm fator de aumento de 16 vezes entre cada, elevando consideravelmente o tempo de execução dos programas. Como os *benchmarks* de NPB consistem de repetições de um grande *loop* principal, cujo número de iterações também aumenta com a classe, optou-se por evitar execuções demasiadamente longas que, aparentemente, não alterariam as características que se desejava observar nas execuções.

O *benchmark* DC não foi incluído neste estudo por referenciar dados em arquivos em disco, cujo tempo de acesso é muito superior ao de dados na memória principal, ao passo que IS foi excluído por apresentar erros nas execuções.

Adicionalmente a NPB, foram testados também o *benchmark* de memória STREAM, na versão 5.10, e uma multiplicação de matrizes. STREAM testa o desempenho de quatro operações sobre *arrays* de números reais (MCCALPIN, 1995), como mostrado na tabela 3.1. O tamanho de cada *array* é definido como sendo maior do que quatro vezes a capacidade do maior cache da máquina. MultMat, um programa de multiplicação de matrizes, foi

desenvolvido em Fortran para calcular o produto de duas matrizes com entradas reais. O tamanho das matrizes também foi ajustado para exceder a capacidade do último nível de cache em todas as máquinas usadas. MultMat implementa a multiplicação de matrizes conforme o algoritmo 1.

Algoritmo 1: Algoritmo de multiplicação de matrizes usado no programa MultMat.

Entrada: matrizes A (linhas de A \times colunas de A), B (linhas de B \times colunas de B) e C (linhas de A \times colunas de B)

Saída: C, contendo o resultado da multiplicação de A por B

para $i = 1$ até linhas de A **faça**

para $j = 1$ até colunas de B **faça**

 soma = 0;

para $k = 1$ até colunas de A **faça**

 soma = soma + $a(i,k) \times b(k,j)$;

fim para

$c(i,j) =$ soma;

fim para

fim para

Para cada aplicação selecionada, foram realizadas várias execuções com diversas quantidades de *threads*, diferentes tamanhos de problema e em diferentes sistemas computacionais para determinar sua escalabilidade.

O desempenho dos sistemas computacionais testados foi caracterizado pela velocidade de processamento, medida em MFLOPS, e pela largura de banda entre processador e memória, medida em MB/s. MFLOPS (*Megafllops*, milhões de operações de ponto flutuante por segundo) é uma medida de velocidade de processamento tipicamente usada no âmbito de computação científica, onde há uso intensivo de cálculos com números reais, também chamados de números de ponto flutuante. A velocidade de processamento em MFLOPS foi obtida pelo *benchmark* Linpack (DONGARRA, 1988; TOY; DONGARRA, 1988), o mesmo usado pelo projeto TOP500 para eleger os 500 supercomputadores mais poderosos do planeta (MEUER et al., 2013) através da resolução de um sistema denso de equações lineares. Embora a medida obtida refira-se apenas ao problema resolvido por Linpack e não indique necessariamente o desempenho total do sistema, seu valor foi usado para auxiliar na comparação entre as diferentes máquinas utilizadas. Já a largura de banda, indicada em *megabytes* por segundo (MB/s), foi determinada pelo *benchmark* STREAM.

Tabela 3.1: As quatro operações realizadas por STREAM.

nome	operação	bytes	FLOPS
COPY	$a(i) = b(i)$	16	0
SCALE	$a(i) = q \times b(i)$	16	1
SUM	$a(i) = b(i) + c(i)$	24	1
TRIAD	$a(i) = b(i) + q \times c(i)$	24	2

Conforme apresentado na seção 1.3.2, granularidade indica a proporção de acessos à memória em relação ao processamento. Duas medidas de granularidade foram adotadas neste trabalho, ambas inspiradas pela definida por Callahan, Cocke e Kennedy (1988). A primeira, *granularidade de loop*, simbolizada por γ_L , aponta a relação entre acessos a variáveis e operações de ponto flutuante em um *loop* paralelo, como definido no código-fonte da aplicação; a segunda, *granularidade efetiva*, representada por γ_E , determina a razão real entre acessos à memória e instruções de ponto flutuante executadas, medidos durante a execução da aplicação. Assim, após a determinação da escalabilidade do programas selecionados, o código-fonte das aplicações foi inspecionado para se calcular a granularidade dos acessos. Para cada *loop* paralelo, a granularidade de *loop* γ_L foi calculada pela equação 3.1.

$$\gamma_L = \frac{\text{número de acessos a arrays}}{\text{número de operações de ponto flutuante}} \quad (3.1)$$

Pela definição de γ_L na equação 3.1, uma aplicação que faz muitos acessos a dados em relação ao processamento possui valores **maiores** de γ_L . Já uma aplicação que faz poucos acessos a dados em proporção ao processamento tem valores **menores** de γ_L , o que é o inverso da definição de granularidade usualmente utilizada, por exemplo, em sistemas com memória distribuída; nestes casos, granularidade indica a proporção de processamento em relação à comunicação.

Foram considerados apenas acessos a dados em *arrays*, pois estes tipicamente são armazenados na memória principal e não nos registradores da CPU, como pode ocorrer com variáveis escalares; além disso, as aplicações estudadas manipulam dados majoritariamente em *arrays*. O processamento é representado por operações aritméticas de ponto flutuante, já que os programas de NPB utilizados lidam principalmente com números reais (exceto, naturalmente, para controles como índices de *arrays* e iterações de *loops*). No contexto deste trabalho, *loops* aninhados foram considerados como um único, uma vez que a maioria dos *loops* de NPB concentra suas operações no nível mais interno.

A granularidade efetiva média γ_E foi calculada pela equação 3.2. Devido à existência

do mecanismo de cache, cada acesso à memória transfere uma linha (ou *bloco*) de cache completa. Como as aplicações operam sobre palavras de dados reais de precisão dupla de b bytes cada e uma linha de cache pode conter n de tais palavras, foi necessária a divisão por n na equação 3.2. Assim, γ_E indica a taxa média de palavras de b bytes acessadas na memória para cada operação de ponto flutuante durante toda a execução do programa.

$$\gamma_E = \frac{\text{total de acessos à memória}}{\text{total de instruções de ponto flutuante executadas} \cdot n} \quad (3.2)$$

No cálculo de granularidade foram consideradas apenas instruções que agem sobre dados do tipo ponto flutuante, ao invés do total de instruções, já que as aplicações têm quantidades diferentes de operações de lógica e de aritmética, manifestadas, por exemplo, em diferentes níveis de aninhamento de *loops* e diferentes deslocamentos (*offsets*) em acessos a *arrays*. Também foi considerado, para fins de simplificação, que as operações de ponto flutuante concluem todas com a mesma duração.

O total de faltas no último nível de cache e o total de instruções de ponto flutuante executadas são medidos durante a execução das aplicações pela unidade de monitoramento de desempenho, um componente embutido da maioria dos processadores. Esses dados podem ser acessados com o auxílio de uma ferramenta de análise dinâmica de aplicações. Registradores contadores medem a quantidade de certos eventos gerados pela aplicação durante sua execução, como, por exemplo, o total de instruções executadas.

Os eventos monitoráveis variam entre arquiteturas e microarquiteturas. No modo de operação utilizado, *event-based sampling* (amostragem baseada em eventos), a ocorrência de certa quantidade de eventos provoca a geração de uma interrupção no processador. A rotina de tratamento de interrupção, fornecida pela ferramenta de análise, registra o contexto da execução, incluindo o valor do contador de programa e as identificações do processo e da *thread*, associando, assim, os eventos com a aplicação causadora. Cada experimento pode exigir várias execuções da aplicação para coletar todos os eventos escolhidos, uma vez que a quantidade de contadores do *hardware* é menor do que a de eventos disponíveis e cada evento é registrado por um contador específico.

As contagens de eventos obtidas pela técnica de *event-based sampling* permitiram a determinação da taxa de uso do barramento de memória, das taxas de faltas nos demais níveis de cache e granularidade efetiva para todas as versões dos programas testados.

Após a determinação das escalabilidades das aplicações e de suas granularidades, associações foram estabelecidas entre escalabilidade, taxa de uso de barramento de memória

e granularidade, tanto efetiva quanto de *loop*. Algumas aplicações foram submetidas a variações em suas granularidades, com consequências importantes para a escalabilidade. Variações na localidade foram em seguida experimentadas para se avaliar os efeitos no comportamento de *speedup*.

Finalmente, considerações sobre o uso automático da granularidade de *loop* na sugestão de quantidade de *threads* foram apresentadas.

Capítulo 4

Escalabilidade de aplicações OpenMP com paralelismo de dados

Para ser possível avaliar os efeitos da granularidade no desempenho das aplicações, é preciso primeiramente observar sua escalabilidade, ou seja, sua capacidade de aproveitar a maior disponibilidade de recursos. Este capítulo apresenta, então, uma análise do comportamento dos programas selecionados, a saber NPB, STREAM e MultMat, em alguns sistemas computacionais com diferentes graus de paralelismo. Uma breve descrição dos computadores utilizados é dada na seção 4.1, enquanto as métricas adotadas e ferramentas de medida são relacionadas na seção 4.2. A seção 4.3 discorre brevemente sobre a quantidade máxima possível de *threads*. A análise de escalabilidade é desenvolvida na seção 4.4 para as aplicações de NPB, na seção 4.5 para o *benchmark* STREAM e na seção 4.6 para o programa de multiplicação de matrizes. As conclusões são indicadas na seção 4.7.

4.1 Hardware utilizado

As aplicações analisadas foram executadas em quatro computadores:

- SGI, com dois processadores Intel Xeon E5420 de quatro *cores* cada em uma configuração UMA, totalizando 8 *cores*, operando a 2,50 GHz, com 64 KB de cache L1 (32 KB para instruções e 32 KB para dados) privativos a cada core, 6 MB de cache L2, compartilhados por pares de *cores*, 16 GB de RAM e executando CentOS Linux 5.7;
- HP, com dois processadores Intel Xeon E5649 de seis *cores* cada operando a 2,53

GHz em uma configuração UMA, totalizando 12 *cores*, com 64 KB de cache L1 (32 KB para instruções e 32 KB para dados), privativos a cada core, 256 KB de cache L2, privativos para cada *core*, 12 MB de cache L3, compartilhados por todos os *cores*, 70 GB de RAM e executando Ubuntu Linux 12.04;

- AMD, com um processador AMD Phenom II X4 940 de 4 *cores* operando a 3 GHz em uma configuração UMA, com 1 MB de cache L1 (512 KB para dados e 512 KB para instruções), privativo a cada *core*, 512 KB de cache L2, também privativos a cada *core*, 6 MB de cache L3, compartilhados por todos os *cores*, 4 GB de RAM e executando Rocks Linux 5.2;
- MTL, com 4 processadores Intel Xeon E7-4860 de dez *cores* cada operando a 2,27 GHz em uma configuração NUMA, totalizando 40 *cores*, com 64 KB de cache L1 (32 KB para dados, 32 KB para instruções) privativos a cada *core*, 256 KB de cache L2 privativos a cada *core* e cache L3 compartilhado por todos os *cores* com 24 MB de capacidade.

4.2 Métricas e ferramentas

Os programas analisados foram compilados pelo GNU Compiler Collection Fortran (FREE SOFTWARE FOUNDATION, 2013) na versão 4.1.2 em SGI, AMD e MTL, e 4.6.2 em HP. As limitações para a escalabilidade apresentadas nas seções 4.4, 4.5 e 4.6 se manifestaram apenas quando otimizações foram ativadas na compilação. O nível de otimização 3, ativado pela opção `-O3` do compilador GCC, resultou nos menores tempos de execução, fazendo com que a opção `-O3` fosse utilizada na compilação de todos os programas analisados. O tempo total de execução (*elapsed time*), necessário para a determinação de *speedup* e eficiência, o tempo de CPU usado pelo sistema (*system time*) e o tempo de CPU (*user time*) usado pelo processo avaliado foram registrados com GNU time 1.7. Os eventos de *hardware*, necessários para a determinação das taxas de uso de barramento e de faltas nos caches, foram obtidas com o auxílio da ferramenta de análise dinâmica de aplicações Intel VTune 9.1 (MALLADI, 2009), disponível apenas na máquina SGI. A tabela 4.1 lista os eventos de *hardware* observados. A largura de banda de memória dos sistemas computacionais foi determinada pelo programa STREAM 5.10 (MCCALPIN, 1995), ao passo que a velocidade em MFLOPS dos *cores* foi estimada pelo *benchmark* Linpack (TOY; DONGARRA, 1988).

Tabela 4.1: Eventos de hardware observados com a ferramenta VTune para execuções no sistema SGI.

<i>nome</i>	<i>descrição</i>
CPU_CLK_UNHALTED.CORE	Quantidade de ciclos de <i>clock</i>
CPU_CLK_UNHALTED.BUS	Quantidade de ciclos do barramento
BUS_TRANS_ANY.ALL_AGENTS	Quantidade de transações no barramento frontal iniciadas por qualquer agente
INST_RETIRED.ANY	Quantidade de instruções executadas
L1D_REPL	Quantidade de linhas de cache trazidas para L1
L2_LINES_IN.SELF.ANY	Quantidade de linhas de cache trazidas para L2
X87_OPS_RETIRED.ANY	Quantidade de instruções X87 em ponto flutuante executadas
SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE	Quantidade de instruções SSE2 em escalares <i>double</i> executadas

4.3 Número máximo de threads

Um *benchmark* sintético foi criado em OpenMP para se iniciar a investigação de escalabilidade, consistindo de dois *loops* paralelos com diversos cálculos, separados por uma barreira. A quantidade máxima possível de *threads* depende do sistema operacional, da biblioteca de *threads*, da quantidade de memória principal disponível e da quantidade de dados locais a cada *thread*. No sistema de teste SGI, a aplicação executa com no máximo 32.288 *threads*. As execuções com 8 até cerca de 20.000 *threads* apresentam um desempenho semelhante, indicando que a sobrecarga de trocas de contexto mais numerosas não foi significativa. A partir de cerca de 20.000 *threads* há redução progressiva de desempenho, chegando a um aumento de 26% do tempo de execução na configuração de máximo número de *threads*.

4.4 NPB

4.4.1 Speedup

Os programas de NPB foram executados nos quatro sistemas computacionais. As figuras 4.1, 4.3 e 4.5 mostram as curvas de *speedup* dos programas em SGI para as classes A, B e C, respectivamente, calculadas pela equação 1.1, ao passo que as figuras 4.2,

4.4 e 4.6 mostram as curvas de eficiência correspondentes, calculadas pela equação 1.2. Em todas as figuras deste trabalho, quantidade de *threads* igual a **um** indica a versão sequencial da aplicação. Para o cálculo de *speedup* e eficiência foram utilizados os tempos de execução (*elapsed time*) médios de 20 execuções, que totalizaram 442 horas. Os desvios absolutos médios de tempo de execução para cada aplicação em cada quantidade de *threads* e em cada classe foram inferiores a 8%. Os desvios padrão do tempo decorrido em cada execução são apresentados no apêndice A, nas tabelas A.1, A.2 e A.3. Os tempos de execução típicos foram menores do que um minuto na classe A, com programas que concluíram em poucos segundos; menores do que cinco minutos na classe B e menores do que quinze minutos na classe C. As figuras mostram que as aplicações, tipicamente, atingem o melhor desempenho com oito *threads*, que é a quantidade de *cores* disponíveis. A aplicação EP, por ser embaraçosamente paralela, apresenta um *speedup* linear, diretamente proporcional à quantidade de *threads*, e permite ganhos contínuos com o uso de mais *cores*. As demais aplicações têm um ritmo menor de crescimento de *speedup*, e algumas deixam de apresentar melhorias antes de a quantidade de *threads* igualar à de *cores*, como MG e SP. A estagnação da curva de *speedup* causa quedas expressivas na eficiência das execuções. Nenhum caso apresenta melhoria de desempenho com mais *threads* do que *cores*; em geral, há piora no desempenho devido às trocas de contexto mais numerosas, resultando em quedas acentuadas na eficiência das execuções. Até mesmo EP apresenta reduções, atingindo eficiência inferior a 40% a partir de 24 *threads*. A maior diminuição de *speedup* a partir de 8 *threads* ocorre com LU, que registra quedas de cerca de 80% nas classes A e B e de 60% na classe C.

Conforme aumenta-se o tamanho do problema, indo-se da classe A para B e depois para C, torna-se mais estável o comportamento de *speedup* dos *benchmarks*, exigindo mais *threads* para haver quedas equivalentes no *speedup*. Isso mostra que, para NPB, os problemas menores são mais sensíveis à quantidade de *threads*, possivelmente associados à maior sobrecarga do paralelismo, dados os curtos períodos de execução.

As quedas no *speedup* que se manifestaram em execuções com 10 *threads* se devem à política de escalonamento OpenMP utilizada. No compilador GCC, a política de escalonamento de iterações do *loop* pelas *threads*, quando não especificada no código-fonte, é adotada como sendo *static* (FREE SOFTWARE FOUNDATION, INC., 2006b). Neste caso, a distribuição de iterações do *loop* pelas *threads* é realizada durante a compilação, com cada *thread* recebendo no máximo um grupo contíguo de iterações (*chunk*). Os grupos são aproximadamente do mesmo tamanho. As políticas de escalonamento *dynamic* e *guided*, por outro lado, distribuem os grupos de iteração para as *threads* conforme estas solicitam.

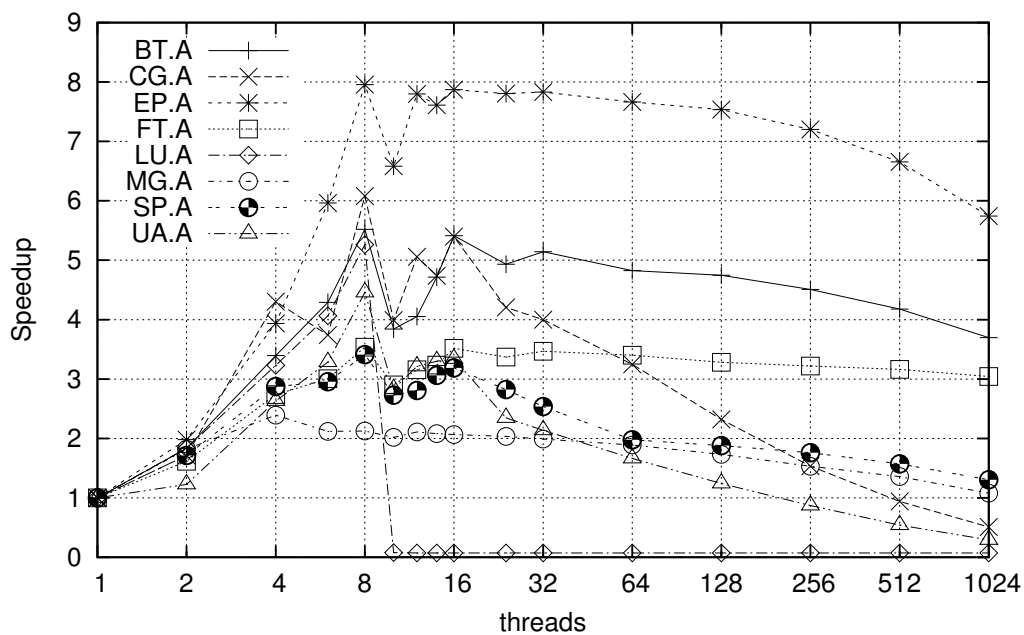


Figura 4.1: Curvas de *speedup* para NPB classe A na máquina SGI.

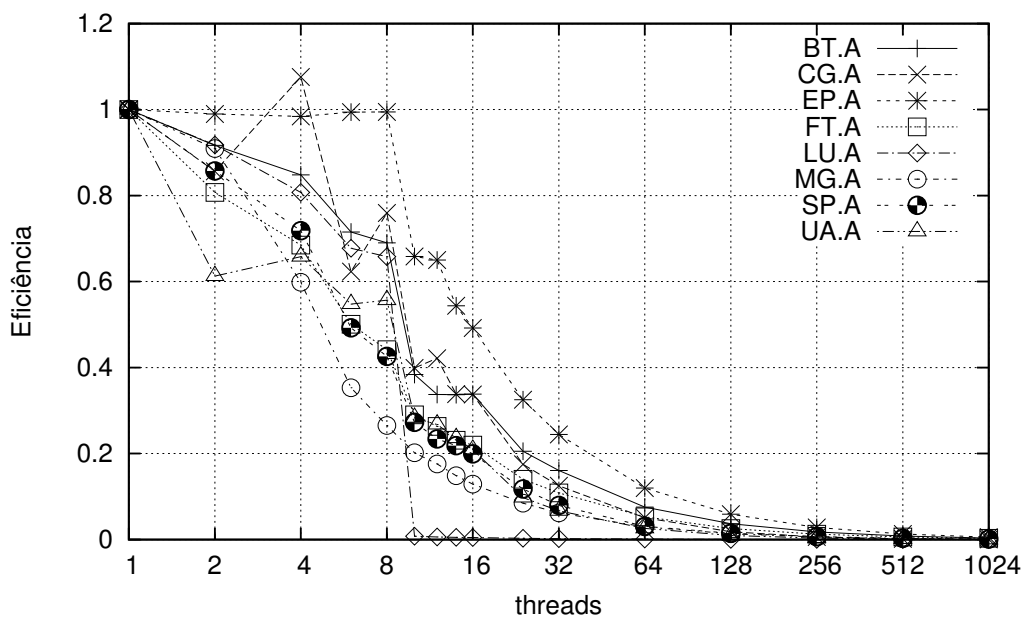


Figura 4.2: Curvas de eficiência para NPB classe A na máquina SGI.

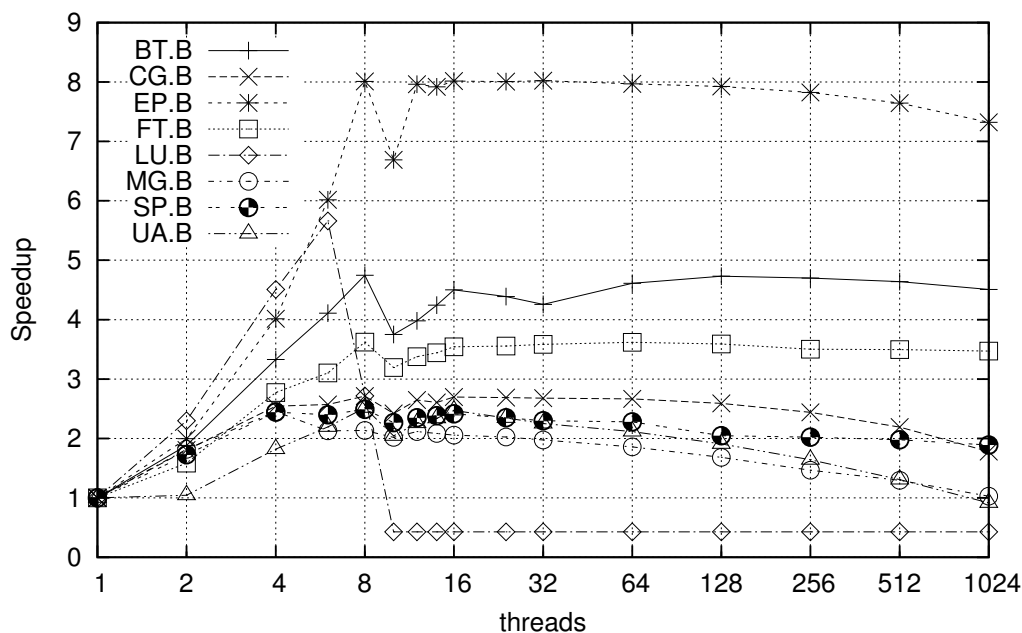


Figura 4.3: Curvas de *speedup* para NPB classe B na máquina SGI.

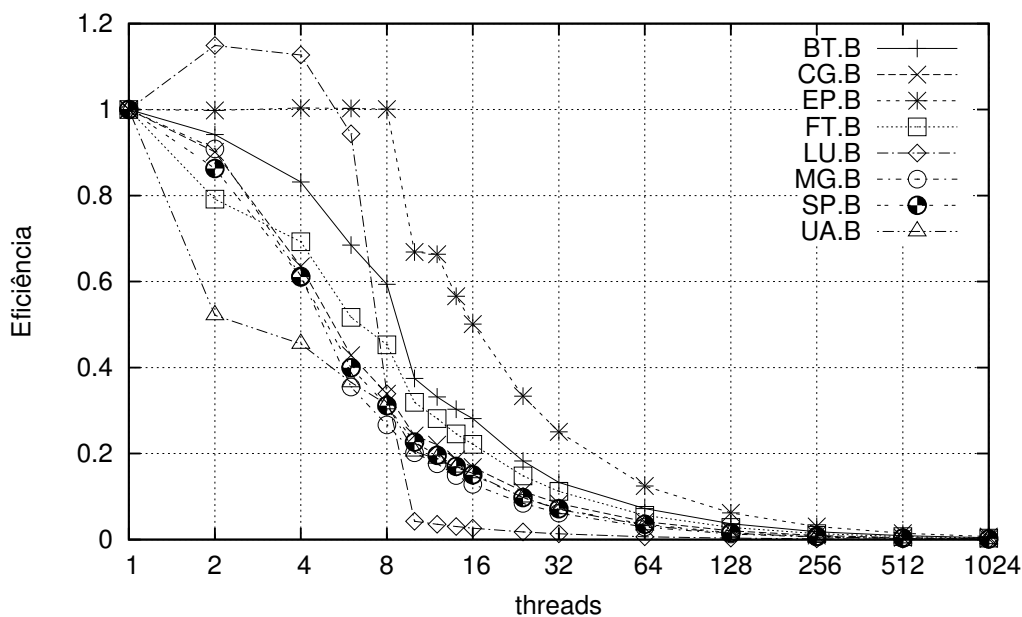


Figura 4.4: Curvas de eficiência para NPB classe B na máquina SGI.

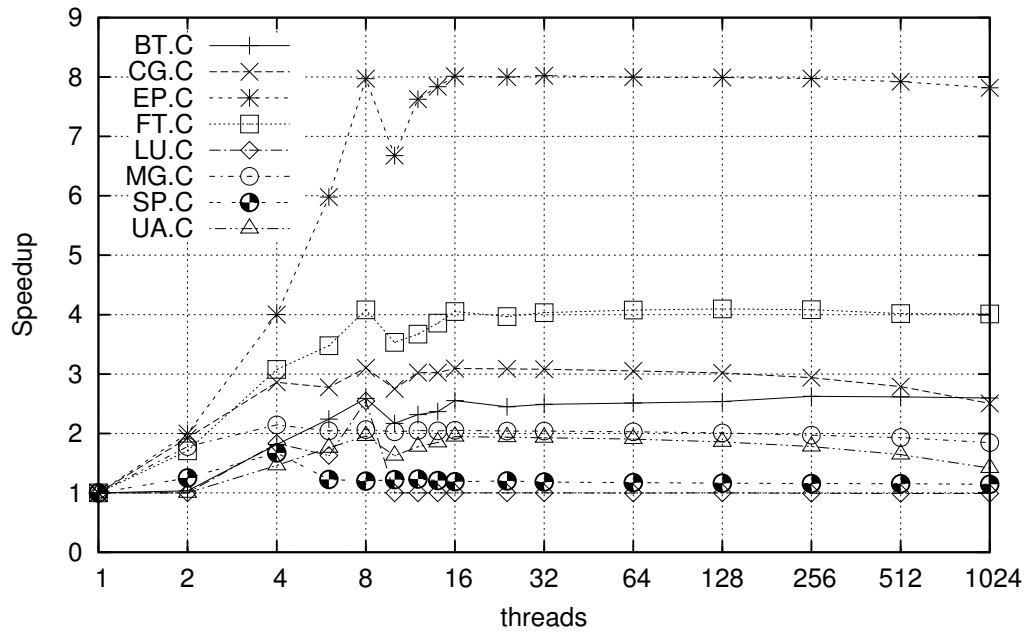


Figura 4.5: Curvas de *speedup* para NPB classe C na máquina SGI.

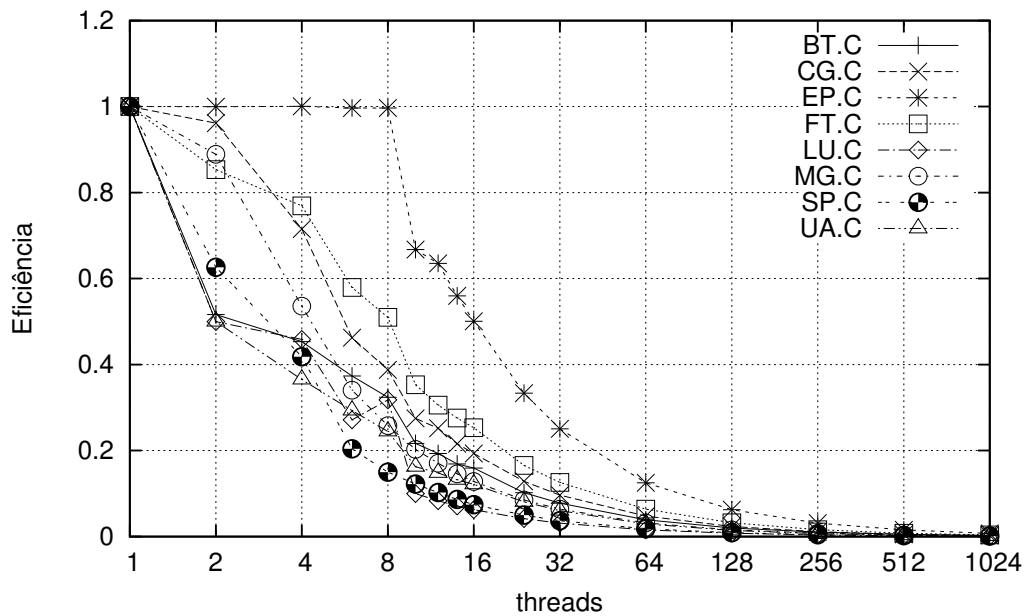


Figura 4.6: Curvas de eficiência para NPB classe C na máquina SGI.

Em *dynamic*, os grupos de iteração têm o mesmo tamanho, exceto, possivelmente, pelo último; já em *guided*, “o tamanho de cada grupo de iteração é proporcional ao número de iterações não atribuídas dividido pelo número de *threads*” (OPENMP ARCHITECTURE REVIEW BOARD, 2008). Em execuções em que a política de escalonamento foi ajustada para *dynamic*, a referida queda no *speedup* com dez *threads* não se manifestou e o desempenho nos demais graus de paralelismo foi semelhante às execuções com *static*, indicando um possível desbalanceamento de carga de trabalho entre as *threads* quando escalonamento *static* é utilizado. Os resultados para a escalonamento *guided* foram semelhantes aos de *static*. Como a escolha de política de escalonamento OpenMP exige alteração no código-fonte da aplicação e afetou apenas as execuções com mais *threads* do que *cores*, todos os resultados apresentados neste trabalho foram obtidos com a política de escalonamento padrão, *static*.

O comportamento das aplicações pode ser melhor compreendido observando-se seus efeitos nos diversos componentes do sistema computacional. Os eventos de *hardware* causados pelas aplicações foram medidos com VTune e registrados para compor as métricas a seguir. As aplicações foram observadas novamente nos três tamanhos de problema e nas mesmas quantidades de *threads*, com exceção de LU, que gerou eventos em quantidade excessiva a partir de dez *threads*, causando interrupções para acesso à unidade de monitoramento em intervalos menores do que 1 ms e impossibilitando o prosseguimento da coleta. O apêndice A traz os desvios padrão para as medições.

4.4.2 Taxa de uso de barramento

A taxa de acesso ao barramento frontal de NPB em SGI em relação ao tempo de execução, calculada conforme equação 4.1, é mostrada na figura 4.7 para a classe A, na figura 4.8 para a classe B e na figura 4.9 para a classe C.

$$\text{taxa de uso do barramento (\%)} = \frac{100 \times \text{BUS_TRANS_ANY_ALL_AGENTS}}{\text{CPU_CLK_UNHALTED_BUS}} \quad (4.1)$$

EP apresentou os menores usos de barramento e os maiores *speedups* nos três tamanhos de problema. Na classe A (figura 4.7), MG e SP fazem maior uso do barramento, acima de 60%, em geral, e apresentam os menores *speedups*; MG, que apresenta uso de barramento superior a 80%, piora seu desempenho em execuções com mais de quatro *threads*. FT tem menor uso de barramento do que SP, no entanto seus *speedups* são parecidos. BT, LU e

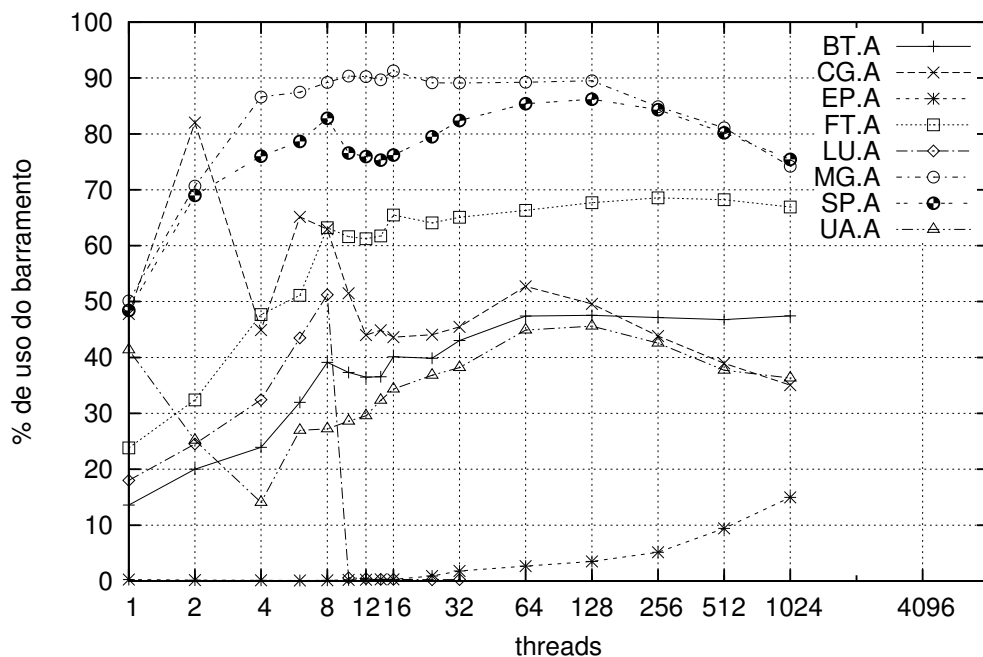


Figura 4.7: Taxas de uso de barramento de memória para NPB classe A na máquina SGI.

UA acessam o barramento em no máximo 50% do tempo, sendo menos de 40% em BT e UA, e conseguem boas melhorias no desempenho com maior grau de paralelismo.

Na classe B, figura 4.8, CG, MG e SP continuaram acessando mais frequentemente o barramento, impedindo melhorias em seus *speedups* com grau de paralelismo maior do que quatro *threads*; coincidentemente, o uso de barramento foi superior a 60%. A curva de *speedup* de UA fica abaixo das curvas de SP e CG na figura 4.3, mas sua menor taxa de uso de barramento relaciona-se com a manutenção do ritmo de melhora de desempenho em quatro, seis e oito *threads*.

As curvas de uso de barramento na classe C (figura 4.9) são semelhantes às da classe B, porém com valores mais acentuados. Todas as aplicações excedem 50% de uso de barramento com oito *threads*. Taxas de acessos ao barramento superiores a 60% para CG, SP, MG e UA combinam com a interrupção de ganhos de desempenho com mais de quatro *threads*. O ritmo de crescimento do *speedup* de FT (figura 4.5), calculado como a diferença do *speedup* dividida pela diferença na quantidade de *threads*, era de 0,68 no intervalo de 2 a 4 *threads*, mas diminuiu para 0,20 em seis *threads* e 0,30 em oito *threads*, coincidindo com o uso de barramento superior a 60% a partir de quatro *threads*. O ritmo de crescimento de *speedup* de UA não sofreu variação significativa, mas seu baixo valor de

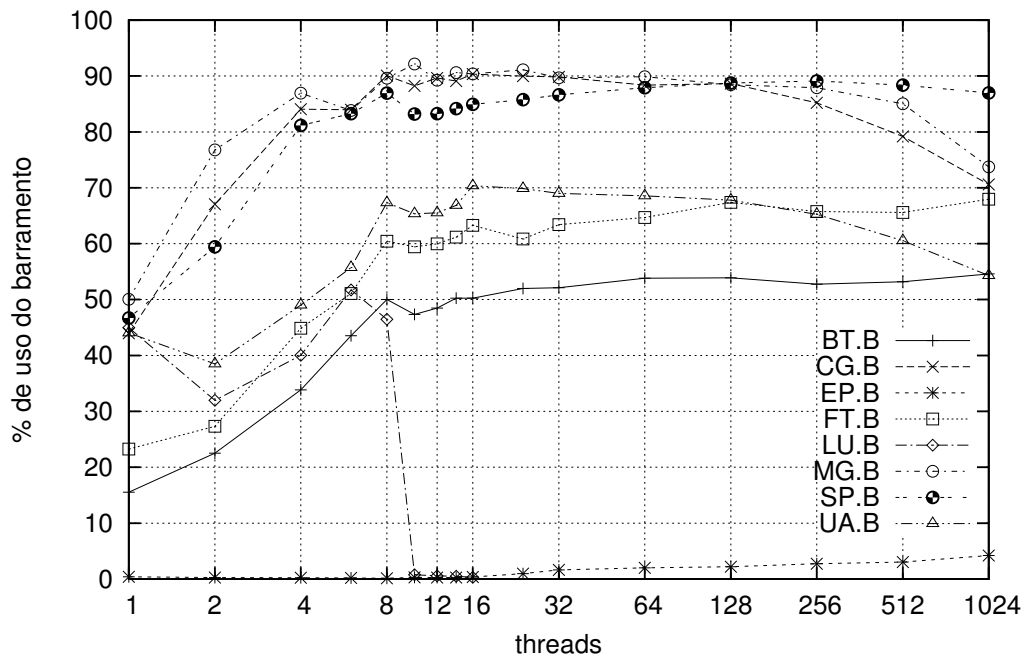


Figura 4.8: Taxas de uso de barramento de memória para NPB classe B na máquina SGI.

speedup máximo e *speedups* menores com o aumento do tamanho do problema concorda com o aumento progressivo de uso de barramento entre as classes A, B e C.

Em todos os casos, até mesmo EP aumenta seu uso de barramento quando há mais *threads* do que *cores*. Segundo Malladi (2009), medições internas feitas pelo fabricante da CPU utilizada apontam que a latência de acesso à memória aumenta rapidamente a partir de 60% de uso do barramento e cresce sem limite a partir de 70%, confirmando os resultados de desempenho das aplicações e justificando a associação de seu comportamento com o uso de barramento. A taxa de uso do barramento de memória mostra-se, então, como uma limitação para a escalabilidade das aplicações observadas.

4.4.3 Total de instruções executadas

Na paralelização pelo modelo de paralelismo de dados, o volume total de processamento é o mesmo independentemente do grau de paralelismo, pois a maior quantidade de *threads* implica menor carga de trabalho para cada uma. Assim, não se espera variação significativa no total de instruções executadas. De fato, para as execuções com até oito *threads*, o total de instruções foi similar, com pequeno incremento em relação à versão

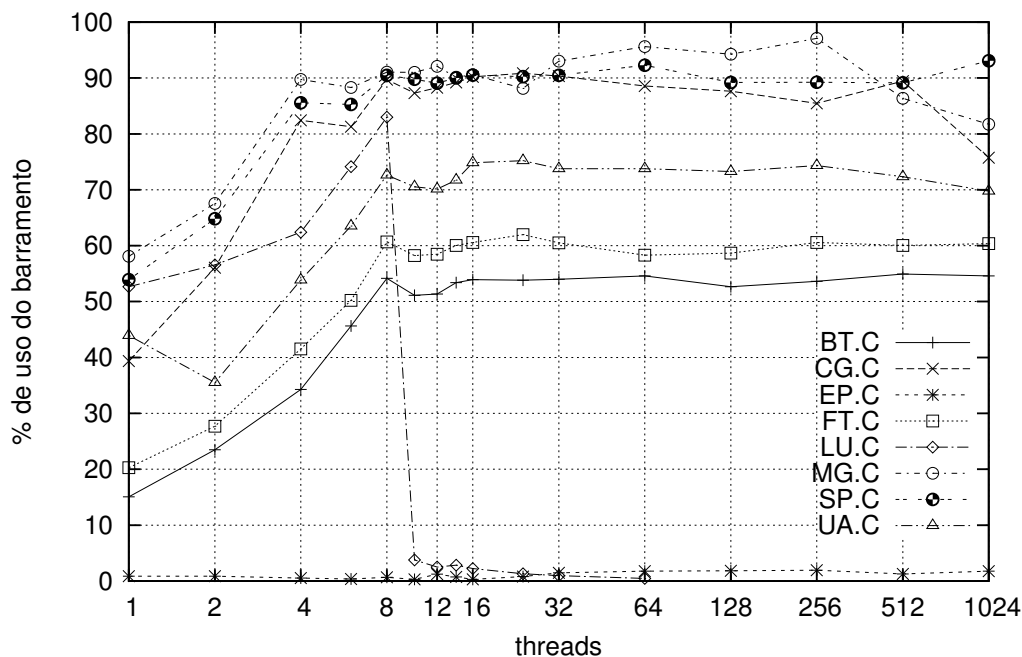


Figura 4.9: Taxa de uso de barramento de memória para NPB classe C na máquina SGI.

sequencial devido às operações de gerenciamento de *threads*. Já o desvio absoluto médio do total de instruções entre as versões sequencial e paralela com dois até 1024 *threads* na classe A foi inferior a 1,4 % para BT, EP e FT; inferior a 5,8% para MG e SP; inferior a 29% para CG e UA; e de 96,8% para LU. Em CG e UA, os maiores aumentos na quantidade de instruções executadas ocorreram com mais de 32 *threads*, enquanto em LU ocorreram já a partir de dez *threads*, resultando em um aumento de duas ordens de magnitude. Nos três programas, o incremento no total de instruções ocorreu com mais *threads* do que *cores*. Na classe B, o desvio médio foi de 9,7% para UA, 6,9% para CG, 89% para LU e inferior a 5% para BT, EP, FT, MG e SP. Novamente, a grande variação em LU ocorreu a partir de dez *threads*. Já na classe C, foram observadas diferentes variações no total de instruções executadas em relação à versão sequencial, excedendo 80% em BT, LU e UA, e igual a 46% em MG, porém inferior a 8,5% em CG, EP, FT e SP.

4.4.4 Tempo de CPU

Embora no intervalo de grau de paralelismo entre sequencial e quantidade de *cores* a contagem de instruções executadas seja similar, o tempo total de processamento teve

outro comportamento. Os gráficos das figuras 4.10, 4.11 e 4.12 mostram o tempo de CPU consumido por todas as *threads* da aplicação (*user time*) de NPB em SGI para as classes A, B e C, respectivamente, em relação à versão sequencial. As aplicações MG, SP e FT sofreram uma duplicação no tempo de CPU na classe A devido a este efeito. No tamanho de problema B, os maiores aumentos foram para os *benchmarks* CG, LU, MG, SP e UA. FT sofreu um aumento médio, ao passo que BT exibiu um aumento pequeno. Finalmente, na classe C houve no mínimo uma duplicação do tempo de CPU, com aplicações apresentando aumentos maiores. SP, por exemplo, teve seu tempo multiplicado por oito (não mostrado no gráfico) nas execuções com oito *threads*. MG e SP apresentam grandes aumentos, enquanto o tempo de CPU de EP é essencialmente inabalável, sofrendo um ligeiro aumento somente na classe A com 1024 *threads*.

Os gráficos mostram, para cada quantidade nt de *threads*, o fator de aumento $x(nt)$ do tempo de CPU $t_{CPU}(nt)$ em relação ao tempo sequencial t_s : $t_{CPU}(nt) = t_s \times x(nt)$. Como $t_{CPU}(nt) \div nt = t_p(nt)$, então a equação 4.2 permite calcular o *speedup* (S) a partir das figuras 4.10, 4.11 e 4.12. A aplicação EP, por exemplo, apresenta o mesmo tempo de CPU usando até 256 *threads*, com $x(nt) = 1$. Seu *speedup* com oito *threads* em qualquer das três classes é $S(8) = 8 \div 1 = 8$. Já FT, por outro lado, tem fator de aumento de tempo de CPU com 8 *threads* igual a 2,176 na classe B; seu *speedup* na mesma situação é $S(8) = 8 \div 2,176 \cong 3,68$.

$$S(nt) = \frac{t_s}{t_p(nt)} = \frac{t_s}{\frac{t_s \times x(nt)}{nt}} = \frac{nt}{x(nt)} \quad (4.2)$$

Os gráficos sugerem que, embora o total de instruções executadas seja essencialmente o mesmo, o maior uso de barramento causa aumento no tempo de CPU, possivelmente gerando ciclos ociosos em que as instruções aguardam a chegada dos dados da memória principal para o cache.

4.4.5 Taxa de faltas no cache

Com relação ao uso de cache, foram observadas as taxas de faltas (*misses*) nos níveis 1 (L1) e 2 (L2). As figuras 4.13, 4.14 e 4.15 mostram a taxa de faltas no cache de dados de L1, calculada pela equação 4.3 como o quociente entre o total de linhas de cache trazidas para o nível 1 e o total de instruções executadas. Conforme discutido por Diamond et al. (2011), a taxa de faltas em L1 sozinha não é um bom indicativo de problemas com a escalabilidade. De fato, a ordem dos programas em relação à taxa de faltas em cada um

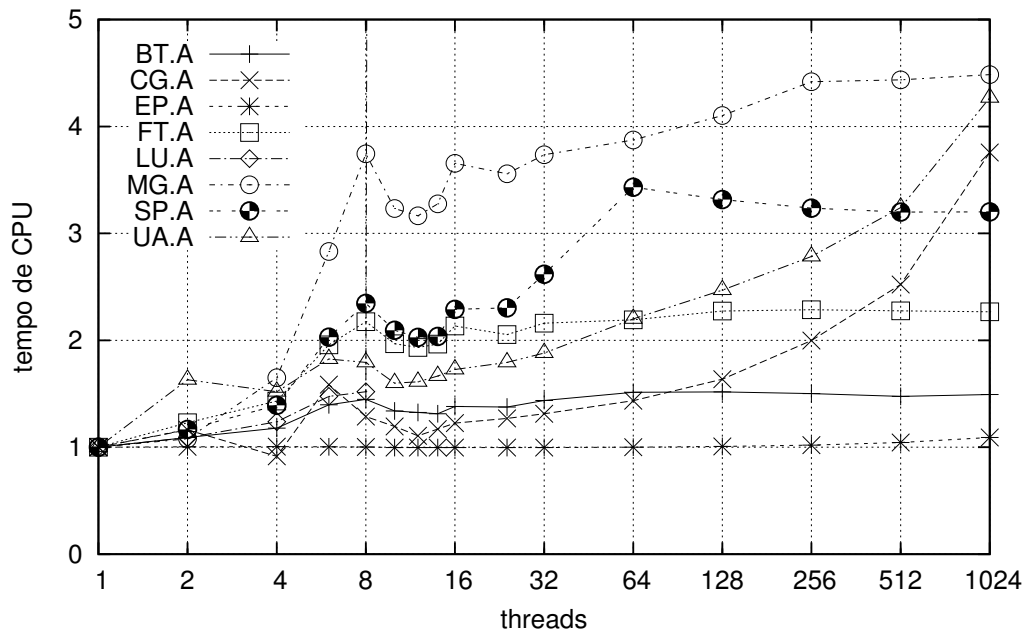


Figura 4.10: Tempos de CPU para NPB classe A na máquina SGI (em relação à versão sequencial).

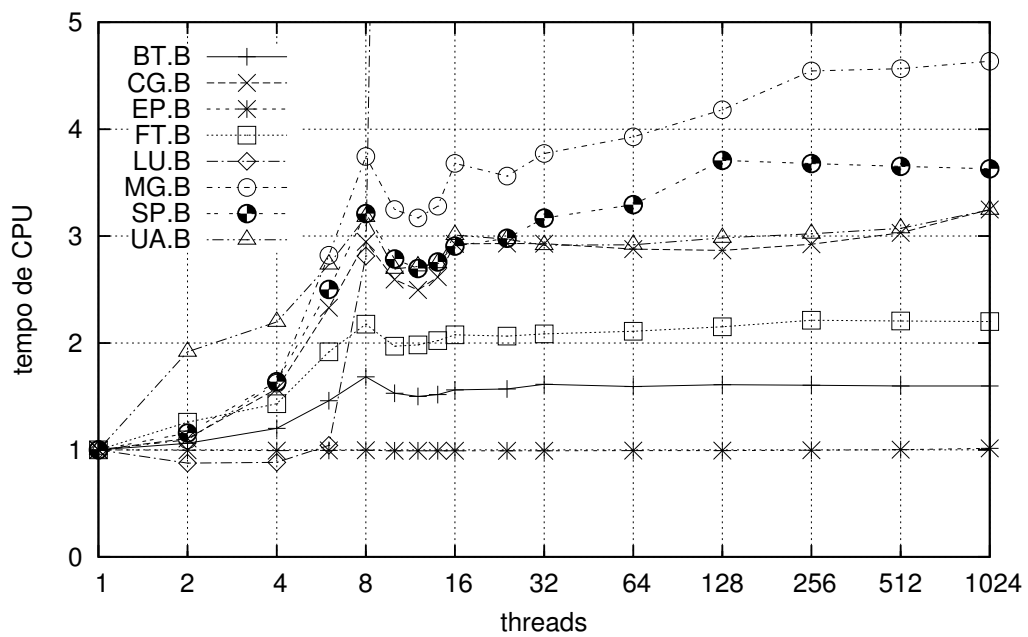


Figura 4.11: Tempos de CPU para NPB classe B na máquina SGI (em relação à versão sequencial).

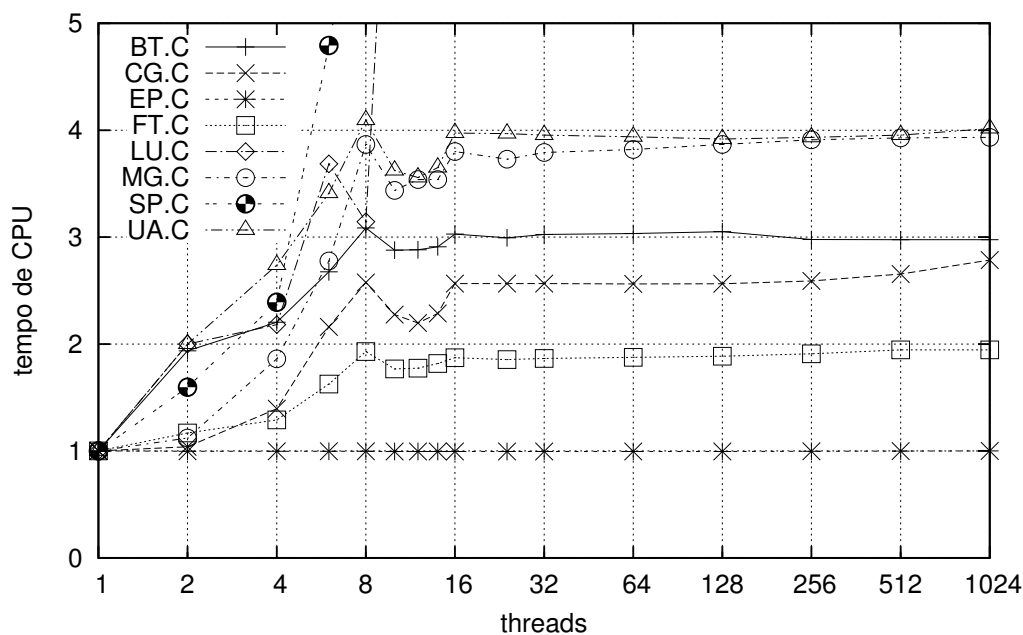


Figura 4.12: Tempos de CPU para NPB classe C na máquina SGI (em relação à versão sequencial).

dos três tamanhos de problema testados não correspondeu com sua ordem na curva de uso de barramento: a aplicação FT tem taxa de faltas em L1 maior do que SP e MG, porém com o menor uso de barramento de memória dos três programas; a grande diferença na taxa de faltas em L1 entre CG e as demais aplicações não se manifestou na taxa de uso de barramento; BT e MG têm taxas de faltas no cache parecidas, mas taxas de uso de barramento bem diferentes. A variação da quantidade absoluta de faltas em L1 com o acréscimo de *threads* não foi significativa o suficiente para gerar uma correspondência com o *speedup* das aplicações.

$$\text{taxa de faltas no cache de dados L1 (\%)} = \frac{100 \times L1D_REPL}{INST_RETIRED.ANY} \quad (4.3)$$

As taxas de faltas no cache de nível 2, por outro lado, permitem certa comparação com o desempenho das aplicações. As figuras 4.16, 4.17 e 4.18 apresentam as taxas de faltas no cache L2 das aplicações em SGI para as classes A, B e C, respectivamente, calculadas pela equação 4.4. A ordem entre as aplicações, da menor taxa de faltas para a maior, é bem próxima da ordem nas curvas de uso de barramento (figuras 4.7, 4.8 e 4.9). As aplicações parecem se dividir em dois grupos, com CG, MG e SP com maiores taxas

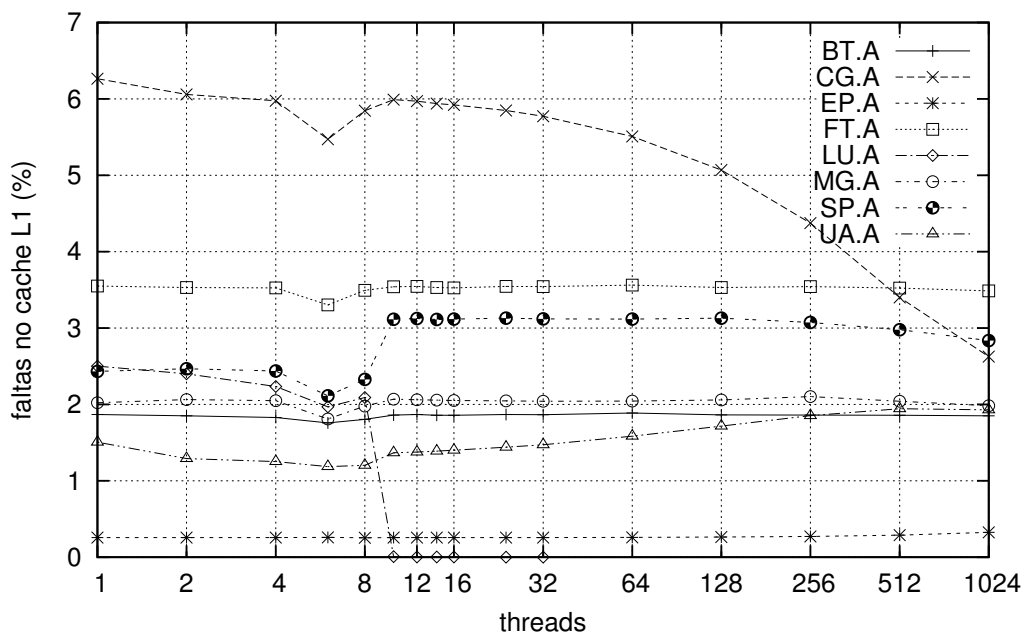


Figura 4.13: Taxas de faltas no cache de dados de nível 1 para NPB classe A em SGI.

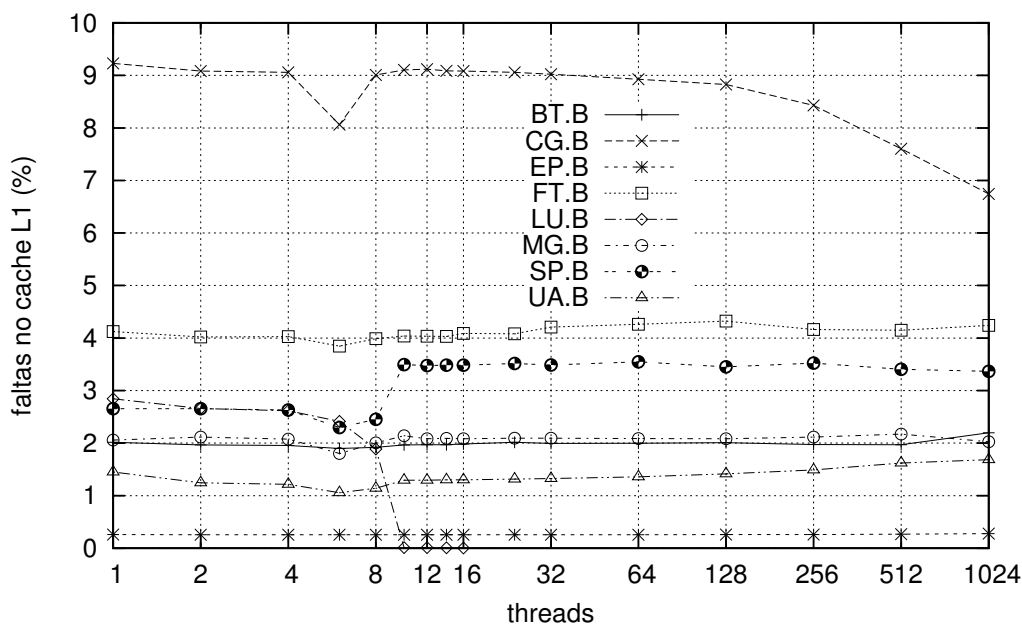


Figura 4.14: Taxas de faltas no cache de dados de nível 1 para NPB classe B em SGI.

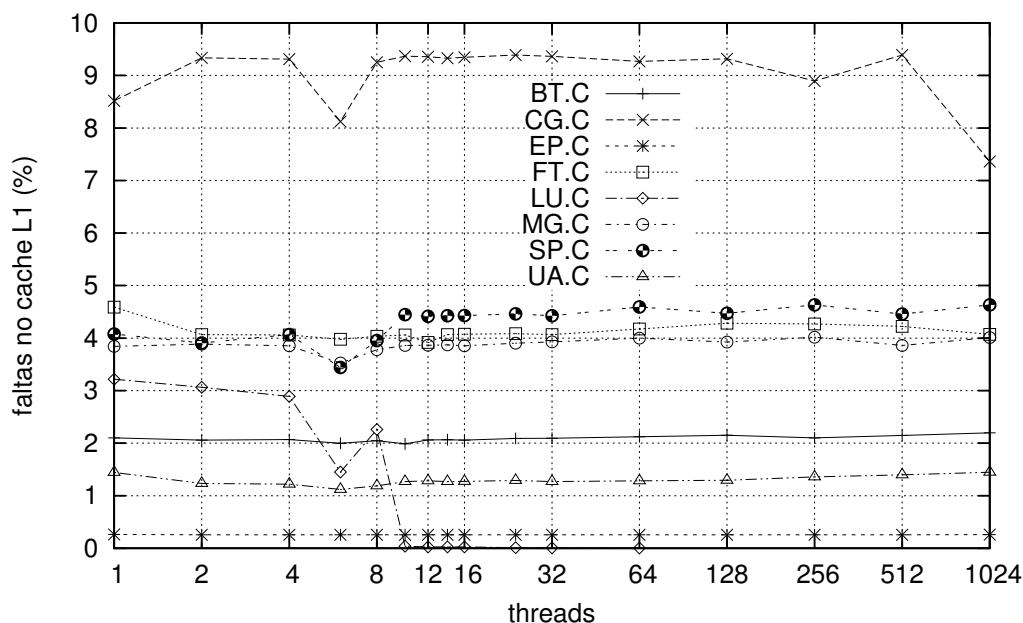


Figura 4.15: Taxas de faltas no cache de dados de nível 1 para NPB classe C em SGI.

de faltas no cache e uso de barramento tipicamente superior a 50%; e BT, EP, FT, LU e UA, com menores taxas de faltas no cache e uso de barramento tipicamente inferior a 50%.

$$\text{taxa de faltas no cache L2 (\%)} = \frac{100 \times L2.LINES.IN.SELF.ANY}{INST.RETIRED.ANY} \quad (4.4)$$

A diferença entre os comportamentos das curvas de uso de barramento e das curvas de faltas no cache pode ser atribuída à diferença de compartilhamento dos dispositivos. Enquanto o barramento de memória é compartilhado por todos os oito *cores* da máquina, o segundo nível de cache é compartilhado por apenas dois *cores*. Na máquina utilizada, a taxa de faltas em L2 não é um indicativo de limitações para a escalabilidade.

Restrições no tempo de uso das demais máquinas usadas nos testes, associadas à curta duração dos programas na classe A limitaram as execuções à classe B para BT, SP e UA e classe C para CG, EP, FT, LU e MG nos outros três sistemas. Em AMD, devido à menor capacidade da memória principal, a aplicação FT não pôde ser executada na classe C, sendo substituída pela classe B. Eventos de *hardware* também não puderam ser observados.

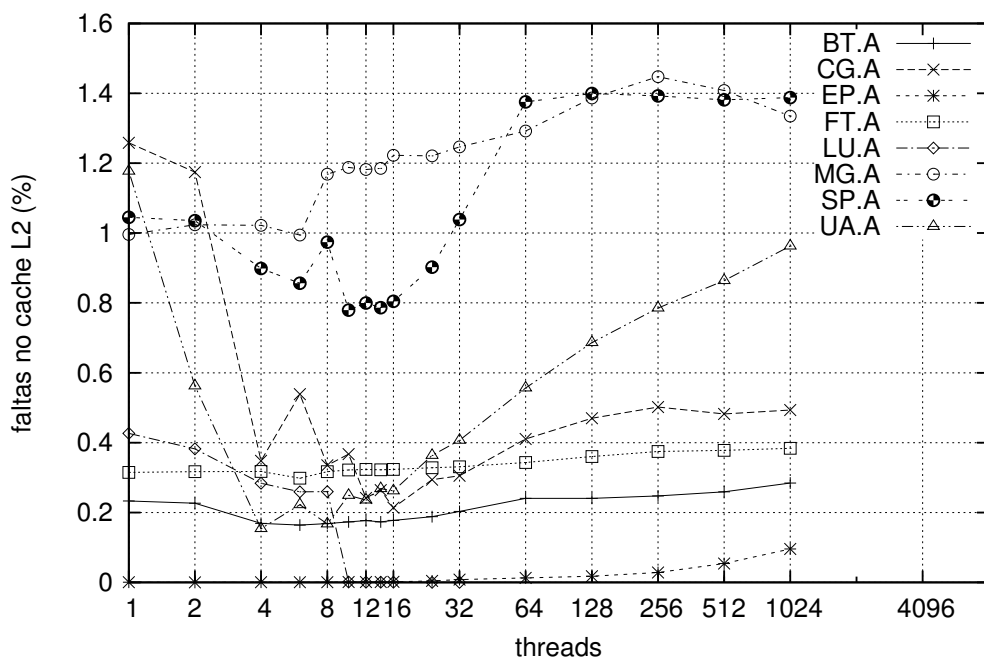


Figura 4.16: Taxas de faltas no cache de nível 2 para NPB classe A em SGI.

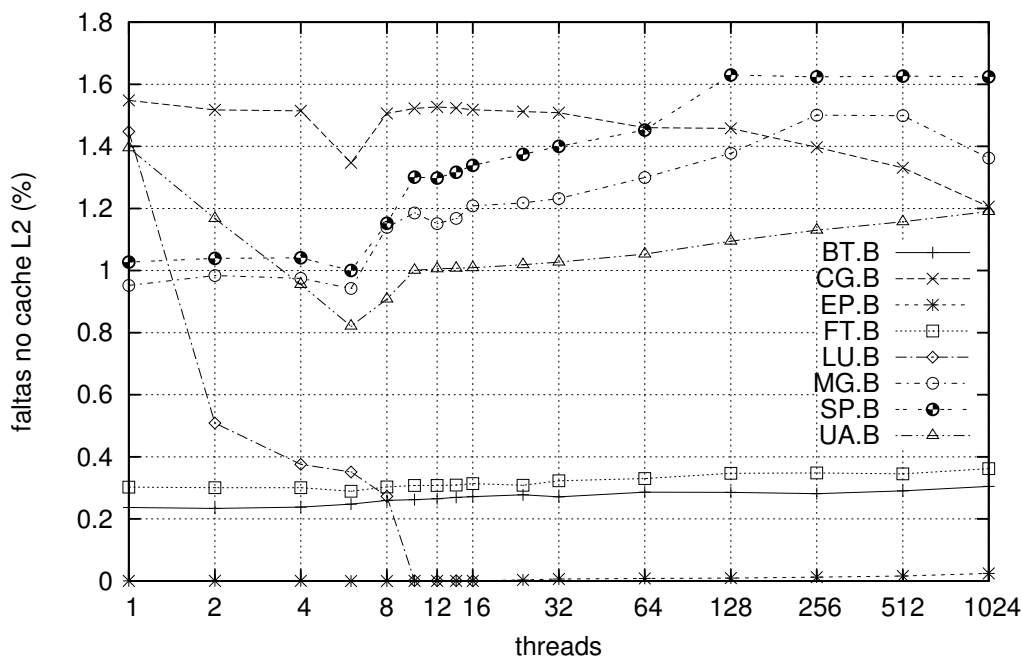


Figura 4.17: Taxas de faltas no cache de nível 2 para NPB classe B em SGI.

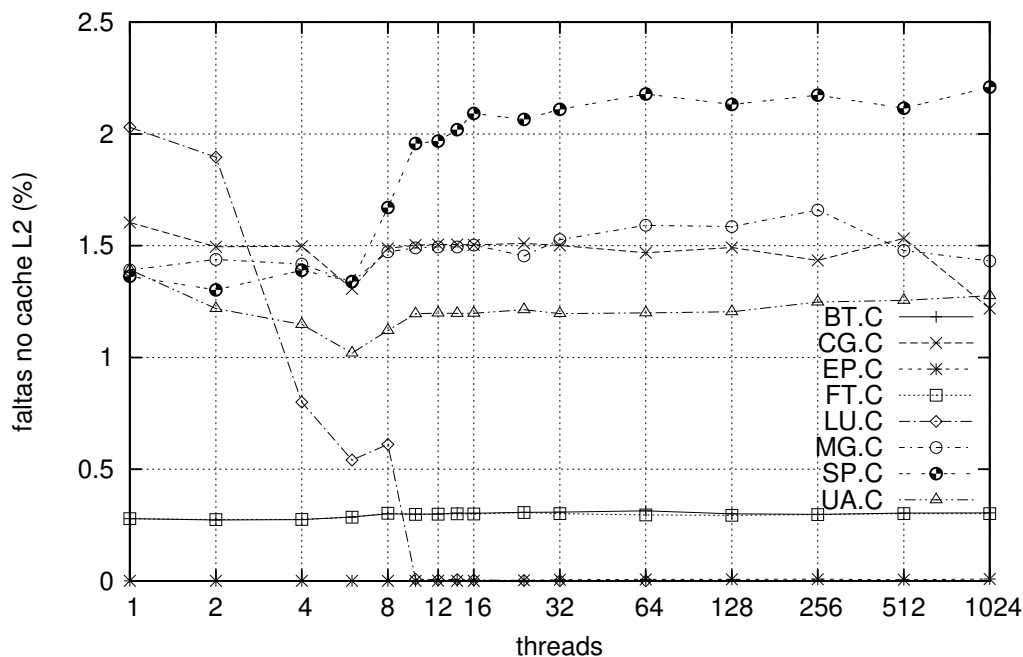


Figura 4.18: Taxas de faltas no cache de nível 2 para NPB classe C em SGI.

O *speedup* e a eficiência de NPB em **HP**, que possui doze *cores*, são mostrados nas figuras 4.19 e 4.20, respectivamente, obtidos de médias de 10 execuções de cada aplicação em cada grau de paralelismo, totalizando 60 horas de uso. Apesar da maior largura de banda de memória disponível, EP não foi capaz de apresentar *speedup* linear, atingindo eficiência de cerca de 70% com 12 *threads*, embora tenha sido a aplicação com melhor escalabilidade. UA, MG e SP apresentaram os menores *speedups*, seguidos de perto por CG e FT. As figuras mostram uma diminuição significativa nas taxas de crescimento das curvas de *speedup* para CG, FT, MG e SP a partir de seis *threads*, e diminuição de *speedup* para SP e MG com doze *threads*. BT e LU, por outro lado, apresentaram melhores ganhos de desempenho com mais *threads*, embora BT sofra uma redução da taxa de melhoria de oito para doze *threads* e LU atinja o pico de desempenho com dez *threads*, diminuindo em seguida. Também nesta máquina nenhuma aplicação apresentou melhoria de desempenho com mais *threads* do que *cores*. Em comparação com SGI, as aplicações obtiveram *speedups* máximos maiores e limitações para a escalabilidade surgiram, pois algumas aplicações que escalavam até a quantidade de *cores* em SGI não conseguiram aproveitar todos os *cores* de HP. Análise mais detalhada, considerando os eventos de *hardware*, é necessária para compreender essa diferença de comportamento.

A figura 4.21 mostra o *speedup* de NPB em **AMD**, ao passo que a figura 4.22 mostra

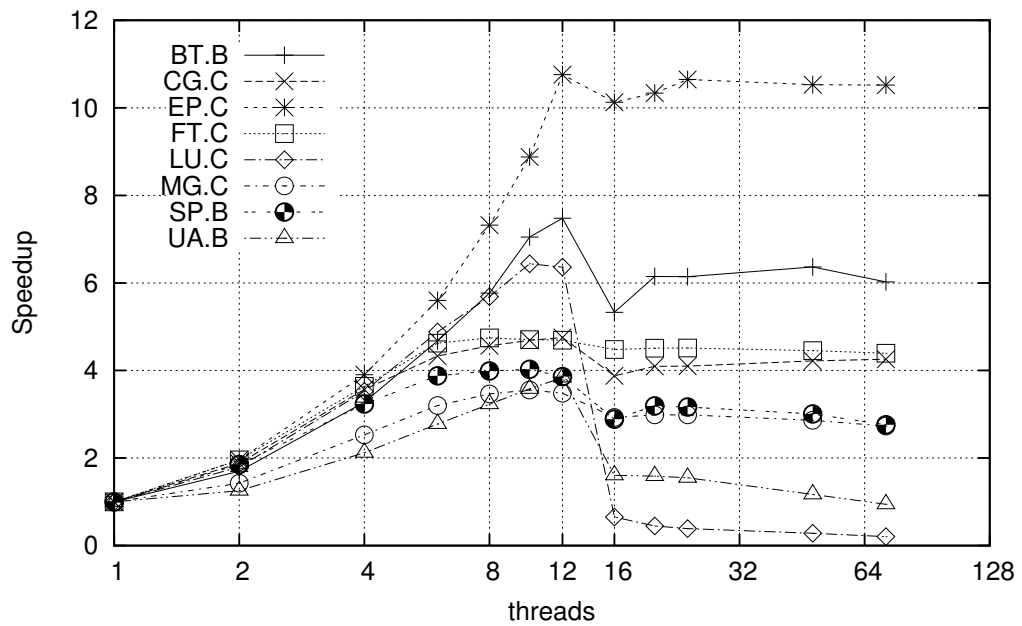


Figura 4.19: Curvas de *speedup* para NPB na máquina HP.

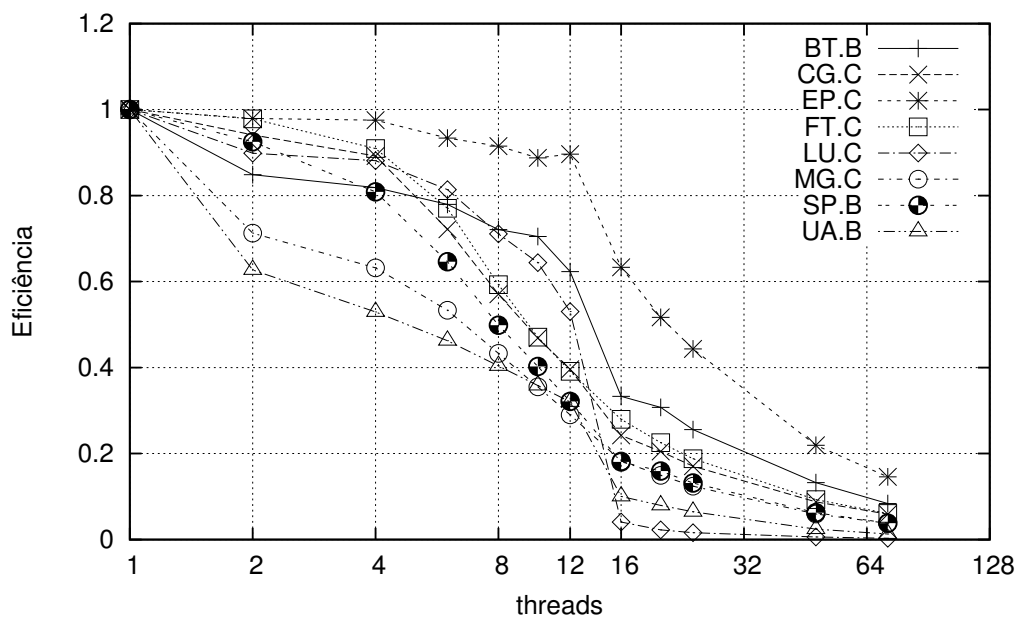


Figura 4.20: Curvas de eficiência para NPB na máquina HP.

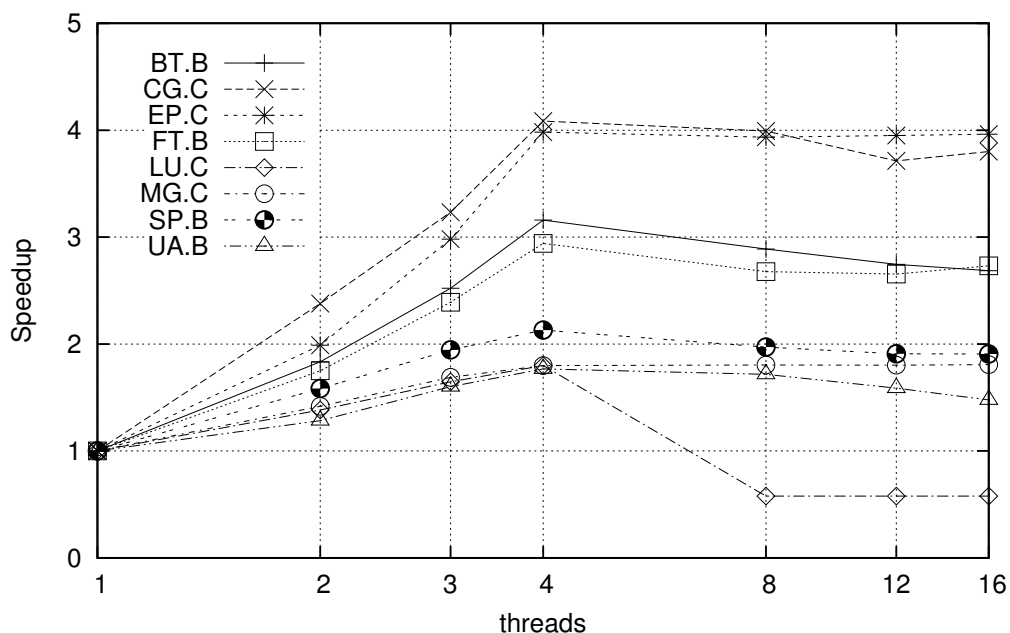


Figura 4.21: Curva de *speedup* para NPB na máquina AMD.

sua eficiência. Nesta máquina de 4 *cores* com um grande cache de nível 3 compartilhado por todos eles, todas as aplicações obtiveram o melhor desempenho com quatro *threads*, com os menores ganhos em MG, SP e UA. EP atingiu *speedup* linear e CG atingiu *speedup* superlinear, com eficiência de 118,97% em 2 *threads*. Nenhuma aplicação desempenhou melhor com mais *threads* do que *cores*, ficando tipicamente mais lentas. A ordem relativa de *speedup* entre os *benchmarks* foi parecida com SGI, exceto por CG, embora esta aplicação tenha escalado até 4 *cores* em ambas as máquinas. A pequena quantidade de *cores* nesta máquina impediu que as limitações vistas em SGI e HP surgissem.

As curvas de *speedup* das execuções iniciais de NPB no sistema MTL são mostradas na figura 4.23. Todas as aplicações sofreram uma queda brusca no desempenho entre 38 e 40 *threads*, possivelmente devido à migração de *threads* para outro domínio NUMA (outro processador), deixando-as distantes de seus dados. As execuções posteriores, realizadas fixando-se cada *thread* a um *core* específico através da variável de ambiente GOMP_CPU_AFFINITY da biblioteca GNU OpenMP (FREE SOFTWARE FOUNDATION, INC., 2006a), não exibiram este comportamento, como indicado nas curvas de *speedup* da figura 4.24 e nas curvas de eficiência da figura 4.25. Foram realizadas 10 execuções em cada grau de paralelismo, totalizando 47 horas. Com afinidade de CPU, a queda de desempenho ocorre quando há mais *threads* do que os 40 *cores* disponíveis, com excessão

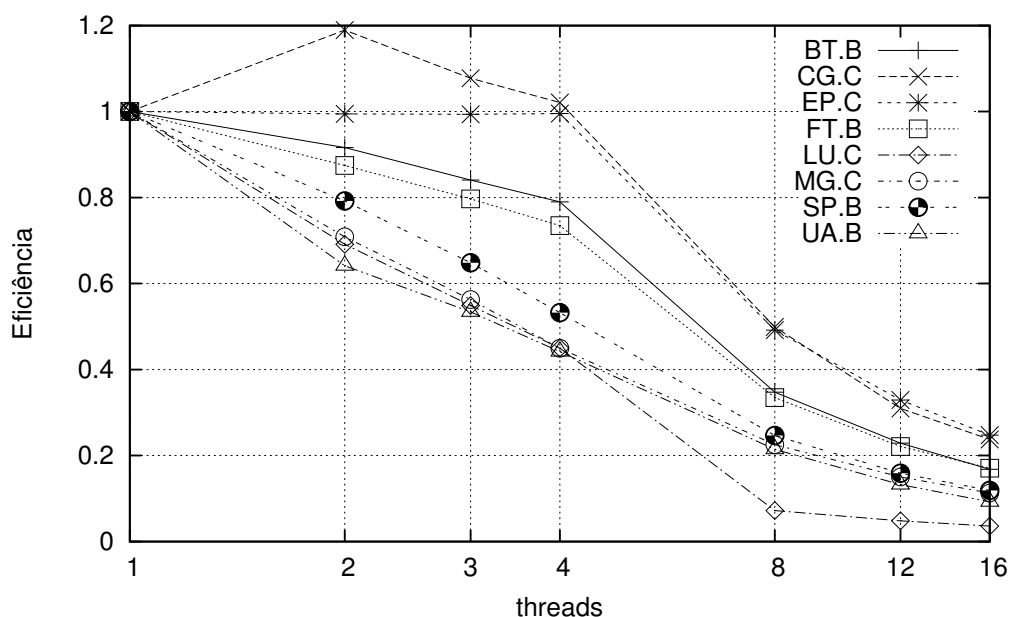


Figura 4.22: Curva de eficiência para NPB na máquina AMD.

de MG, que tem melhor *speedup* com 38 *threads* e redução de desempenho com 40 *threads*. SP tem curva de *speedup* similar à de MG, porém cessa de crescer em 34 *threads* e decai em 44 *threads*, com três intervalos em que há certa estabilização do *speedup*: 18 a 24 *threads*, 26 a 32 *threads* e 34 a 40 *threads*, o que se reflete em quedas serrilhadas em sua eficiência. MG e SP atingem eficiência máxima de apenas 30% a partir de dez *threads*, chegando a cerca de 20% em 40 *threads*. EP tem *speedup* linear e eficiência de 100% até 40 *threads*. BT, CG, FT e LU têm escalabilidades semelhantes entre si. BT apresenta três regiões de estabilidade: 22 e 24 *threads*, 26 a 32 *threads* e 34 a 40 *threads*; seu *speedup* máximo em 34 *threads* é 1,06% superior ao *speedup* com 40 *threads*. LU tem uma região de estagnação de 32 a 38 *threads* e *speedup* máximo com 40 *threads*. FT e CG também têm *speedup* máximo com 40 *threads*. UA apresenta melhoria de apenas 2,8 vezes em seu tempo de execução com até 10 *threads*, porém exibe um salto na taxa de ganhos a partir de 12 *threads*, melhorando sua eficiência de 28,41% com 10 *threads* para 54,23% com 40 *threads*. LU também exibe aumento na taxa de melhoria de *speedup* em duas situações: com mais de 10 *threads* e com mais de 22 *threads*. A figura 4.25 mostra que dez *threads*, exatamente a quantidade de *cores* em cada processador da máquina, é um ponto de mudança de comportamento de eficiência para todos os programas, exceto BT e EP, com redução da taxa de queda e até mesmo aumento da eficiência. O mesmo fenômeno não se

Tabela 4.2: Aumento médio de *speedup* máximo para NPB em MTL com o uso de afinidade de CPU.

<i>benchmark</i>	ganho (%)
LU	27,06
MG	19,66
CG	14,18
UA	10,34
SP	7,39
FT	7,20
BT	4,61
EP	2,78

observa em 20 ou em 30 *threads*. Uma comparação com os resultados de SGI, HP e AMD aponta semelhanças no comportamento de MG e SP, além de, naturalmente, EP.

Afinidade de CPU foi usada somente em MTL, pois as demais máquinas tinham organização de memória do tipo UMA, sem possibilidade de penalização por acessos a certas posições da memória, o que certamente não significa que afinidade de CPU seria prejudicial para as execuções. A tabela 4.2 indica os ganhos médios de *speedup* máximo de NPB com o uso de afinidade de CPU em MTL em relação à sua ausência. Os programas mais beneficiados por afinidade de CPU foram, em geral, os que apresentaram maior taxa de uso de barramento em SGI.

4.5 STREAM

4.5.1 Speedup

O *benchmark* de memória STREAM também foi executado nos quatro sistemas computacionais disponíveis. Suas curvas de *speedup* são mostradas na figura 4.26, ao passo que sua eficiência é mostrada na figura 4.27. Execuções com muito mais *threads* do que a quantidade de *cores* disponíveis não foram realizadas, pois já foi mostrado que não são vantajosas neste tipo de aplicação; por isso há diferentes quantidades de *threads* nas figuras de *speedup* e de eficiência. Os ganhos de desempenho com a adição de *threads* são bem modestos, atingindo um *speedup* máximo de apenas 7,56 no sistema com 40 *cores*; nos demais sistemas, as variações no *speedup* deixam de ser significativos a partir de 4 *threads*. Assim como observado em NPB, há uma alteração expressiva na taxa de aumento de *speedup* de STREAM em MTL com mais de dez *threads*, suficiente para estabilizar a eficiência das execuções em 19%. Não foram observadas reduções do tempo de execução

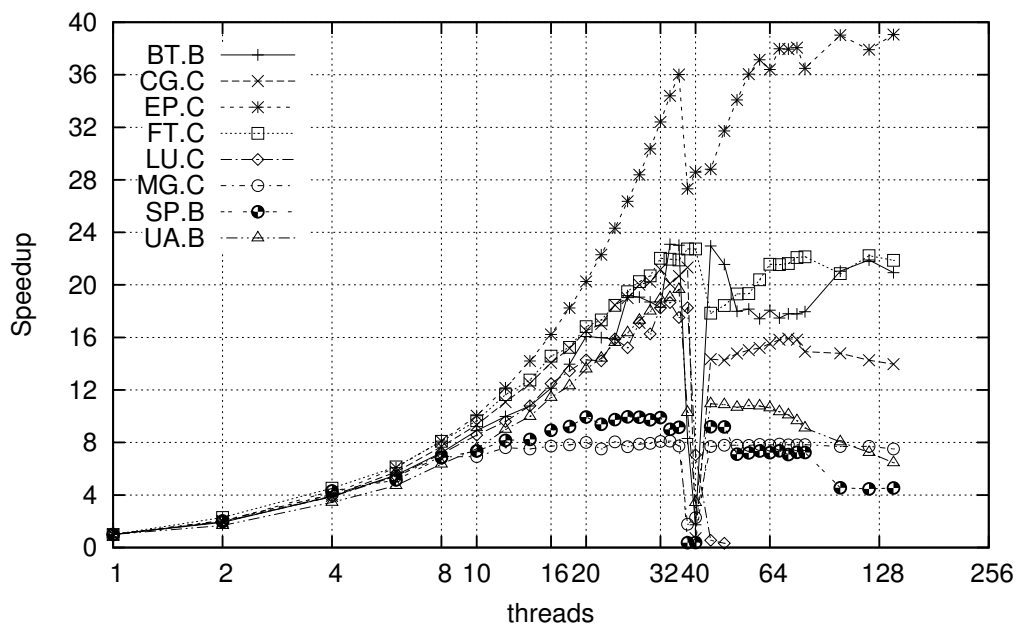


Figura 4.23: Curvas de *speedup* para NPB na máquina MTL sem afinidade de CPU.

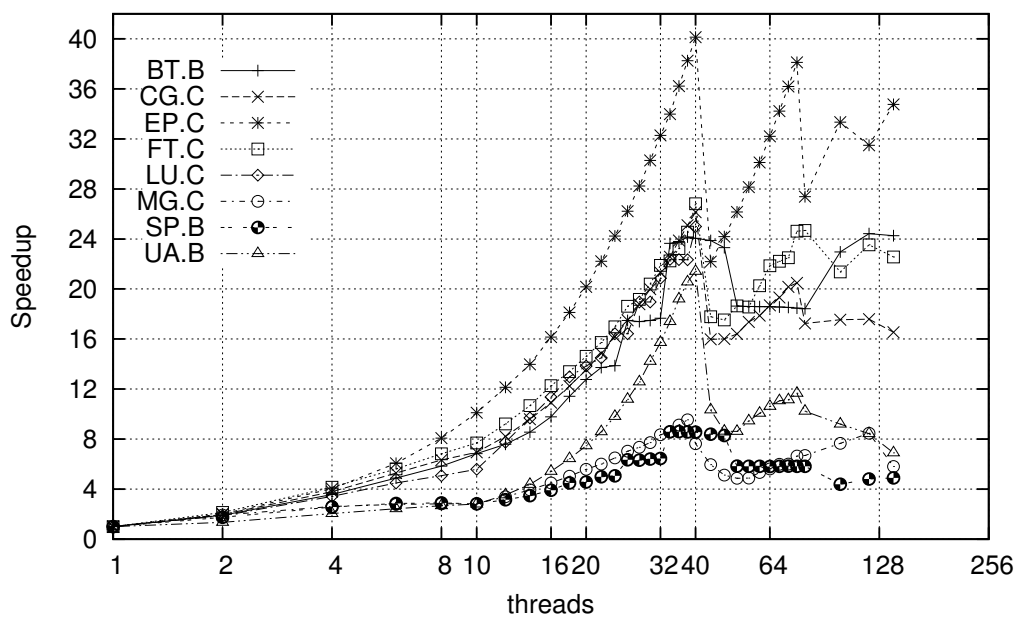


Figura 4.24: Curvas de *speedup* para NPB na máquina MTL com afinidade de CPU ativa.

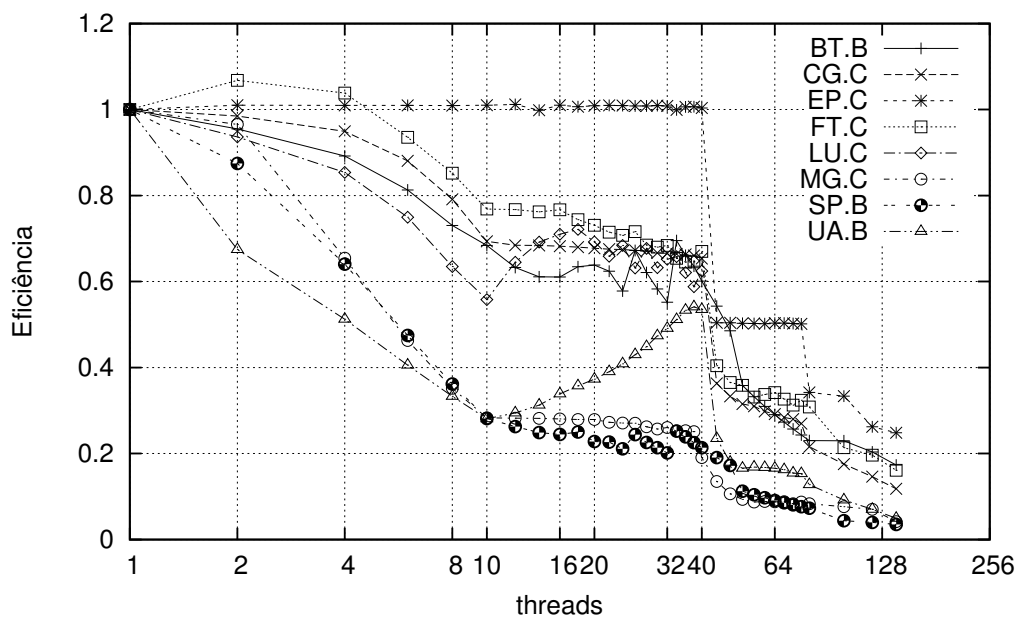


Figura 4.25: Curvas de eficiência para NPB na máquina MTL com afinidade de CPU ativa.

de STREAM com mais *threads* do que *cores*. A figura 4.28 mostra que há um aumento crescente no tempo de CPU registrado em cada sistema até a quantidade de *threads* ser igual à quantidade de *cores*, exceto em MTL, onde o tempo de CPU se estabiliza em cerca de 5,25 vezes o tempo da versão sequencial.

4.5.2 Taxa de uso de barramento e taxas de faltas no cache

As taxas de uso de barramento, de faltas no cache de nível 1 e de faltas no cache de nível 2 de STREAM em SGI são indicadas na tabela 4.3. A escalabilidade ruim de STREAM se deve à intensa competição pelo barramento de memória, indicada pelo elevado uso do barramento, acima de 70% desde a execução sequencial, e altas taxas de faltas em L2, o que é um comportamento esperado para uma aplicação que se dispõe a medir a largura de banda do canal de acesso à memória. O aumento da taxa de uso deste barramento conforme o acréscimo de *threads* torna as instruções de acesso à memória mais lentas, uma vez que a chance de uma *thread* encontrar o barramento ocupado e precisar aguardar sua liberação é alta. Os maiores tempos de espera por acessos à memória justificam os aumentos de tempo de CPU, uma vez que a medição de tempo de CPU de

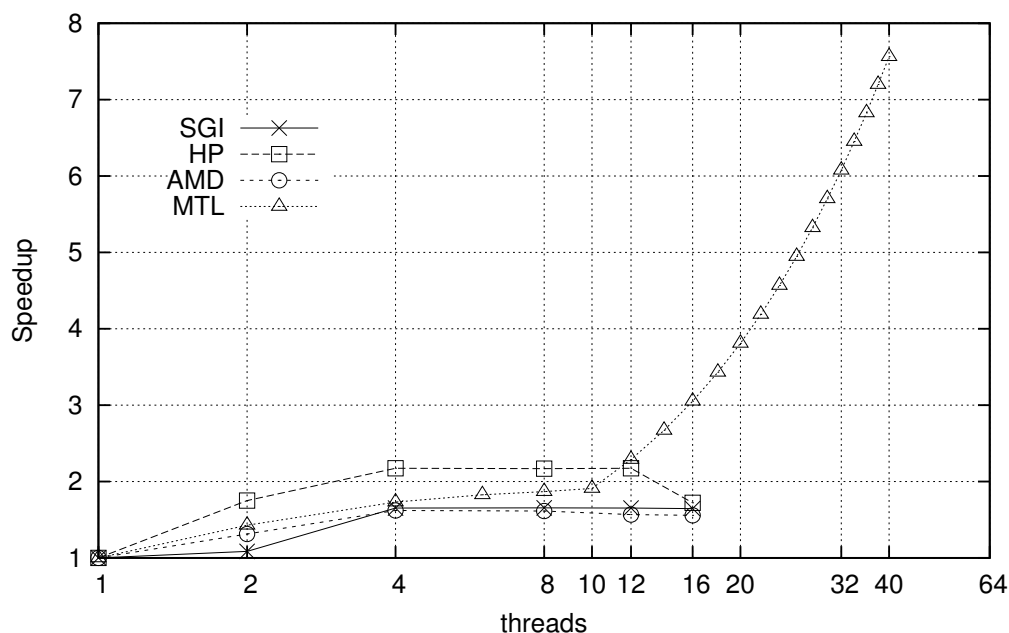


Figura 4.26: Curvas de *speedup* para STREAM nos quatro sistemas testados.

um processo feita pelo sistema operacional é ininterrupta e independente das atividades que estiverem sendo realizadas pela aplicação. A estabilização do tempo de CPU em **MTL** evidencia, por sua vez, uma vantagem da descentralização do subsistema de memória na organização NUMA, pois a competição pelo acesso à memória se concentra tipicamente nas *threads* que executam nos *cores* de um mesmo nó NUMA, pouco afetando as *threads* que executam nos demais nós.

Para confirmar que o aumento do tempo de CPU em execuções com maior grau de paralelismo se deve às maiores competições pelo acesso ao barramento de memória, o *benchmark* STREAM foi executado na versão sequencial, chamada aqui de *STREAM-base*, em **SGI**, concorrentemente com uma carga de trabalho extra. Tal carga de trabalho adicional consistiu também de STREAM, nas versões sequencial e paralela com diversos graus de paralelismo. Foram medidos o tempo decorrido e o tempo de CPU de *STREAM-base* para cada carga de trabalho extra em co-execução. Em todas as execuções, **um** processo de *STREAM-base* foi executado concorrentemente com **um** processo da carga de trabalho adicional. A figura 4.29 mostra o tempo de CPU e o tempo decorrido de cada execução de *STREAM-base* com cargas de trabalho extra em relação ao tempo correspondente de *STREAM-base* sem a presença de cargas de trabalho adicionais. Embora o total de operações de processamento e de acesso à memória de *STREAM-base* seja o mesmo, já

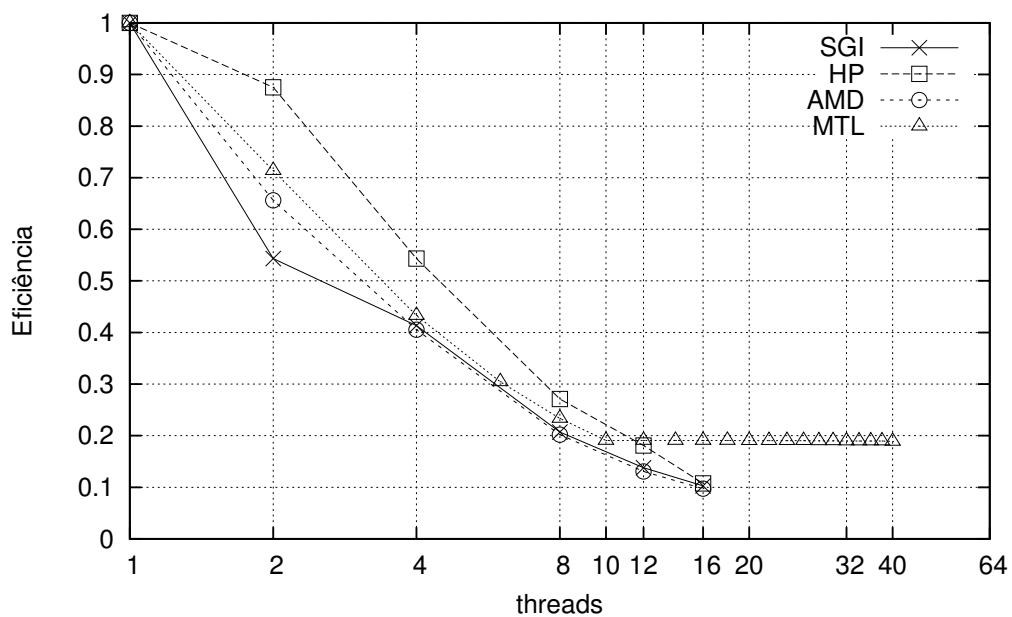


Figura 4.27: Curvas de eficiência para STREAM nos quatro sistemas testados.

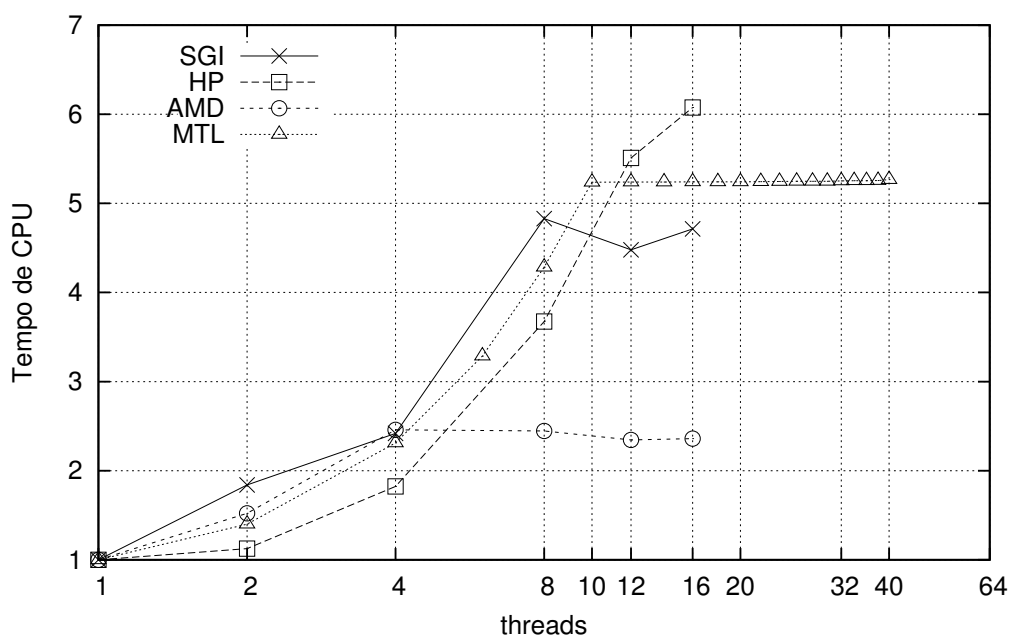


Figura 4.28: Tempo de CPU para STREAM nos quatro sistemas testados (em relação à versão sequencial).

Tabela 4.3: Taxas de uso de barramento e de faltas no cache para STREAM em SGI.

<i>threads</i>	uso de barramento (%)	faltas em L1 (%)	faltas em L2 (%)
sequencial	74,6 ±3,31	4,89±0,14	4,84±0,24
2	82,24±5,62	4,21±0,38	4,16±0,39
4	88,19±5,62	4,11±0,12	4,09±0,22
8	89,76±8,69	4,01±0,25	3,85±0,26
12	91,13±3,10	4,28±0,35	4,24±0,17
16	90,56±3,02	4,28±0,06	4,25±0,24

que o programa não sofreu alterações, o tempo de CPU e o tempo decorrido aumentaram conforme mais *threads* foram utilizadas pela carga de trabalho adicional, que provocaram maiores contenções pelo barramento compartilhado.

4.6 Multiplicação de matrizes

Um programa de multiplicação de matrizes, MultMat, foi desenvolvido em Fortran para encerrar a análise de escalabilidade. O tamanho das matrizes A, B e C para a operação $C_{1000 \times 1000} = A_{1000 \times 30000} \cdot B_{30000 \times 1000}$ foi escolhido de forma a exceder a capacidade do último nível de cache em todas as máquinas usadas. A figura 4.31 mostra seu *speedup* e a figura 4.30 mostra sua eficiência nos sistemas SGI, AMD e MTL; o sistema HP não estava disponível para este teste. Os maiores *speedups* foram atingidos na quantidade de *threads* igual à quantidade de *cores* de cada máquina e corresponderam a 80%, 106% e 72% do *speedup* linear em SGI, AMD e MTL, respectivamente, com eficiência superior a 74%. O aumento de tempo de CPU de MultMat foi inferior a 29% nos três sistemas, excedendo 60% apenas com mais *threads* do que *cores* em SGI, indicando baixa taxa de uso de barramento, compatível com os *speedups* observados. As medições de uso de barramento e de faltas no cache são apresentadas na tabela 4.4 e confirmam a menor taxa de acessos à memória.

4.7 Conclusões

As análises da seção anterior conduzem à conclusão de que, para os programas e máquina testados, a taxa de faltas no cache de nível 1 não é indicativo de escalabilidade. A taxa de faltas no cache de nível 2 é uma melhor estimativa; contudo, é limitada pelo baixo grau de compartilhamento do cache pelos *cores* na máquina utilizada. A taxa de uso do barramento de memória se mostrou como um indicador mais acurado para

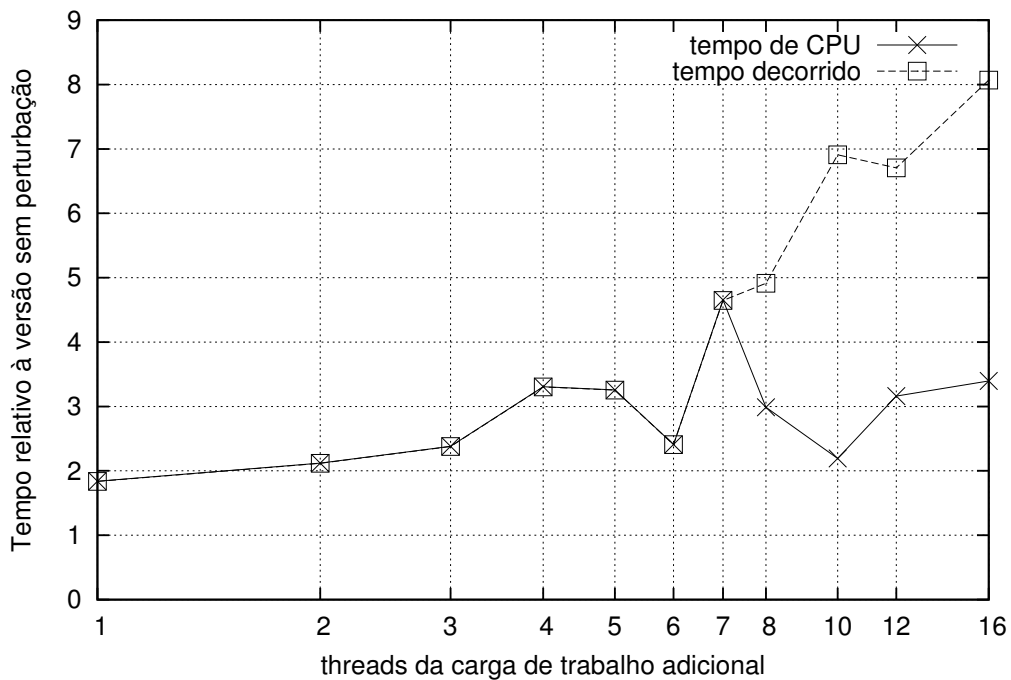


Figura 4.29: Tempo de CPU e tempo decorrido de *STREAM-base* para cada carga de trabalho adicional em SGI, em relação à execução sem perturbação.

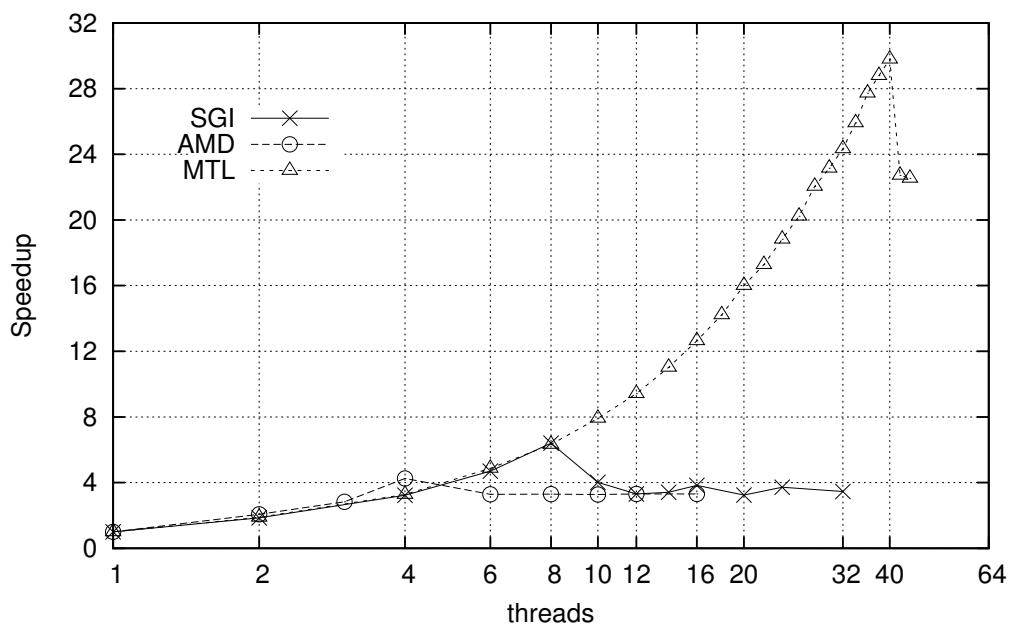


Figura 4.30: Curvas de *speedup* para MultMat nos três sistemas testados.

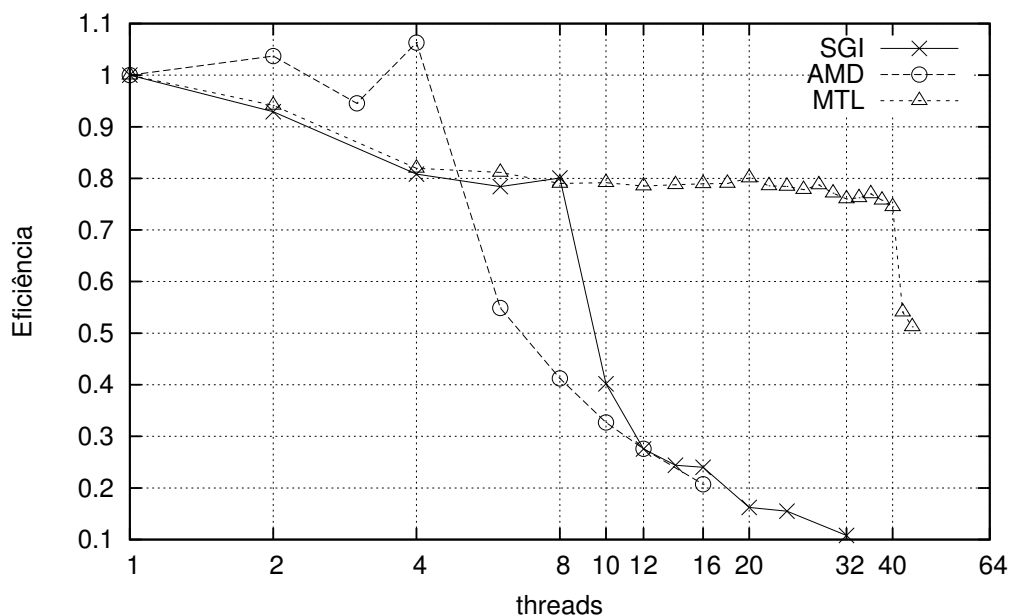


Figura 4.31: Curvas de eficiência para MultMat nos três sistemas testados.

Tabela 4.4: Taxas de uso de barramento e de faltas no cache para MultMat em SGI.

<i>threads</i>	uso de barramento (%)	faltas em L1 (%)	faltas em L2 (%)
sequencial	21,41±0,53	19,53±1,52	1,81±0,11
2	39,16±2,97	19,52±1,62	1,8 ±0,04
4	43,18±0,83	19,51±1,10	1,8 ±0,16
8	60,5 ±3,84	19,39±1,15	1,67±0,15
10	88,03±3,74	19,58±1,34	2,27±0,10
12	87,95±4,91	19,58±1,73	2,94±0,15
16	79,89±1,43	19,58±0,48	2,44±0,07
20	84,06±2,25	19,6 ±1,31	3,01±0,26
24	77,54±4,72	19,57±0,51	2,72±0,08
32	81,4 ±4,30	19,58±0,85	2,71±0,25

a escalabilidade, indicando restrições para o aumento de desempenho com taxas de uso superiores a 70%.

Capítulo 5

Influência da granularidade dos acessos na escalabilidade de aplicações paralelas

Conforme exposto no capítulo 1, granularidade é, no contexto deste trabalho, a proporção entre acessos à memória e processamento. Neste capítulo é abordada a relação entre granularidade e desempenho de execução paralela como uma possível limitação para a escalabilidade. A seção 5.1 inicia com medições de propriedades relevantes dos sistemas computacionais utilizados. Na seção 5.2, são apresentados os valores de granularidade de *loop* para as aplicações observadas, enquanto suas granularidades efetivas são apresentadas na seção 5.3. A relação entre granularidade e medidas indicativas de desempenho é estabelecida na seção 5.4. Variações na granularidade são estudadas na seção 5.5, ao passo que variações na localidade são estudadas na seção 5.6. A seção 5.7 conclui com considerações sobre o uso de granularidade na determinação automática da quantidade de *threads*.

5.1 Caracterização dos sistemas computacionais

Algumas propriedades dos sistemas computacionais relevantes para a análise da granularidade foram registradas para as quatro máquinas testadas. A tabela 5.1 relaciona a velocidade de CPU, a largura de banda de memória e o *machine balance* de cada computador. A coluna MFLOPS indica a velocidade média de um único *core* no problema resolvido por Linpack em execuções unicamente sequenciais com *arrays* de 200 posições. A largura de banda foi considerada como o maior valor medido pelo componente Triad do *benchmark* STREAM entre execuções sequenciais e paralelas com quantidade de *threads* variando de 2 até o número de *cores* disponíveis. Duas medidas estão presentes: MBytes/s,

Tabela 5.1: Velocidade de processamento de um *core* e hipotética total, largura de banda e β_M dos sistemas computacionais testados.

máquina	MFLOPS 1 <i>core</i>	MFLOPS total	MBytes/s	MWords/s	β_M
SGI	1548,7± 33	12389,6	5739,5± 95	717,4	0,057903
HP	1736,8± 83	20841,6	21434,7± 224	2679,3	0,128555
AMD	1865,1± 40	7460,4	5259,9± 903	657,5	0,088132
MTL	1365,8±115	54600,0	36903,8±1485	4613,0	0,084487

que é a medida utilizada por STREAM, e MWords/s, obtida dividindo-se o valor médio em MBytes/s pela quantidade de *bytes* usada para representar um número real de precisão dupla; nos processadores e compilador utilizados, cada número real de precisão dupla é representado por oito *bytes*. A relação β_M (*machine balance*) entre largura de banda de memória (em MWords/s) e velocidade de processamento foi calculada segundo a equação 1.6, porém considerando uma velocidade hipotética do processador como sendo a velocidade média em MFLOPS de um *core* no Linpack multiplicada pela quantidade de *cores* disponível.

5.2 Granularidade de loop

A granularidade de *loop*, γ_L , foi determinada pela equação 3.1. Com exceção de Mult-Mat, as aplicações observadas possuem diversos *loops* paralelos, cada um com sua própria granularidade. Para os fins desta análise foi escolhido apenas um valor de granularidade de *loop* para cada aplicação, ou componente de aplicação, como no caso de STREAM, indicado na tabela 5.2. A escolha deste valor para as aplicações de NPB exigiu a determinação dos trechos dos programas que mais consumiram tempo de execução. A figura 5.1 mostra a distribuição do tempo de execução das aplicações de NPB classe B na máquina SGI pelas suas funções, onde *primeira*, *segunda*, *terceira*, *quarta* e *quinta* indicam as cinco funções que executaram por mais tempo, enquanto *outras* representa o somatório de tempo de execução das demais funções. Esta distribuição de tempo foi obtida a partir da execução paralela com 8 *threads*, e em graus de paralelismo inferiores os resultados foram similares. As aplicações CG, FT e MG destacam-se na figura 5.1 por possuírem uma função que ocupa ao menos 50% do tempo de execução. No caso de MG, a segunda função mais demorada é similar à primeira, essencialmente aumentando o intervalo de abrangência para mais de 80%. A segunda função mais demorada de EP é uma chamada a uma rotina da biblioteca matemática a partir da primeira função. Nestes quatro casos, a granularidade assumida para representar o programa é a granularidade do *loop* paralelo

Tabela 5.2: Granularidade de loop das aplicações de NPB, STREAM e MultMat.

programa		γ_L
NPB	BT	0,856
	CG	1,800
	EP	0,211
	FT	1,333
	LU	0,807
	MG	1,667
	SP	1,894
	UA	1,615
STREAM	STREAM	2,500
	COPY	∞
	SCALE	1,637
	ADD	3,000
	TRIAD	1,500
MultMat		1

principal da função mais demorada, a saber *conj-grad* (CG), *MAIN* (EP), *fftz2* (FT) e *resid* (MG). A análise dos demais programas não é trivial devido à menor diferença dos tempos de execução entre as funções principais; é de se esperar que a análise seja menos precisa quanto menor for a participação da função escolhida no tempo total da aplicação. BT e SP são estruturalmente parecidos, diferenciando-se no algoritmo de resolução do problema, implementado nas funções *x_solve*, *y_solve* e *z_solve*. Cada uma destas funções é composta unicamente de um grande *loop* paralelo com a mesma granularidade, sendo este o valor adotado para representar a respectiva aplicação. Para LU foi adotada, para simplificar, a granularidade da função mais custosa, com a ressalva de que não é representativa da aplicação como um todo, já que a função ocupa menos de 40% do tempo de execução. UA é um caso especial, por fazer uso de *locks* OpenMP para garantir a consistência de alguns dados compartilhados por suas *threads* e também por ter função mais custosa cobrindo um intervalo relativamente curto do tempo de execução (inferior a 30%). Não obstante, foi adotada a granularidade da função principal, com a mesma ressalva aplicada a LU.

Em MultMat, devido à soma parcial, o acesso ao *array C*, fora do *loop* mais interno, não foi considerado na granularidade uma vez que o *loop* interno é executado mais frequentemente do que o acesso a *C*. Tamanhos de matrizes maiores aumentam a diferença; para os tamanhos de matrizes $C_{1000 \times 1000} = A_{1000 \times 30000} \cdot B_{30000 \times 1000}$, o *loop* interno é executado 4197 vezes mais do que a linha que acessa $c(j,i)$, resultando em uma granularidade igual a cerca de 1,00024.

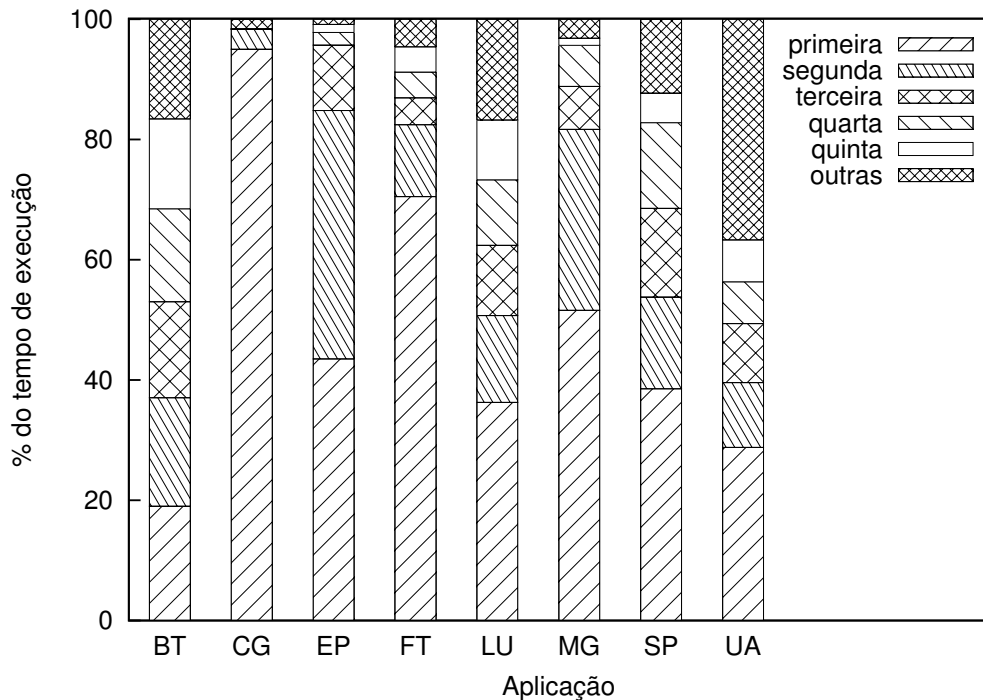


Figura 5.1: Tempo de execução das cinco funções mais demoradas de cada aplicação de NPB classe B em SGI.

Os valores de γ_L para STREAM presentes na tabela 5.2 foram determinados considerando inicialmente os *loops* principais como um todo, na linha STREAM, e separadamente, nas linhas COPY, SCALE, ADD e TRIAD. O valor ilimitado de γ_L para COPY é devido à ausência de processamento sobre os dados dos dois *arrays* envolvidos.

5.3 Granularidade efetiva

A granularidade efetiva γ_E , que indica a taxa real média com que dados foram acessados na memória em relação ao processamento de ponto flutuante, foi calculada pela equação 3.2 a partir da observação de eventos do *hardware*. Na equação 3.2, n assumiu o valor **8**, indicando que cada linha de cache foi ocupada por oito palavras de dados do tipo ponto flutuante de precisão dupla, uma vez que uma linha de cache nos processadores empregados tem capacidade para 64 *bytes* e cada palavra de dados de tal tipo no compilador utilizado ocupa 8 *bytes*. Os valores obtidos de γ_E são mostrados nas figuras 5.2, 5.3 e 5.4 para NPB em SGI. Os desvios padrão correspondentes são apresentados no apêndice A.

A vantagem do uso do mecanismo de cache é notada imediatamente, pois enquanto a maior granularidade de *loop* de NPB na tabela 5.2 é de 1,894 em SP, indicando, em média,

1894 acessos a *arrays* a cada 1000 *flops*, a mesma aplicação exibe granularidade efetiva média igual a 80 acessos à memória para cada 1000 *flops* na configuração de 8 *threads* na classe C. Se o processador não possuísse memória cache, todos os 1894 acessos a *arrays* de SP, representados por γ_L , seriam realizados na memória principal a cada 1000 *flops* executadas. Nos três tamanhos de problema, a aplicação EP apresenta granularidade efetiva inferior a 1 acesso para cada 1000 *flops* com até 8 *threads*, e mesmo nas execuções com mais *threads* do que *cores* esse valor não excede 6 acessos em 1000 *flops*, o que é compatível com seu baixo valor de granularidade de *loop* na tabela 5.2. BT apresenta o segundo menor valor de γ_E nas três classes e o terceiro menor de γ_L , bem próximo ao segundo valor. FT tem granularidade de *loop* maior do que EP e BT, e maior granularidade efetiva do que ambos. O valor de γ_L de LU é o segundo menor de NPB, porém sua curva de γ_E oscila, ora sendo a terceira menor, ora sendo a quarta menor. CG tem, em geral, granularidade efetiva superior às demais aplicações e o segundo maior γ_L de NPB. MG e SP têm granularidades efetivas semelhantes, apesar da diferença absoluta de cerca de 0,2 entre suas granularidades de *loop*. UA tem γ_E inferior a 0,02 com até 8 *threads* na classe A, porém 0,06 na classe B e 0,07 na classe C; sua granularidade de *loop* é próxima à de CG. As figuras 5.2, 5.3 e 5.4 indicam dois grupos de aplicação: BT, EP, FT e LU com γ_E tipicamente inferior a 0,03 para até 8 *threads*; e CG, MG, SP e UA, com γ_E excedendo 0,05.

De modo geral, é possível estabelecer uma correspondência entre granularidade de *loop* e granularidade efetiva, pois aos baixos valores de γ_L na tabela 5.2 corresponderam baixos valores de γ_E , e altos valores de γ_L corresponderam a altos valores de γ_E , embora a ordem dos programas segundo γ_L não seja exatamente a mesma que segundo γ_E . A diferença de granularidade de *loop* entre BT e SP relaciona-se com a diferença significativa entre suas curvas de γ_E . Os elevados valores de γ_L de CG e MG são compatíveis com seus elevados valores de γ_E , sendo que a maior granularidade efetiva de CG pode estar relacionada à aleatoriedade de seus acessos, conforme a descrição do *benchmark*. FT possui granularidade de *loop* intermediária e γ_E tipicamente maior do que os programas do primeiro grupo. Os resultados para LU e UA são menos significativos, como exposto na seção 5.2.

A figura 5.5 mostra a granularidade efetiva de MultMat em SGI. Esta aplicação tem valores de γ_E equivalentes aos valores intermediários nos gráficos de NPB, apesar de baixo valor de γ_L na tabela 5.2.

A granularidade efetiva de STREAM em SGI, tanto do programa completo quanto

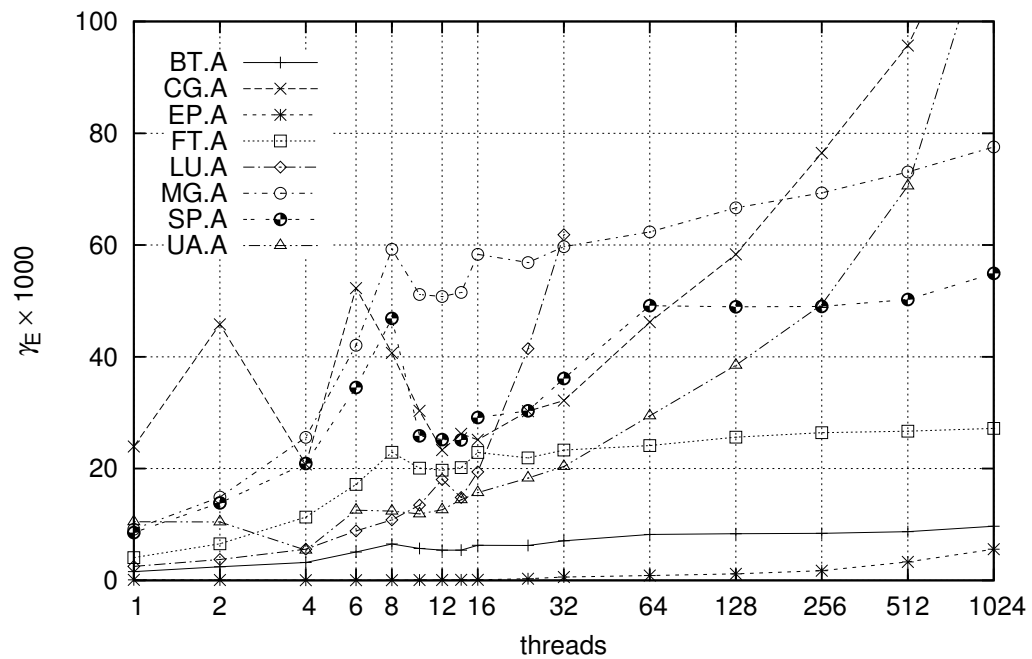


Figura 5.2: Granularidade efetiva de NPB classe A na máquina SGI.

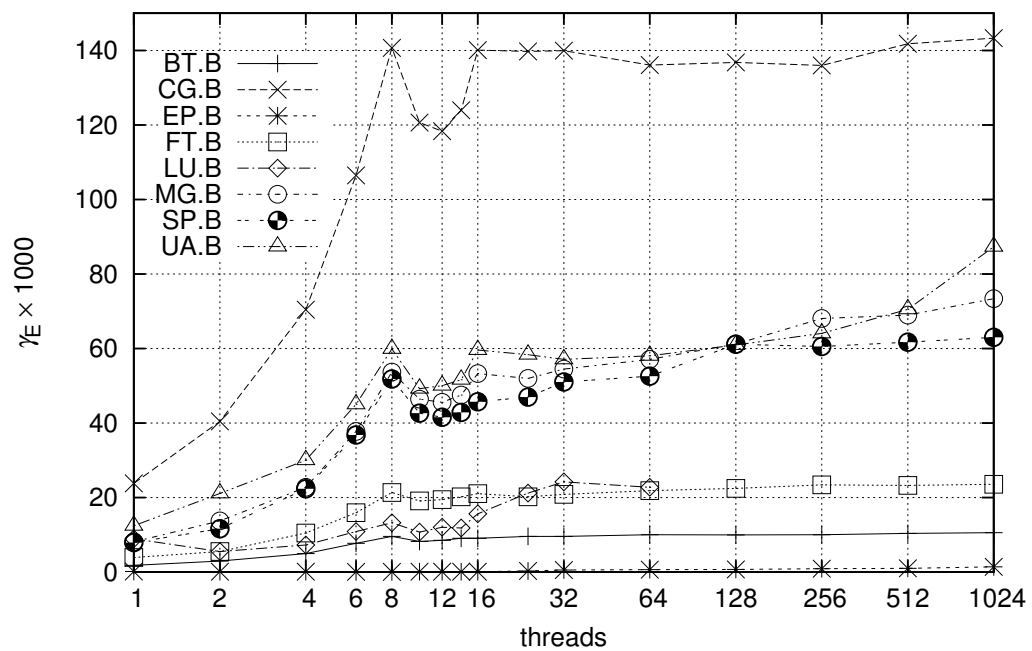


Figura 5.3: Granularidade efetiva de NPB classe B na máquina SGI.

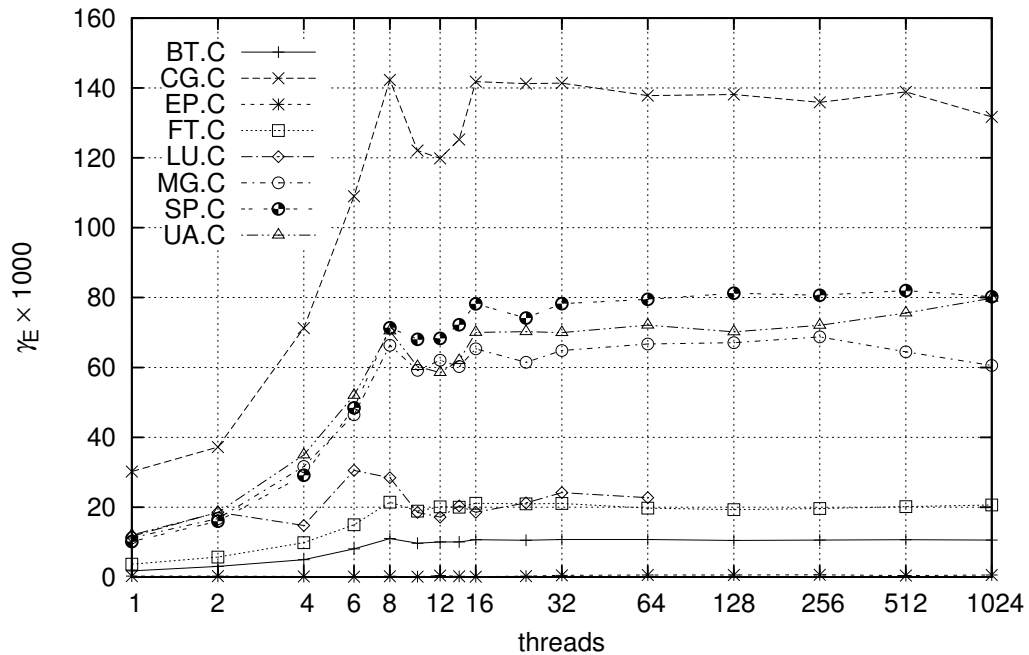


Figura 5.4: Granularidade efetiva de NPB classe C na máquina SGI.

de seus quatro *loops* paralelos, é mostrada na figura 5.6. Seus valores elevados de granularidade de *loop* na tabela 5.2 são compatíveis com os numerosos acessos à memória em relação ao processamento, com granularidade efetiva ultrapassando 100 acessos por 1000 *flops* já em 2 *threads*. O valor ilimitado de γ_L para o componente COPY faz com que sua granularidade efetiva seja a maior deste trabalho, atingindo 1440 acessos a cada 1000 *flops* (executados em outras partes do programa) com 8 *threads*. A ordem relativa entre os demais componentes sob γ_L , a saber ADD, SCALE e TRIAD, reflete-se na granularidade efetiva, com a curva de ADD superior à de SCALE e esta superior à de TRIAD. Um aspecto importante dos resultados de TRIAD é que cada *loop* paralelo tem seu próprio valor de γ_L , que não é o mesmo quando se considera hipoteticamente os quatro componentes como um único *loop*. A granularidade efetiva média do programa como um todo é maior do que a granularidade efetiva média de SCALE e de TRIAD, ligeiramente menor do que a de ADD e chega a ser 64,4% menor do que a de COPY. A diferença nas granularidades entre os trechos do programa indica diferentes limitações ao paralelismo. A consequência imediata desta observação é que uma ferramenta que considere granularidade de *loop* na análise automática de escalabilidade deve determinar γ_L para cada *loop* paralelo, permitindo possivelmente diferentes quantidades de *threads* para diferentes trechos do programa.

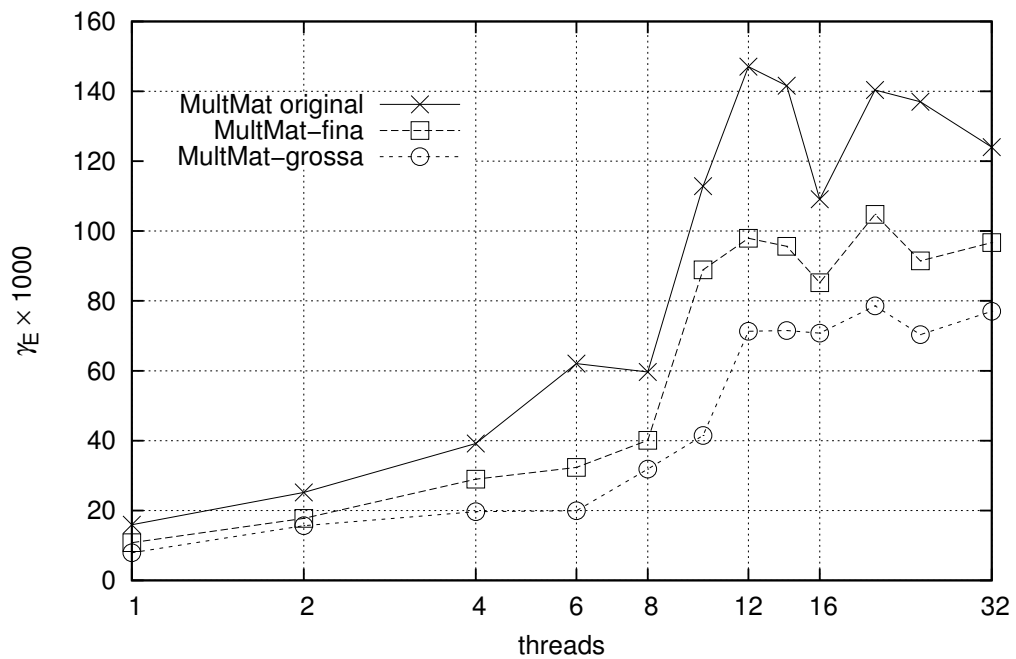


Figura 5.5: Granularidade efetiva de MultMat e de suas variações na máquina SGI.

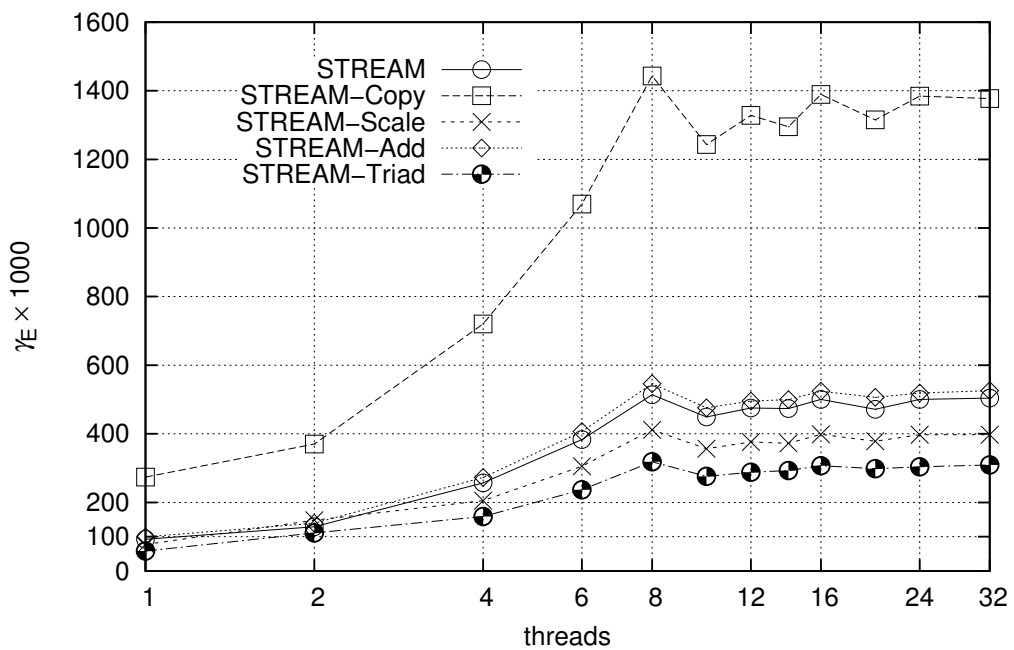


Figura 5.6: Granularidade efetiva de STREAM na máquina SGI.

5.4 Relação com o desempenho

A taxa de uso de barramento de memória, apresentada na seção 4.4 para NPB, e a granularidade efetiva, apresentada na seção 5.3, são ambas métricas baseadas na contagem absoluta de acessos à memória, com a primeira relacionando-se com o tempo decorrido e a segunda com instruções de aritmética de ponto flutuante. Logo, é imediato que haja forte correspondência entre as duas observações, embora uma correlação perfeita e proporcional seja improvável, já que os programas realizam outras operações além de cálculos em ponto flutuante, e mesmo no conjunto destas há diferenças de duração entre instruções diversas. Ainda assim, programas com elevada taxa de uso de barramento nas figuras 4.7, 4.8 e 4.9 exibiram granularidade efetiva elevada nas figuras 5.2, 5.3 e 5.4, como CG, MG e SP, enquanto programas com uso mais ameno de barramento exibiram granularidade efetiva baixa, como EP e BT.

Uma vez que granularidade efetiva está relacionada com taxa de uso de barramento, então também pode ser associada com a escalabilidade. As figuras 4.1, 4.3 e 4.5, que mostram as curvas de *speedup* de NPB em SGI nas classes A, B e C, respectivamente, manifestam características semelhantes às das curvas de γ_E , mostradas nas figuras 5.2, 5.3 e 5.4. Os maiores *speedups* são de, principalmente, EP, BT e FT, que apresentam baixa granularidade efetiva; os menores *speedups* são de, principalmente, MG e SP, que possuem granularidade efetiva elevada. Assim como em *speedup*, há mais inversões de comportamento das curvas de granularidade efetiva com o aumento de *threads* nos problemas menores. Os curtos tempos de execução na classe A, em muitos casos inferiores a um minuto, sugerem maior sobrecarga da paralelização. Segue-se, então, que granularidade de *loop* é associada à escalabilidade, pois as aplicações com baixo valor de γ_L , EP e BT, apresentaram *speedups* elevados em SGI, tipicamente acima de 4, conseguindo escalar até a quantidade de *threads* igual ao número de *cores*. A aplicação FT, que possui valor intermediário de γ_L , apresentou *speedups* intermediários, por volta de 4. Já as aplicações com valor de γ_L elevado, MG e SP, apresentaram *speedups* inferiores a 4 em SGI e, nos tamanhos de problema B e C, escalaram apenas até 4 *threads*, falhando em aproveitar todos os recursos computacionais disponíveis. Mesmo nos casos de LU e UA, em que o *loop* paralelo escolhido para representar a aplicação é pouco significativo para o comportamento total (em relação ao *loop* escolhido para as demais aplicações), há indícios de uma correspondência entre granularidade de *loop* e escalabilidade, pois ambas as aplicações, com γ_L inferior à de CG e SP, exceto LU.B, conseguem apresentar ganhos de desempenho com até 8 *threads*, ainda que modestos nas classes B e C. O menor valor de γ_L para LU é

compatível com seus maiores valores de *speedup* em relação a UA. Os pequenos *speedups* de UA em SGI nas classes B e C, inferiores a 2,5, também são justificados pelo uso de *locks* OpenMP, que atuam essencialmente serializando os acessos por eles protegidos.

Similarmente ao que ocorre com NPB, os elevados valores de γ_L para STREAM correspondem à sua elevada taxa de uso de barramento em SGI, indicada na tabela 4.3, enquanto o baixo valor de γ_L de MultMat causa uma taxa de uso de barramento mais moderada, como mostrado na tabela 4.4. Também nestes dois programas a granularidade de *loop* sugere uma associação com a escalabilidade.

A comparação direta entre γ_L e β_M é inadequada devido ao mecanismo de cache, que possui diferentes níveis, capacidades e graus de compartilhamento entre os sistemas computacionais; ainda assim, é esperado que β_M mais elevado garanta melhor escalabilidade para um mesmo valor de γ_L . De fato, a maior largura de banda de HP em comparação a SGI, refletida em sua maior proporção β_M , garantiu melhores escalabilidades para as aplicações com granularidade γ_L maior. As aplicações CG, MG e SP, que em SGI apresentam limitação clara à escalabilidade em 4 *threads*, conseguiram escalar em HP no mínimo até 6 *threads*, como observado na figura 4.19, com CG e MG diminuindo consideravelmente seu ritmo de crescimento de *speedup* somente em 8 *threads*. Esta melhoria, contudo, não foi proporcional ao incremento de 122,02% em β_M ; a escalabilidade de FT, por outro lado, foi prejudicada na execução em HP e apresentou estagnação a partir de 6 *threads*, o que pode estar associado a especificidades do algoritmo utilizado.

Desconsiderando as situações em que há mais *threads* do que *cores*, já abordadas como não causando melhorias na execução dos programas, na máquina AMD não se manifestaram na escalabilidade limitações severas o suficiente para impedir ganhos com o acréscimo de *threads*, pois todas as aplicações testadas, incluindo STREAM, conseguiram obter diminuições no tempo de execução com mais *threads*, mesmo que modestas. As aplicações com menor γ_L apresentaram menores *speedups*, e as com maiores γ_L , maiores *speedups*. O comportamento inesperado do *speedup* de CG pode estar associado a potenciais diferenças na localidade de seus acessos e à diferente organização de cache entre as duas máquinas. A pequena quantidade de *cores* em AMD prejudica a apreciação de potenciais benefícios com seu maior β_M em relação a SGI, embora teriam valor de β_M próximos se possuíssem a mesma quantidade de *cores*.

O sistema MTL, embora possua a segunda menor proporção β_M , apresenta escalabilidade superior aos demais. Apesar do baixo valor de granularidade de *loop* de EP garantir a maior curva de *speedup* na figura 4.24 e os valores elevados de granularidade de MG, SP

e STREAM induzirem as curvas de *speedup* mais baixas, não há uma separação clara entre o comportamento de *speedup* de BT, CG, FT e LU, que apresentam γ_L variando de 0,856 até 1,800. Além de diferentes localidades de acesso entre as aplicações, a organização de memória NUMA contribui para tal resultado; na prática, impedindo-se que *threads* sejam executadas em *cores* diferentes de onde já executaram e assumindo que os dados são mantidos pela biblioteca de *threads* e pelo sistema operacional no nó NUMA mais próximo ao *core* que os usam, a competição pelos acessos à memória se dá majoritariamente entre as *threads* que executam nos dez *cores* de cada nó NUMA, e não simplesmente entre todos os 40 *cores* presentes. Uma estratégia de sugestão automática de quantidade de *threads* precisa considerar tais fatores.

Embora a granularidade de *loop* γ_L , tal como definida neste trabalho, tenha razoável associação com benefícios da paralelização quando se considera apenas as aplicações de STREAM e MultMat, a associação de comportamento entre NPB e STREAM com base em γ_L é falha. As grandes diferenças na granularidade efetiva entre NPB e STREAM são inconsistentes com as pequenas diferenças em γ_L , especialmente entre TRIAD e FT. O valor intermediário de γ_L na tabela 5.2 para o *loop* TRIAD de STREAM, ligeiramente inferior ao de FT, não garantiu que TRIAD tivesse escalabilidade melhor do que FT. Na verdade, nos sistemas AMD e SGI, tanto STREAM como um todo (mostrado na seção 4.5) quanto seus quatro componentes considerados separadamente apresentaram *speedups* pequenos, inferiores a 1,7, e cessaram de manifestar melhorias com a paralelização a partir de 4 *threads*. As diferenças no código-fonte das aplicações apontam três possíveis motivos para esta discrepância: processamento que não foi considerado em γ_L , o modo de acesso aos dados e tamanho dos *arrays* manipulados. Enquanto cada um dos quatro componentes de STREAM é um *loop* simples, sem aninhamento, FT possui dois *loops* aninhados na função mais custosa, resultando em operações de controle de iterações mais frequentes. Também há processamento no acesso aos *arrays* de FT para cálculo de índices, enquanto STREAM acessa seus *arrays* diretamente com a variável de controle do *loop*. Os *arrays* de STREAM são acessados sequencialmente na memória, resultando em localidade espacial máxima, porém nenhuma posição é reusada dentro do *loop*, tornando nula a localidade temporal. O tamanho dos *arrays*, ajustado para exceder em no mínimo quatro vezes a capacidade de cache do processador, contribui para a ausência dos dados no cache no início de cada *loop*, provocando os frequentes acessos à memória observados na tabela 4.3. As menores taxas de uso de barramento de FT, mostradas nas figuras 4.7, 4.8 e 4.9, sugerem que este *benchmark* tem maior localidade de acessos a dados, indicando que a localidade também é relevante na determinação da quantidade de *threads*.

Tabela 5.3: Granularidade de *loop* de aplicações de NPB e MultMat após alterações.

programa		γ_L original	γ_L novo
NPB	EP	0,211	0,267
	FT	1,333	0,800
	MG	1,667	1,000
<i>MultMat-fina</i>		1	1,667
<i>MultMat-grossa</i>			0,500

5.5 Variações na granularidade

Algumas aplicações tiveram sua granularidade de *loop* alterada a fim de se avaliar os efeitos na escalabilidade. Devido à maior simplicidade de alteração na granularidade, as aplicações EP, FT e MG de NPB, e também STREAM e MultMat, tiveram suas operações de ponto flutuante e acessos a *arrays* dentro do *loop* paralelo ajustados de forma a obter os novos valores de γ_L indicados na tabela 5.3. As listagens de código-fonte dos *loops* originais e modificados dos referidos programas são mostradas no apêndice B. De EP foram suprimidas uma operação de raiz quadrada, uma de logaritmo, uma divisão e uma multiplicação; em FT foram acrescentadas uma operação de raiz quadrada e uma de logaritmo; em MG, foram adicionadas multiplicações; MultMat-fina recebeu mais três acessos a *arrays* e uma soma; finalmente, MultMat-grossa foi acrescida de uma soma, uma multiplicação e uma divisão. Em todos os casos, as aplicações deixaram de implementar o algoritmo inicial, emitindo resultados errôneos para o problema original, o que foi indiferente para a análise conduzida neste trabalho.

5.5.1 NPB

A granularidade efetiva γ_E em SGI da versão modificada de EP foi igual a um décimo de γ_E da versão original; no entanto, a quantidade absoluta de acessos à memória foi reduzida pela metade, indicando que há outros acessos à memória envolvidos nas rotinas de biblioteca matemática além dos considerados em γ_L . A granularidade efetiva de FT foi igual a cerca de um décimo de γ_E da versão original; e a de MG foi, em média, 16% inferior à de sua contraparte.

Os tempos de execução dos programas alterados foram medidos e seu *speedup* comparado com a versão original. As figuras 5.7, 5.8, 5.9 e 5.10 mostram as curvas de *speedup* de EP, FT e MG na classe C nos sistemas SGI, HP, AMD e MTL, respectivamente, tanto na versão original quanto na versão com γ_L alterado. A figura 5.7 mostra que EP com γ_L 26,5% superior que sua versão original, embora em termos absolutos apenas ligeiramente

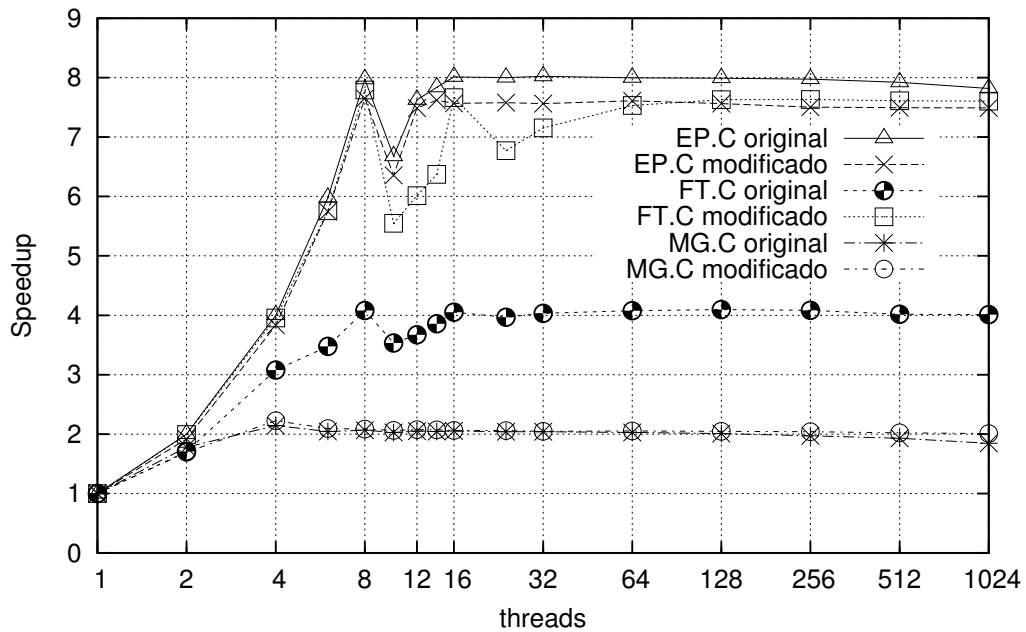


Figura 5.7: Curvas de *speedup* para as versões originais e com granularidade modificada de EP, FT e MG na máquina SGI.

maior, não apresentou mudanças expressivas em sua escalabilidade em SGI, continuando a apresentar ganhos com o maior uso de *threads*; seu *speedup*, porém, foi 4,07% menor, em média, do que o do programa original. Já FT, com γ_L igual a cerca de 60% da versão original, apresentou grandes melhorias na escalabilidade, com *speedup* máximo até 91% superior ao de sua versão com maior valor de granularidade. MG, por outro lado, com γ_L também igual a cerca de 60% da granularidade da versão original, atingiu *speedup* apenas 3,76% superior à de sua versão sem modificação nas execuções com 4 *threads*. A influência sentida pelo tempo de execução foi variada. EP executou mais rapidamente sem as operações custosas de raiz quadrada e logaritmo, enquanto FT executou até 100% mais demoradamente pela inclusão de tais operações. MG, mesmo realizando mais processamento do que anteriormente, concluiu suas execuções em menos tempo do que sua versão não modificada, no mínimo 4,5% e no máximo 12,4% mais rapidamente que sua contraparte.

No sistema HP, os resultados foram parecidos com SGI. A figura 5.8 mostra que a curva de *speedup* de FT modificado continuou crescendo até doze *threads*, representando uma melhoria de 58,1% em relação à aplicação original nas execuções com quantidade de *threads* igual ao número de *cores*, embora os tempos de execução tenham sido cerca de o

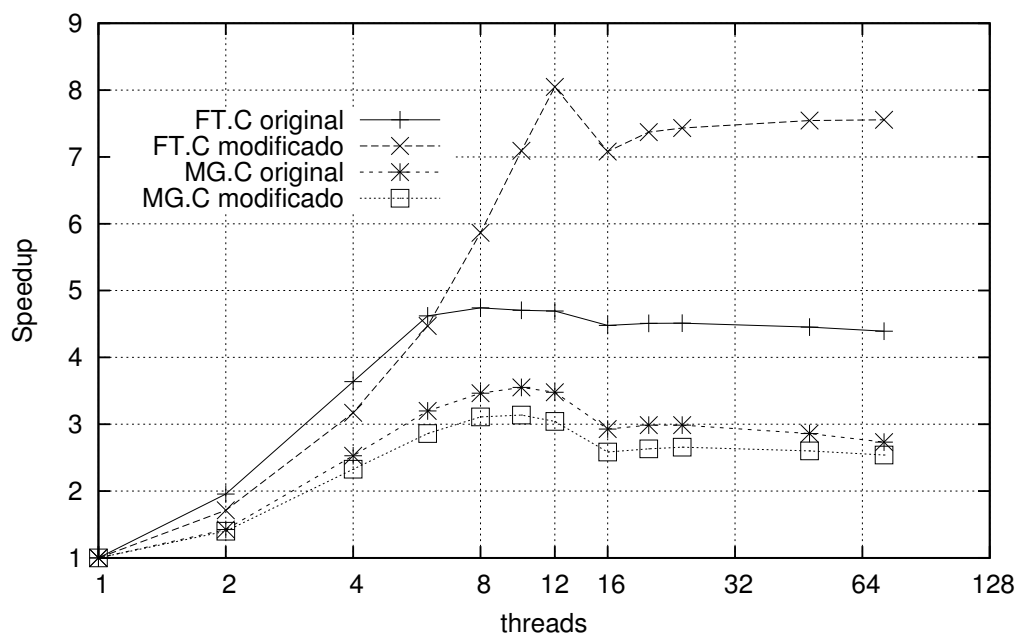


Figura 5.8: Curvas de *speedup* para as versões originais e com granularidade modificada de FT e MG na máquina HP.

dobro da versão original. A versão alterada de MG atingiu *speedup* máximo 11,7% inferior que sua contraparte, porém suas execuções concluíram em menos tempo: a versão com γ_L menor executou ao menos 6,1% mais rapidamente, com 12 *threads*, e até 16,4% mais rapidamente, com 2 *threads*, do que o programa original nos mesmos graus de paralelismo. EP não foi executado em HP.

A escalabilidade em AMD das aplicações selecionadas, exibida na figura 5.9, também aponta maiores *speedups* para a alteração em FT. A diferença entre os *speedups* máximos das duas versões foi de 13,9% com 4 *threads*; o tempo de execução, por sua vez, foi em média 66% maior na versão alterada. Ao contrário das máquinas anteriores, o *speedup* de MG em AMD foi sutilmente maior na versão alterada, em média 1,1% superior. Seu tempo de execução foi cerca de 15% mais rápido do que o original em todos os graus de paralelismo até 4 *threads*. EP também não foi executado em AMD.

Em MTL houve uma pequena diminuição do *speedup* de EP com as modificações, inferior a 4,4%, como mostrado na figura 5.10. Todas as execuções foram mais rápidas que suas correspondentes originais devido à remoção de duas operações matemáticas custosas. A aplicação FT modificada atingiu *speedup* máximo 30% superior à versão original com 40 *threads*, porém seu tempo de execução foi até o triplo do programa original. Esta maior

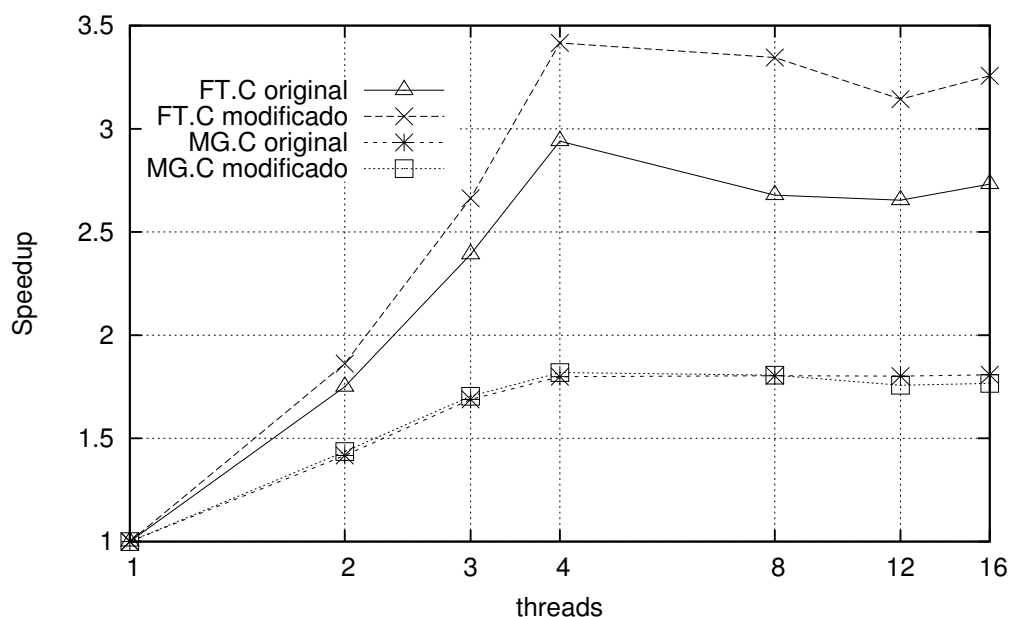


Figura 5.9: Curvas de *speedup* para as versões originais e com granularidade modificada de FT e MG na máquina AMD.

diferença no tempo de execução em relação aos demais sistemas utilizados é compatível com a maior carga de processamento da alteração de FT e com a menor velocidade de processamento de cada *core* em MTL, como exposto na tabela 5.1. As mudanças fizeram o *speedup* de MG ser em média 5,2% menor que sua versão original, sendo 14,7% inferior nas execuções com 40 *threads*. Seu tempo de execução, contudo, foi em média 25% mais rápido.

Sob o ponto de vista da escalabilidade, FT se beneficiou do maior volume de processamento, atingindo *speedups* maiores, enquanto MG foi ligeiramente prejudicado. Considerando o desempenho absoluto, contudo, FT foi prejudicado com o maior processamento e MG foi beneficiado. A taxa de uso de barramento de acesso à memória, identificada no capítulo 4 como uma importante limitação para a escalabilidade em um sistema *multicore*, é mostrada na figura 5.11 para os três programas alterados em SGI e confirma as diferenças nos tempos e na escalabilidade das execuções. FT teve seu uso de barramento diminuído em até 95% em relação à sua versão original, enquanto MG apresentou redução média de 9% na frequência de uso do barramento em comparação com sua contraparte sem alterações. Os acessos à memória de MG são tão frequentes que o maior volume de processamento diminuiu sua taxa de uso de barramento para abaixo de

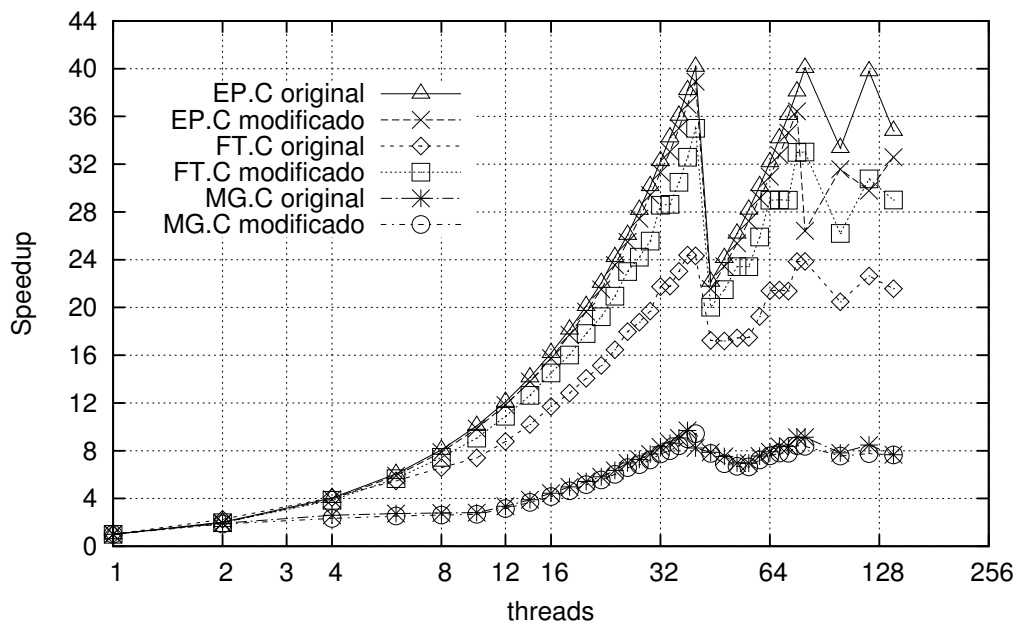


Figura 5.10: Curvas de *speedup* para as versões originais e com granularidade modificada de EP, FT e MG na máquina MTL.

80%, minimamente reduzindo a competição pelo acesso à memória e consequentemente diminuindo a latência no acesso à memória.

5.5.2 MultMat

A granularidade efetiva das variações de MultMat em SGI é mostrada na figura 5.12; neste gráfico observa-se que tanto *MultMat-fina* quanto *MultMat-grossa* apresentaram γ_E menor do que a versão original da aplicação. Suas taxas de uso de barramento, indicadas na figura 5.13, associam-se com a granularidade efetiva, com MultMat original apresentando o maior uso, seguido de *MultMat-fina* e *MultMat-grossa*, nesta ordem, apesar do maior valor de granularidade de *loop* de *MultMat-fina* indicar intuitivamente, segundo os resultados da seção 5.4, maior frequência de acessos do que sua versão original. A quantidade absoluta de acessos ao barramento de memória sofreu um aumento de menos de 10% em *MultMat-fina* em relação à versão original, porém a quantidade absoluta de *flops* foi 50% maior que a da versão sem modificação, sugerindo que otimizações do compilador no acesso aos dados, como uso mais frequente de registradores, podem ter sido responsáveis pelo menor uso de barramento. Isto mostra uma limitação da obtenção de γ_L

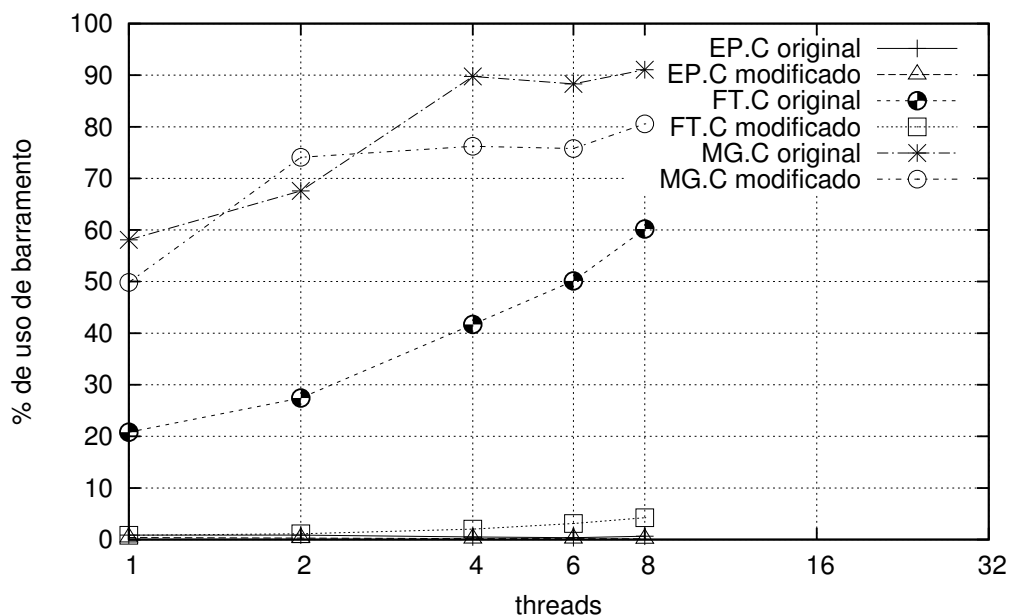


Figura 5.11: Taxas de uso de barramento de memória para as versões originais e com granularidade modificada de EP, FT e MG na máquina SGI.

diretamente a partir do código-fonte da aplicação, pois desta forma está sujeito a alterações pelo compilador. O compilador mostra-se, então, como o candidato mais apropriado para a determinação da granularidade de *loop*, pois tem acesso a quais instruções do processador serão realmente utilizadas. Houve pequena diferença entre as escalabilidades das duas versões modificadas, como indicado pelas curvas de *speedup* da figura 5.14. Em AMD e SGI, MultMat-grossa teve *speedups* levemente maiores que MultMat-fina; já em MTL, em execuções sem uso de afinidade de CPU, o contrário ocorreu, porém com diferenças mais sutis nos *speedups*. A comparação com a escalabilidade da versão original de MultMat nas mesmas máquinas, mostrada na figura 4.30, mostra que a variação de granularidade testada não afetou a capacidade da aplicação de obter ganhos com o paralelismo.

5.5.3 STREAM

Nos experimentos com as aplicações anteriores, a alteração artificial no volume de processamento e nos acessos aos dados foi feita no código-fonte dos programas, estando sujeitas a mudanças durante as etapas de otimização de código do compilador. Uma abordagem diferente de variação de granularidade foi utilizada em STREAM. Para evi-

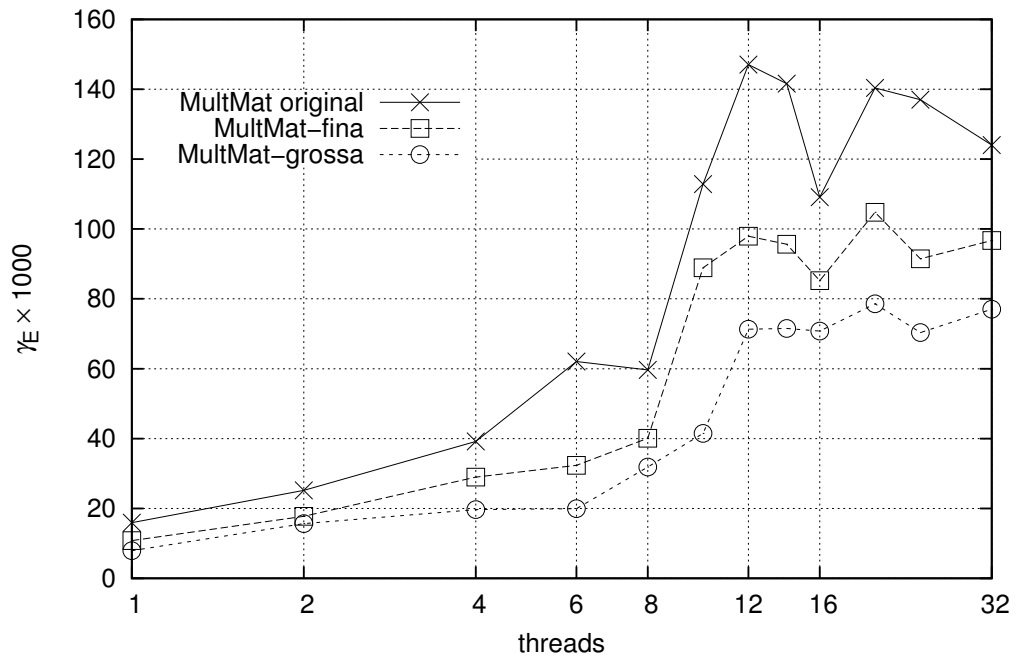


Figura 5.12: Granularidade efetiva para as versões original e com granularidade modificada de MultMat na máquina SGI.

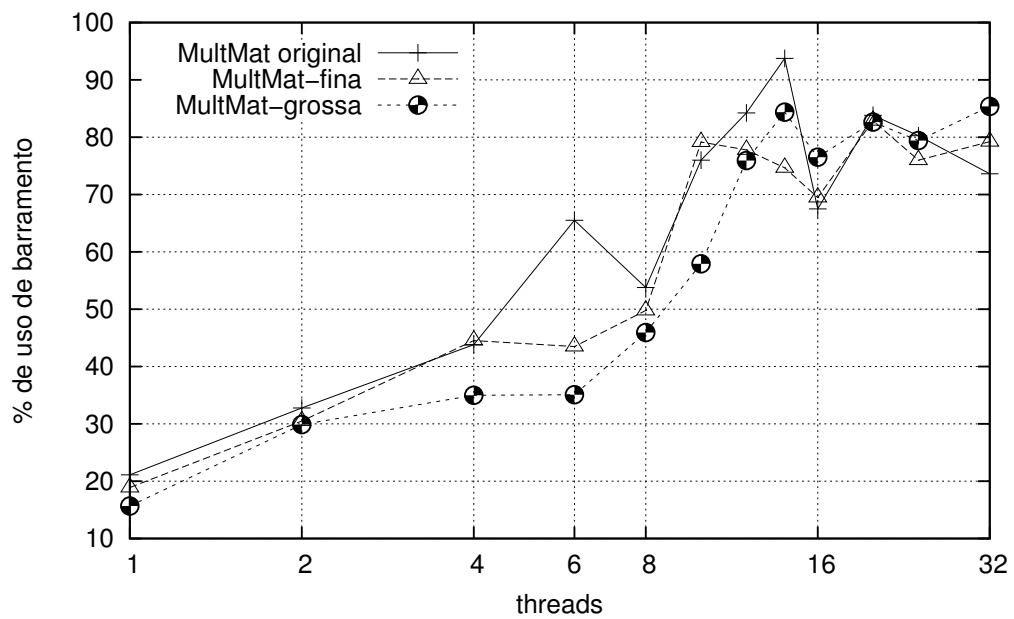


Figura 5.13: Taxas de uso de barramento de memória para as versões original e com granularidade modificada de MultMat na máquina SGI.

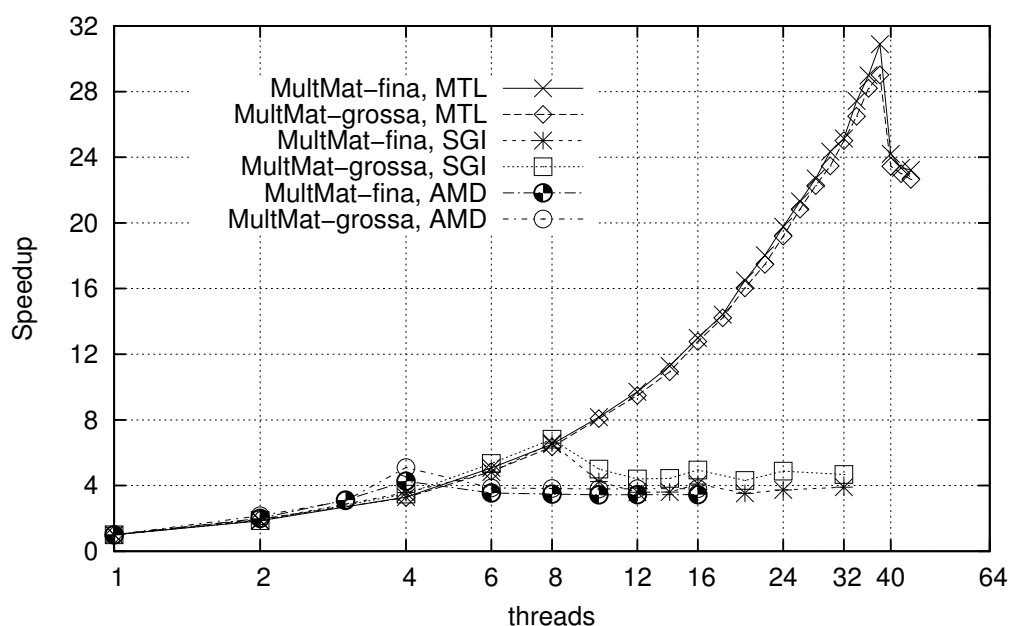


Figura 5.14: Curvas de *speedup* para as versões de MultMat com granularidade alterada nas máquinas MTL, SGI e AMD.

tar que otimizações do compilador afetassem o processamento introduzido, instruções do processador em linguagem de montagem (*assembly*) foram adicionadas aos *loops* de STREAM dentro de blocos demarcados pelas palavras-chave `__asm__`, para indicar trecho de código em linguagem de montagem, e `__volatile__`, que indica que uma instrução ou trecho *assembly* tem efeitos colaterais importantes (FREE SOFTWARE FOUNDATION, INC, 2012), impedindo que o compilador as considere inúteis e as remova. A unidade básica de processamento incluída consistiu da instrução *MULSD*, pertencente ao subconjunto de instruções SSE2 (*Streaming SIMD Extensions 2*) e que realiza uma multiplicação entre números escalares de ponto flutuante com precisão dupla. Os dois operandos da multiplicação foram fixados em registradores da CPU. Os componentes de STREAM foram acrescidos dos volumes de processamento adicional 1, 2, 5, 10, 15, 20, 25 e 30 aos seus *loops* paralelos, sem prejuízo das operações já existentes. Os componentes foram então compilados e executados separadamente na máquina SGI. O código-fonte das mudanças em STREAM é apresentado no apêndice B, na listagem B.10, ao passo que o código-fonte do programa auxiliar que ajusta sua granularidade é apresentado na listagem B.11. As figuras 5.15, 5.16, 5.17 e 5.18 mostram a escalabilidade de COPY, SCALE, ADD e TRIAD, respectivamente. As execuções com maiores volumes de processamento tiveram

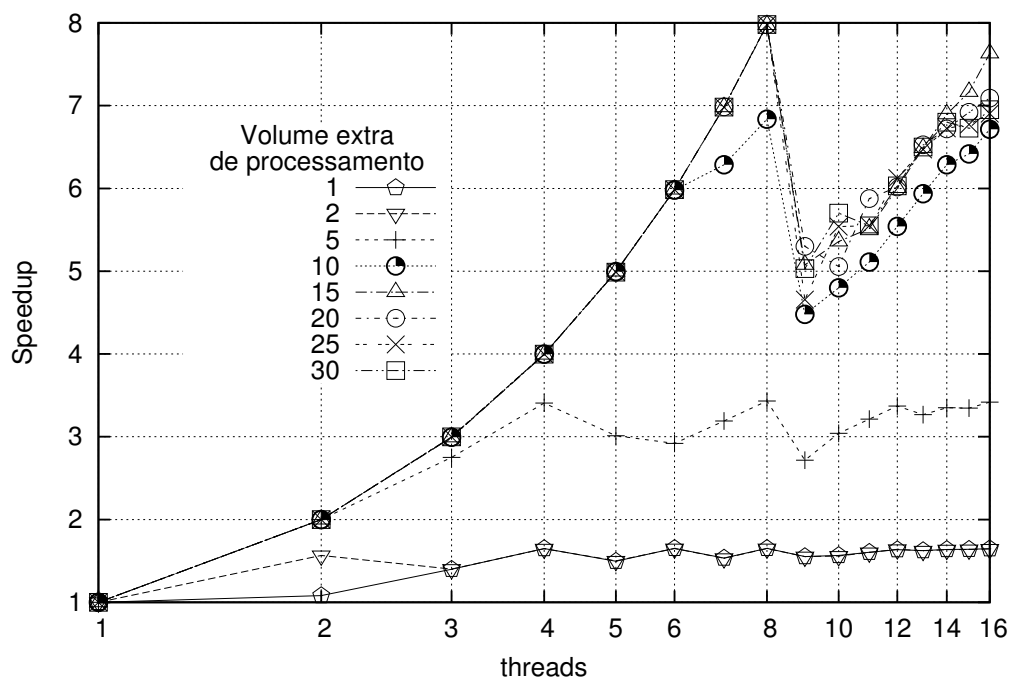


Figura 5.15: Curvas de *speedup* para o componente COPY de STREAM com variações no volume de processamento na máquina SGI.

melhor escalabilidade, embora tenha sido necessário grandes volumes de cálculos para a obtenção de *speedups* lineares. As execuções com volume extra de processamento igual a uma multiplicação e a duas multiplicações atingiram *speedup* máximo inferior a 2 em todas as quantidades de *threads* testadas. As execuções com 20, 25 e 30 multiplicações adicionais manifestaram *speedup* linear em todos os componentes, com ganhos no desempenho proporcionais à quantidade de *threads* usadas. ADD e TRIAD exigiram volumes de processamento maiores para habilitar ganhos com o paralelismo, pois as execuções com 10 multiplicações cessaram de apresentar melhorias no *speedup* a partir de 6 *threads*, atingido *speedup* máximo igual a 5 com 8 *threads*, ao passo que as execuções com 15 multiplicações tiveram seu ritmo de crescimento de *speedup* reduzidos a partir de 6 *threads*, atingindo *speedup* máximo inferior a 7,5 com 8 *threads*. Em COPY e SCALE, por outro lado, as execuções com 10 multiplicações conseguiram *speedup* máximo ligeiramente inferior a 7 com oito *threads*, com ganhos expressivos em relação às execuções com 6 *threads*.

As taxas de uso de barramento de COPY, SCALE, ADD e TRIAD em SGI são mostradas, respectivamente, nas figuras 5.19, 5.20, 5.21 e 5.22. Todas as execuções com 1 e 2 multiplicações adicionais ultrapassaram 70% de uso de barramento e as execuções com 5 multiplicações adicionais ultrapassaram 70% com no máximo 4 *threads*. Já as execuções

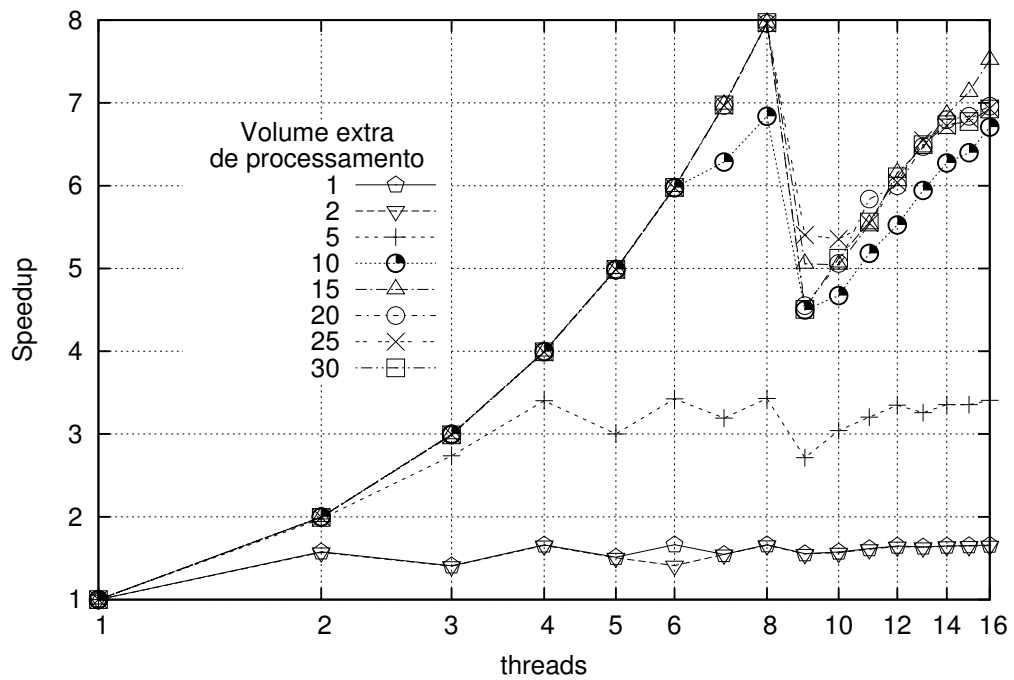


Figura 5.16: Curvas de *speedup* para o componente SCALE de STREAM com variações no volume de processamento na máquina SGI.

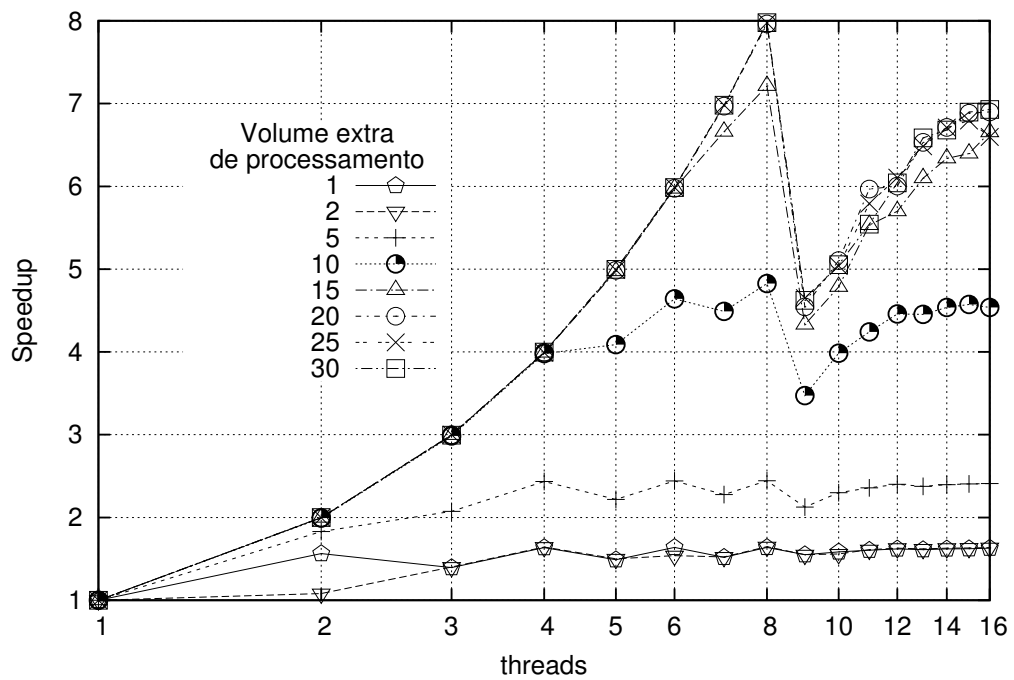


Figura 5.17: Curvas de *speedup* para o componente ADD de STREAM com variações no volume de processamento na máquina SGI.

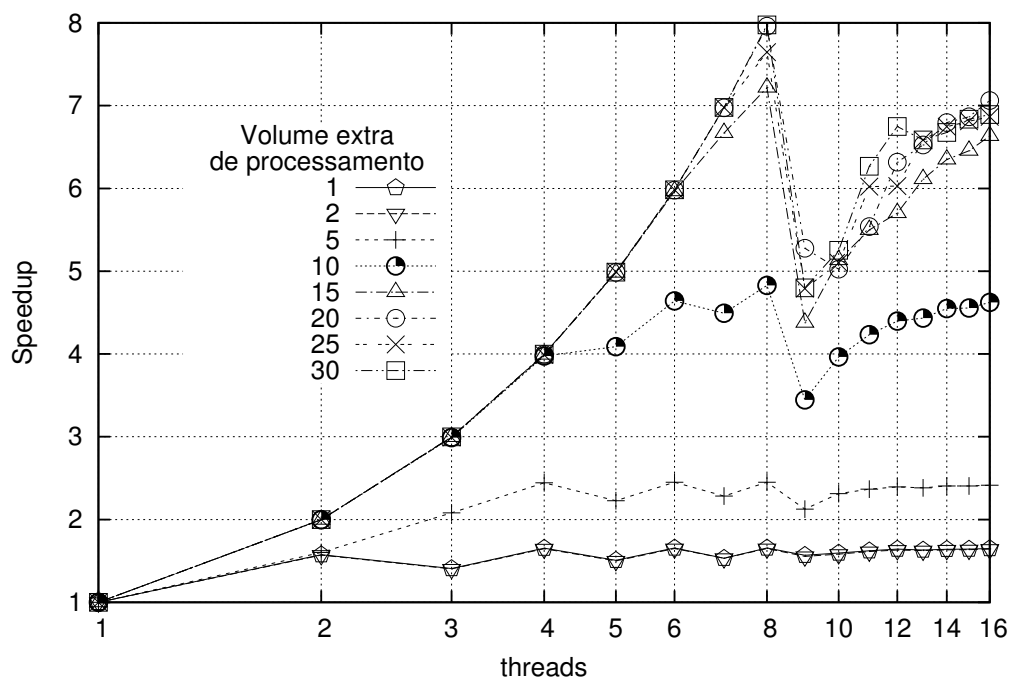


Figura 5.18: Curvas de *speedup* para o componente TRIAD de STREAM com variações no volume de processamento na máquina SGI.

com 20, 25 e 30 multiplicações sequer atingiram 50% de uso de barramento com até 8 *threads*. As execuções de COPY e SCALE com 10 multiplicações, que apresentam diminuição do ritmo de crescimento de *speedup* a partir de 6 *threads*, excederam 50% de uso de barramento nas mesmas 6 *threads*, porém só superaram 70% com mais *threads* do que os oito *cores* da máquina. Em ADD e TRIAD, as execuções com 10 multiplicações excederam 50% de uso de barramento já em 4 *threads* e superaram os 70% de uso de barramento com seis *threads*, exatamente no grau de paralelismo em que cessam os ganhos de desempenho. Como, segundo a tabela 3.1, ADD e TRIAD manipulam 3 *arrays* e COPY e SCALE manipulam apenas 2, foi necessário maior volume de processamento para compensar os acessos mais frequentes a dados na memória em ADD e TRIAD e para igualar a granularidade dos quatro componentes.

5.6 Variações na localidade

Conforme visto na seção 5.4, algumas aplicações com valores próximos de granularidade de *loop* γ_L não tiveram comportamentos parecidos de escalabilidade, sugerindo que a localidade dos acessos aos dados pode ser um fator influente na paralelização. Nesta

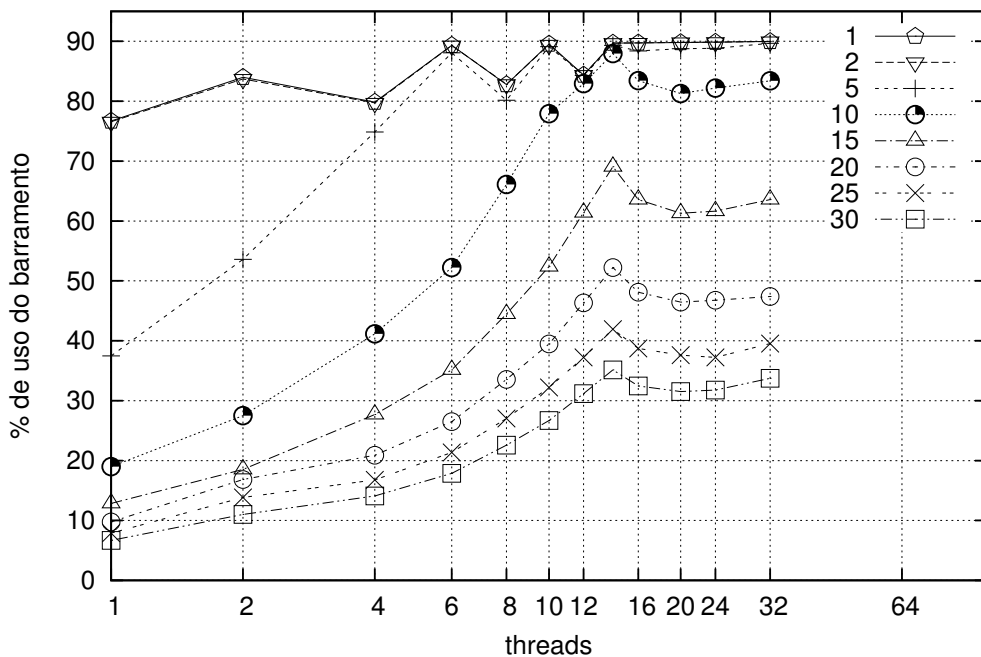


Figura 5.19: Taxas de uso de barramento de memória para o componente COPY de STREAM com variações no volume de processamento na máquina SGI.

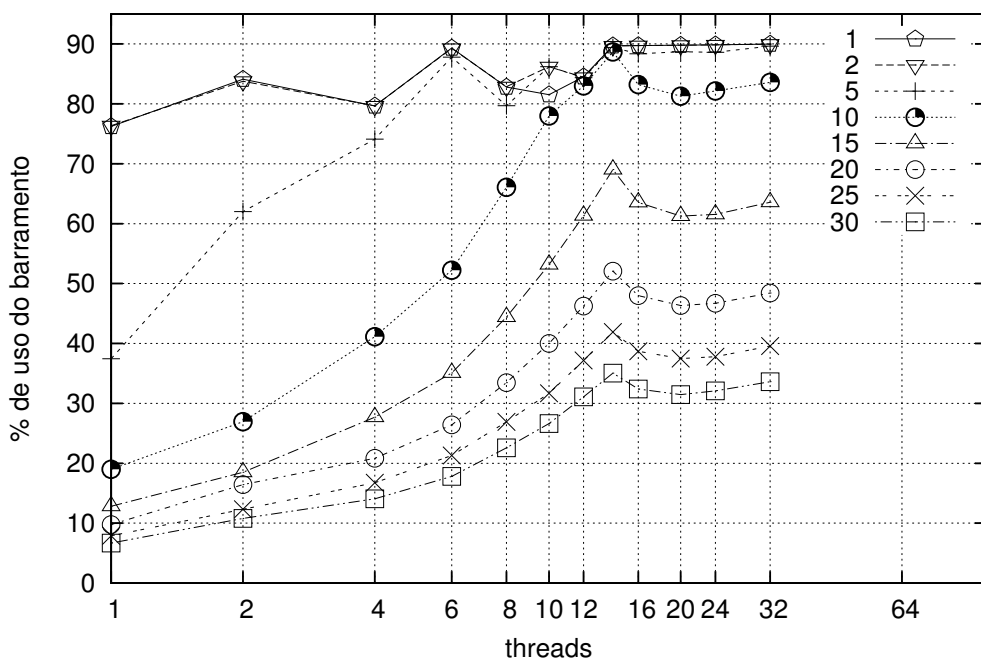


Figura 5.20: Taxas de uso de barramento de memória para o componente SCALE de STREAM com variações no volume de processamento na máquina SGI.

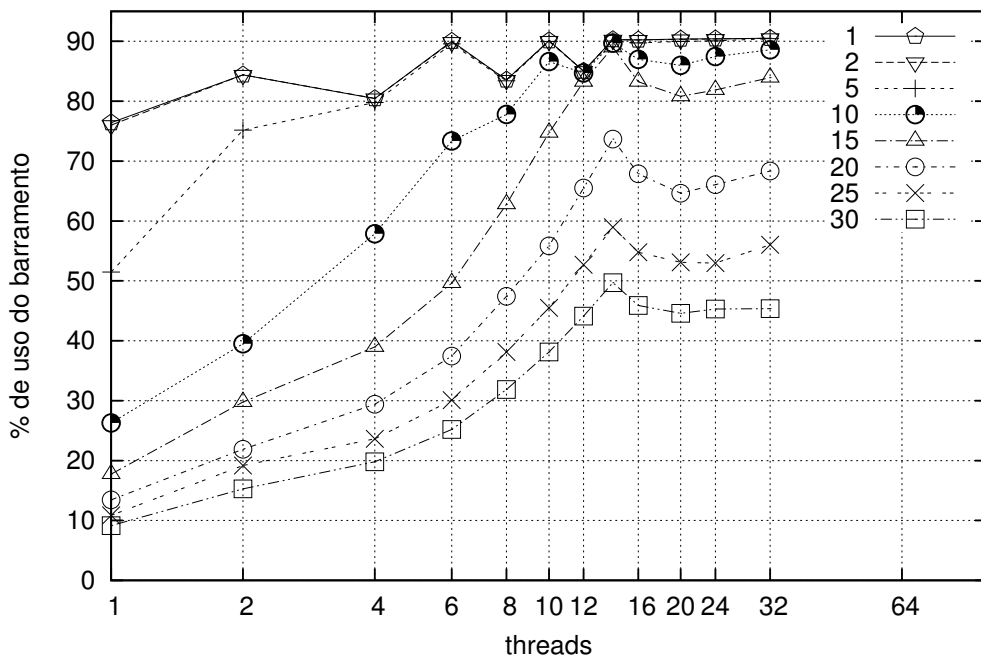


Figura 5.21: Taxas de uso de barramento de memória para o componente ADD de STREAM com variações no volume de processamento na máquina SGI.

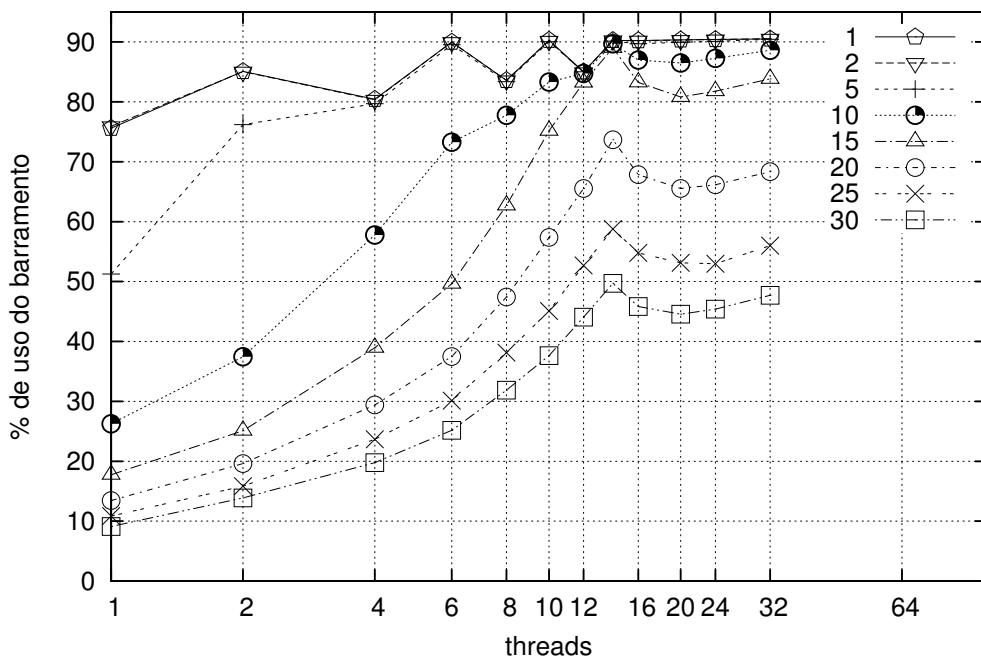


Figura 5.22: Taxas de uso de barramento de memória para o componente TRIAD de STREAM com variações no volume de processamento na máquina SGI.

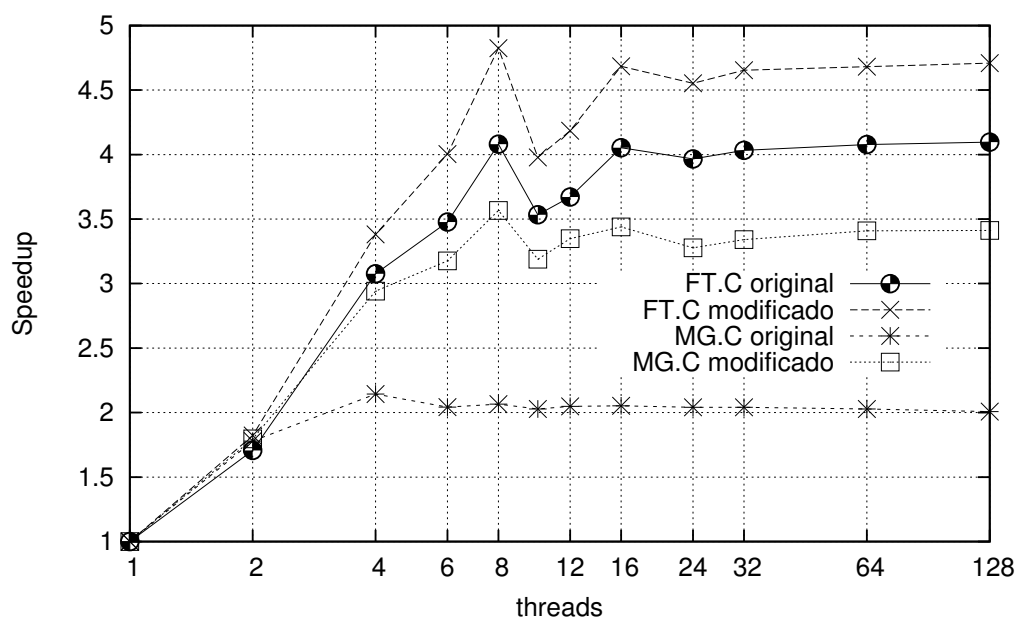


Figura 5.23: Curvas de *speedup* para as versões originais e as com *loops* invertidos de FT e MG na máquina SGI.

seção, são apresentados resultados de desempenho de algumas aplicações após alteração da localidade dos acessos. As mudanças foram realizadas em FT, MG e MultMat e consistiram na inversão dos *loops* aninhados dentro da função mais custosa de cada aplicação. Em FT, os dois *loops* mais internos foram alternados; em MG, o *loop* mais interno foi trocado pelo mais externo; em MultMat, o *loop* mais interno, da variável de iteração k , foi invertido com o *loop* de nível imediatamente superior, da variável de controle j , transformando o grupo de *loops* ijk em ikj e resultando no programa *MultMat-inv*. Ao contrário do ocorrido na seção 5.5, a alteração da localidade dos acessos aos dados não altera o resultado do problema tratado na aplicação.

A escalabilidade de FT e MG é comparada com suas versões de localidade modificada nas figuras 5.23 e 5.24 nos sistemas SGI e MTL, respectivamente. Em SGI, a inversão dos *loops* causou melhoria na escalabilidade, provocando aumento de até de 18% no *speedup* máximo de FT e de até 66% no *speedup* máximo de MG. Não obstante, o tempo de execução foi prejudicado, sofrendo aumento médio de 6% em FT e de espantosos 500% em MG, o que aponta para um menor aproveitamento dos dados no cache e conseqüente maior uso de barramento. MG também sofreu aumento excessivo de seu tempo de execução em MTL, que foi em média 580% superior ao tempo das execuções sem inversão de *loops*;

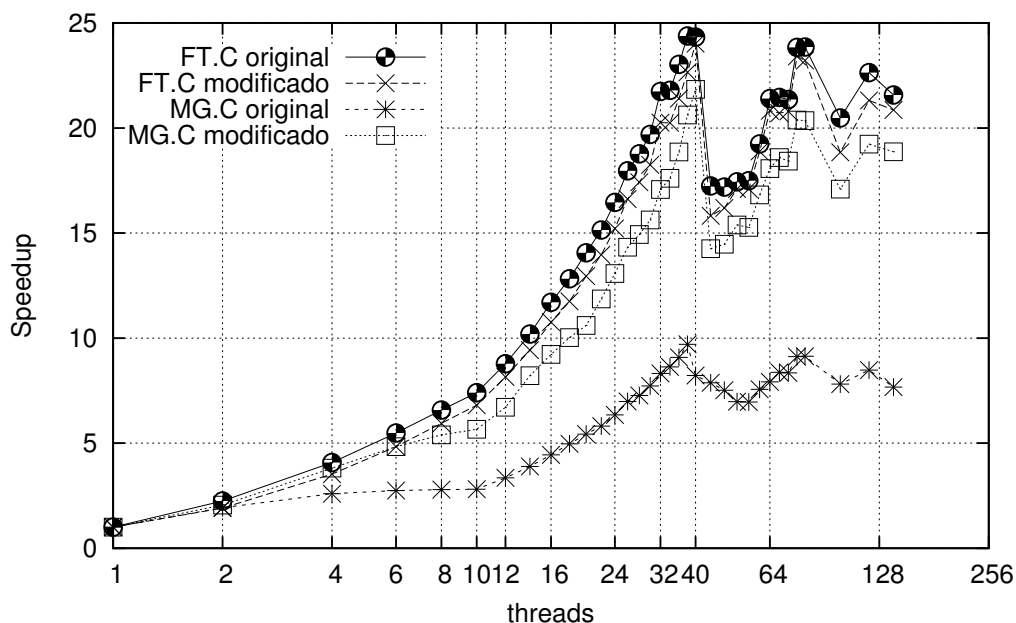


Figura 5.24: Curvas de *speedup* para as versões originais e as com *loops* invertidos de FT e MG na máquina MTL.

sua escalabilidade, contudo, foi tal a ponto de permitir *speedup* máximo elevado, superior a 21, enquanto sua versão original atinge *speedup* máximo de apenas 10. O tempo de execução de FT com inversão de *loops* foi, em média, 34% superior ao da versão original, e sua curva de *speedup* foi inferior à da versão original, diferentemente do observado em SGI.

MultMat teve suas curvas de *speedup* registradas na figura 5.25 para os sistemas AMD e SGI e na figura 5.26 para o sistema MTL. Nas três máquinas, a inversão de *loops* provocou diminuição do *speedup* em relação à versão não modificada, embora a curva de *speedup* não deixe de ser crescente com o aumento de *threads* até a quantidade de *cores*. O tempo de execução, todavia, foi grandemente melhorado, com redução média de 52% em SGI, 87% em AMD e 63% em MTL em relação à versão original nas execuções sequenciais e paralelas com quantidade de *threads* no máximo igual à quantidade de *cores*. A figura 5.27 mostra que MultMat com inversão de *loops* fez uso elevado do barramento em SGI, excedendo 70% de taxa de uso já a partir de 2 *threads* e superando 80% a partir de quatro *threads*, enquanto sua versão sem alteração não excedeu 70% de taxa de uso de barramento. A figura 5.27 apresenta a granularidade efetiva γ_E das duas versões de MultMat em SGI e mostra uma estabilização na proporção de acessos à memória por

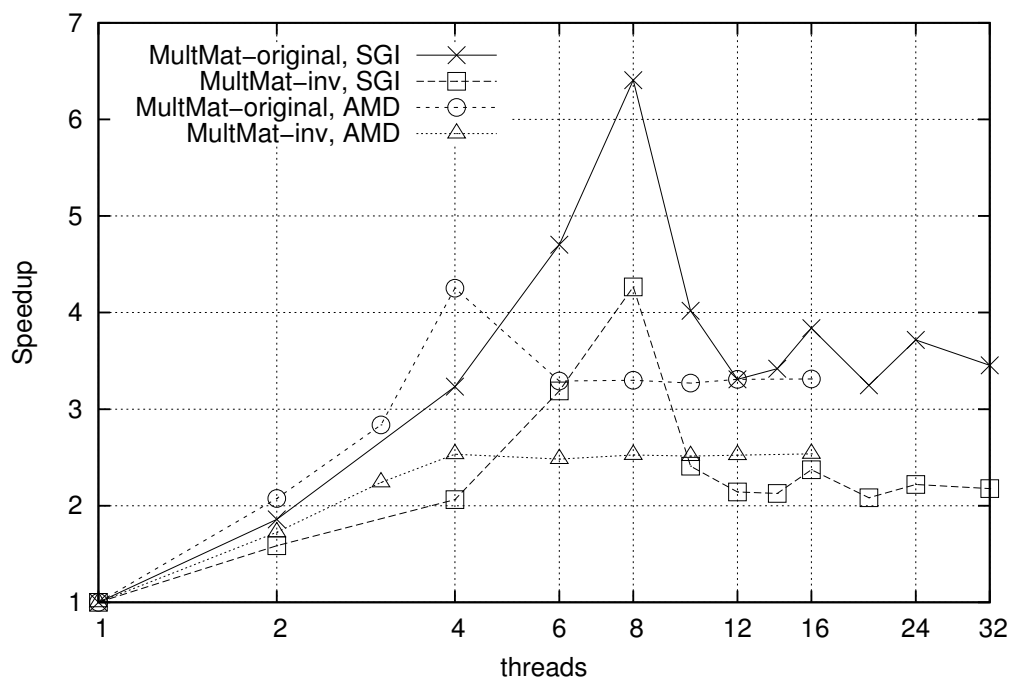


Figura 5.25: Curvas de *speedup* para a versão original e a com *loops* invertidos de MultMat nas máquinas SGI e AMD.

instruções de ponto flutuante executadas por MultMat-inv a partir de 4 *threads*, enquanto MultMat-original apresentou aumentos em sua proporção γ_E em todas as execuções com até 8 *threads*.

Nesta seção, foi mostrado que algumas aplicações com as mesmas operações, mesmo volume de processamento e mesma quantidade de acesso aos dados, e conseqüentemente mesma granularidade de *loop*, manifestaram escalabilidades distintas. Valores desiguais de granularidade efetiva, conseqüência de diferenças na localidade de acesso aos dados, fizeram com que as aplicações obtivessem benefícios distintos com a paralelização. Os efeitos da localidade foram sentidos também para uma mesma aplicação entre diferentes sistemas computacionais, como sugerido pela diferença do comportamento de *speedup* entre as duas versões de FT observadas. Estes resultados apontam para a necessidade de consideração da localidade, além do volume de processamento e da quantidade de acessos aos dados, em um mecanismo automático para sugestão de quantidade de *threads*.

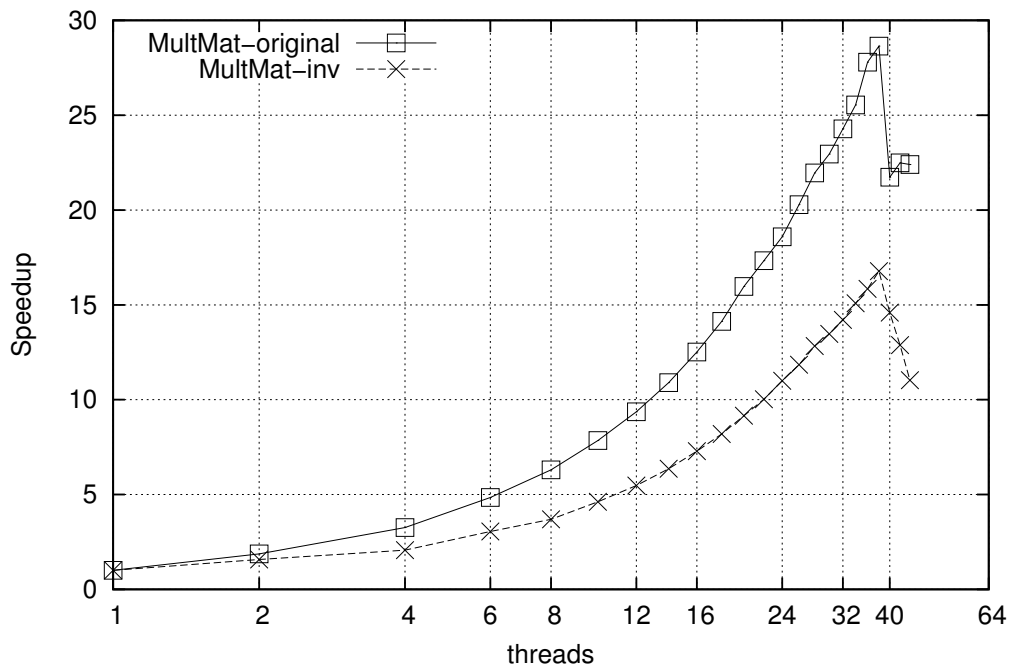


Figura 5.26: Curvas de *speedup* para a versão original e a com *loops* invertidos de MultMat na máquina MTL.

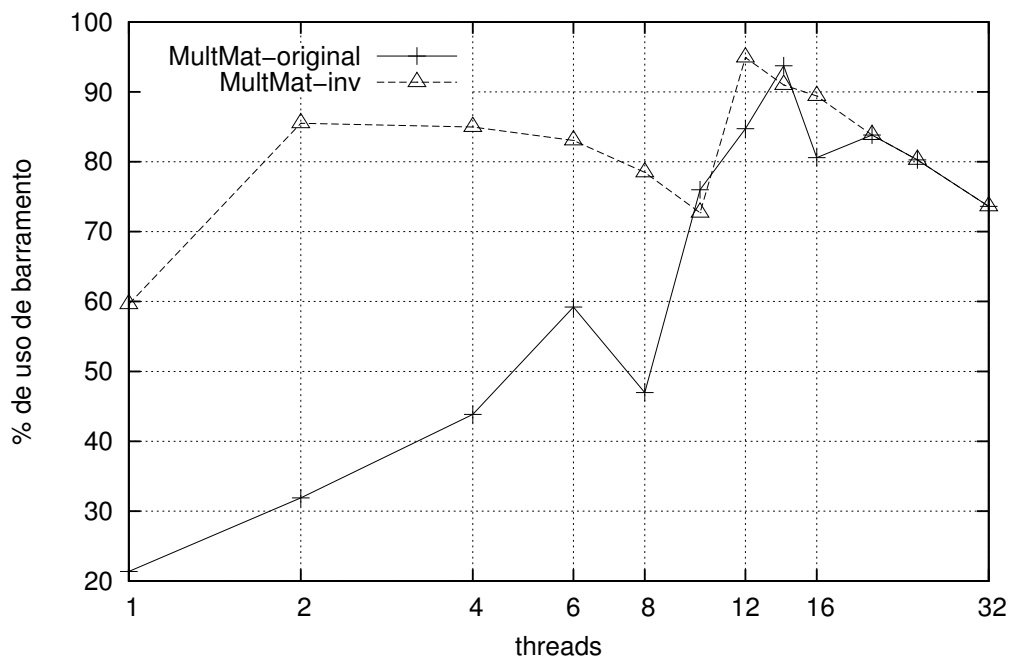


Figura 5.27: Taxas de uso do barramento de memória para a versão original e a com *loops* invertidos de MultMat na máquina SGI.

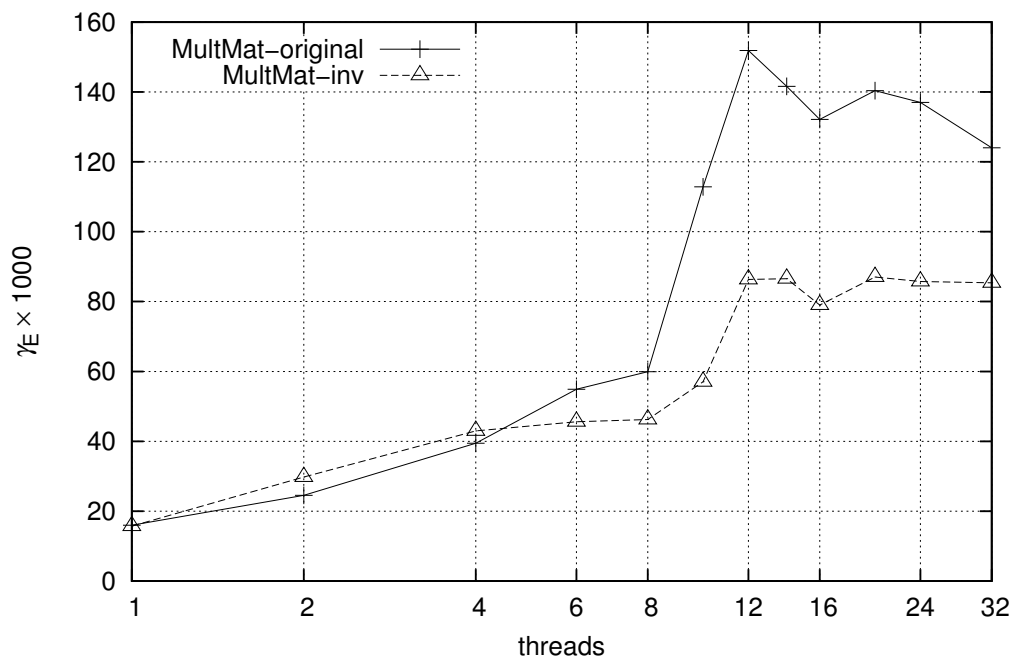


Figura 5.28: Granularidade efetiva para a versão original e a com *loops* invertidos de MultMat na máquina SGI.

5.7 Conclusões

Este capítulo investigou a associação da granularidade dos acessos com o desempenho de aplicações paralelas. O barramento de ligação com a memória foi identificado no capítulo 4 como um importante gargalo para o paralelismo, porém a quantidade absoluta de acessos à memória não foi suficiente para indicar, isoladamente, concorrências pelo uso do barramento ou canal de comunicação. A granularidade efetiva, γ_E , que indica a proporção de acessos à memória em relação ao processamento, se mostrou claramente como associada à capacidade de obtenção de ganhos com o paralelismo. A métrica γ_L , ou granularidade de *loop*, foi apresentada como uma estimativa para a granularidade efetiva. De modo geral, valores elevados de granularidade de *loop* corresponderam a maiores contenções pelo acesso à memória e a maiores valores de granularidade efetiva, enquanto valores menores de γ_L foram compatíveis com menores contenções e melhor escalabilidade. Diminuições artificiais de γ_L , através do acréscimo de processamento, resultaram em maiores valores de *speedup*, porém às custas de maiores tempos de execução artificiais.

A obtenção de γ_L diretamente a partir do código-fonte da aplicação sofre a desvan-

tagem de potencialmente não representar corretamente as operações que serão realizadas pelo processador, pois o código-fonte está sujeito a mudanças durante a etapa de compilação devido a otimizações. Além disso, a precisão do uso desta métrica depende grandemente da consideração de todo o processamento realizado, não apenas de operações matemáticas em ponto flutuante, e de sua obtenção individual para cada *loop* paralelo da aplicação. O compilador se torna, então, o candidato mais apropriado para determinar o valor de γ_L , já que tem acesso direto à forma final do programa. Uma possível abordagem para o acesso às informações necessárias para se determinar γ_L seria a consulta à representação intermediária do programa sendo processado através de alterações no código-fonte do compilador. No compilador GCC, por exemplo, o código-fonte da aplicação é inicialmente traduzido para uma linguagem chamada GENERIC e, após algumas etapas da compilação, convertido para GIMPLE, uma representação intermediária independente de máquina baseada na estrutura de dados árvore (FREE SOFTWARE FOUNDATION, INC, 2011). Os acessos a *arrays* são representados por nós do tipo ARRAY_REF em GENERIC. A macro FOR_EACH_LOOP permite consultar todos os nós de *loops* da estrutura do programa em GIMPLE; entre eles, os nós do tipo GIMPLE_OMP_FOR, que representam *loops* paralelos OpenMP. O incremento da variável de iteração, bem como os respectivos limites inferior e superior, são também armazenados na forma GIMPLE e acessíveis através dos campos *header* e *latch* do nó que representa o *loop*. Em outros compiladores, os detalhes dos acessos às informações são diferentes, porém a abordagem é semelhante.

A granularidade de *loop* é, com as considerações mencionadas no parágrafo anterior, um componente viável para ser usado em uma determinação automática de quantidade de *threads*. Por outro lado, também foi mostrado que variações na localidade causaram alterações na frequência de acesso à memória, afetando a escalabilidade das aplicações. Desta forma, um mecanismo de sugestão automática do grau de paralelismo precisaria considerar, além da granularidade, a localidade dos acessos a dados.

5.7.1 Considerações sobre modelagem analítica

A modelagem analítica de quantidade de *threads* proposta por Sun, Byna e Holmgren (2009), apresentada na seção 2.3, estabelece uma fórmula fechada para a determinação da quantidade de *threads* em aplicações com *loops* paralelos após execução instrumentada de sua versão sequencial. O modelo assume que o tempo de acesso aos dados na memória independente de seu grau de paralelismo. Porém, como mostrado nas seções anteriores,

Tabela 5.4: Valores das variáveis da equação 5.1 para o componente TRIAD de STREAM na máquina SGI.

R	$B_{sustained}$	$P_{sustained}$	H_{L1}	H_{L2}	F	L_{word}	L_{cache}
0,5	5739,5	281	0,9511	0,9516	1	8	64

há uma correlação significativa entre quantidade de *threads* e tempo de acesso aos dados, pois maiores quantidades de *threads* provocam contenções pelo acesso ao barramento de memória compartilhado, afetando a escalabilidade das aplicações. Assim, cabe testar a validade do modelo.

Assumindo sobrecarga nula na predição de acessos para busca antecipada de dados, a quantidade n_{opt} de *threads* que o modelo criado pelos autores indica de forma a não causar competição pelo barramento de acesso à memória é calculada pela equação 5.1. Conforme explicado na seção 2.3, I é o total de instruções em cada iteração do *loop* considerado; R é a proporção de instruções de acesso à memória em relação ao total de instruções do *loop* da aplicação; $B_{sustained}$ é a largura de banda de memória, em MB/s, medida pelo *benchmark* STREAM; $P_{sustained}$ é a velocidade, em MFLOPS, atingida por um *core* na execução sequencial da aplicação; H_{L1} é a taxa de acertos no cache de nível 1, enquanto H_{L2} é a taxa de acertos no cache de nível 2; F é o fator de reuso espacial; L_{word} é o tamanho, em *bytes*, de cada palavra de dados manipulada; e F_{cache} é o tamanho da linha de cache, em *bytes*. Tais valores foram medidos, com a ferramenta VTune e com inspeção do código-fonte, para o componente TRIAD de STREAM em SGI e seus valores são indicados na tabela 5.4. I é irrelevante para o resultado final e não foi determinado.

$$n_{opt} = \frac{I \times (1 - R) \times B_{sustained}}{P_{sustained} \times I \times R \times [(1 - H_{L1}) \times (1 - H_{L2})] \times [F \times L_{word} + (1 - F) \times L_{cache}]}$$
(5.1)

A aplicação dos valores da tabela 5.4 na equação 5.1 resulta em n_{opt} igual a 1078 *threads*, o que é exageradamente superior ao grau de paralelismo **dois** (com 2 *threads*, portanto) que resultou no maior *speedup* de TRIAD em SGI. Mesmo trocando arbitrariamente o valor de $P_{sustained}$ pelo valor obtido pelo *benchmark* Linpack na mesma máquina SGI, igual a 1548,7 MFLOPS, o grau de paralelismo continuaria excessivo, com n_{opt} igual a 195 *threads*. Com a medida de $P_{sustained}$ para TRIAD, as taxas de acerto ao cache precisariam ser ambas inferiores a 7,5% para que n_{opt} fosse inferior a 3 *threads*. Esta modelagem, portanto, não se aplica aos programas e sistemas computacionais testados.

Apesar de a quantidade sugerida de *threads* ter sido centenas de vezes superior à

quantidade de *threads* de melhor desempenho obtida experimentalmente para o *benchmark* STREAM, a simplicidade do modelo avaliado em representar diversos componentes envolvidos na execução paralela pode ser aproveitada.

$$n_{opt} = \frac{I \times (1 - R) \times B_{sustained}}{P_{sustained} \times I \times R \times [(1 - H_{L1}) \times (1 - H_{L2})] \times [F \times L_{word} + (1 - F) \times L_{cache}]}$$
 (5.2)

Uma deficiência identificada para o modelo proposto consiste em não considerar o caráter temporal da localidade de referência dos acessos a dados. Conforme exposto na seção 5.6, a localidade dos acessos é um componente influente nas limitações à escalabilidade de uma aplicação. Não obstante, a informação de localidade de referência na equação 5.2, representada pelo *fator de reúso* F , abrange apenas o aspecto espacial e ignora a localidade temporal.

A equação 5.2 pode ser ajustada para fornecer uma estimativa de quantidade de *threads* mais próxima das observações experimentais. Uma possível alternativa para determinação do número de *threads* a partir dos mesmos princípios empregados na equação 5.2, com a substituição do quociente $\frac{R}{1-R}$ pela variável γ_L (eq. 3.1) e da proporção $\frac{B_{sustained}}{P_{sustained}}$ pelo *balanceamento de máquina*, β_M (eq. 1.6), é sugerida pela equação 5.3, onde L é a informação de localidade de referência. Calculado pela equação 5.4 com base no fator de reúso espacial, F , e no fator de reúso temporal, T , o valor de L indica a quantidade de palavras de 8 *bytes* acessadas pela aplicação, em média, para cada linha de cache utilizada.

$$n_{opt} = \frac{\beta_M}{\gamma_L \times (1 - H_{L1}) \times (1 - H_{L2}) \times L}$$
 (5.3)

$$L = \{F \times L_{word} + (1 - F) \times L_{cache}\} + \{T \times L_{word} + (1 - T) \times L_{cache}\}$$
 (5.4)

A equação 5.3 foi aplicada em TRIAD. Este componente do *benchmark* STREAM não faz reúso dos dados de *arrays* acessados pelo *loop*, portanto o valor de T é zero. O fator de reúso espacial, F , e as taxas de acerto em cada nível do cache, H_{L1} e H_{L2} , são indicados na tabela 5.4. A proporção β_M para a máquina SGI é dada na tabela 5.1, enquanto a granularidade de *loop* γ_L é indicada na tabela 5.2. Como β_m e γ_L envolvem *palavras* ao invés de *bytes*, os valores para L_{word} e L_{cache} , obtidos da tabela 5.4, precisam ser divididos por 8, para indicar quantidade de *palavras* de 8 *bytes* e harmonizar com β_m e γ_L . A fórmula resulta no valor de 1,812 para n_{opt} , correspondente a **uma thread**. Tal

como usada no presente trabalho, γ_L não considera o processamento além de operações sobre dados em ponto flutuante. Incluindo-se o incremento da variável de iteração do *loop*, γ_L assume o valor **1**, elevando o valor de n_{opt} para 2,72, correspondente ao uso de **2 threads**. As execuções experimentais de TRIAD em SGI atingiram maior *speedup* com 4 *threads*, porém com ganho irrisório de desempenho em relação à execução com 2 *threads*, apontando para uma melhoria na sugestão do grau de paralelismo pela equação 5.3 em relação à versão original do modelo.

Cabem ressalvas a esses resultados, contudo. A proporção β_M considera a velocidade de um *core* multiplicada pela quantidade de *cores* disponíveis. Usando β'_M como a proporção entre a largura de banda de memória e a velocidade de um único *core*, o valor de n_{opt} cresce para 14,5. Além disso, β_M é calculado a partir da execução sequencial do *benchmark* Linpack. Se for considerado a velocidade de processamento em MFLOPS de STREAM em β_M , o valor de n_{opt} excede a quantidade de 540 *threads*.

Desenvolvimentos adicionais são necessários para se refinar a equação 5.3, bem como as avaliações mais abrangentes para validar ou inviabilizar sua utilização na determinação de grau de paralelismo.

Capítulo 6

Conclusões

A presente dissertação apresentou uma análise dos efeitos da granularidade dos acessos à memória na escalabilidade de aplicações paralelas em processadores *multicore*, com vistas a seu uso para a sugestão automática da quantidade de *threads*. As aplicações estudadas empregaram o modelo de paralelismo de dados através de *loops* paralelos com a interface de programação OpenMP.

Foi observado que o *speedup* máximo ocorre tipicamente na quantidade de *threads* igual à quantidade de *cores* disponíveis. Execuções com mais *threads* do que *cores* não trouxeram benefício para o tempo de execução, que tipicamente sofreu aumento conforme as trocas de contexto se tornaram mais numerosas. Quando poucos *cores* foram considerados, todos os programas escalaram, cada um a seu ritmo, indicando que em tal situação o uso de todos os recursos disponíveis pode ser uma estratégia viável. Em processadores com *cores* mais numerosos, surgem limitações à capacidade de algumas aplicações de apresentar ganhos com o paralelismo, causando desde reduções nos ganhos com o acréscimo de *threads* até pioras no desempenho com graus de paralelismo maiores, provocando quedas expressivas na eficiência das execuções.

Percebe-se, então, que o número de *threads* não é o único determinante da escalabilidade. A taxa de uso do barramento de acesso à memória, para os programas e máquinas testados, se mostrou como um indicador acurado para a obtenção de ganhos com execuções paralelas, apontando restrições para o aumento de desempenho com taxas de uso superiores a 70%. As taxas de faltas nos caches, métricas tradicionais de indicação de desempenho, falharam em sugerir o comportamento de *speedup* devido ao baixo grau de compartilhamento dos níveis de cache entre os *cores*.

O efeito de usos mais intensos de barramento de memória foi tão notável que influ-

enciou o tempo de CPU das aplicações, apesar do volume total de processamento ser o mesmo para uma mesma aplicação com tamanho de problema fixo. A maior competição pelo acesso à memória causou atrasos na execução das operações, possivelmente gerando ciclos ociosos em que as instruções aguardaram a chegada dos dados da memória principal para o cache. A aplicação MG, por exemplo, faz uso tão intenso do barramento de acesso à memória que a introdução de processamento em seus *loops* diminuiu a frequência dos acessos à memória (mas não sua quantidade absoluta), minimamente diminuindo a latência nos acessos e habilitando maiores *speedups*. A organização de memória NUMA, devido a uma associação mais forte, em comparação com UMA, entre módulos de memória e grupos de *cores*, suaviza esta limitação para a escalabilidade, essencialmente restringindo a competição pela memória aos *cores* locais, embora isto dependa de uma boa distribuição dos dados que os posicione perto dos *cores* que os acessam.

Enquanto a taxa de uso do barramento de memória é uma medida da frequência dos acessos em relação ao tempo de execução, a granularidade efetiva, γ_E , indica a proporção de acessos à memória em relação ao processamento. Os resultados das análises deste trabalho apontaram uma associação clara entre a granularidade efetiva e a capacidade de se obter ganhos com o paralelismo.

A métrica γ_L , ou granularidade de *loop*, que indica a proporção entre acessos a *arrays* e operações aritméticas de ponto flutuante no código-fonte da aplicação, foi apresentada como uma estimativa para a granularidade efetiva. De modo geral, valores elevados de granularidade de *loop* corresponderam a maiores contenções pelo acesso à memória e a maiores valores de granularidade efetiva, enquanto valores menores de γ_L foram compatíveis com menores contenções e melhor escalabilidade, permitindo, assim, estabelecer uma correspondência entre granularidade de *loop* e granularidade efetiva. Diminuições artificiais de γ_L , através do acréscimo de processamento, resultaram em maiores valores de *speedup*, porém às custas de maiores tempos de execução, já esperados com o maior volume de operações.

Uma vez que granularidade efetiva pode ser estimada do código-fonte, torna-se possível, a princípio, incorporar a granularidade de *loop* a um mecanismo automático de sugestão de quantidade de *threads* que não necessite de execuções prévias da aplicação. Por outro lado, a obtenção de γ_L diretamente a partir do código-fonte da aplicação sofre a desvantagem de potencialmente não representar corretamente as operações que serão realizadas pelo processador, pois o código-fonte está sujeito a mudanças durante a etapa de compilação devido a otimizações. Embora a granularidade de *loop* γ_L , tal como definida neste

trabalho, tenha razoável associação com benefícios da paralelização quando se considera apenas as aplicações pertencentes a um mesmo conjunto, como NPB e STREAM, a associação de comportamento entre aplicações de grupos diferentes com base em γ_L é falha. As diferenças no código-fonte das aplicações apontam três possíveis motivos para esta discrepância: processamento que não foi considerado em γ_L , o modo de acesso aos dados e tamanho dos *arrays* manipulados.

Uma ferramenta que considere granularidade de *loop* na análise automática de escalabilidade deve, portanto, determinar γ_L para cada *loop* paralelo, permitindo possivelmente diferentes quantidades de *threads* para diferentes trechos do programa. O compilador se torna, então, o candidato mais apropriado para determinar o valor de γ_L , já que tem acesso direto à forma final do programa.

Foi mostrado que algumas aplicações com as mesmas operações, mesmo volume de processamento e mesma quantidade de acesso aos dados, e conseqüentemente mesma granularidade de *loop*, manifestaram escalabilidades distintas. Valores desiguais de granularidade efetiva, consequência de diferenças na localidade de acesso aos dados, fizeram com que as aplicações obtivessem benefícios distintos com a paralelização. Os efeitos da localidade foram sentidos também para uma mesma aplicação entre diferentes sistemas computacionais, como sugerido pela diferença do comportamento de *speedup* entre as duas versões de FT observadas. Estes resultados apontam para a necessidade de consideração da localidade, além do volume de processamento e da quantidade de acessos aos dados, em um mecanismo automático para sugestão de quantidade de *threads*.

6.1 Trabalhos Futuros

Uma solução para o problema da determinação automática da quantidade de *threads* sem necessidade de execuções prévias da aplicação não é trivial. Variações na localidade causaram alterações na frequência dos acessos à memória, afetando a escalabilidade das aplicações. Desta forma, não só a granularidade dos acessos a dados precisa ser considerada, mas também sua localidade. Assim sendo, vislumbram-se as seguintes estratégias de continuidade do presente trabalho:

- Implementar a determinação de γ_L dentro de um compilador, considerando todo o processamento dentro do loop;
- Avaliar técnicas de estimativa de localidade a partir do código-fonte;

- Avaliar técnicas de determinação de localidade a partir de execuções da aplicação e sua aplicabilidade na determinação da quantidade de *threads*;
- Buscar uma relação matemática que associe *machine balance* com granularidade de *loop* e localidade para indicar o número de *threads*.

Na busca por estes objetivos, considera-se que um compilador poderia, no momento de sua instalação, verificar a largura de banda do sistema de memória, a velocidade de processamento de cada *core* e o tempo de execução de cada instrução *assembly* do processador usado que atuasse somente em registradores e assim registrar o tempo relativo entre as operações, possibilitando determinar γ_L com maior precisão.

Apêndice A

Desvio padrão dos experimentos

Este apêndice apresenta os desvios padrão de medidas obtidas para os experimentos discutidos nos capítulos 4 e 5. As medidas coletadas foram tempo decorrido, taxa de uso de barramento, taxas de faltas (*misses*) nos caches de nível 1 e de nível 2 e granularidade efetiva (γ_E). Em cada tabela, o desvio padrão para cada aplicação em cada grau de paralelismo é indicado como uma porcentagem da média da medida correspondente. Um “-” indica que o programa, na respectiva quantidade de *threads*, teve sua execução interrompida antes do término por exceder a duração máxima de 15 minutos estabelecida para cada teste.

Tabela A.1: Desvio padrão do tempo decorrido de NPB classe A na máquina SGI.

<i>threads</i>	BT.A	CG.A	EP.A	FT.A	LU.A	MG.A	SP.A	UA.A
1	0,1309	0,0000	0,1051	0,2320	0,4131	0,1425	0,0263	0,3376
2	3,1220	8,5600	3,6995	5,2419	1,2961	6,6833	8,9238	0,2500
4	0,2720	1,2926	3,5101	0,1729	0,2560	0,0000	0,2438	0,2405
6	3,7828	10,6423	0,8805	3,3194	2,2192	8,7694	7,0393	2,9959
8	0,8184	2,1323	0,2233	0,3005	0,4000	0,2886	1,2769	1,5187
10	1,1692	5,0673	1,0760	0,7229	8,1986	0,3602	1,6289	0,4653
12	0,4242	2,5511	2,0147	0,3304	0,0000	0,5379	1,1320	0,5942
14	0,5196	7,1604	2,0785	0,2552	0,0000	0,2968	1,1323	0,7174
16	0,3646	2,3999	0,8404	0,5441	0,0000	0,2839	1,3335	2,8635
24	0,2670	2,1048	0,6120	0,3191	0,0032	0,2622	0,3340	0,5801
32	0,2927	3,4971	0,6714	0,2681	0,0285	0,3921	0,2593	0,3756
64	0,1526	3,0569	0,9211	0,2461	0,2575	0,5150	0,2339	0,3180
128	0,1993	4,4900	0,1898	0,2539	0,0943	0,4142	0,1532	0,2372
256	0,1817	4,5993	0,2798	0,2490	0,1707	0,2107	0,0961	0,4655
512	0,3223	7,4360	0,2065	0,2689	0,2553	0,3317	0,2725	0,6250
1024	0,6158	4,8014	0,1782	0,2401	0,0889	2,4785	0,5482	0,8190

Tabela A.2: Desvio padrão do tempo decorrido de NPB classe B na máquina SGI.

<i>threads</i>	BT.B	CG.B	EP.B	FT.B	LU.B	MG.B	SP.B	UA.B
1	0,6159	0,0872	1,0722	0,1235	19,6197	0,0690	0,0437	0,2309
2	2,2526	0,1441	2,2771	5,3129	17,0218	8,8696	9,6718	5,9566
4	0,2645	0,0894	0,7228	0,3075	7,0648	0,0846	0,0818	0,2035
6	3,2391	3,6017	0,5422	2,0641	6,0262	9,2733	7,2160	3,6318
8	0,1093	0,0923	0,8680	0,3173	188,2318	0,2220	0,3866	0,2470
10	0,1187	0,3898	0,2031	0,4516	-	0,2615	0,0694	0,2909
12	0,3250	0,0853	1,5639	0,4294	-	0,2498	0,0514	0,4602
14	0,0406	0,2755	1,1407	0,3084	-	0,1556	0,0720	0,1013
16	0,1494	0,0647	0,2530	0,5839	-	0,1179	0,0363	0,4207
24	0,0261	0,0618	0,3912	0,3353	-	0,1562	0,0643	0,2249
32	0,2053	0,0736	0,0956	0,2176	-	0,1319	0,0365	0,1758
64	0,0304	0,1510	0,1732	0,1170	-	0,2209	0,0345	0,1097
128	0,0456	0,4854	0,4062	0,0549	-	0,4815	0,0262	0,0958
256	0,0186	0,6791	0,2663	0,1509	-	0,6307	0,2701	0,6641
512	0,1122	1,6405	0,2159	0,0552	-	0,6230	0,0831	1,1501
1024	0,1032	1,9213	0,0742	0,0965	-	0,9405	0,1224	0,4320

Tabela A.3: Desvio padrão do tempo decorrido de NPB classe C na máquina SGI.

<i>threads</i>	BT.C	CG.C	EP.C	FT.C	LU.C	MG.C	SP.C	UA.C
1	0,0004	8,5014	0,3111	0,1402	0,0010	0,0456	0,2800	0,0901
2	2,2677	5,5918	1,0732	4,6106	0,0006	0,1042	7,9021	0,0800
4	0,0634	0,1087	0,2797	0,7887	0,8490	0,0556	0,1392	0,0976
6	0,1103	5,4714	0,8896	1,3259	8,8774	2,8757	0,7716	0,0885
8	0,0333	0,1070	0,8541	0,4351	1,0595	0,0896	0,1656	0,1413
10	4,5107	1,0743	0,7941	2,3463	0,0005	0,8562	0,3192	0,2800
12	2,0560	0,1210	2,4598	0,4610	0,0006	0,3591	0,0681	0,3496
14	0,1064	0,1406	1,3015	0,5166	0,0006	0,1702	0,0379	0,0924
16	0,1968	0,0670	0,1689	0,9077	0,0006	0,1709	0,0548	0,0598
24	0,1446	0,1261	0,1911	1,2357	0,0006	0,1779	0,0845	0,0412
32	0,2153	0,0845	0,0613	0,7807	0,0006	0,0670	0,0387	0,0707
64	0,0318	0,1251	0,1304	0,1602	0,3263	0,0794	0,0108	0,0865
128	0,0502	0,1131	0,1620	0,1224	0,0000	0,0382	0,0310	0,0312
256	0,0235	0,3036	0,0644	0,1006	0,6639	0,1066	0,0826	0,2097
512	0,0352	0,4551	0,0791	0,0881	0,9279	0,0562	0,0333	0,0756
1024	0,0419	1,0669	0,1100	0,0288	0,9735	0,1259	0,0505	0,1598

Tabela A.4: Desvio padrão da taxa de uso de barramento de NPB na máquina SGI.

<i>threads</i>	BT.A	CG.A	EP.A	FT.A	LU.A	MG.A	SP.A	UA.A
1	0,3077	0,5153	11,2739	0,1333	0,6378	0,1996	4,7862	0,2350
2	12,3186	0,6679	19,9947	20,9096	15,5263	22,5706	25,4124	17,0452
4	0,4229	13,0964	37,7330	0,7243	0,9065	0,2532	19,1143	3,3207
6	5,2152	9,1468	28,6651	3,1188	5,3173	5,3679	16,5984	4,1457
8	1,2160	1,8927	10,0094	0,3544	1,5565	0,5148	10,6765	5,2016
<i>threads</i>	BT.B	CG.B	EP.B	FT.B	LU.B	MG.B	SP.B	UA.B
1	28,0543	0,4000	22,4216	0,1039	23,0809	1,1836	9,9755	0,5986
2	26,8145	8,8680	13,7300	12,5444	18,6997	14,7742	25,5927	14,1684
4	14,6774	0,1517	9,4330	0,2716	13,1916	1,3384	10,6635	0,2412
6	8,0585	5,1606	22,2983	1,1814	7,4457	6,2821	6,4594	3,7838
8	2,0232	0,1378	37,3853	0,2237	54,5353	0,4012	4,2888	0,4936
<i>threads</i>	BT.C	CG.C	EP.C	FT.C	LU.C	MG.C	SP.C	UA.C
1	0,0903	5,4816	60,7538	2,2988	0,1789	0,1761	10,5587	0,4854
2	12,7283	6,7023	24,4921	16,3854	13,3776	12,4628	8,5233	0,8459
4	0,1986	0,1030	36,1646	0,5710	0,9638	0,0933	6,6532	0,1432
6	0,1297	4,2865	49,1955	0,2611	1,8483	0,5762	4,3604	2,4590
8	0,2859	0,2611	49,1955	0,2611	1,8483	0,5762	4,3604	2,4590

Tabela A.5: Desvio padrão da taxa de faltas no cache de nível 1 de NPB na máquina SGI.

<i>threads</i>	BT.A	CG.A	EP.A	FT.A	LU.A	MG.A	SP.A	UA.A
1	0,1985	0,0772	0,4910	0,0292	0,1179	0,1872	23,0630	0,0337
2	0,1077	0,0203	0,3030	0,0514	0,2231	0,7163	22,1769	0,0902
4	0,1135	0,4397	0,1753	0,1914	0,1607	0,2641	21,7511	0,2829
6	0,4538	3,6954	3,1586	0,5685	1,3600	1,6932	20,2523	0,8791
8	0,1603	1,6916	0,0977	0,2980	0,6840	0,4704	20,7593	0,1855
<i>threads</i>	BT.B	CG.B	EP.B	FT.B	LU.B	MG.B	SP.B	UA.B
1	1,3552	0,0347	1,2454	0,0104	3,4757	0,7096	30,5853	0,0349
2	0,0446	0,0787	0,2560	0,1671	2,0145	0,1982	27,9103	1,8702
4	1,3307	0,1020	0,4346	0,0617	5,0448	0,8450	28,4159	0,0853
6	0,6762	3,5722	0,1352	0,1485	2,1001	2,2442	28,4331	2,1670
8	1,4219	0,5109	0,1620	0,1709	55,4326	0,3150	24,9342	5,2896
<i>threads</i>	BT.C	CG.C	EP.C	FT.C	LU.C	MG.C	SP.C	UA.C
1	0,1597	17,3053	2,3503	0,4837	0,1300	0,0519	7,8528	0,5078
2	3,5494	0,1241	0,1769	0,4671	8,2545	0,1377	9,1179	8,3207
4	0,1287	0,0797	0,1618	0,1129	3,0794	0,0846	7,2115	0,2699
6	0,2066	3,1517	0,1852	0,0569	20,1901	0,1936	3,6424	5,0805
8	0,0735	0,1930	0,1890	0,1200	5,5499	0,1174	5,8929	0,1632

Tabela A.6: Desvio padrão da taxa de faltas no cache de nível 2 de NPB na máquina SGI.

<i>threads</i>	BT.A	CG.A	EP.A	FT.A	LU.A	MG.A	SP.A	UA.A
1	0,0610	0,1519	22,0913	0,1331	0,9293	0,1533	6,3453	0,5115
2	0,1093	0,2272	15,8584	0,3683	0,3052	0,3125	1,4214	0,4740
4	0,2426	12,7021	19,6342	0,2415	0,4767	0,4848	19,0238	2,6428
6	1,3423	3,8949	19,1500	0,3940	1,6546	2,0257	21,9577	1,4694
8	1,4205	4,8962	7,0368	0,4067	2,8803	0,1345	27,3233	3,7455
<i>threads</i>	BT.B	CG.B	EP.B	FT.B	LU.B	MG.B	SP.B	UA.B
1	0,5568	0,6950	23,7355	0,0574	35,8319	1,1001	9,8575	0,0160
2	0,6940	0,0773	26,7371	0,0819	14,2262	0,2007	7,4815	0,2102
4	2,1560	0,1400	12,3354	0,1115	4,1128	0,7711	8,1904	0,0907
6	1,8531	3,5183	12,3227	0,2341	2,7091	2,4458	6,2320	2,0609
8	0,8891	0,4505	14,6387	0,2419	55,3422	0,3140	1,6796	0,2725
<i>threads</i>	BT.C	CG.C	EP.C	FT.C	LU.C	MG.C	SP.C	UA.C
1	0,1878	12,0279	43,9453	0,4477	0,1807	0,1642	2,9255	0,6898
2	4,1277	0,1224	18,8102	0,2097	9,5114	0,1459	14,1848	4,6303
4	0,1198	0,1107	34,9969	0,1109	2,1416	0,1055	2,7358	0,1203
6	0,2570	3,1434	40,4521	0,1838	16,3045	0,2605	0,5393	5,3855
8	0,2406	0,2006	21,1818	1,1378	5,1503	0,1999	16,1949	0,1416

Tabela A.7: Desvio padrão do tempo decorrido de NPB na máquina HP.

<i>threads</i>	BT.B	CG.C	EP.C	FT.C	LU.C	MG.C	SP.B	UA.B
1	0,0598	0,0433	0,1460	0,0992	0,8319	0,2757	0,2737	0,2657
2	0,7534	2,0642	0,2429	0,2696	2,8499	0,1105	4,5096	0,1195
4	0,6634	0,3099	0,7164	0,9704	0,4693	0,3737	3,6231	0,8074
6	0,3730	0,3589	0,1603	0,3181	0,7735	0,2593	3,9900	0,9686
8	0,9188	0,3513	0,9308	0,1179	6,3696	0,2463	4,2027	0,6858
10	0,7241	0,2270	0,2069	0,1787	4,3693	1,3436	3,1281	0,3027
12	1,8855	0,1023	0,0245	0,1500	1,3183	0,8492	0,9577	0,6846
16	0,4689	0,1075	1,2068	0,1651	2,2273	1,5310	1,7999	0,7130
20	0,3128	0,0832	0,8539	0,2661	0,5596	0,4865	3,4796	1,0121
24	0,8063	0,1682	0,7021	0,2514	0,3392	0,8178	2,4372	0,5498
48	0,8582	0,0763	0,4799	0,3361	0,2219	1,2569	1,7249	0,9135
72	1,4975	0,0388	0,7998	0,3733	0,0008	0,8143	0,9232	1,6595

Tabela A.8: Desvio padrão do tempo decorrido de NPB na máquina AMD.

<i>threads</i>	BT.B	CG.C	EP.C	FT.C	LU.C	MG.C	SP.B	UA.B
1	1,3540	4,2192	0,5714	0,4796	0,1718	1,0134	2,9719	1,6836
2	0,8136	2,1751	0,2471	0,6202	0,8158	0,9935	0,9690	1,1012
3	0,8037	2,0355	0,4875	0,8067	0,2261	0,7314	1,3124	0,8610
4	0,1418	0,8076	0,3292	0,3633	0,2773	4,5975	0,5429	7,6214
8	2,3966	1,6905	0,7847	5,9385	0,0002	2,3757	1,2415	4,0031
12	1,8007	3,9082	0,8603	1,2806	0,0002	0,8928	1,2888	1,0454
16	2,0332	2,8851	0,1878	1,3498	0,0005	0,9262	1,3002	2,4716

Tabela A.9: Desvio padrão do tempo decorrido de NPB na máquina MTL sem afinidade de CPU.

<i>threads</i>	BT.B	CG.C	EP.C	FT.C	LU.C	MG.C	SP.B	UA.B
1	0,5393	1,7986	1,4426	1,0215	1,7629	0,8360	1,7193	2,3313
2	0,7150	1,8034	0,3567	1,6636	0,9079	1,9740	3,0306	0,8349
4	0,6861	1,9679	0,5664	1,7990	0,8924	2,3392	2,3363	0,1720
6	2,1971	2,6667	0,1748	4,3153	2,4316	6,2383	7,3088	1,1869
8	1,9829	3,2562	0,3180	5,2848	2,7144	6,7856	8,4951	0,5356
10	4,2308	4,4334	3,5779	4,1852	4,2579	16,9510	11,4866	0,6492
12	3,4101	4,9923	0,5172	6,0919	4,7215	16,4336	10,4296	0,9643
14	4,3698	6,7517	0,0223	6,5257	5,9431	19,9287	12,6456	1,3988
16	5,0298	7,9635	0,1086	8,4849	4,7831	20,6311	12,7256	1,6712
18	5,8715	10,2321	0,2833	8,3840	7,3516	21,9875	14,8662	2,1975
20	6,5293	12,2322	0,3596	9,2394	7,5268	21,6668	13,9205	1,9162
22	6,2661	17,0277	0,3205	10,2925	8,5895	23,4362	16,5010	2,1811
24	7,6176	16,7478	0,2644	11,4916	7,9303	22,7920	14,5077	2,2651
26	7,1277	18,3755	0,0000	12,4758	8,6911	23,4775	16,4737	2,5075
28	7,1648	20,9627	0,0267	12,4164	8,9569	23,9844	17,4254	2,5357
30	9,6584	21,8261	0,3197	14,0837	10,7924	22,7599	16,0939	3,4677
32	7,9937	24,2572	0,1154	16,2032	13,2486	23,8246	15,0426	2,8904
34	8,8290	28,1647	0,2264	15,8729	11,3190	24,2486	19,2408	6,2800
36	8,2167	26,2647	3,0693	19,0985	13,2003	24,0255	19,2538	3,0489
38	192,8234	28,5664	17,7710	18,7623	12,7279	158,7356	29,9393	132,3439
40	57,3558	82,5081	17,8715	19,7810	118,6776	159,2146	46,8213	92,0520
44	8,0928	13,1039	0,5324	10,9287	7,3056	19,8418	18,5919	6,3658
48	7,9436	14,3475	1,6594	11,7160	-	19,7205	18,7323	9,4808
52	8,5487	13,7390	1,3833	11,9018	-	21,0318	18,0539	9,8371
56	5,9856	15,1271	0,3906	12,8168	-	20,5583	18,5930	6,4282
60	9,4679	15,9690	4,2422	13,5945	-	20,7992	17,5376	8,8588
64	6,5443	15,8307	1,5908	15,2157	-	21,7038	17,3737	4,3920
68	9,9364	16,7138	1,8057	15,5414	-	21,5578	17,8521	7,3785
72	6,4060	16,8114	2,3893	15,3996	-	21,4439	18,1211	3,4330
76	8,3963	16,5937	2,5593	17,2049	-	22,0933	17,7349	6,2542
80	6,8883	13,7317	1,5293	16,3145	-	21,3956	17,2477	6,5798
100	10,8031	14,4725	0,8742	14,0398	-	21,1506	24,2420	4,9169
120	12,1819	12,8737	1,8126	16,5959	-	21,1551	22,0848	3,9928
140	11,0858	13,3280	0,8220	15,4851	-	20,9204	23,0224	2,4295

Tabela A.10: Desvio padrão do tempo decorrido de NPB na máquina MTL com afinidade de CPU ativa.

<i>threads</i>	BT.B	CG.C	EP.C	FT.C	LU.C	MG.C	SP.B	UA.B
1	0,3669	0,0543	0,4342	0,0411	0,5417	0,0670	0,0750	0,0633
2	0,2101	2,2810	1,5397	1,9901	2,7134	2,4032	3,7010	3,2780
4	0,3196	2,9436	0,1332	3,8868	5,2118	9,3638	7,2998	6,1896
6	0,1359	4,3997	0,4337	5,1775	6,1375	11,7829	9,0198	8,2074
8	0,4009	6,3741	0,3814	7,3828	9,7870	12,8461	10,4738	9,8049
10	0,1510	7,0998	0,4636	8,3531	10,1150	13,3624	11,6215	10,9316
12	0,1178	7,2615	0,4976	8,6611	9,7444	12,8386	11,9109	10,5914
14	0,1955	6,9883	0,2766	9,2729	6,6174	12,5281	12,4300	10,1175
16	0,0698	7,1095	0,1201	9,1117	7,1576	12,2297	12,3429	9,5743
18	0,1732	7,3091	0,1840	8,7248	6,8958	11,9202	12,5288	8,8834
20	0,1555	10,5296	1,5774	7,6708	7,5331	11,3620	12,9734	8,2887
22	0,4146	10,1260	1,7627	7,3574	7,5339	11,5853	12,5682	7,8487
24	0,2618	9,7195	0,3829	7,0955	7,3809	11,3508	12,5173	7,3944
26	0,4431	9,5903	1,7695	6,8505	7,4622	11,0117	12,2562	6,7760
28	0,3710	8,7971	0,4709	6,6373	8,1736	10,8090	12,2341	6,1794
30	0,0992	8,5920	0,5376	6,5991	8,0605	10,7464	12,2600	5,2669
32	0,1833	8,3128	0,2652	6,3453	9,2231	10,5359	12,2526	4,7370
34	0,5014	8,3041	0,3096	6,2731	8,2713	10,7540	11,9873	3,7957
36	0,5538	8,3684	1,8656	6,0819	8,2465	10,6315	12,1909	3,6699
38	0,4410	13,6015	0,4722	6,6018	8,0744	10,2654	12,1282	2,8889
40	0,3913	22,7301	0,7551	16,3259	9,8590	18,2853	12,0854	3,1243
44	0,2503	5,3700	0,1948	7,4338	9,2702	11,4535	11,9258	8,8696
48	0,6815	6,3433	0,2638	6,5785	-	10,2251	11,7180	5,1476
52	0,2002	6,8291	0,0962	8,8777	-	10,3356	11,9548	5,5294
56	0,3688	7,1474	0,1685	8,5005	-	10,4512	12,0161	4,7805
60	0,1572	7,0285	0,1702	7,7030	-	10,3841	11,9370	4,0059
64	0,1534	7,0717	0,1925	7,0209	-	10,5513	11,9519	2,7975
68	0,1987	6,9898	0,1867	7,4699	-	10,4799	12,0077	2,4553
72	0,3537	6,9048	0,0755	6,9974	-	10,6074	11,7281	1,9985
76	0,2283	11,0142	0,1536	7,5592	-	10,6171	11,5706	1,6271
80	0,2244	18,1084	0,0505	7,9246	-	10,4634	11,2098	2,3939
100	0,1839	6,9517	0,0697	9,4616	-	13,2098	12,9296	2,6844
120	0,1459	16,1666	0,1812	7,3095	-	10,4579	12,6604	3,0254
140	0,1218	6,8685	0,0955	9,6820	-	13,1445	12,7863	3,2706

Tabela A.11: Desvio padrão de γ_E de NPB na máquina SGI.

<i>threads</i>	BT.A	CG.A	EP.A	FT.A	LU.A	MG.A	SP.A	UA.A
1	0,2769	0,1739	11,2961	0,0857	0,9044	0,0850	9,4034	0,4059
2	12,3844	0,2054	19,5130	18,9497	15,9333	18,7507	28,0420	16,6551
4	0,3513	12,8459	37,8432	0,0578	0,9157	0,0368	30,6604	3,0867
6	3,4871	1,5246	28,0947	1,6447	3,6841	3,3692	32,0063	4,3094
8	1,3788	3,4695	10,1192	0,0796	1,5086	0,2629	36,3937	4,4086
<i>threads</i>	BT.B	CG.B	EP.B	FT.B	LU.B	MG.B	SP.B	UA.B
1	32,6345	0,8120	22,1286	0,1109	26,8326	0,7323	3,5984	0,1217
2	30,7790	9,2756	13,7330	9,8170	18,2702	17,4567	8,0030	16,1486
4	24,3420	0,0359	9,2461	0,0753	21,8181	0,1603	2,2756	0,2249
6	15,5189	1,8461	22,2577	0,7764	12,1813	1,1601	4,5246	0,2026
8	6,1324	0,1217	37,3318	0,1347	0,4503	0,2048	4,8329	0,3620
<i>threads</i>	BT.C	CG.C	EP.C	FT.C	LU.C	MG.C	SP.C	UA.C
1	0,0742	12,9600	61,0720	2,5055	0,1295	0,1626	14,4165	0,5782
2	13,0418	4,6377	24,4832	15,7400	11,8498	7,4723	11,7926	6,6132
4	0,0442	0,0615	36,1630	0,0819	1,4992	0,0305	11,9039	0,0708
6	0,6213	1,9250	49,2256	0,2263	10,7170	0,1997	5,6572	2,8881
8	0,3128	0,0717	25,1504	0,9555	1,3219	0,0641	9,6644	0,1513

Tabela A.12: Desvio padrão do tempo decorrido para as versões com granularidade alterada de EP, FT e MG na máquina SGI.

<i>threads</i>	EP.C	FT.C	MG.C	<i>threads</i>	EP.C	FT.C	MG.C
1	0,0089	4,8163	0,1116	16	0,5449	0,0580	0,0657
2	0,0158	4,2418	7,2700	24	0,5099	0,0934	0,0413
4	0,0769	0,0889	0,0393	32	0,2946	0,0695	0,1521
6	0,0339	5,0848	2,7037	64	0,1797	0,0795	0,1275
8	0,0452	0,0822	0,0639	128	0,0848	0,0834	0,0225
10	0,5141	0,8486	0,0600	256	0,1329	0,2411	0,0337
12	1,7990	0,0974	0,2097	512	0,3185	0,3911	0,0167
14	0,1484	0,1244	0,1311	1024	0,2968	0,9106	0,0263

Tabela A.13: Desvio padrão do tempo decorrido para as versões com granularidade alterada de FT e MG na máquina HP.

<i>threads</i>	FT.C	MG.C	<i>threads</i>	FT.C	MG.C
1	0,0669	0,0364	12	0,3316	1,0778
2	0,1738	0,3411	16	0,3786	1,2034
4	0,9774	0,5989	20	0,0539	1,5691
6	0,6031	2,6448	24	0,2205	0,9445
8	0,2500	0,8356	48	0,4699	1,2831
10	0,3541	0,8676	72	0,1668	1,1574

Tabela A.14: Desvio padrão do tempo decorrido para as versões com granularidade alterada de FT e MG na máquina AMD.

<i>threads</i>	FT.C	MG.C
1	0,0603	1,5437
2	0,7580	0,5090
3	0,7616	0,2414
4	0,7068	0,4888
8	1,1570	0,6869
12	1,0953	0,7805
16	1,4447	0,2649

Tabela A.15: Desvio padrão do tempo decorrido para as versões com granularidade alterada de EP, FT e MG na máquina MTL.

<i>threads</i>	EP.C	FT.C	MG.C	<i>threads</i>	EP.C	FT.C	MG.C
1	0,0106	0,2128	0,5704	34	0,4448	0,9196	6,6769
2	0,0100	0,2440	1,8966	36	0,5889	1,4034	7,0427
4	0,0465	0,4169	4,6239	38	0,5703	1,0419	6,7416
6	0,0002	0,5308	5,8957	40	0,0193	1,2131	6,9127
8	0,0007	0,7142	6,5912	44	0,2406	0,7576	4,5140
10	0,0081	0,8623	7,1295	48	0,1578	0,8021	4,8998
12	0,0796	0,8865	7,2149	52	0,1200	1,0517	6,0007
14	0,0696	0,9305	7,0344	56	0,0507	0,9063	6,1760
16	0,1060	0,9851	7,0269	60	0,0060	1,1182	6,0041
18	0,1192	0,9920	6,9961	64	0,2551	0,6325	5,5097
20	0,5810	1,0343	7,0852	68	0,4126	0,8624	4,1664
22	1,5357	1,0767	6,9339	72	0,1748	0,7258	3,8597
24	0,9365	0,9883	7,4541	76	0,8519	0,8616	4,0451
26	0,1715	0,9893	6,7140	80	0,1334	0,8133	3,8688
28	0,4155	1,0405	6,9573	100	0,2603	0,9030	6,4356
30	0,0109	0,9399	6,4973	120	0,2454	1,0972	6,5830
32	0,1580	0,9557	6,3512	140	0,2512	0,8954	5,7923

Tabela A.16: Desvio padrão da taxa de uso de barramento para as versões com granularidade alterada de EP, FT e MG na máquina SGI.

<i>threads</i>	EP.C	FT.C	MG.C
1	28,9827	5,5465	0,3185
2	15,2798	5,2354	1,7298
4	41,5940	2,8155	0,1245
6	15,5408	0,8836	1,9526
8	60,3839	1,5961	0,7801

Tabela A.17: Desvio padrão de γ_E para as versões com granularidade alterada de EP, FT e MG na máquina SGI.

<i>threads</i>	EP.C	FT.C	MG.C
1	28,9318	5,5149	0,1638
2	15,7124	5,0387	2,2818
4	41,6708	2,3953	0,2516
6	15,5000	0,7622	0,4867
8	60,4040	1,5970	0,7491

Tabela A.18: Desvio padrão do tempo decorrido para as versões com *loops* invertidos de FT e MG na máquina SGI.

<i>threads</i>	FT.C	MG.C
1	0,6122	0,0323
2	3,3369	3,8233
4	0,1676	0,0990
6	0,4980	0,2705
8	0,4627	0,1674
10	2,2880	0,2934
12	0,6249	0,0789
16	0,5945	0,2460
20	0,6368	0,3300
24	0,9982	0,6965
28	1,0703	0,7500
32	0,6348	0,6860
64	0,1299	0,1046
96	0,0899	0,0396
128	0,0748	0,0330

Tabela A.19: Desvio padrão do tempo decorrido para as versões com *loops* invertidos de FT e MG na máquina HP.

<i>threads</i>	FT.C	MG.C	<i>threads</i>	FT.C	MG.C
1	0,1716	0,4159	16	0,4222	0,5574
2	0,0554	0,5163	20	0,4236	0,2994
4	0,5957	1,3417	24	0,2910	0,3637
6	0,1669	1,3730	36	0,2504	0,2885
8	0,0780	1,1654	48	0,2733	0,3489
10	0,1066	0,5433	60	0,1677	0,2510
12	0,3532	1,0060	72	0,2534	0,5558

Tabela A.20: Desvio padrão do tempo decorrido para as versões com *loops* invertidos de FT e MG na máquina AMD.

<i>threads</i>	FT.C	MG.C
1	2,1447	3,3449
2	0,4662	1,7179
3	0,6138	1,8943
4	0,3103	0,7156
8	0,7487	1,7676
12	0,8175	1,8951
16	0,1779	1,9141

Tabela A.21: Desvio padrão do tempo decorrido para as versões com *loops* invertidos de FT e MG na máquina MTL.

<i>threads</i>	FT.C	MG.C	<i>threads</i>	FT.C	MG.C
1	2,4527	0,5728	34	3,4512	8,8348
2	1,2168	0,3848	36	3,3818	7,5108
4	1,9155	0,9581	38	3,6175	8,4926
6	2,5217	2,8655	40	4,2256	8,6174
8	3,3621	4,5885	44	2,8133	5,2331
10	3,6010	5,3636	48	2,5442	5,1268
12	3,5695	3,4379	52	2,1782	4,5522
14	3,7068	6,9828	56	1,8692	4,7971
16	4,0118	9,8294	60	1,8736	4,4711
18	4,2813	14,3973	64	1,7810	5,1887
20	4,1665	16,6934	68	2,4448	5,1873
22	4,0509	14,7115	72	1,8654	4,7843
24	3,8505	13,8382	76	1,9576	5,7367
26	3,5624	12,2157	80	2,0805	6,4418
28	3,5248	11,8118	100	3,7090	5,9388
30	3,6203	10,8504	120	4,0604	7,3047
32	3,1942	8,4620	140	1,9421	6,7627

Tabela A.22: Desvio padrão do tempo decorrido para as versões de MultMat na máquina SGI.

<i>threads</i>	MultMat	MultMat-fina	MultMat-grossa	MultMat-inv
1	0,0807	0,0623	0,0263	0,0643
2	2,1203	2,6776	3,7032	10,4112
4	1,4019	0,5718	1,1136	1,6141
6	8,4327	8,9537	0,8836	4,8792
8	1,1867	8,0106	2,1700	0,0177
10	4,6923	4,6021	2,3550	4,9489
12	2,0743	1,9069	1,6971	2,2818
14	4,5159	2,4550	1,8779	1,4761
16	10,4256	5,8962	7,9774	9,1575
20	0,9688	0,9356	1,4925	0,4580
24	6,0105	2,4778	6,4102	3,5544
32	6,7057	6,8431	4,8593	2,8170

Tabela A.23: Desvio padrão do tempo decorrido para as versões de MultMat na máquina AMD.

<i>threads</i>	MultMat	MultMat-fina	MultMat-grossa	MultMat-inv
1	6,4600	7,5743	10,3232	0,4050
2	13,2211	13,3133	18,4480	0,0910
3	22,8683	11,0981	25,4568	0,6121
4	1,2957	1,2327	0,9075	0,5906
6	2,0801	3,2354	4,2199	0,5285
8	0,9510	0,4995	1,2567	0,0860
10	0,7120	0,6697	1,0003	0,2672
12	0,3979	0,4014	0,3790	0,2392
16	0,4627	0,3204	0,2928	0,1297

Tabela A.24: Desvio padrão do tempo decorrido para as versões de MultMat na máquina MTL.

<i>threads</i>	MultMat	MultMat-fina	MultMat-grossa	MultMat-inv
1	0,2660	0,4407	0,4475	3,4636
2	1,4406	0,5470	1,1585	4,3263
4	2,3854	0,5025	1,4652	18,5127
6	1,2950	1,3028	1,2788	22,5849
8	3,4302	1,9085	0,4843	18,8100
10	2,8951	1,7226	1,6057	19,7431
12	2,9097	0,6969	0,5007	18,8652
14	3,5873	1,8484	2,2529	19,3293
16	2,7209	1,8011	2,1449	18,8441
18	1,5735	2,1233	1,0863	19,6229
20	0,8159	1,6880	0,8638	19,1617
22	1,0014	0,8110	1,4338	19,7453
24	3,2661	0,6748	1,5404	20,2162
26	0,9729	0,5624	1,9048	20,2895
28	1,1923	2,1925	0,8433	20,3321
30	2,3700	0,6687	1,0362	19,6061
32	2,2151	0,5682	0,2577	19,5846
34	2,9036	0,6610	0,9265	19,0958
36	0,9398	0,6269	0,8740	18,6403
38	1,4308	0,7041	2,2113	19,4529
40	2,4566	14,9663	14,5756	12,0742
42	3,1690	1,7167	3,0979	11,2346
44	2,3580	1,6761	2,2476	13,4955

Tabela A.25: Desvio padrão do tempo decorrido para STREAM nas máquinas SGI, AMD e HP.

<i>threads</i>	1	2	4	8	12	16
SGI	3,1088	7,3104	1,5170	6,2264	4,5752	1,6143
AMD	2,1013	9,4365	7,4867	5,4896	9,0308	3,7521
HP	4,8039	2,0310	2,9249	8,7282	6,3625	1,2511

Tabela A.26: Desvio padrão do tempo decorrido para STREAM na máquina MTL.

<i>threads</i>	desvio	<i>threads</i>	desvio
1	6,6661	22	2,6254
2	3,4029	24	0,8258
4	1,7034	26	4,6588
6	5,0472	28	7,9514
8	6,7245	30	3,7747
10	5,5815	32	1,3253
12	6,3927	34	6,7354
14	7,5326	36	3,9597
16	1,7842	38	3,6727
18	4,5695	40	4,2655
20	3,4723		

Tabela A.27: Desvio padrão de γ_E para STREAM e seus componentes na máquina SGI.

<i>threads</i>	STREAM	COPY	SCALE	ADD	TRIAD
1	6,1464	6,1303	0,9414	7,6794	5,7072
2	8,6869	6,7968	2,9945	5,9538	7,4831
4	2,5342	0,3208	7,8135	2,6044	5,1082
6	8,4518	5,1137	8,0341	8,2440	1,3527
8	8,4108	3,7693	6,6245	4,3236	7,6277
10	8,1157	3,9893	2,2573	3,2404	6,9240
12	8,0036	0,3869	4,0543	8,4450	8,0662
14	0,7615	8,1318	5,8630	3,7560	5,0856
16	4,3462	6,2902	5,4064	3,1597	8,8946
20	1,5146	2,6115	5,0083	0,5487	1,8555
24	6,3609	8,9595	5,6248	3,9854	4,2832
32	4,2524	3,1011	8,2724	6,5097	6,3416

Tabela A.28: Desvio padrão do tempo decorrido para STREAM com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	6,8729	8,6661	0,8160	1,3256	9,2379	1,4529	1,3239	6,7553
2	1,6014	4,1996	6,9440	4,8698	2,1525	7,2634	3,7022	7,2763
3	8,7761	6,2507	6,0955	1,4608	2,9710	4,5307	1,2623	0,3208
4	7,3120	0,3528	2,0853	6,9824	2,4294	4,9047	3,7713	9,3024
5	3,5709	4,5873	0,6279	2,8087	6,0402	1,9519	9,5640	7,6416
6	6,1515	6,5080	2,5114	8,3040	3,7714	6,2137	5,5803	2,5475
7	2,4644	1,6758	4,0084	5,4354	6,2065	5,2707	5,7561	3,5186
8	5,6235	7,8414	0,5010	8,0529	2,7462	4,2723	7,3553	6,3171
9	8,8596	7,9832	9,1258	4,8999	9,9351	8,6898	2,5415	6,0866
10	5,1978	5,0529	4,3906	8,9693	1,2666	9,9709	1,5168	3,7310
11	1,6467	5,5252	9,1663	7,8533	0,7959	4,9225	1,3718	6,4193
12	2,7639	1,8728	4,4722	5,5101	6,1451	1,8275	1,8271	5,0048
13	9,8107	0,9529	9,9047	9,7459	9,6427	2,4462	5,8325	4,8406
14	7,4991	0,2231	3,8098	8,7656	0,1940	5,3267	2,4966	1,8408
15	0,8518	1,6629	9,6941	1,6477	6,5854	1,0659	8,0670	9,3493
16	2,9387	2,5393	4,8594	9,0838	4,3668	6,6865	4,0886	4,1775

Tabela A.29: Desvio padrão do tempo decorrido para STREAM-COPY com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	0,7948	2,9595	9,7657	8,8737	6,2934	6,9969	7,6554	8,3185
2	3,9048	9,6604	5,7387	3,6681	3,9652	5,5068	9,5491	5,6974
3	1,0551	8,5325	7,5602	0,6905	0,1532	0,8369	4,4659	0,0434
4	0,3382	8,1427	4,0419	6,2590	2,2512	1,3200	4,0592	3,0459
5	4,2795	3,8250	1,9196	0,5729	0,8219	9,5750	8,8914	4,7268
6	9,2354	4,6301	8,3948	3,2005	0,1369	7,9440	8,8980	1,1921
7	6,4765	6,4582	1,8826	6,6297	7,2951	6,3485	6,6731	7,6333
8	4,4912	0,7150	3,8923	6,7423	2,0350	7,9515	9,7883	6,3145
9	1,7765	1,7079	6,8874	2,5984	1,2829	5,7788	7,3252	0,5183
10	0,4089	5,7200	3,7189	0,5458	3,6640	2,6169	1,7379	0,1405
11	9,0751	3,6205	6,7702	6,3702	9,9690	3,4433	4,0035	4,4601
12	4,1582	7,8957	1,2025	6,1932	5,8473	0,9908	2,5077	7,6238
13	2,6987	9,3951	0,2223	3,9816	5,1739	7,5475	4,5000	5,5828
14	3,2675	8,2188	6,1286	6,9315	0,8357	7,8665	7,0720	9,9108
15	1,4869	3,8422	6,2810	1,4559	7,2855	0,2844	5,9161	1,4437
16	8,1802	7,1185	7,6370	4,0275	8,1093	0,1447	1,6513	0,8080

Tabela A.30: Desvio padrão do tempo decorrido para STREAM-SCALE com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	9,8060	4,8419	8,4559	5,9161	0,9553	4,7946	3,4478	9,3184
2	5,9824	3,8284	1,4505	9,7379	1,7349	4,9713	2,9298	4,3014
3	0,1660	6,0156	1,7609	7,1721	5,6797	5,2418	3,9279	7,7491
4	4,3632	0,2953	9,1986	8,1096	3,0560	2,1999	9,6217	2,8620
5	7,0418	8,0776	8,7781	7,9971	2,8722	2,2259	7,3154	8,8546
6	6,0543	8,7659	8,5925	7,7892	3,7372	1,5223	2,0906	3,9032
7	7,5380	3,8515	1,0753	3,2177	9,0933	5,0033	0,9668	3,4564
8	5,2986	0,1653	1,5660	8,3546	2,3652	1,1878	1,2166	9,4070
9	9,2654	9,9947	7,4041	2,1376	2,2206	4,7195	0,9922	8,2750
10	3,4854	9,5847	6,0642	7,2227	1,1070	8,1548	1,1259	8,6450
11	2,0063	2,2012	1,8627	1,0995	7,2045	2,8295	4,5560	2,5031
12	2,9948	6,1220	0,8577	5,3600	7,3097	2,0744	4,7670	6,5751
13	2,0691	2,1711	8,7126	4,2897	6,8906	9,7048	2,5647	0,3760
14	9,2895	8,6289	7,5987	0,3965	6,7837	8,7246	9,0414	8,7900
15	0,9258	0,9041	9,8895	8,1303	3,7335	4,4455	0,6334	6,7283
16	0,5674	1,4911	2,0884	7,8772	3,5655	6,8554	4,4522	5,6346

Tabela A.31: Desvio padrão do tempo decorrido para STREAM-ADD com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	3,3220	1,5151	7,9646	3,9022	4,1641	4,0210	4,0631	4,7285
2	3,7883	5,8135	4,1610	9,2610	3,3326	7,3377	2,1017	4,1743
3	6,0017	7,7508	3,4428	6,3482	5,9615	5,1199	6,2552	6,9240
4	8,6122	1,5487	5,5930	3,5953	9,1255	3,8675	8,3549	2,4475
5	5,3826	6,3196	6,3497	9,5466	0,3405	0,4128	4,2751	4,1288
6	6,2263	8,4361	3,3899	9,5588	5,7738	5,4916	3,7331	1,7755
7	3,2424	7,1759	8,1237	9,2039	2,2958	4,3789	6,1278	0,9081
8	5,9277	1,7208	4,5033	5,0532	5,5883	2,8583	7,5008	0,9708
9	9,1778	3,8505	0,5174	9,5184	4,2633	4,7925	3,6472	0,4896
10	3,2286	7,0371	0,0484	9,0024	2,5287	3,7816	0,7779	5,7710
11	0,9575	8,9017	4,9749	3,2533	3,2806	1,1028	4,1614	9,2083
12	2,8236	8,6647	4,2615	8,4118	1,5230	1,7623	9,3827	0,7009
13	5,6127	9,9001	0,2192	9,8760	4,6926	3,8665	0,3656	7,9213
14	0,9035	0,4141	6,9237	3,4322	4,1956	7,7016	9,2032	5,1531
15	6,6033	4,1781	8,4063	9,8839	5,2809	2,5677	9,0922	8,1045
16	1,2324	3,3537	6,5163	2,7555	5,1159	5,8990	3,4563	0,7287

Tabela A.32: Desvio padrão do tempo decorrido para STREAM-TRIAD com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	4,7019	3,9161	1,4681	0,4131	7,9195	3,9736	4,3896	2,8171
2	7,7727	7,9284	9,3663	6,3595	5,9034	3,0631	6,8691	6,3422
3	4,2283	5,8768	3,2678	9,9647	3,1757	1,7341	7,0650	2,6914
4	0,1084	8,7392	4,2465	0,3746	2,5323	8,9128	3,7503	7,2342
5	2,8289	5,2184	7,6473	0,7484	9,1920	2,0369	3,5655	6,9647
6	9,9652	2,9318	3,3242	5,8687	5,9949	0,1934	2,2109	0,2232
7	6,0702	5,4787	0,1879	9,2459	7,2128	7,2529	1,9373	7,3212
8	5,9921	6,1838	7,6957	8,5245	5,0966	1,4461	5,7586	7,9255
9	6,6645	3,4059	8,6739	5,8565	5,4428	2,2393	2,8212	5,4080
10	5,1711	6,1454	1,2767	1,1660	6,3388	3,4876	1,3892	2,4090
11	8,9663	1,5772	1,6549	6,1791	8,8301	3,5922	3,5003	4,8222
12	9,7760	1,1961	3,3467	4,8726	2,6421	9,1053	2,7981	9,3066
13	2,5112	1,4719	5,1630	7,9540	3,7113	7,9842	3,3621	8,8824
14	4,1296	4,6388	0,0484	0,4684	8,1264	1,4376	2,8774	7,0927
15	3,0148	4,5323	3,2718	1,8449	8,1245	6,7721	6,6671	7,9005
16	7,9682	0,0138	2,7731	0,6103	9,1191	5,5712	9,9169	1,6303

Tabela A.33: Desvio padrão da taxa de uso de barramento para STREAM com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	8,9190	4,2476	4,0474	4,2300	9,0104	5,9886	4,0413	6,8628
2	3,9068	4,3502	0,2355	2,9114	0,3660	5,0829	3,7709	7,5469
3	1,5612	3,9653	4,0381	0,6611	4,6350	0,8718	0,2968	6,1949
4	5,5131	7,4221	5,0554	6,8811	2,4221	2,3055	5,8308	1,3411
5	6,5531	9,8782	5,5711	5,5635	5,8668	9,6125	2,4263	9,7735
6	3,9627	2,6619	2,6849	4,3286	7,7448	6,4558	1,8755	9,3060
7	0,4211	5,9136	9,9672	5,0561	6,7854	0,2639	1,2510	2,2985
8	7,6860	6,3064	9,1796	0,1081	8,6119	5,0104	1,4493	5,1650
9	4,8885	7,0204	0,7285	0,7553	6,6328	3,1549	0,5288	0,5955
10	5,8168	3,2137	4,9241	3,5616	9,6695	6,7996	2,8676	0,0906
11	2,7132	2,8348	5,1468	9,4986	3,0987	6,3978	1,7971	0,7847
12	2,7041	0,9766	0,8928	1,3160	5,9870	2,3420	6,4810	0,8755
13	9,3624	7,2095	1,6308	5,9953	0,3644	2,1595	6,5908	6,1812
14	5,3732	1,5149	9,7428	5,0427	8,3145	2,6104	5,1333	1,0277
15	5,4452	0,2801	0,5263	8,5439	6,6778	2,3234	9,3285	9,3819
16	3,3000	0,2213	0,6979	9,2870	2,5634	7,1789	0,1625	1,9258

Tabela A.34: Desvio padrão da taxa de uso de barramento para STREAM-COPY com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	7,2433	6,8667	2,7756	4,7076	8,7634	3,6341	9,0396	9,5143
2	4,8441	3,4930	5,6289	2,7869	2,1394	1,7146	4,8862	0,4758
3	7,0342	0,8182	6,8459	2,8104	5,5110	3,8405	2,2517	9,5998
4	4,5653	1,7883	4,2249	2,2880	1,0087	9,7550	4,0176	8,2521
5	6,6218	6,7932	2,9597	5,3852	0,4273	1,9993	4,8995	5,2714
6	5,4923	0,5284	8,0583	7,6318	2,2431	2,9445	8,1075	9,2772
7	3,7628	4,9534	2,0876	9,2738	8,7939	4,3393	8,8735	3,3592
8	6,1276	3,0984	5,6472	7,1363	2,8535	9,6649	5,3884	9,4752
9	6,4581	8,3481	4,8604	6,8854	0,3474	9,7599	2,1568	5,8397
10	0,2884	0,2152	3,4715	2,5314	3,1597	1,5790	1,8087	6,9224
11	6,5325	3,8963	6,1962	5,3264	8,2356	5,0698	8,6856	4,3632
12	8,1682	4,3328	1,4995	1,0216	3,9977	6,8879	0,4969	0,4557
13	5,2360	5,3573	7,3411	5,5834	5,1173	9,4979	1,4231	5,4056
14	9,7131	4,8946	7,9371	2,8728	6,4736	9,7458	9,7952	3,0061
15	3,6421	5,9914	8,3324	1,8777	1,0612	7,0180	6,2409	9,2294
16	1,3508	7,7405	0,2510	5,3485	4,6284	0,7479	5,8042	9,8644

Tabela A.35: Desvio padrão da taxa de uso de barramento para STREAM-SCALE com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	3,2155	3,4409	4,8700	4,1262	1,6225	6,7433	0,2751	6,7499
2	4,6435	9,1600	0,6970	1,6610	9,2940	4,9157	0,7169	1,2674
3	7,4536	0,8042	6,9768	4,3050	4,7280	9,8758	4,7592	3,7342
4	5,7236	7,2047	7,3737	8,6894	2,2747	8,9501	2,5757	5,4903
5	2,3910	7,4456	9,6164	4,0135	4,1890	9,8915	0,7634	8,8325
6	9,0515	1,4604	0,4935	8,3455	6,3760	1,2104	9,6129	3,8297
7	2,0146	6,5896	8,1347	6,7426	6,4654	2,8939	0,4769	2,1890
8	0,0986	7,8505	0,8784	2,3733	6,8006	3,4541	7,8635	9,1916
9	0,8997	7,4799	3,2051	5,0887	7,3715	3,9685	3,9212	6,4229
10	5,4288	4,4147	4,7684	1,8049	5,6251	4,3813	5,6346	7,6397
11	0,9709	3,7693	4,3823	7,4363	6,6632	4,8592	9,6253	6,7617
12	2,7097	0,5037	9,1350	9,5103	3,9578	6,9985	8,7019	4,8575
13	4,4785	1,9070	9,9461	1,8499	5,8755	3,8673	8,2729	1,3043
14	8,2820	3,0413	3,1092	3,9071	7,4225	8,7438	1,5467	8,3934
15	2,5131	5,9291	5,8297	9,1762	0,7882	5,4550	5,9380	3,4979
16	5,9587	5,0729	3,0082	9,9165	2,0715	1,7101	4,7739	6,5499

Tabela A.36: Desvio padrão da taxa de uso de barramento para STREAM-ADD com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	6,0577	7,4740	2,8145	3,4408	7,6909	7,6596	2,3602	8,2179
2	1,0700	3,5558	0,4796	5,9280	6,0194	7,2598	9,4552	7,7833
3	8,1569	9,1504	9,4181	5,5427	7,8626	7,0409	4,4785	5,8624
4	4,0251	7,4894	7,3920	4,6182	9,2839	9,0649	3,4336	5,3416
5	6,5389	6,2481	8,7824	4,2298	3,9077	1,1426	2,4477	4,9777
6	4,6984	2,9273	0,9057	0,7178	0,1871	0,3609	8,5012	8,3440
7	9,5113	7,9193	3,8866	7,3739	4,9602	8,3651	3,2363	8,9853
8	5,8545	0,6283	3,6035	5,1384	9,6932	7,0371	0,4800	6,2322
9	3,2852	9,2625	0,4620	7,1929	0,4051	2,9097	2,1705	5,1035
10	5,8370	3,0762	5,8213	6,0241	3,4371	4,3225	4,3680	2,9484
11	2,2418	8,2546	0,3224	7,2020	6,6197	3,5587	6,1874	2,4742
12	4,1870	9,7909	7,6126	3,8802	6,8280	8,0927	0,1124	0,1132
13	7,3551	0,5743	7,3061	7,7602	3,4840	9,4766	2,8637	9,3210
14	2,5528	8,6850	5,3451	5,9899	3,0075	9,7131	8,9383	5,2494
15	7,9677	9,2607	2,4514	4,5875	2,8194	8,6388	7,0617	7,0064
16	8,4296	4,6743	0,8866	5,2576	2,7670	0,9990	5,3708	0,1221

Tabela A.37: Desvio padrão da taxa de uso de barramento para STREAM-TRIAD com diferentes volumes de processamento extra na máquina SGI.

<i>threads</i>	volume de processamento extra							
	1	2	5	10	15	20	25	30
1	6,7647	0,5541	7,6515	1,3063	3,8331	5,0305	5,6985	4,7088
2	2,8028	7,6446	7,9835	9,1477	3,7433	8,5989	4,2358	2,3076
3	0,2744	7,2913	0,3714	3,9376	1,2658	4,5763	2,9812	5,6055
4	2,1700	8,6342	6,3732	2,5801	2,6387	4,1308	2,4710	9,4034
5	4,6850	0,1225	0,7097	8,5180	5,1529	6,4082	3,2268	7,9557
6	4,0528	1,2104	7,1034	7,7960	9,8092	1,3393	0,1036	0,0836
7	8,6306	0,4750	4,0212	9,8964	5,0513	7,0024	5,5018	7,2213
8	5,6366	1,8750	9,8014	8,2753	6,0059	2,2723	7,6787	0,6908
9	2,3948	8,3883	9,2089	7,5478	4,7965	2,4357	5,5035	8,8493
10	3,6460	2,6069	6,6453	3,4552	3,9462	6,7489	3,5388	2,5768
11	7,2239	7,5600	2,4731	2,2752	4,5624	7,9750	9,4966	0,1990
12	9,8500	9,2980	8,4743	5,8559	1,5703	6,1529	6,5467	3,9651
13	4,5413	5,7556	1,5129	9,3378	8,1912	7,0164	8,1871	1,8372
14	9,6233	4,8323	5,2924	3,5695	1,5812	8,8313	6,1462	8,8051
15	6,3913	8,6194	1,0803	0,9537	6,5943	0,5769	1,1527	6,4443
16	9,8749	9,6270	2,3002	1,4452	5,7799	8,8469	5,4103	0,3212

Apêndice B

Código-fonte das alterações de granularidade

Este apêndice apresenta as listagens de código-fonte das alterações de granularidade de *loop* de alguns dos programas analisados, discutidas no capítulo 5. As versões originais dos *loops* selecionados de EP, FT, MG e MultMat são mostradas nas listagens B.1, B.3, B.5 e B.7, respectivamente. Os *loops* modificados de EP, FT e MG são mostrados nas listagens B.2, B.4 e B.6, respectivamente. O *loop* de MultMat-fina é mostrado na listagem B.8, enquanto o *loop* de MultMat-grossa é mostrado na listagem B.9. A listagem B.10 apresenta o *loop* principal de STREAM com um bloco de processamento que é ajustado pelo programa da listagem B.11.

Listagem B.1: *Loop* selecionado de EP.

```
1      do 140 i = 1, nk
2          x1 = 2.d0 * x(2*i-1) - 1.d0
3          x2 = 2.d0 * x(2*i) - 1.d0
4          t1 = x1 ** 2 + x2 ** 2
5          if (t1 .le. 1.d0) then
6              t2 = sqrt(-2.d0 * log(t1) / t1)
7              t3 = (x1 * t2)
8              t4 = (x2 * t2)
9              l = max(abs(t3), abs(t4))
10             qq(l) = qq(l) + 1.d0
11             sx = sx + t3
12             sy = sy + t4
13         endif
14 140 continue
```

Listagem B.2: *Loop selecionado de EP após alteração de granularidade.*

```

1      do 140 i = 1, nk
2          x1 = 2.d0 * x(2*i-1) - 1.d0
3          x2 = 2.d0 * x(2*i) - 1.d0
4          t1 = x1 ** 2 + x2 ** 2
5          if (t1 .le. 1.d0) then
6              t2 = 1
7              t3 = (x1 * t2)
8              t4 = (x2 * t2)
9              l = max(abs(t3), abs(t4))
10             qq(l) = qq(l) + 1.d0
11             sx = sx + t3
12             sy = sy + t4
13         endif
14 140 continue

```

Listagem B.3: *Loop selecionado de FT.*

```

1      do k = 0, lk - 1
2          do j = 1, ny
3              x11 = x(j, i11+k)
4              x21 = x(j, i12+k)
5              y(j, i21+k) = x11 + x21
6              y(j, i22+k) = u1 * (x11 - x21)
7          enddo
8      enddo

```

Listagem B.4: *Loop selecionado de FT após alteração de granularidade.*

```

1      do k = 0, lk - 1
2          do j = 1, ny
3              x11 = sqrt(x(j, i11+k))
4              x21 = log(x(j, i12+k))
5              y(j, i21+k) = x11 * x21
6              y(j, i22+k) = u1 * (x11 - x21)
7          enddo
8      enddo

```

Listagem B.5: *Loop selecionado de MG.*

```

1      do i3=2, n3-1
2          do i2=2, n2-1
3              do i1=1, n1
4                  u1(i1) = u(i1, i2-1, i3) + u(i1, i2+1, i3)
5                  >          + u(i1, i2, i3-1) + u(i1, i2, i3+1)

```

```

6           u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
7   >           + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
8           enddo
9           do i1=2,n1-1
10          r(i1,i2,i3) = v(i1,i2,i3)
11   >          - a(0) * u(i1,i2,i3)
12 C-----
13 c Assume a(1) = 0           (Enable 2 lines below if a(1) not= 0)
14 C-----
15 c   >          - a(1) * ( u(i1-1,i2,i3) + u(i1+1,i2,i3)
16 c   >          + u1(i1) )
17 C-----
18   >          - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
19   >          - a(3) * ( u2(i1-1) + u2(i1+1) )
20           enddo
21           enddo
22           enddo

```

Listagem B.6: *Loop* selecionado de MG após alteração de granularidade.

```

1           do i3=2,n3-1
2           do i2=2,n2-1
3           do i1=1,n1
4           u1(i1) = u(i1,i2-1,i3) * u(i1,i2+1,i3)
5   >           * u(i1,i2,i3-1) * 3.2
6           u2(i1) = u(i1,i2-1,i3-1) * u(i1,i2+1,i3-1)
7   >           * u(i1,i2-1,i3+1) * 7.8
8           enddo
9           do i1=2,n1-1
10          r(i1,i2,i3) =
11   >          a(0) * u(i1,i2,i3)
12 C-----
13 c Assume a(1) = 0           (Enable 2 lines below if a(1) not= 0)
14 C-----
15 c   >          - a(1) * ( u(i1-1,i2,i3) + u(i1+1,i2,i3)
16 c   >          + u1(i1) )
17 C-----
18   >          * a(2) * 7.4 * ( u2(i1) + u1(i1-1) + u1(i1+1)
19   >          * a(3) * ( u2(i1-1) + u2(i1+1) ) )
20           enddo
21           enddo
22           enddo

```

Listagem B.7: Loop de MultMat.

```
1  do i=1, lin_c
2      do j=1, col_c
3          soma = 0
4              do k=1, col_a
5                  soma = soma + a(k,i) * b(j,k)
6              enddo
7          c(j,i) = soma
8      enddo
9  enddo
```

Listagem B.8: Loop de MultMat-fina.

```
1  do i=1, lin_c
2      do j=1, col_c
3          soma = 0
4              do k=1, col_a
5                  soma = soma + a(k,i) * b(j,k)
6                  a(k,i) = c(j,i) + b(j,k)
7              enddo
8          c(j,i) = soma
9      enddo
10 enddo
```

Listagem B.9: Loop de MultMat-grossa.

```
1  do i=1, lin_c
2      do j=1, col_c
3          soma = 0
4              do k=1, col_a
5                  soma = 1.7 * soma + a(k,i) * b(j,k) / 2.3
6              enddo
7          c(j,i) = soma
8      enddo
9  enddo
```

Listagem B.10: Loop principal de STREAM com ajuste de granularidade.

```
1  for (k = 0; k < NTIMES; k++) {
2      times[0][k] = mysecond();
3  #ifdef COPY
4      #pragma omp parallel for
5      for (j = 0; j < N; j++) {
6          c[j] = a[j];
7          bloco_processamento();
8      }
9  #endif
10     times[0][k] = mysecond() - times[0][k];
11
12     times[1][k] = mysecond();
13 #ifdef SCALE
14     #pragma omp parallel for
15     for (j = 0; j < N; j++) {
16         b[j] = scalar * c[j];
17         bloco_processamento();
18     }
19 #endif
20     times[1][k] = mysecond() - times[1][k];
21
22     times[2][k] = mysecond();
23 #ifdef ADD
24     #pragma omp parallel for
25     for (j = 0; j < N; j++) {
26         c[j] = a[j] + b[j];
27         bloco_processamento();
28     }
29 #endif
30     times[2][k] = mysecond() - times[2][k];
31
32     times[3][k] = mysecond();
33 #ifdef TRIAD
34     #pragma omp parallel for
35     for (j = 0; j < N; j++) {
36         a[j] = b[j] + scalar * c[j];
37         bloco_processamento();
38     }
39 #endif
40     times[3][k] = mysecond() - times[3][k];
41 }
```

Listagem B.11: Programa de ajuste de granularidade de STREAM.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6     int i, n;
7
8     if (argc < 2) {
9         fprintf(stderr, "Uso: %s volume_de_processamento\n"
10             "volume e' a quantidade de operacoes basicas a efetuar\n",
11             argv[0]);
12         return 0;
13     }
14
15     n = atoi(argv[1]);
16     if (n < 1)
17         n = -n;
18
19     fputs("#define bloco_processamento()", stdout);
20     if (n > 0) {
21         puts(" __asm__ __volatile__ ( \\");
22         for (i = n; i > 0; --i) {
23             puts("\t\t\"mulsd %%xmm14, %%xmm15\\n\\t\t\" \\");
24         }
25         puts(": : : \"xmm14\", \"xmm15\")");
26     }
27
28     putchar('\n');
29
30     return 0;
31 }
```

Referências

- ADVANCED MICRO DEVICES, INC. *AMD Processors for Servers and Workstations: AMD Opteron Processor*. 2013. Disponível em: <<http://products.amd.com/pt-br/OpteronCPUDetail.aspx?id=567&f1=Six-Core+AMD+Opteron%E2%84%A2&f2=&f3=Sim&f4=&f5=512&f6=Socket+F+%281207%29&f7=&f8=45nm+SOI&f9=&f10=4800&f11=6&>>. Acesso em: 3 abr. 2013.
- AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS SPRING JOINT COMPUTER CONFERENCE, 1967, Atlantic City. *Proceedings...* New York: ACM, 1967. p. 483–485.
- BARKER, K. et al. A performance evaluation of the Nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, v. 18, n. 4, p. 453–469, dez. 2008.
- BELL, S. et al. Tile64 - processor: A 64-core SoC with mesh interconnect. In: INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE (ISSCC), 2008, San Francisco. *Proceedings...* [S.l.]: IEEE, 2008. p. 88–598.
- BLAGODUROV, S. et al. A case for NUMA-aware contention management on multicore systems. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT '10), 19. *Proceedings...* New York: ACM, 2010. p. 557–558. ISBN 978-1-4503-0178-7.
- BUNT, R. B.; MURPHY, J. M. The measurement of locality and the behaviour of programs. *The Computer Journal*, Oxford, v. 27, n. 3, p. 238–253, ago. 1984.
- CALLAHAN, D.; COCKE, J.; KENNEDY, K. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, Orlando, v. 5, n. 4, p. 334–358, ago 1988.
- CASAZZA, J. *First the Tick, Now the Tock - Intel Microarchitecture (Nehalem)*. 2010. Disponível em: <<http://www.intel.com/content/dam/doc/white-paper/intel-microarchitecture-white-paper.pdf>>. Acesso em: 21 set. 2011.
- CONWAY, P.; HUGHES, B. The AMD Opteron northbridge architecture. *IEEE Micro*, Los Alamitos, v. 27, n. 2, p. 10–21, mar. 2007.
- DIAMOND, J. et al. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In: INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS '11), 2011, Austin. *Proceedings...* Washington: IEEE Computer Society, 2011. p. 32–43.

- DONGARRA, J. The LINPACK benchmark: An explanation. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1., 1987, Atenas. *Proceedings...* London: Springer-Verlag, 1988. p. 456–474.
- FENG, H. et al. *Unstructured Adaptive (UA) NAS Parallel Benchmark, Version 1.0*. Moffett Field, CA, 2004. Disponível em: <<http://www.nas.nasa.gov/assets/pdf/techreports/2004/nas-04-006.pdf>>. Acesso em: 12 mai. 2011.
- FREE SOFTWARE FOUNDATION. *GCC, the GNU Compiler Collection - GNU Project*. 2013. Disponível em: <<http://gcc.gnu.org/>>. Acesso em: 13 mar. 2013.
- FREE SOFTWARE FOUNDATION, INC. *GOMP_CPU_AFFINITY - GNU libgomp*. 2006. Disponível em: <http://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fCPU_005fAFFINITY.html>. Acesso em: 30 mar. 2013.
- FREE SOFTWARE FOUNDATION, INC. *Implementing FOR construct - GNU libgomp*. 2006. Disponível em: <<http://gcc.gnu.org/onlinedocs/gcc-4.2.4/libgomp/Implementing-FOR-construct.html#Implementing-FOR-construct>>. Acesso em: 19 abr. 2013.
- FREE SOFTWARE FOUNDATION, INC. *OMP_NUM_THREADS - GNU libgomp*. 2006. Disponível em: <http://gcc.gnu.org/onlinedocs/libgomp/OMP_005fNUM_005fTHREADS.html>. Acesso em: 30 mar. 2013.
- FREE SOFTWARE FOUNDATION, INC. *GIMPLE - GNU Compiler Collection (GCC) Internals*. 2011. Disponível em: <<http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html>>. Acesso em: 04 mar. 2011.
- FREE SOFTWARE FOUNDATION, INC. *Assembler Instructions with C Expression Operands*. 2012. Disponível em: <<http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm>>. Acesso em: 10 jan. 2013.
- GRAMA, A. et al. *Introduction to Parallel Computing*. 2. ed. Essex: Pearson Education, 2003. 636 p.
- GREWE, D.; WANG, Z.; O'BOYLE, M. F. P. A workload-aware mapping approach for data-parallel programs. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE AND EMBEDDED ARCHITECTURES AND COMPILERS (HIPEAC '11), 6., 2011, Heraklion. *Proceedings...* New York: ACM, 2011. p. 117–126.
- GSCHWIND, M. et al. Synergistic processing in Cell's multicore architecture. *Micro, IEEE*, Los Alamitos, v. 26, n. 2, p. 10–24, 2006.
- GUPTA, S. et al. Locality principle revisited: A probability-based quantitative approach. In: INTERNATIONAL PARALLEL DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 26., 2012, Shanghai. *Proceedings...* Washington: IEEE Computer Society, 2012. p. 995–1009.
- GUSTAFSON, J. L. Reevaluating Amdahl's law. *Communications of the ACM*, New York, v. 31, n. 5, p. 532–533, mai. 1988.

- HOOD, R. et al. Performance impact of resource contention in multicore systems. In: INTERNATIONAL SYMPOSIUM ON PARALLEL DISTRIBUTED PROCESSING (IPDPS), 2010, Atlanta. *Proceedings...* [S.l.]: IEEE, 2010. p. 1–12.
- INTEL CORPORATION. *An Introduction to the Intel QuickPath Interconnect*. 2009. Disponível em: <<http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf>>. Acesso em: 03 abr. 2013.
- JIN, H.; FRUMKIN, M.; YAN, J. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Moffet Field, 1999. Disponível em: <<http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf>>. Acesso em: 12 mai. 2011.
- LI, Y. et al. Compiler-assisted data distribution for chip multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT '10), 19., 2010, Vienna. *Proceedings...* New York: ACM, 2010. p. 501–512.
- MALLADI, R. *Using Intel VTune Performance Analyzer events/ratios & optimizing applications white paper*. 2009. Disponível em: <<http://software.intel.com/file/8486>>. Acesso em: 28 set. 2011.
- MARS, J.; TANG, L.; SOFFA, M. L. Directly characterizing cross core interference through contention synthesis. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE AND EMBEDDED ARCHITECTURES AND COMPILERS (HIPEAC '11), 6., 2011, Heraklion. *Proceedings...* New York: ACM, 2011. p. 167–176.
- MCCALPIN, J. D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. 1991–2007. Um relatório técnico continuamente em atualização. Disponível em: <<http://www.cs.virginia.edu/stream/>>. Acesso em: 6 jul. 2012.
- MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, p. 19–25, dez. 1995.
- MEUER, H. et al. *Introduction and Objectives — TOP500 Supercomputer Sites*. 2013. Disponível em: <<http://www.top500.org/project/introduction/>>. Acesso em: 10 abr. 2013.
- NASA. *NAS Parallel Benchmarks*. 2013. Disponível em: <<http://www.nas.nasa.gov/publications/npb.html>>. Acesso em: 28 set. 2011.
- NICOLAU, A.; KEJARIWAL, A. How many threads to spawn during program multithreading? In: INTERNATIONAL CONFERENCE ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC'10), 23., 2010, Houston. *Proceedings...* Berlin: Springer-Verlag, 2011. p. 166–183.
- NOVILLO, D. OpenMP and automatic parallelization in GCC. In: GCC DEVELOPERS' SUMMIT, 2006, Ottawa. *Proceedings...* 2006. p. 261–270. Disponível em: <<http://www.airs.com/dnovillo/Papers/gcc2006.pdf>>. Acesso em: 27 out. 2010.

OLUKOTUN, K. et al. The case for a single-chip multiprocessor. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS (ASPLOS VII), 7., 1996, Cambridge, Massachusetts. *Proceedings...* New York: ACM, 1996. p. 2–11.

OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface, Version 3.0*. 2008. Disponível em: <<http://www.openmp.org/mp-documents/spec30.pdf>>. Acesso em: 13 jan. 2012.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. 2. ed. San Francisco: Morgan Kaufmann Publishers Inc., 1996.

ROGERS, B. M. et al. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. *ACM SIGARCH Computer Architecture News*, New York, v. 37, n. 3, p. 371–382, jun. 2009.

RUSSEL, S. J.; NORVIG, P. *Artificial Intelligence, A Modern Approach*. Upper Saddle River: Prentice Hall, Inc., 1995.

SHIN, J. et al. A 40nm 16-core 128-thread CMT SPARC SoC processor. In: INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE (ISSCC), 2010, San Francisco. *Proceedings...* [S.l.]: IEEE, 2010. p. 98–99.

SMITH, A. J. Cache memories. *ACM Computing Surveys*, New York, v. 14, n. 3, p. 473–530, set. 1982.

SUN, X.-H.; BYNA, S.; HOLMGREN, D. Modeling data access contention in multicore architectures. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS '09), 15., 2009, Shenzhen. *Proceedings...* Washington: IEEE Computer Society, 2009. p. 213–219.

SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, San Francisco, v. 30, n. 2, mar. 2005. Disponível em: <<http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990>>. Acesso em: 26 mar. 2013.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2. ed. São Paulo: Pearson Prentice Hall, 2005. 695 p.

TILERA CORPORATION. *TILE-Gx8072 Specification Brief*. 2013. Disponível em: <http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8072_PB041-02.pdf>. Acesso em: 03 abr. 2013.

TOURNAVITIS, G. et al. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI '09), 2009, Dublin. *Proceedings...* New York: ACM, 2009. p. 177–187.

TOY, B.; DONGARRA, J. *Linpack benchmark, calculates FLOPS (versão em C)*. 1988. Disponível em: <<http://www.netlib.org/benchmark/linpackc.new>>. Acesso em: 10 set. 2012.

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: maximizing on-chip parallelism. In: 25 YEARS OF THE INTERNATIONAL SYMPOSIA ON COMPUTER ARCHITECTURE (ISCA '98), 1998, Barcelona. *Proceedings...* New York: ACM, 1998. p. 533–544.

WANG, Z.; O'BOYLE, M. F. Mapping parallelism to multi-cores: a machine learning based approach. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP '09), 14., 2009, Raleigh. *Proceedings...* New York: ACM, 2009. p. 75–84.

WEINBERG, J. et al. Quantifying locality in the memory access patterns of HPC applications. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (SC '05), 2005, Seattle. *Proceedings...* Washington: IEEE Computer Society, 2005. p. 1–12.

WILKINSON, B.; ALLEN, M. *Parallel programming - techniques and applications using networked workstations and parallel computers*. 2. ed. Upper Saddle River: Pearson Education, 2004. 465 p.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, New York, v. 23, n. 1, p. 20–24, mar. 1995.

ZIAKAS, D. et al. Intel QuickPath Interconnect architectural features supporting scalable system architectures. In: SYMPOSIUM ON HIGH PERFORMANCE INTERCONNECTS (HOTI '10), 18., 2010, Mountain View. *Proceedings...* Washington: IEEE Computer Society, 2010. p. 1–6.

Glossário

API – Do inglês *application programming interface*. Protocolo projetado para servir de ligação entre peças de software.

CPU – Do inglês *central processing unit*. Unidade Central de Processamento.

FSB – Do inglês *front-side bus*. Barramento frontal, que liga processador à memória.

HT – Do inglês *Hyper-Transport*. Tecnologia de canal de comunicação entre chips da AMD.

MFLOPS – Do inglês *millions of floating point operations per second*. Milhões de operações de ponto flutuante por segundo.

MIPS – Do inglês *millions of instructions per second*. Milhões de instruções por segundo.

ML – Do inglês *machine learning*. Aprendizado de máquina.

MPI – Do inglês *message passing interface*. Protocolo de passagem de mensagem para paralelização.

NPB – Do inglês *NAS Parallel Benchmarks*. Conjunto de programas de teste de desempenho de computadores.

NUMA – Do inglês *non-uniform memory access*. Acesso não uniforme à memória.

QPI – Do inglês *QuickPath Interconnect*. Tecnologia de canal de comunicação entre chips da Intel.

RAM – Do inglês *random-access memory*. Memória de acesso aleatório.

SSE2 – Do inglês *Streaming SIMD Extensions 2*. Subconjunto de instruções de ponto flutuante da arquitetura de conjunto de instruções x86.

UMA – Do inglês *uniform memory access*. Acesso uniforme à memória.

X87 – Subconjunto de instruções de ponto flutuante da arquitetura de conjunto de instruções x86.