UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ESTUDO EXPLORATÓRIO SOBRE FONTES DE *OS JITTER* NO KERNEL LINUX

ELDER VICENTE DE PAULO SOBRINHO

Uberlândia, Minas Gerais
2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ELDER VICENTE DE PAULO SOBRINHO

ESTUDO EXPLORATÓRIO SOBRE FONTES DE *OS JITTER* NO KERNEL LINUX

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Redes de Computadores

Orientador: Prof. Dr. Rivalino Matias Júnior

Uberlândia, Minas Gerais
2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Estudo Exploratório Sobre Fontes de *OS Jitter* no Kernel Linux**" por **Elder Vicente de Paulo Sobrinho** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 19 de Junho de 2012

Orientador: _____
Prof. Dr. Rivalino Matias Júnior
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Lúcio Borges de Araújo
Universidade Federal de Uberlândia

_____
Prof. Dr. Antônio Augusto Medeiros Fröhlich
Universidade Federal de Santa Catarina

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Junho de 2012

Autor:	**Elder Vicente de Paulo Sobrinho**
Título:	**Estudo Exploratório Sobre Fontes de *OS Jitter* no Kernel Linux**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Aos meus pais, Manoel e Leonídia e às minhas irmãs Erica e Taíse.

À minha amada Juliana Vaz.

Agradecimentos

A Deus,
que sempre está a iluminar os meus passos.

Ao Prof. Rivalino,
meu orientador, pelas oportunidades e desafios que me foram proporcionados, por seu apoio, incentivo e inúmeros conselhos. A você, Rivalino, minha eterna gratidão.

Aos meus pais, Manoel e Leonídia,
pelo amor, apoio e estímulo que incondicionalmente sempre é fornecido. Por compreenderem minha ausência. A vocês dedico todo meu esforço.

À minhas irmãs Erica e Taíse,
por me proporcionar momentos indescritíveis e, principalmente, pela felicidade.

A minha amada Juliana Vaz,
por sempre estar ao meu lado e compreender minha ausência.

Aos amigos de laboratório,
pela convivência intensa, pelas risadas, amizades e conselhos. Em especial, a Ana, ao Hiran, ao Lucas e ao Romerson.

Resumo

O fenômeno *OS Jitter* tem sido considerado um importante fator na computação distribuída de alto desempenho. Neste trabalho, apresentam-se os resultados de um estudo experimental que quantifica os efeitos de diferentes fontes de *OS Jitter* no sistema operacional Linux. Para tanto, utilizou-se o método estatístico DOE para projetar e conduzir experimentos estatisticamente controlados. Diferentemente de estudos anteriores, constatou-se que a topologia do processador, principalmente em relação ao compartilhamento da *cache*, tem influência significativa em termos de *OS Jitter*. Além disso, verificou-se que, a fim de reduzir o impacto do *OS Jitter* em uma dada aplicação, o número de fases computacionais do seu algoritmo é significativamente mais importante do que o número de processos distribuídos ou nós de computação utilizados.

Palavras Chave: *OS Jitter*, Arquitetura de Sistemas Operacionais, Computação de Alto Desempenho

Abstract

The OS Jitter phenomenon has been considered an important factor in high-performance distributed computing. In this thesis, we present the results of an experimental study that quantifies the effects of different sources of OS Jitter in the Linux operating system. We use the DOE method to conduct controlled experiments statistically planned. Differently than previous studies, we found that the processor topology, especially regarding the shared processor cache, has the most significant influence in terms of OS Jitter. Also, we found that in order to reduce the impact of OS Jitter on a given application, the number of computational phases in the algorithm is significantly more important than the number of distributed processes or compute nodes.

Keywords: OS Jitter, Architecture of System Operation, High Performance Computing

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas e Siglas

ACS	Accelerated Critical Sections
DNA	Deoxyribonucleic Acid
APIC	Advanced Programmable Interrupt Controller
ANOVA	Analysis of Variance
AP	Application Processors
API	Application Program Interface
ALU	Arithmetic Logic Unit
AMP	Asymmetric Multiprocessing
ARTiS	Asymmetric Real-Time Scheduler
BIOS	Basic Input/Output System
BSP	Bootstrap Processor
CC-NUMA	Cache Coherent Non Uniform Memory Access
COMA	Cache Only Memory Architecture
CPU	Central Processing Unit
CSV	Comma Separated Values
Cosy	Compound System Calls
CNK	Compute Node Kernel
IPC	Comunicação Interprocessos
DOE	Design of Experiments
PDE	Equações Diferenciais Parciais
FPU	Floating Point Unit
pdf	Função de Densidade de Probabilidade
HÁ	High Availability
HPC	High Performance Computing
ILP	Instruction Level Parallelism
IPI	Interrupção Inter-Processador
ICR	Interrupt Command Register
ICC	Interrupt Controller Communication
IRT	Interrupt Redirection Table
IRQ	Interrupt Request

KB	Kilobyte
K-S	Kolmogorov-Smirnov
LLC	Last Level Cache
LWK	Lightweight kernel
LB	Load Balancing
LAPIC	Local Advanced Programmable Interrupt Controller
LVT	Local Vector Table
VLIW	Very Long Instruction Words
MB	Megabyte
MP	Message Passing
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
NORMA	No Remote Memory Access
NUMA	Non Uniform Memory Access
PA	Parallelism-Aware
PC	Personal Computer
POSIX	Portable Operating System Interface
POST	Power-On Self-Test
PU	Processing Unit
PIT	Programmable Interrupt Timer
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SFD	SF-Driven
SMT	Simultaneous Multithreading
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SBC	Sociedade Brasileira de Computação
SQE	Soma dos Quadrados dos Erros
SQT	Soma dos Quadrados Totais
SMP	Symmetric Multiprocessing
TLP	Thread-Level parallelism

TLB	Translation Address Table
UMA	Uniform Memor Access
UF	Utility Factor

CAPÍTULO 1 – INTRODUÇÃO

1.1 – Contextualização

A evolução científica e tecnológica nas diversas áreas do conhecimento tem exigido, cada vez mais, poder computacional para a execução de modelos complexos usados em pesquisas científicas e em diversas atividades industriais, as quais, na sua maioria, requerem elevado tempo de computação [Buyya 1999].

Na era dos grandes avanços tecnológicos, a Sociedade Brasileira de Computação (SBC) realizou, em 2006, um seminário para discutir os grandes desafios da pesquisa em ciência da computação. Nesse seminário, foram discutidas questões ligadas a "problemas centrais", os quais exigem pesquisas que busquem, não apenas conquistas incrementais, mas que forneçam avanços significativos à ciência. O principal resultado do referido seminário foi produzir um conjunto de cinco grandes desafios para a ciência da computação no Brasil [Leon F. de Carvalho et al. 2006], sendo:

1. Gestão da Informação em grandes volumes de dados multimídia distribuídos;
2. Modelagem computacional de sistemas complexos artificiais, naturais e sócio-culturais e da interação homem-natureza;
3. Impactos para a área da computação da transição do silício para novas tecnologias;
4. Acesso participativo e universal do cidadão brasileiro ao conhecimento;
5. Desenvolvimento tecnológico de qualidade: sistemas disponíveis, corretos, seguros, escaláveis, persistentes e ubíquos.

Dos desafios listados, muitos estão associados a questões tecnológicas em diversos níveis, abrangendo uma infinidade de áreas que vão desde a elaboração de elementos básicos usados na fabricação de componentes eletrônicos, passando pela microarquitetura utilizada no hardware até os mais complexos sistemas distribuídos. Nesse sentido, faz-se necessário propor soluções escaláveis que possam lidar de forma eficiente com o crescente aumento do volume de dados científicos, provenientes das mais diferentes fontes. Isso implica, dentre outras coisas, no desenvolvimento de soluções relacionadas à transmissão, ao armazenamento, ao processamento e à manipulação de dados científicos. Considerando a globalização científica e os desafios propostos pela SBC, depara-se com inúmeras áreas do conhecimento que necessitam de grandes requisitos computacionais, como exemplificados a seguir.

De início, a área de estudos meteorológicos (meteorologia) proporciona a previsão de fenômenos naturais e climáticos e necessita de ambientes computacionais robustos, com

aplicações executando paralelamente e capazes de manipular grandes volumes de dados que são gerados e acessados em tempo real. De forma geral, essas previsões são feitas com base em modelos analíticos e numéricos que representam um determinado domínio, de tal forma que esses domínios possam ser globais ou regionais e podem representar o mundo todo ou apenas uma região, uma cidade, um estado, dentre outros [Osthoff et al. 2011]. Em razão da atual capacidade computacional disponível, os modelos globais têm uma resolução espacial limitada, o que não representa satisfatoriamente os modelos regionais [Osthoff et al. 2011]. Já os domínios regionais têm uma boa resolução, entretanto, são restritos a uma determinada região e necessitam da prévia execução dos modelos globais a fim de conhecer as condições atmosféricas na fronteira do domínio [Osthoff et al. 2011]. Assim, devido à necessidade de grande poder computacional e das limitações de tempo de execução, esses modelos são analisados em múltiplos computadores, em que a carga de trabalho pode ser distribuída.

Outros exemplos podem ser observados em relação às novas necessidades da engenharia, como a busca por eficiência energética, por novos materiais semicondutores, dentre outros. Estas áreas têm levado ao aumento da demanda por novos materiais com características muito particulares. Para projetar os novos materiais, é necessário caracterizar – predizendo as propriedades do novo material (dureza, condutividade, resistência, etc.) e fabricar – e criar formas de produzir este material em laboratório. Ambas as etapas são passíveis de abordagens computacionais, entretanto, não são triviais em termos de algoritmos e custo computacional [Hammond 2011]. Assim, a modelagem de novos materiais que podem ser empregados em diversas áreas está, atualmente, intimamente ligada à computação de alto desempenho que, por meio da execução de modelagens e simulações, fornece elementos necessários ao entendimento da natureza atômica dos materiais.

O campo das ciências biológicas é outro exemplo em que diversas áreas de pesquisa dependem de armazenamento de grandes volumes de dados e da execução de complexos algoritmos que manipulam dados de diversas fontes. Mais especificamente nessa área, observam-se as pesquisas em genética, como, por exemplo, o projeto do Genoma Humano, que tem como principal objetivo identificar cada gene do DNA (*Deoxyribonucleic Acid*) humano. Para compreender o genoma e as bases moleculares de uma determinada doença, uma comparação é realizada entre sequências moleculares normais e patológicas, sendo que a combinação dessas sequências exige um grande poder de processamento [Fosdick 1996]. Há também, pesquisas que visam a descoberta de novas drogas para a produção de remédios e utilizam técnicas de modelagem molecular possibilitando compreender o modo de ação em nível molecular e atômico de uma determinada estrutura.

Outras áreas de pesquisa como a astronomia, física, engenharia mecânica, engenharia elétrica são exemplos que possuem requisitos computacionais de processamento e armazenamento muito semelhantes às descritas anteriormente. De forma geral, a computação científica fornece meios para simular e resolver modelos complexos que representam, com certa exatidão, um dado problema, sendo que, quanto maior a precisão da solução desejada, maiores são os requisitos computacionais.

Nesse contexto, a computação científica é uma vasta área de estudo voltada para o uso intensivo de recursos computacionais de alto desempenho para descobertas científicas. Dentre as disciplinas que compreendem a computação científica, destaca-se a computação de alto desempenho (ou HPC – *High Performance Computing*), fornecendo recursos que possibilitam aos cientistas e engenheiros a resolução de problemas complexos por meio de programas de computador que fazem uso de redes de comunicação de baixa latência e de processamento e armazenamento de dados de alto desempenho [Fosdick 1996]. Mesmo com o grande aumento do poder computacional dos computadores convencionais, o que, segundo Gordon Moore [Moore 2000], dobra a cada dezoito meses, muitas tarefas ainda continuam exigindo tempo considerável de computação. Nesses casos, o conceito de processamento paralelo é fundamental, pois visa dividir uma tarefa em um subconjunto de tarefas menores para que possam ser processadas por diferentes processadores ao mesmo tempo, estando eles fortemente ou fracamente acoplados [Garg e De 2006]. Os aglomerados de processadores, também chamados de *Clusters*, têm sido usados, cada vez mais, para o processamento paralelo/distribuído de alto desempenho, como alternativa ou complemento dos supercomputadores. Em uma arquitetura típica de um *Cluster* HPC é comum existirem centenas ou milhares de nós de computação, em que cada nó executa seu próprio sistema operacional para gerenciar e supervisionar os recursos de hardware e software daquele nó. Segundo dados publicados na lista de supercomputadores do site Top500, que relaciona os 500 maiores supercomputadores do mundo, aproximadamente 450 destes executam um sistema operacional Linux, ou seja, mais de 90% usam esta plataforma.

Em um *Cluster*, o sistema operacional de cada nó possui, além dos processos relativos a programas dos usuários, várias tarefas internas que devem ser executadas com certa regularidade, tais como rotina para sincronização da *cache* de disco, rotinas para manipulação de interrupções, dentre outras. Desta forma, observa-se que, durante o período de execução de um processo do usuário, este processo sofre, por diversas vezes, a interferência dessas tarefas internas do sistema operacional. Tais interferências têm sido investigadas e são chamadas de "*OS Jitter"* (ex. [De et al. 2007], [Mann e Mittaly 2009] e [IBM 2010]) ou "*OS Noise*" (ex.

[De et al. 2007], [Gioiosa et al. 2004], [Agarwal et al. 2005], [Tsafrir et al. 2005], [Garg e De 2006] e [Beckman et al. 2006]). Neste trabalho, o termo *OS Jitter* será adotado para representar a interferência que o sistema operacional causa na execução das aplicações.

Recentemente, o fenômeno do *OS Jitter* tem sido considerado um dos principais fatores causadores de atrasos no tempo total de processamento em ambientes de *Cluster* HPC, além de fatores já conhecidos como latência da rede de interconexão, latência dos dispositivos de I/O, configuração de nós não homogêneos, dentre outros. Estudos [Jones et al. 2003], [Gioiosa et al. 2004], [Agarwal et al. 2005], [Tsafrir et al. 2005], [Garg e De 2006] reportam que os efeitos do *OS Jitter* se agravam à medida que o tamanho do *Cluster* aumenta em número de nós, de tal forma que, isso pode limitar a eficiência computacional. Portanto, considera-se importante o estudo e o desenvolvimento de métodos/técnicas para reduzir os efeitos do *OS Jitter*, especialmente em ambientes de HPC.

1.2 – Relevância do Trabalho

O surgimento de interconexões de redes com baixa latência, padronização da computação paralela e distribuída e a criação de processadores com alto poder computacional também são alguns dos fatores que têm impulsionado o crescimento da computação em ambientes baseados em *Clusters*.

Atualmente, sistemas HPC são formados por centenas ou milhares de nós que, com o advento dos sistemas multiprocessados, chegam a alcançar dezenas ou centenas de milhares de processadores, o que significa um poder computacional equivalente a de uma elevada quantidade de operações de ponto flutuante por segundo (ex. Peta Flops) [Top 500 2011]. Ao longo do texto, os termos processadores e núcleo são usados como sinônimos.

Como mencionado anteriormente, as aplicações HPC geralmente usam o modelo de programação paralela baseado em *bulk-synchronous* [Tsafrir et al. 2005], onde cada processo é composto por fases computacionais separadas por barreiras (operações de sincronização coletivas). Assim, pode-se observar que os processos mais rápidos devem esperar os processos retardatários que, de alguma forma, sofreram algum tipo de atraso (ex. *OS Jitter*) e, finalmente, um novo ciclo possa ser iniciado.

Para exemplificar o contexto anteriormente especificado, observa-se a situação em uma simulação climática. Para que o próximo ciclo seja executado, é preciso realizar operações de sincronização para encontrar a maior pressão em uma matriz que está distribuída em diversos nós, mas a pressão máxima não pode ser determinada até que todos os nós tenham finalizado o cálculo do valor da pressão. As barreiras podem comprometer a

escalabilidade e a eficiência computacional introduzindo uma limitação no sistema, visto que, um processo retardatário irá propagar o atraso e impedir o progresso das próximas fases computacionais. Por sua vez, a granularidade (tempo de duração das fases computacionais) varia conforme a implementação, mas, geralmente, ela está na ordem de alguns milissegundos.

Em termos de *OS Jitter*, o melhor desempenho possível em um *Cluster* HPC é quando, teoricamente, todos os nós iniciam e terminam cada ciclo no mesmo instante, de forma que o processador não fique ocioso (ver Figura 1.1b) [Agarwal et al. 2005]. Entretanto, essa situação é improvável no mundo real (ver Figura 1.1a), pois pode haver situações em que os nós não são homogêneos (hardware e software) e, mesmo em ambientes completamente homogêneos, ainda existem fatores que levam a atrasos que não são determinísticos [Agarwal et al. 2005].

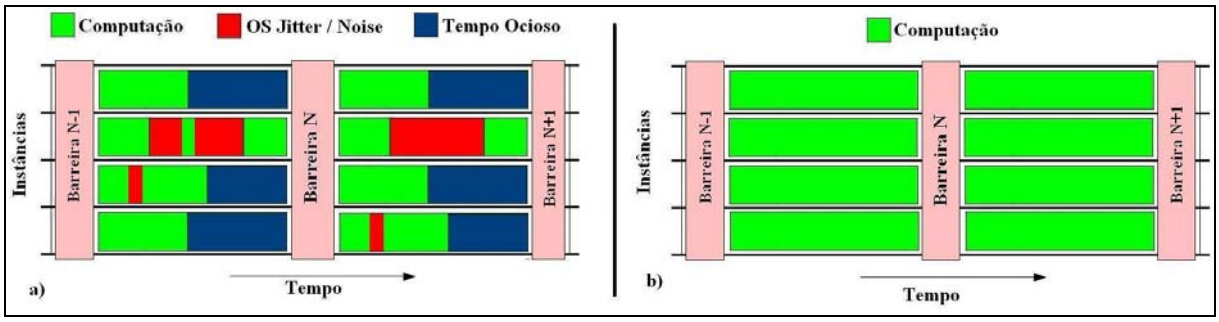


Figura 1.1 Programação paralela *bulk-synchronous*. Adaptado de [Tsafrir et al. 2005]

Existem ainda limitações quanto ao nível de paralelismo possível de ser atingido pelos programas. Nesse sentido, sabe-se que determinadas características do programa limitam o seu desempenho, de forma que, essas características estão relacionadas ao fato de não ser possível desenvolver um código totalmente paralelo [Carvalho 2001].

Assim, dentre as limitações existentes no mundo real, descritas anteriormente, todas são, até certo ponto, conhecidas e estão sendo investigadas por pesquisadores (ex. [Carter et al. 2007], [Hwang et al. 2000], [Fromm e Treuhaft 1996]). Diversas propostas têm sido apresentadas para reduzir a influência desses fatores que provocam atrasos. Neste trabalho, o foco está somente nos atrasos relativos ao sistema operacional, conhecidos como *OS Jitter*.

De forma geral, atividades relativas ao gerenciamento do núcleo (*kernel*) do sistema operacional podem introduzir atrasos durante a fase de computação dos programas do usuário. Estudos anteriores (ex. [Tsafrir et al. 2005], [De et al. 2007], [Mann e Mittaly 2009]) discutem os diversos tipos de interferências (*OS Jitter*) do sistema operacional na execução de processos no espaço do usuário, em especial, para processos de HPC. Dentre essas

interferências, destaca-se a execução de processos administrativos e rotinas periódicas do *kernel*, como, por exemplo, a interrupção de *timer*. Segundo [Gioiosa et al. 2010], a média de degradação do desempenho proveniente do *OS Jitter*, em um único nó, está entre 1% e 2%, o que, em um ambiente de *Cluster* HPC, com centenas de nós, torna-se uma limitação, restringindo a escalabilidade e degradando, de forma mais acentuada, o desempenho global do sistema.

A contribuição da presente pesquisa para a redução do *OS Jitter* prioriza esses dois problemas (processos administrativos e rotinas periódicas do *kernel*). A plataforma onde este trabalho foi realizado é baseada em um sistema operacional de propósito geral, nesse caso, o Linux. A escolha deste sistema operacional deve-se, principalmente, à disponibilidade do seu código fonte e por ser, atualmente, o sistema operacional mais usado em *Clusters* HPC, executando em mais de 90% dos computadores/*Clusters* apresentados na lista dos principais supercomputadores do mundo [Top 500 2011].

1.3 – Objetivos da Pesquisa

1.3.1 – Objetivo Geral

Realizar um estudo exploratório quantitativo para identificação das principais fontes de *OS Jitter* no *kernel* do Linux.

1.3.2 – Objetivos Específicos

- Identificar experimentalmente os principais fatores do Linux que são causadores de *OS Jitter* no contexto de aplicações de HPC.
- Avaliar quantitativamente os efeitos do *OS Jitter* do Linux.
- Através de simulação, estimar o impacto do *OS Jitter* do Linux sobre uma aplicação de HPC, baseada em múltiplas fases de computação, executando em múltiplos nós de computação de um *Cluster*.

1.4 – Desenvolvimento da Pesquisa

Neste ponto, é apresentada uma visão geral das principais atividades desempenhadas durante a realização dessa pesquisa de dissertação.

1.4.1 – Revisão da literatura

Nesta etapa, foi realizada uma pesquisa bibliográfica englobando conceitos necessários para o entendimento do problema investigado. As principais fontes de consulta envolveram livros clássicos na área de sistemas operacionais, bem como manuais fornecidos

por fabricantes de processadores e artigos científicos obtidos em anais de conferências e periódicos nacionais e internacionais.

Inicialmente, realizou-se um estudo teórico visando o entendimento de aspectos fundamentais dos principais componentes envolvidos na pesquisa. A Seção 2.2 (Aspectos Arquiteturais) detalha recursos e características que, atualmente, são incluídas no projeto e fabricação de processadores, apresentando, para isso, a Lei de Amdahl e a taxionomia de Flynn que, por sua vez, agrupa os equipamentos conforme características comuns. A Subseção 2.2.1 (Processamento SMP e AMP) trata de aspectos relativos à forma como ocorre o processamento em sistemas multiprocessados (SMP – *symmetric multiprocessing* e AMP – *asymmetric multiprocessing*). A seção 2.3 (Computação em *Cluster*) está relacionada ao ambiente de execução e programação paralela (ex. MPI – *message passing interface*) tipicamente usados em *Clusters* de HPC.

Por sua vez, na Seção 3.2 (Tratamento de Interrupções) são apresentados conceitos relativos ao projeto e implementação de sistemas operacionais, tais como o Linux, em plataforma x86. Em seguida, na Subseção 3.2.1 (*Advanced Programmable Interrupt Controller* – APIC), são abordadas questões relativas à estrutura física de projetos de processadores multiprocessados, neste caso, com foco nas interrupções de *timer*. A Seção 3.3 (Trabalhos Correlatos) apresenta diversos trabalhos encontrados na literatura que estão relacionados à problemática do *OS Jitter*. A Subseção 3.3.1 (Interferência dos Modos de Operação) expõe a necessidade de execução das chamadas de sistema no modo privilegiado (*kernel mode*). Logo após, em 3.3.2 (Interferência de Cache) são apresentados estudos relativos à sobreposição e à falta de dados na memória *cache.* Por fim, a subseção 3.3.3 (Interferência do Ambiente) apresenta diversos estudos na área de *OS Jitter*.

1.4.2 – Estudo Experimental

Segundo [Wazlawick 2009], uma caracterização clássica das formas de pesquisa consiste na classificação da pesquisa experimental e não experimental. A pesquisa não experimental consiste no estudo de fenômenos sem a intervenção sistemática do pesquisador, ou seja, o pesquisador atua apenas observando e tirando conclusões a partir de um arcabouço teórico preconcebido. Para exemplificar esse tipo de pesquisa, pode ser citado um estudo diário em empresa de desenvolvimento de software cujo objetivo é detectar determinadas práticas que previamente foram catalogadas. Já a pesquisa experimental condiciona o pesquisador a provocar mudanças sistemáticas no meio alvo da pesquisa para observar se cada intervenção produz os resultados esperados.

O presente estudo classifica-se como uma pesquisa experimental exploratória, em que experimentos controlados foram realizados para avaliar e quantificar os efeitos do *OS Jitter* de sistemas Linux, realizando uma comparação entre o modelo atual de processamento SMP e o modelo de processamento AMP, em conjunto com uma abordagem de processamento *tickless*.

Para este estudo, utilizou-se a técnica de experimentos estatisticamente controlados (DOE – *Design of Experiments*) para o planejamento dos experimentos [Montgomery 2005]. Essa técnica fornece elementos para encontrar e interpretar as influências de uma variável e/ou fator sobre uma variável resposta em um objeto de estudo. Assim, a presente pesquisa investiga, de forma conjunta, diversas fontes de *OS Jitter* quantificando-as para apontar aquelas fontes com maior contribuição para a variação no tempo de resposta de uma aplicação HPC de prova. Para tanto, foram aplicados diferentes métodos de análise de dados (ex. Kruskal-Wallis e Kolmogorov-Smirnov) para fornecer avaliações estatisticamente confiáveis sobre os resultados [Campos 1979].

1.5 – Escopo da Pesquisa

Nesse trabalho, são estudados seis fatores, sendo cada fator analisado em dois níveis. Os fatores considerados são:

1) *runlevel* do sistema operacional, para seus respectivos valores de 1 e 5;
2) ocorrência ou não de *timers* de software do sistema operacional no processador;
3) ocorrência ou não de pedidos de interrupção (IRQs) no processador;
4) compartilhamento ou não do processador (*processor affinity*);
5) ocorrência ou não da interrupção de *timer* no processador;
6) compartilhamento ou não de *cache* de processador com uma dada carga de fundo.

Para a avaliação dos fatores mencionados anteriormente, foram realizados dois experimentos. O Experimento #1 é composto pelos cinco primeiros fatores e o Experimento #2 introduz o sexto fator. Vale ressaltar que todos os experimentos foram planejados e realizados seguindo a abordagem DOE [Montgomery 2005]. Com os resultados dos experimentos, por meio de simulação, estimamos a influência do *OS Jitter* em um ambiente de *Cluster* com múltiplos nós e diversas fases.

Muitos dos fatores investigados nesta pesquisa já foram individualmente ou coletivamente estudados em outros trabalhos. Contudo, após extensa revisão bibliográfica, não foi encontrado um estudo que adotou métodos rigorosos de análise de dados, coletados por meio de instrumentação de *kernel* e implementando múltiplas replicações, para avaliar de forma mais precisa a contribuição, individual ou por meio de interações, dos fatores

investigados neste trabalho para o *OS Jitter* de um sistema operacional. Outro ponto importante a destacar-se é que, nos trabalhos anteriores, muitos testes foram realizados com sistemas operacionais, ou versões do *kernel* do Linux, que não incorporavam diversos aspectos arquiteturais que são implementados nas versões mais recentes destes sistemas. Por exemplo, escalonamento *tickless* e regulação automática da voltagem do processador são características que influenciam significativamente no *OS Jitter*. Também, nos trabalhos experimentais encontrados na literatura, muitos resultados não levam em consideração o erro experimental que pode, muitas vezes, introduzir vieses na análise dos resultados. Estes e outros aspectos são considerados no presente estudo.

Esse estudo está estruturado nos capítulos especificados a seguir: O Capítulo 2 aborda conceitos e técnicas da computação de alto desempenho. Logo após, o Capítulo 3 descreve e relaciona a problemática do *OS Jitter* e apresenta o "estado da arte" neste campo de pesquisa. Em seguida, o Capítulo 4 trata dos métodos usados na análise e do projeto dos experimentos deste trabalho. Já o Capítulo 5 apresenta um estudo com a realização de experimentos controlados para avaliar e quantificar os efeitos do *OS Jitter* em um sistema operacional Linux. Posteriormente, o Capítulo 6 apresenta a análise dos resultados. Por fim, no Capítulo 7, são apresentadas as conclusões e perspectivas para trabalhos futuros.

CAPÍTULO 2 – COMPUTAÇÃO DE ALTO DESEMPENHO

2.1 – Introdução

Os *Clusters* podem ser classificados em: alta disponibilidade (*High Availability* – HA) [Sloan 2004], balanceamento de carga (*Load Balancing* – LB) [Sloan 2004] e processamento distribuído ou paralelo, também conhecidos como *Clusters* de alto desempenho (*High Performance Computing* – HPC) [Prabhu 2008], sendo este último o foco do presente trabalho. Ambientes de *Clusters* HPC são criados para permitir a execução paralela de programas, fornecendo um aumento do poder computacional que não seria possível usando um único computador.

Ao longo dos últimos anos, a capacidade computacional dos nós de computação, que compõem as estruturas de um *Cluster*, tem aumentado consideravelmente [Top 500 2011]. Isso se deve, em grande parte, ao surgimento de novos materiais semicondutores proporcionando a capacidade de desenvolvimento de processadores e arquiteturas que procuram explorar e tirar o melhor proveito de certas características dos programas. Recentemente, estas novas tecnologias têm explorado principalmente a questão do paralelismo, que pode ser observado tanto em nível de instruções quanto em nível de software [Culler et al. 1999]. O paralelismo de instruções ou ILP (*instruction level parallelism*) está ligado à forma pela qual a microarquitetura do processador busca e manipula os dados/instruções. Já o paralelismo em nível de software é atingido por meio da divisão de uma tarefa em múltiplos fluxos de execução que executam simultaneamente em múltiplos processadores e/ou núcleos que, por sua vez, podem estar agrupados em um mesmo *chip* e/ou vários *chips* e/ou, ainda, por meio de diversos computadores interligados por uma rede de interconexão. Neste caso, tais processadores podem ou não ser idênticos [Lastovetsky 2003].

As diversas arquiteturas multiprocessadas podem ser divididas de acordo com a organização do compartilhamento de hardware, sendo duas categorias principais [Pitanga 2008]: 1) fortemente acoplada, em que todos os processadores podem acessar todos os recursos (ex. memória, dispositivos, etc.) como, por exemplo, em sistemas SMP e AMP; 2) fracamente acoplados, em que um sistema é construído interconectando sistemas independentes, cada um com seus próprios recursos (ex. processador, memória, dispositivos de entrada e saída, etc.) que, por sua vez, são gerenciados pelo seu próprio sistema operacional como é o caso dos *Clusters*. Nessa segunda estrutura, cada sistema é independente e é utilizado como nó de computação.

Uma arquitetura multiprocessada também pode ser classificada quanto às características de processamento de cada processador, podendo ser simétrico (*symmetric)* ou assimétrico (*asymmetric*) [Padua 2011]. Em um sistema com processamento simétrico, todos os processadores possuem a mesma função, ou seja, qualquer processador pode executar qualquer tarefa do sistema. De forma oposta, devido às características específicas da microarquitetura, no processamento assimétrico existem processadores com certo nível de especialização; por exemplo, um processador é dedicado a determinados tipos de tarefas (ex. CPU-*bound*), enquanto que o outro se dedica a tarefas diferentes (ex. *memory-bound*).

Atualmente, com o intuito de melhorar o desempenho dos processadores, os fabricantes, além do paralelismo referente a processos (que é atingido com arquiteturas multiprocessadas), também introduzem o paralelismo de instruções (ILP), alcançado por meio da implementação de tecnologias/recursos na microarquitetura que, por sua vez, são preparadas para manipular e executar paralelamente múltiplos dados e instruções [González et al. 2010]. Para exemplificar estas tecnologias, tem-se: *pipeline*, *pipeline superscalar*, *very long instruction words* (VLIW), *out-of-order execution*.

2.2 – Aspectos Arquiteturais

Nas últimas décadas, o poder computacional alcançado pelos processadores tem crescido de forma relevante o que se justifica, essencialmente, ao progresso tecnológico que pemitiu aos fabricantes de processadores incorporarem, em seus projetos, recursos como *pipeline*, *pipeline superscalar*, *very long instruction words* (VLIW) e *out-of-order execution*.

O princípio envolvendo *pipeline* [Chevance 2005] é baseado na premissa de que o trabalho necessário para executar uma única instrução, que pode gastar vários ciclos de CPU, é dividido em certo número de etapas mais simples. Cada etapa é executada por uma unidade de processamento (unidade funcional) independente, de forma que, essas unidades são colocadas uma após a outra e, ao final da etapa, os dados são encaminhados para a etapa subsequente até a conclusão de todas elas [Chevance 2005]. Vale ressaltar que as etapas subsequentes podem ser executadas apenas quando a etapa anterior for concluída. Como exemplo, a Figura 2.1 representa um *pipeline* simples de quatro etapas (*instruction fetch*, *decode*, *execution*, *write results*). No instante inicial, apenas uma etapa do trabalho é realizada, mas, à medida que o *pipeline* é preenchido, outras etapas vão sendo executadas simultaneamente até o momento em que, para cada ciclo, ocorre a execução completa de uma instrução. Na prática, a engenharia necessária para projetar um *pipeline* não é tão simples, pois podem ocorrer conflitos entre dados, resultando em um tempo maior do que um ciclo de

CPU para execução de uma única instrução em cada etapa. Além disso, a dependência de dados e os desvios condicionais podem atrasar o *pipeline*, fazendo-o aguardar um resultado ou avaliação de uma expressão condicional. Por fim, apresenta-se o problema do aumento da complexidade dos projetos de hardware, o que pode elevar, de forma significativa, o custo de produção dos processadores.

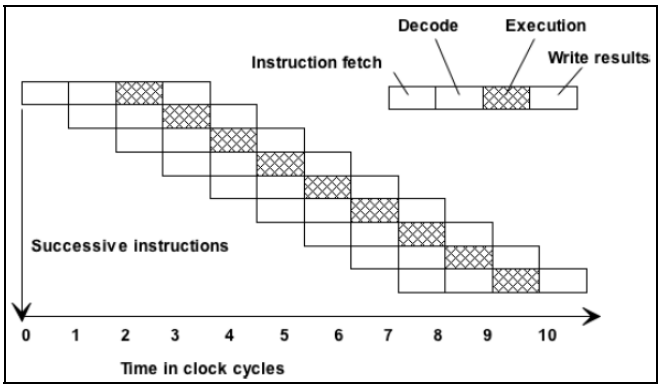


Figura 2.1 Representação de um *pipeline* simples [Chevance 2005]

O recurso *superscalar* ou *pipeline superscalar* é a capacidade do *pipeline* de executar mais de uma etapa simultaneamente. A Figura 2.2 representa um *pipeline superscalar* de quatro etapas, executando duas etapas por vez, ou seja, em cada ciclo de CPU há duas etapas sendo executadas [Chevance 2005].

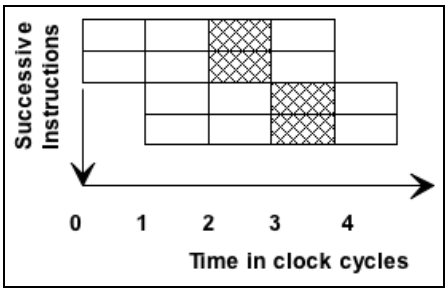


Figura 2.2 Representação *superscalar* [Chevance 2005]

Nas arquiteturas que implementam o recurso *very long instruction words* (VLIW) [Chevance 2005], cada instrução é composta por diversas operações primitivas que podem ser executadas de forma concorrente. Considerando que, em uma arquitetura tradicional RISC (*Reduced instruction set computer*), uma operação primitiva é representada por 32 bits, e que, em uma arquitetura VLIW uma operação primitiva pode ser descrita, por exemplo, com 128 bits, então, tem-se uma simultaneidade de quatro operações por vez. Entretanto, para explorar esta funcionalidade, é necessário que o compilador identifique, no código fonte, situações em que as instruções VLIW possam ser usadas para que, assim, o arquivo binário seja produzido

de forma a empregar essas instruções [Chevance 2005]. A Figura 2.3 mostra um *pipeline* de quatro estágios, em que a etapa *execution* pode realizar operações equivalentes a três operações primitivas (VLIW).

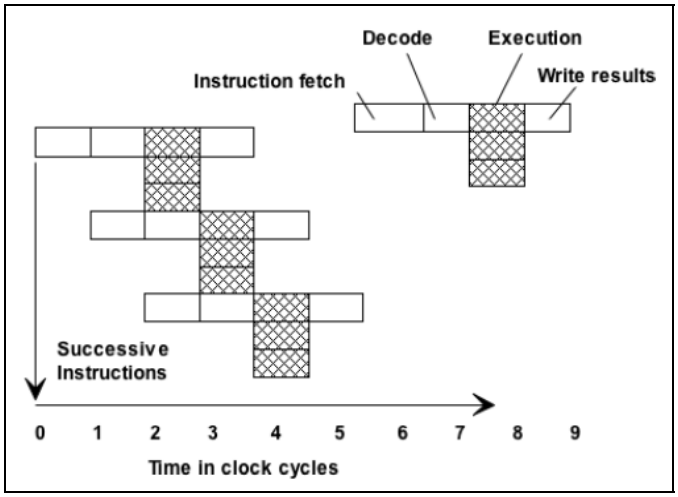


Figura 2.3 Representação VLIW [Chevance 2005]

Os processadores modernos contêm várias unidades funcionais e de diferentes tipos (ex. ALU – *arithmetic logic unit*, FPU – *floating point unit*). O recurso *out-of-order execution* [Chevance 2005] é um mecanismo que, por meio da verificação de dependência e conflito de instruções, permite a execução das instruções fora de ordem sem seguir a estrutura original do programa, fazendo uso simultâneo de múltiplas unidades funcionais de propósito geral (ex. ALU) e de propósito específico (ex. FPU).

Um computador paralelo pode ser fabricado utilizando-se vários processadores e/ou núcleos, de forma que, estes estão equipados com diversas tecnologias que exploram o paralelismo de instruções (ILP). Conforme a arquitetura empregada, o desempenho pode variar, já que, também está relacionada a outros fatores como vazão e latência da rede de interconexão, forma com que o código fonte da aplicação foi escrito, qualidade das otimizações realizadas pelo compilador, frequência do processador, velocidade de acesso à memória, capacidade e velocidade da memória *cache*, dentre outros [Mattson et al. 2005].

Com relação ao software, o paralelismo classifica-se como implícito ou explícito [Hwang 2010]. O paralelismo explícito é obtido quando o desenvolvedor projeta e escreve o código fonte informando "quando" e "como" paralelizar as suas operações. Já o paralelismo implícito não possui, no código fonte, as informações explícitas sobre a execução paralela. Nesse caso, o compilador gera um binário, obtendo certo nível de paralelismo microarquitetural, valendo-se de recursos existentes na microarquitetura como *pipeline, out-of-order execution*, *branch prediction*, etc. Recentemente, a combinação destes dois níveis de

paralelismo (implícito e explícito) tem sido uma prática cada vez mais utilizada [Rauber e Rünger 2010].

Uma questão importante em paralelismo computacional é como mensurar o desempenho alcançado por um programa. Determinadas características do programa limitam o desempenho e, essas características, estão relacionadas com o fato de não ser possível desenvolver um código totalmente paralelo, já que, certas operações são puramente sequenciais. Assim, o tempo de execução dessas operações sequenciais é independente da quantidade de processadores [Carvalho 2001]. Neste sentido, Gene Amdahl observou, em 1967, que essa relação entre a parte sequencial e a parte paralela do código limita o desempenho dos programas [Chevance 2005]. Amdahl identificou que o aumento no desempenho máximo (*speedup* máximo) está restrito à parte sequencial do código. Em seus estudos, Amdahl sugere que o *speedup* máximo relativo à execução de determinado programa, usando p processadores, é definido como a razão entre o tempo gasto para execução com um único processador e o tempo gasto para execução com p processadores. Assim, considerando a fração do tempo de determinado programa com p processadores sendo $T(p)$, e $T(1)$ a fração do tempo gasto com apenas um processador, o *speedup* máximo pode ser descrito matematicamente como [Chevance 2005]:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.1}$$

Considerando p processadores e um código com partes sequenciais e paralelas, tem-se [Chevance 2005]:

$$S(p) = \frac{1}{T(s) + \frac{T(p)}{p}} \tag{2.2}$$

onde $T(s)$ é a fração de tempo gasto nas partes sequenciais, $T(p)$ é a fração do tempo gasto nos pontos paralelos e p é a quantidade total de processadores.

A Equação 2.2 é conhecida como Lei de Amdahl e expressa as limitações para o aumento do *speedup* máximo de um programa. Consequentemente, para um programa com 20% do código sequencial usando 2 processadores, o *speedup* máximo é igual a 1,6666, apresentando, portanto, um ganho máximo de 66,66%.

Em uma situação em que a quantidade de processadores tende ao infinito, o *speedup* máximo que pode ser obtido depende da porção do código que não é executada paralelamente, ou seja:

$$S(p) = \frac{1}{T(s)} \tag{2.3}$$

A Lei de Amdahl convenciona diversos pressupostos que não se aplicam no mundo real. Ela não considera o custo de operações de comunicação e/ou sincronização que são inseridas em uma aplicação paralela, assim como os efeitos do *OS Jitter* estudados neste trabalho. Além disso, sua expressão considera o tamanho do problema constante independente da quantidade de processadores e, por fim, assume que as partes sequenciais e paralelas possuem peso invariante [Carvalho 2001]. Em razão dessas simplificações, John Gustafson, em 1988, considerou que, para utilizar mais processadores, é preciso aumentar o tamanho do problema executado em paralelo e, para isso, ele apresentou a seguinte fórmula para calcular o *speedup* máximo [Carvalho 2001]:

$$S(p) = T(s) + (1 - T(s)) \times p \tag{2.4}$$

Pela Equação 2.4, quando *T(s)* tende a zero, o *speedup* máximo é igual à quantidade de processadores, de forma que, para aplicações de grandes dimensões onde a parte sequencial, *T(s)*, é pequena, a Equação 2.4 mostra-se mais realista.

No contexto do processamento paralelo, um importante conceito é o de granularidade [Padua 2011], referindo-se ao tamanho das seções do código passíveis de serem executadas em paralelo. Neste caso, considera-se que o paralelismo do programa pode ser de granularidade fina ou grossa. Na granularidade fina, as seções paralelas são representadas apenas por um pequeno número de instruções. Já a granularidade grossa é considerada quando o tamanho das seções paralelizáveis exibe um grande número de instruções. Juntamente com o conceito de granularidade, também surgiram definições que agrupam os equipamentos conforme suas características comuns. Uma delas foi proposta em 1972, por Michael Flynn, que classificou as arquiteturas de computadores com base na quantidade de fluxos de dados e instruções existentes em um dado instante, produzindo quatro categorias [Mattson et al. 2005]: 1) *Single Instruction Single Data* (SISD); 2) *Single Instruction Multiple Data* (SIMD); 3) *Multiple Instruction Single Data* (MISD); 4) *Multiple Instruction Multiple Data* (MIMD). Essa classificação ficou conhecida como taxionomia de Flynn.

Um sistema SISD é um computador convencional de *Von Neumann* com único processador que executa instruções, sequencialmente, na ordem em que são repassadas pela unidade de controle. Apenas uma instrução é executada por vez, pois há apenas um fluxo de dados e instruções [Mattson et al. 2005]. A Figura 2.4 representa um típico sistema SISD.

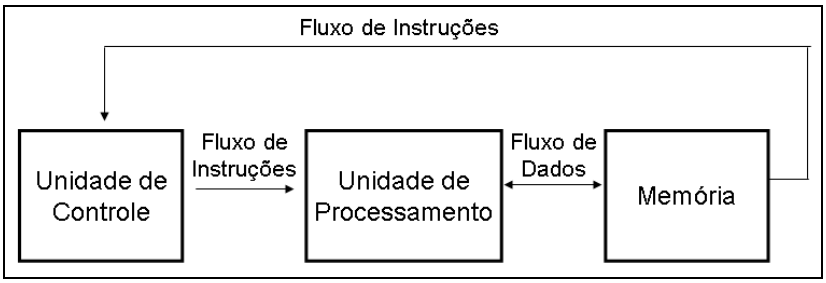


Figura 2.4 Arquitetura *single instruction single data* (SISD)

Já em sistemas SIMD, uma mesma instrução é processada em diversos fluxos de dados (ver Figura 2.5), sendo equivalente ao paralelismo de dados em que uma única instrução é executada paralelamente usando múltiplas cópias do dado [Mattson et al. 2005]. Tais sistemas possuem várias unidades de processamento gerenciadas por uma única unidade de controle. Essa estrutura é muito comum em problemas que envolvem multiplicação de matrizes e manipulação de imagens e, geralmente, usada em processadores matriciais e vetoriais [Pitanga 2008].

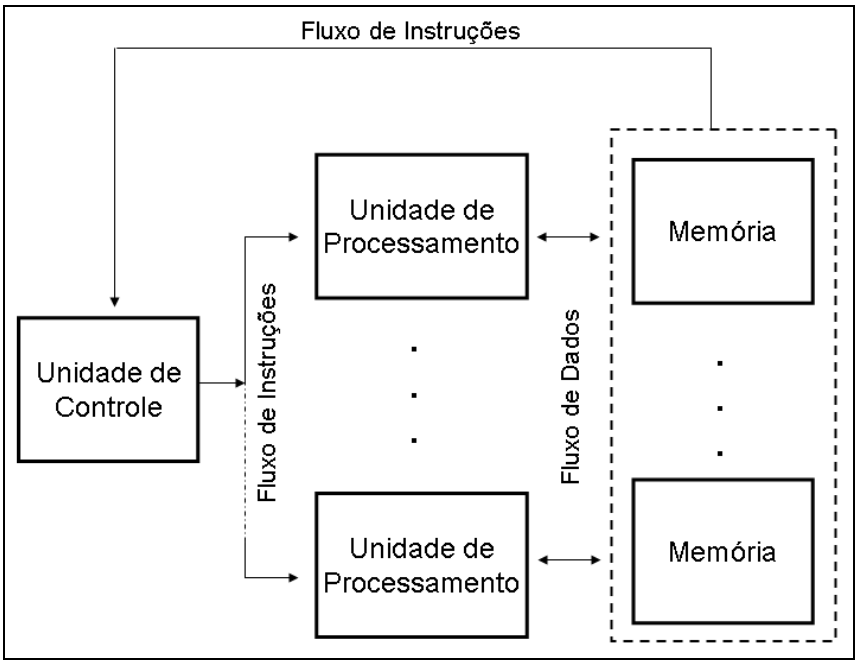


Figura 2.5 Arquitetura *single instruction multiple data* (SIMD)

Nos sistemas MISD, diversas instruções são executadas em um mesmo fluxo de dados, ou seja, múltiplos processadores executando diferentes instruções sobre um único conjunto de dados. O fluxo de dados percorre todas as unidades de processamento, considerando que o resultado oriundo da unidade de processamento anterior seja a entrada da próxima unidade. Ainda, segundo [Pitanga 2008], não há nenhuma arquitetura classificada como puramente MISD.

O modelo usado em sistemas MIMD (ver Figura 2.6) é aquele no qual cada processador atua de forma independente e paralelamente com múltiplos fluxos de dados e instruções. Atualmente, é a arquitetura mais difundida no mercado de computadores, também conhecida como arquitetura multiprogramada [Pitanga 2008].

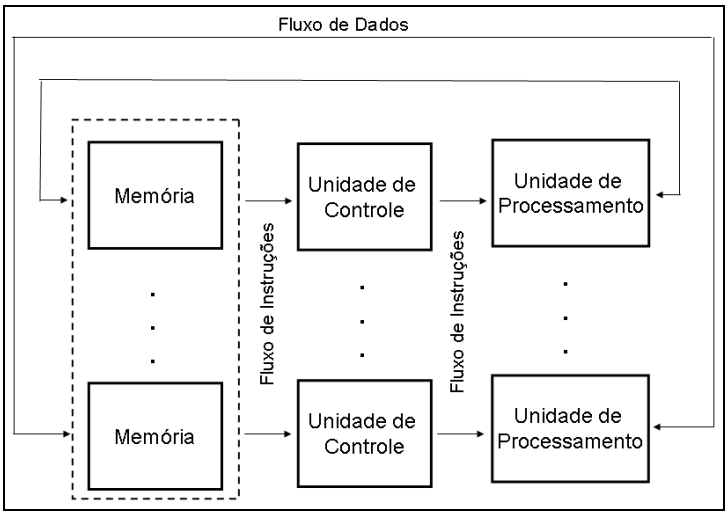


Figura 2.6 Arquitetura *multiple instruction multiple data* (MIMD)

Dentre os modelos descritos por Flynn, atualmente, grande parte dos processadores paralelos são MIMD [Tanenbaum 2005]. A categoria MIMD pode ser dividida em duas subclasses: multiprocessadores de memória compartilhada (*shared-memory multiprocessors*) e multicomputadores (*message passing multicomputers*) [Pitanga 2008]. A grande diferença entre essas subclasses é a forma de acesso à memória e a forma pela qual ocorre a comunicação entre os processos. Na subclasse *shared-memory*, também representada pelas arquiteturas UMA (*uniform memor access*), COMA (*cache only memory architecture*), NUMA (*non uniform memory access*), nCC-NUMA (*non cache coherent non uniform memory access*) e CC-NUMA (*cache coherent non uniform memory access*), a comunicação ocorre por meio do compartilhamento de dados que estão em um mesmo espaço de memória, ou seja, a memória global que é compartilhada entre todos. Na subclasse *message passing,* conhecida como arquitetura NORMA (*no remote memory access*), cada nó tem sua própria memória sendo gerenciada pelo seu sistema operacional, de forma que, não há um compartilhamento global de memória e a comunicação acontece através de troca de mensagens [Pitanga 2008]. Assim, a memória pertencente a outro processador é considerada uma memória remota, pois está localizada em um espaço de endereçamento distinto, ou seja, não há como usar dados compartilhados na memória para efetuar a comunicação entre processos, já que, é necessário enviar e receber mensagens pela rede de interconexão [Pitanga 2008]. A Figura 2.7 apresenta um resumo com as diversas ramificações da arquitetura MIMD.

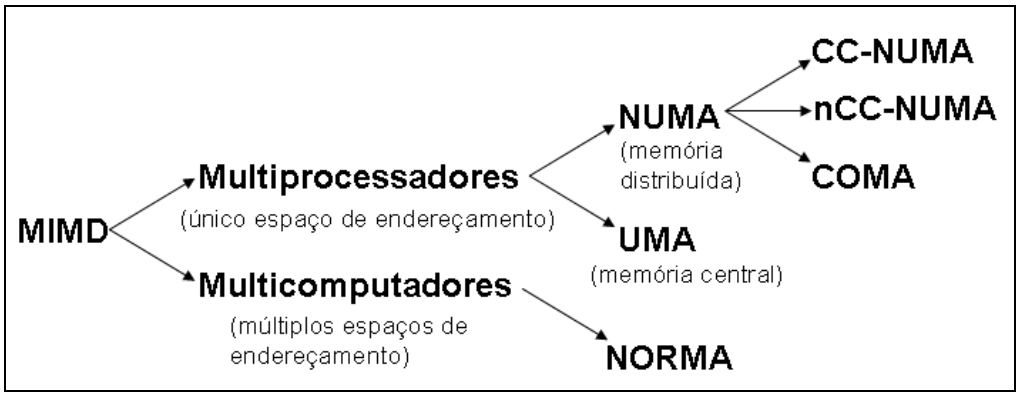


Figura 2.7 Subcategorias da arquitetura MIMD [Pitanga 2008]

A principal vantagem da arquitetura de multiprocessadores é a forma de compartilhamento de dados entre processos que é muito rápida [Pitanga 2008]. Porém, os computadores que a utilizam são caros e possuem certo limite físico para a quantidade de processadores, limitando a escalabilidade destes sistemas. Neste sentido, a arquitetura de multicomputadores é mais escalável, porém, a programação se mostra mais complexa devido à necessidade dos processos trocarem mensagens de forma explícita. Além disso, dependendo da quantidade de mensagens enviadas e recebidas, o desempenho pode ser comprometido. Vale ressaltar ainda que, as arquiteturas fortemente acopladas, também referidas como multiprocessadas, embora aumentem a capacidade de processamento e/ou escalabilidade (aumentando a quantidade de processadores), também sofrem com algumas limitações físicas do hardware que estão intimamente ligadas à capacidade de manter constantes as baixas latências e altas taxas da largura de banda [Tanenbaum 2005].

A memória, em grandes multiprocessadores, costuma ser subdividida em diversos módulos e, conforme a implementação do compartilhamento da memória, tais sistemas podem ser classificados em UMA, NUMA, COMA e CC–NUMA. Os sistemas UMA [Tanenbaum 2005] têm como principal característica o fato de que cada processador atinge o mesmo tempo de acesso a todos os módulos de memória, ou seja, cada palavra de memória pode ser lida tão rapidamente quanto qualquer outra. Isso facilita a implementação de programas, pois o desenvolvedor não precisa preocupar-se com o princípio da localidade de referência dos dados [Pitanga 2008]. Em uma arquitetura UMA simples, os processadores são ligados aos módulos de memória por intermédio de um único barramento. Quando o processador precisa ler algum dado da memória, inicialmente, ele verifica se o barramento está ocupado, caso positivo, o processador fica aguardando até sua liberação. Para um pequeno número de processadores, a contenção é aceitável, mas para uma centena de processadores torna-se inviável. Assim, o sistema fica limitado pela largura de banda do barramento e a maioria dos

processadores ficará ociosa durante grande parte do tempo. Devido ao aumento do tempo de acesso à memória, à medida que o número de processadores aumenta, as arquiteturas UMA são limitadas pela quantidade de processadores [Tanenbaum 2005].

De forma contrária aos sistemas UMA, os sistemas NUMA não possuem tempo de acesso aos módulos de memória uniforme para todos os processadores. Geralmente, há um módulo de memória próximo de cada processador e o acesso a este módulo é mais rápido do que em módulos distantes. Todos os programas UMA podem ser executados em máquinas NUMA, mas, devido ao tempo de acesso à memória, o seu desempenho será menor do que em uma máquina UMA de mesma capacidade de processamento [Tanenbaum 2005].

O fato das aplicações necessitarem de acesso à memória remota toda vez que um dado de memória não local for acessado tem um grande impacto no desempenho, por isso, é importante que cada processador tenha memória *cache* [Tanenbaum 2005]. Tal fato implica em adicionar coerência de *cache*, que pode ser feito pelo monitoramento do barramento [Tanenbaum 2005]. Essa abordagem, na qual estão presentes *caches* coerentes, é conhecida como CC–NUMA. Uma abordagem bastante utilizada em sistemas multiprocessados de grande porte é o multiprocessador baseado em diretório. A idéia básica é manter um banco de dados que informa onde e em que estado está cada linha da *cache* [Tanenbaum 2005]. Uma desvantagem dos sistemas NUMA e CC–NUMA refere-se ao tempo de acesso a uma memória remota ser muito maior do que acessar uma memória local. Quando a quantidade de dados é maior do que a *cache*, o desempenho é comprometido por conta das altas taxas de *cache miss* [Tanenbaum 2005].

Já os sistemas COMA são um tipo alternativo de sistema que tenta contornar este problema do tempo de acesso usando a memória principal de cada processador como uma memória *cache* [Tanenbaum 2005].

Várias técnicas de programação podem ser usadas nas arquiteturas de multiprocessadores, mas, em geral, a comunicação entre processos é baseada no princípio de escrever um dado na memória para que, posteriormente, ele possa ser lido por outros processos. Os mecanismos de sincronização podem ser constituídos de seções críticas implementadas com semáforos e/ou monitores que garantem a exclusão mútua [Pitanga 2008]. Em ambientes em que a arquitetura é fortemente acoplada, o paralelismo pode ser atingido usando-se *threads* e/ou múltiplos processos que compartilham um espaço de memória, tendo, como exemplos, o padrão POSIX de *threads* (Pthread) [Rauber e Rünger 2010] e a biblioteca (API – *application program interface*) OpenMP [Quinn 2004] que estão disponíveis para diversas linguagens e sistemas operacionais. Em ambientes com acoplamento

fraco, normalmente, formados por diversos nós interligados por uma rede de interconexão, o paralelismo é alcançado com o uso de bibliotecas que implementam técnicas de *message passing* (MP), como o padrão MPI [Gropp et al. 1999]. Contudo, devido à complexidade das arquiteturas envolvidas, atualmente, um modelo híbrido (OpenMP e MPI) também tem sido bastante usado [Quinn 2004]. Neste trabalho, a etapa experimental considerou um sistema fortemente acoplado, composto de único nó de computação multiprocessado (ver Seção 5.2). Já na etapa de simulação, avaliou-se um sistema fracamente acoplado, baseado em um *Cluster* composto de múltiplos nós de computação (ver Seção 6.4).

2.2.1 – Processamento SMP e AMP

Na categoria de multiprocessadores, é possível classificá-los quanto ao tipo de processamento realizado pelo conjunto de processadores da arquitetura, sendo divididos em processamento AMP (*asymmetric multiprocessing*) e processamento SMP (*symmetric multiprocessing*).

Um sistema AMP foca no particionamento ou especialização dos processadores. Em nível de hardware, geralmente, são formados por processadores com diferentes características (ex. diferentes conjuntos de instruções arquiteturais, diferentes velocidades de execução) [Vajda e Brorsson 2011].

No caso de processadores com diferentes conjuntos de instruções (*instruction set*) podemos ter processadores que realizam processamento de propósito geral e outros que realizam processamento específico (ex. *digital signal processor* – DSP, *graphics processing unit* – GPU). Como exemplo deste tipo de processador tem-se o *Cell,* produzido pela IBM em parceria com a Sony [Costigan e Scott 2007].

No caso de diferentes velocidades de execução, tem-se, por exemplo, um conjunto de processadores "rápidos" (alta frequência de operação, complexo *pipeline out-of-order*, alto consumo de energia) e um conjunto de processadores "lentos" (baixa frequência de operação, *pipeline* simples, baixo consumo de energia), entretanto, todos os processadores têm o mesmo *instruction set* [Fedorova et al. 2009]. Como exemplo deste tipo de processador tem-se o "*big.LITTLE*", fabricado pela ARM [Greenhalgh 2011]

Segundo [Fedorova et al. 2009], com processamento AMP, os processadores mais rápidos são favoráveis para executar programas *single-thread* e sequenciais, já que, esses programas não se beneficiam diretamente da arquitetura multiprocessada, utilizando apenas o poder de processamento de um único processador. Por outro lado, as aplicações paralelas são mais apropriadas para executar em ambientes com múltiplos processadores convencionais.

Isso acontece porque a relação entre desempenho e consumo de energia dos processadores com mesma área de superfície é mais eficiente quando há diversos processadores com mesma frequência de operação do que quando há apenas alguns processadores rápidos. Assim, constata-se que os processadores rápidos são bons para executar aplicações CPU-*Bound*, uma vez que há um uso mais eficiente dos recursos implementados na microarquitetura do processador (ex. *pipeline superscalar*, *branch prediction*, etc). Já para programas *memory-bound*, os processadores lentos são melhores, pois passam a maior parte do tempo de execução buscando dados da memória. Para obter o melhor ganho de um sistema AMP, [Fedorova et al. 2009] sugere um escalonador denominado *parallelism-aware* (PA) em que programas sequenciais e *single-threaded* serão executados nos processadores rápidos e programas paralelos serão executados nos processadores lentos. Os resultados indicam que, em programas com grandes fases sequenciais, há um ganho no desempenho de até 26%. Ainda segundo [Fedorova et al. 2009], no processamento SMP e AMP, observam-se dois tipos de carga de trabalho:

1) *Efficiency specialization* classifica a carga de trabalho como de programas com intenso uso de memória e/ou processador. Os programas de intenso uso de processador são caracterizados pelo uso eficiente de recursos da microarquitetura (ex. *pipeline*). Já nos programas de uso intenso de memória, como há muitas requisições à memória, o processamento é frequentemente interrompido, de forma que, há uso ineficiente do processador. Com o processamento SMP não é possível obter a melhor relação de desempenho e consumo de energia para ambas as aplicações. Assim, ou tem-se um SMP implementado com processadores rápidos de microarquitetura complexa (ideal para aplicações de uso intenso de processador) ou tem-se um SMP de processadores lentos com microarquitetura simples (ideal para programas com intenso uso de memória). Dessa forma, um sistema AMP atenderá as aplicações de ambos os tipos [Saez et al. 2010].
2) *Thread-level parallelism (TLP) specialization* classifica a carga de trabalho como programas paralelos ou sequenciais. Estudos anteriores [Saez et al. 2010] mostram que, comparando processadores de área e capacidade de processamento equivalente, é mais eficiente executar um código paralelo em um grande número de pequenos processadores simples do que em um pequeno número de processadores complexos.

O trabalho de [Saez et al. 2010] propõe um escalonador denominado *Comprehensive AMP* (CAMP) que identifica a carga de trabalho conforme o tipo (TLP ou *Efficiency specialization*). O escalonador CAMP é baseado em uma métrica denominada *Utility Factor*

(UF), que indica quanto o desempenho do programa irá melhorar se todas as suas *threads* forem executadas em processadores rápidos. Ao estimar o desempenho, a métrica UF é calculada, para uma quantidade constante de instruções, indicando o tempo de conclusão nos dois tipos de processadores (rápidos e lentos). Desta forma, UF ajuda o escalonador a identificar quais *threads* são mais indicadas para serem executadas nos processadores rápidos. Comparando CAMP com outros escalonadores (SFD – *SF-Driven* e PA – *Parallelism-Aware*), que implementam apenas um tipo de classificação da carga de trabalho e usando diversas configurações AMP com diferentes tipos de carga de trabalho (paralela, parcialmente sequencial, *single-threaded*, *memory-intensive* e CPU-*intensive*), o escalonador CAMP (diferentemente dos outros escalonadores) mostrou-se eficiente para identificar a carga de trabalho.

Com o intuito de melhorar o desempenho, os programas *multithreaded* podem criar várias *threads* que executam de forma paralela em ambientes com múltiplos processadores. Nestes programas, as seções críticas são usadas para garantir que, em um dado instante, apenas uma *thread* tenha acesso à sua região de dados compartilhados. Entretanto, as seções críticas podem introduzir uma "serialização" na execução das *threads,* o que pode reduzir significativamente o desempenho e escalabilidade dos programas. Baseado em uma arquitetura de processamento assimétrico, o trabalho [Suleman et al. 2009] propõe um mecanismo denominado *Accelerated Critical Sections* (ACS) a fim de acelerar a execução das seções críticas. O mecanismo consiste em enviar uma requisição aos processadores de maior capacidade computacional para execução das seções críticas e a porção paralela do código é executada nos processadores de menor capacidade de processamento. Comparando o mecanismo ACS em relação a ambientes de processamento SMP com diversos processadores de mesma capacidade, os resultados indicam uma redução média de 34% no tempo de execução. Já comparando o ACS em relação a ambientes AMP composto por múltiplos processadores de diferentes capacidades, há uma redução de 23% no tempo de execução.

Com o crescente aumento do número de núcleos nos processadores, [Qingbo et al. 2009] avalia a existência de problemas de comunicação em sistemas multicore SMP. Os resultados mostram que a largura de banda de memória ainda continua restringindo o desempenho dos programas e, além disso, um sistema operacional rodando em ambiente SMP requer complexas estruturas de dados compartilhadas, o que, por conta da contenção, aumenta a latência interna do sistema.

Já [Li et al. 2007b] descreve um escalonador baseado em processamento assimétrico, com suporte eficiente para ambientes *symmetric multiprocessing* e *non-uniform memory*

access. Os resultados indicam um bom desempenho do escalonador para ambas as plataformas, chegando a atingir um *speedup* que varia entre 1,07 e 1,16.

Por sua vez, [Piel et al. 2005] discute um modelo de escalonador denominado *Asymmetric Real-Time Scheduler* (ARTiS) que fornece propriedades de tempo real em ambientes SMP. A idéia básica deste modelo é dividir os processadores em conjuntos, onde alguns dos conjuntos são dedicados a tarefas com algum requisito de tempo real. Os resultados indicam uma redução da latência no espaço de usuário, que, em grande parte dos casos (99%), fica na ordem de 18 microssegundos. Nesse sentido, o sistema pode ser considerado um sistema *hard real-time*.

Por fim, [Happe et al. 2009] apresenta um estudo experimental sobre um escalonador de sistemas operacionais de propósito geral em ambientes SMP, de forma a identificar fatores críticos que podem determinar o desempenho dos programas. Por meio de diversos experimentos, os resultados identificam que diferentes fatores influenciam no desempenho, como, por exemplo, as políticas de balanceamento de carga em ambiente multiprocessado, estratégias de prioridade das tarefas, tamanho do *time-slice* no escalonamento *time-share* e as heurísticas usadas pelo escalonador para otimizar a utilização geral do sistema identificando o tipo das tarefas (CPU-*Bound*, *I/O-Bound*, *interactive tasks*, etc.).

2.3 – Computação em *Cluster*

Até o início da década de noventa, o paradigma de programação por passagem de mensagem (MP) já era amplamente empregado em grandes computadores paralelos com memória distribuída [Marc Snir 1996]. Nesta época, cada fabricante fornecia sua própria implementação MP. Em 1992, foi criado um grupo denominado *Message Passing Interface Forum* (MPIF), composto por mais de 80 pessoas representando 40 empresas de diversas áreas (fabricantes, indústrias, universidades, etc.) que tinha como principal objetivo padronizar o paradigma da programação de passagem de mensagem e fornecer uma biblioteca de programação eficiente e portável. Assim, surge o projeto MPI que, ao invés de selecionar um padrão dentre as diversas implementações existentes na época, reuniu as melhores práticas e características de implementações MP disponíveis até então. O projeto MPI foi amplamente influenciado por trabalhos da IBM, Intel's, Express, nCUBE's Vertex, p4, PARMACS, dentre outros, e, mais recentemente, tem suportado as linguagens C, C++ e Fortran [Marc Snir 1996].

Atualmente, o mecanismo de MP apresenta duas grandes vantagens. A primeira é que os programas desenvolvidos são portáveis de forma a facilitar que qualquer computador possa

executar os programas, sem que haja a necessidade de uma configuração específica de hardware. Outra vantagem é que esse modelo de programação fornece, ao desenvolvedor, o controle explícito sobre a forma na qual a memória é usada por cada processo, uma vez que a forma de acessar a memória pode determinar o desempenho global do sistema.

De modo geral, um programa desenvolvido nos moldes MP consiste em uma determinada quantidade de processos, onde cada um executa seu código cooperando com os demais para a solução de um dado problema. O código executado por cada processo pode ou não ser idêntico, além disso, a comunicação dos processos é feita por intermédio de rotinas fornecidas por bibliotecas (ex. MPI) que enviam e recebem mensagens conforme a necessidade dos processos em execução. Diferentemente de outros modelos de programação paralela, neste enfoque, a comunicação segue o padrão cooperativo, ou seja, um dado enviado não é recebido enquanto o processo destino não chamar a rotina correspondente para o recebimento [Gropp et al. 1999].

Hoje, encontra-se um grande número de bibliotecas MP (MPI, OpenMPI, etc.) que fornece as rotinas básicas do padrão MPI, mas, internamente, diferem-se na forma como são implementadas. Dois exemplos dessa diferença são a forma com que os *buffers* são alocados e preenchidos e, a outra, se as rotinas são bloqueantes ou não bloqueantes (*send*/*receive*) [Dongarra et al. 2003]. Essas variações nas implementações podem ter impacto significativo no desempenho do programa, pois o *overhead* causado pelo software pode tornar-se demasiadamente significativo, visto que, a latência total pode ser dominada pelo tempo de criação da mensagem, e não pelo tempo real de envio da mensagem por meio da rede de comunicação. A largura de banda ocupada por um programa de transferência de mensagens é, muitas vezes, dominada pela quantidade de vezes em que os dados devem ser copiados, entre os componentes do programa, durante uma comunicação. É importante destacar que bibliotecas mal projetadas podem resultar em múltiplas cópias de dados, reduzindo o desempenho global da aplicação [Dongarra et al. 2003].

A Figura 2.8 ilustra um programa típico implementado usando MPI. O processo “pai” (P0) dispara a execução de 4 processos “filhos” (P1, P2, P3 e P4) que são executados remotamente nos nós de computação do *Cluster* e, individualmente, resolverão uma parte do problema. Cada processo executa seu código e comunica-se com outros processos, sendo que a comunicação consiste em enviar e receber dados para distribuição das tarefas ou coleta de resultados. Ao final, o processo pai encarrega-se de integrar os resultados parciais e compor o resultado final. Este modelo é conhecido como mestre/escravo, onde o processo mestre tem a

função de coordenar a execução das atividades e os processos escravos executam as tarefas atribuídas pelo mestre.

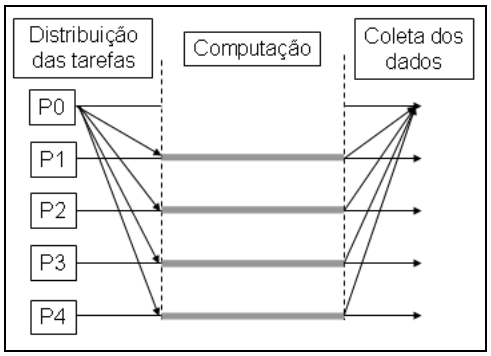


Figura 2.8 Representação de um problema em MPI

Conforme exemplo da Figura 2.8, além de conhecer o problema e dominar uma linguagem de programação compatível (ex. Fortran, C, C++), o programador também deve ter conhecimento da técnica de *message passing* (ex. MPI) para que seu programa possa se beneficiar do processamento distribuído nos nós de computação do *Cluster*.

CAPÍTULO 3 – INFLUÊNCIA DO SISTEMA OPERACIONAL NA EXECUÇÃO DE APLICAÇÕES – *OS JITTER*

3.1 – Introdução

A crescente demanda por recursos computacionais durante os últimos anos fornece indícios suficientes para acreditar que, futuramente, a busca por esses recursos será mais agressiva, desafiando a capacidade da ciência em superar os atuais limites tecnológicos. Por esse motivo, faz-se necessário estudar as questões relacionadas às limitações da escalabilidade computacional. Neste sentido, pesquisas sobre *OS Jitter* estão alinhadas ao futuro e ao desenvolvimento da ciência da computação.

As interferências do sistema operacional (*OS Jitter*) têm sido consideradas como um importante fator que compromete o desempenho de programas distribuídos em grande escala. O impacto no desempenho, geralmente, está relacionado à ocorrência de atrasos nas operações de sincronização dos processos que, por sua vez, estão distribuídos ao longo de diversos nós de computação em um ambiente de processamento distribuído de alto desempenho. Assim, utiliza-se o termo *OS Jitter* para referenciar a interferência "sentida" por determinado processo causada pela operação do sistema operacional.

Algumas técnicas têm sido empregadas para reduzir a influência do sistema operacional no tempo de execução das aplicações como, por exemplo: o uso de tipos específicos de sistemas operacionais (LWK – *Lightweight kernels*) [Ferreira et al. 2008]; customização de um sistema operacional com *kernel* de propósito geral (ex. Linux) [Morari et al. 2011]; dedicar o uso de certos processadores ou *threads* (SMT) a atividades administrativas do sistema operacional [Mann e Mittaly 2009]; sincronização do *OS Jitter* através dos nós (*gang-scheduling* ou *co-scheduling*) [Jones 2011].

Das técnicas citadas, os sistemas *Lightweight kernels* apresentam-se como uma das opções de sistemas operacionais que, muitas vezes, são usados nos nós de computação em plataformas de HPC com a intenção de reduzir os efeitos do *OS Jitter* [Morari et al. 2011], [De e Mann 2010]. Diferentemente de um *kernel* convencional, sistemas LWKs oferecem um conjunto limitado de serviços [Ferreira et al. 2008]. Um LWK usado em sistemas de grande escala, geralmente, não fornece suporte à execução multitarefa, nem mesmo ao tratamento de interrupções [De e Mann 2010], [De et al. 2008]. Comumente, estes não implementam mecanismos de interrupções periódicas de *timer* ou mesmo mecanismos para tratar *TLB misses* [Morari et al. 2011]. Assim, em sistemas LWK praticamente não há ocorrência de *OS Jitter* e, por isso, apresentam melhor escalabilidade do que em sistemas com *kernel*

convencional [Ferreira et al. 2008]. Entretanto, o uso de sistemas baseados em LWK implica em modificar ou mesmo portar os programas, que usam certas funcionalidades não implementadas, para a respectiva plataforma [De et al. 2008]. Como exemplo de implementação LWK tem-se o *IBM Compute Node Kernel* (CNK), usado no supercomputador Blue Gene para aplicações HPC [Morari et al. 2011]. Segundo [Morari et al. 2011], a abordagem LWK é uma boa opção para ambientes onde as aplicações HPC necessitam apenas das bibliotecas científicas e uma implementação MPI. Todavia, tecnologias como virtualização ([Nussbaum et al. 2009]), *scripts*, coleta de rastros de execução (*traces*), informações de *debugging* e outras mais, que necessitam de suporte tanto em nível de software como em nível de sistema operacional, estão sendo cada vez mais necessárias em aplicações HPC. Para muitos desses recursos tecnológicos, é necessário usar um sistema operacional de propósito geral customizado para ambientes de HPC ou estender os serviços fornecidos pelos sistemas LWK, entretanto, essas abordagens resultam em um possível aumento nos níveis de *OS Jitter*.

Outra técnica adotada para reduzir o *OS Jitter* é o escalonamento coordenado (*co-scheduling*), que procura reduzir o impacto do *OS Jitter* aumentando a sobreposição das paralisações sentidas pelos processos [Jones 2011]. Neste caso, durante a execução dos processos paralelos, sempre há dois intervalos periódicos em que cada um é dedicado a determinadas atividades, sendo o intervalo de maior duração dedicado exclusivamente ao escalonamento do processo paralelo e o de menor duração dedicado a outras atividades, como escalonamento de processos administrativos do sistema operacional e/ou tratamento de interrupções [Jones 2011]. Como os intervalos são sincronizados por entre os vários processadores que compõem o *Cluster*, ao longo do tempo cria-se, então, uma sobreposição das atividades do sistema operacional causando a redução nos efeitos do *OS Jitter*. Conforme [Chakravarthi et al. 2002], não é uma tarefa fácil manter a alta precisão (ex. nanossegundos) na sincronização do *clock*, uma vez que a variação na temperatura e na pressão altera a frequência natural de oscilação do cristal.

A Figura 3.1 mostra a execução de um processo paralelo utilizando quatro processadores. Os retângulos sem preenchimento representam um instante de tempo gasto pelo processo paralelo e os retângulos hachurados representam um instante de tempo gasto em outras atividades do sistema operacional (*OS Jitter*). Do lado esquerdo da Figura 3.1, apresenta-se a ocorrência de *OS Jitter* durante a execução de um processo paralelo em ambiente onde há escalonamento coordenado, ou seja, ao longo do tempo observa-se uma sobreposição do *OS Jitter* que acaba por atenuar os efeitos globais do *OS Jitter*. Entretanto, o

lado direito da Figura 3.1 mostra o comportamento do *OS Jitter* quando não há escalonamento coordenado, demonstrando que o *OS Jitter* pode ocorrer a qualquer momento e de forma não sincronizada, acarretando situações em que as interferências têm efeito acumulativo através dos processadores que compõem o *Cluster*.

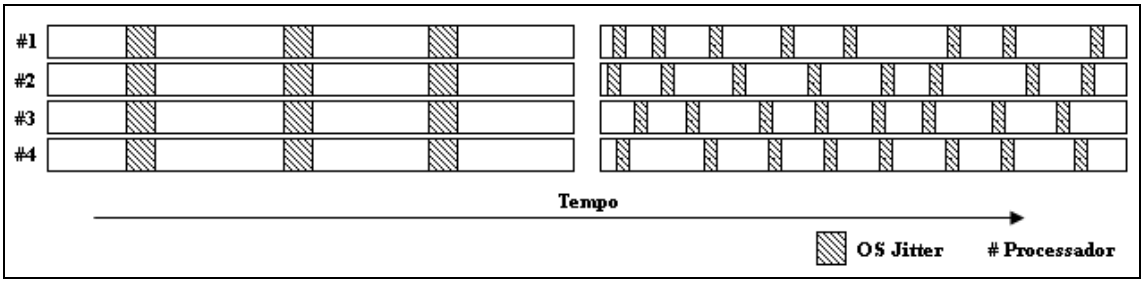


Figura 3.1 Escalonamento coordenado

Considerando um ambiente típico de *Cluster* HPC, em que os processos são formados por diversos ciclos computacionais (*bulk-synchronous*), onde a cada execução do próximo ciclo faz-se necessário que todos os nós tenham concluído a execução do ciclo atual, ou seja, quando atividades administrativas do sistema operacional são executadas de maneira não sincronizada entre os nós, resulta em um aumento do tempo de execução necessário para concluir os processos. Dessa forma, o escalonamento coordenado diminui os efeitos do *OS Jitter* e aumenta a escalabilidade do ambiente [Morari et al. 2011].

Os autores do trabalho [Tsafrir et al. 2005] citam que a sincronização de fontes de *OS Jitter*, como as interrupções de *timer,* pode realmente aliviar seus efeitos; porém, em certa escala (tamanho do *Cluster*) pode ser difícil ou impossível implementar tal abordagem. Além disso, a referida proposta sofre potencialmente com a penalidade do custo (*overhead*) indireto gerado pelas interrupções de *timer*, que pode ser relativamente "elevado". Ainda segundo os autores, a principal desvantagem da referida técnica é que ela trata o sintoma do problema e não a sua causa e, além disso, a única forma de extinguir o *OS Jitter* de granularidade fina é eliminando completamente as interrupções de *timer*. Neste sentido, [Tsafrir et al. 2005] propõe um mecanismo denominado "*smart timers*", que atua agregando eventos próximos, reduzindo o *overhead* e evitando "*clock ticks*" periódicos desnecessários.

O trabalho [Fröhlich et al. 2011] compara as estratégias *one-shot* e *periodic* (ver seção 3.2.1) usadas na geração da interrupção de *timer*. Para isso, foi projetado e implementado um novo modelo de evento baseado em timers *one-shot*, onde a programação do *timer* é baseada no intervalo do próximo evento. Os experimentos demonstram que em termos de precisão e interferência com a aplicação, quando corretamente projetada, a interrupção de *timer*

periódica é equivalente a interrupção de *timer one-shot*, e em alguns casos, até mesmo melhor.

O surgimento de arquiteturas multiprocessadas e/ou *multithread* permitiu a criação de novas técnicas com a finalidade de tratar o *OS Jitter* por meio de uma abordagem em que certos processos são executados em um conjunto específico de processadores. Desse modo, alguns processadores são usados para tratar as fontes de *OS Jitter*, enquanto que os processos do usuário são executados em outros processadores. O uso dessa técnica demonstra bons resultados, porém, ela apresenta perda de parte da capacidade computacional em cada nó. Contudo, com o aumento da quantidade de processadores por nó, essa perda é cada vez menor em termos percentuais [Mann e Mittaly 2009]. Outro ponto desfavorável é que o sistema operacional deve estar preparado de forma a escalonar *threads* de *kernel* e interrupções nos processadores que foram destinados a esse propósito, evitando ainda a migração de processos entre as duas "classes" de processadores [Mann e Mittaly 2009].

Com base no que foi exposto anteriormente, fica evidente que conhecer os fatores que contribuem para o *OS Jitter* em um sistema operacional, bem como o quanto cada fator colabora para o atraso total, é de fundamental importância para o projeto e implementação de sistemas operacionais para ambientes de HPC, assim como, para o desenvolvimento de novas técnicas e o aperfeiçoamento das técnicas já existentes. Geralmente, os projetistas de sistemas operacionais fazem uso dessas informações para tomar decisões sobre quais serviços implementar, bem como para decompor e escalonar esses serviços, minimizando o impacto no desempenho global do sistema [Ferreira et al. 2008]. Na literatura, um dos principais fatores causadores de *OS Jitter* nos sistemas operacionais modernos é o tratamento assíncrono de eventos pelo *kernel*, em especial as interrupções de hardware. A seguir, é apresentado um detalhamento sobre o gerenciamento de interrupções, a fim de subsidiar as demais seções do trabalho.

3.2 – Tratamento de Interrupções

As interrupções são eventos produzidos por um sinal elétrico proveniente de dispositivos (hardware) que alteram a sequência de instruções executadas pelo processador [Bovet e Cesati 2000]. Este mecanismo permite ao dispositivo comunicar-se com o processador para notificar alterações no seu estado (ex. uma tecla pressionada). Assim, uma interrupção dispara a execução de uma rotina do sistema operacional que foi implementada para tratar eventos de uma determinada natureza. Após o evento ser tratado, o retorno do processo original é permitido sem prejuízo para sua execução. Para que o processador possa

interromper a execução de um processo e retornar a ele mais tarde, sem comprometer seu estado interno, é necessário executar operações para salvar e restaurar os contextos de hardware e software do processo interrompido. A atividade de salvamento e restauração dos valores dos contextos de processos é denominada troca de contexto (*context switch*).

A implementação da troca de contexto é uma operação delicada que envolve a manipulação de registradores[1] e "*flags*" específicas de cada processador. A implementação da *preempção* por tempo tem como base as interrupções geradas pelo hardware programável de temporização, que são geradas em intervalos regulares (ex. a cada *n* milissegundos) e são tratadas pelo manipulador de interrupção (*interrupt handler*). O intervalo de tempo entre cada sinal gerado pelo hardware de temporização programável é chamado de *clock tick*. Os sinais são gerados em intervalos de tempo conforme a frequência na qual o hardware foi programado (ex. 100Hz é equivalente a um intervalo de 10 milissegundos) [Bovet e Cesati 2000]. Nas últimas versões do Linux, é possível compilar o *kernel* para usar diferentes frequências (ex. 100Hz, 250Hz, 1000Hz). Nesse sistema operacional, o campo *counter* do descritor de cada processo especifica quantos *clock ticks* do tempo de processador é atribuído ao processo, sendo que o *quantum* sempre é múltiplo de um *clock tick*. O valor do campo *counter* é decrementado a cada *clock tick* e, ao atingir um valor negativo, outro processo é escalonado, atribuindo o processador ao novo processo [Bovet e Cesati 2000].

3.2.1 – *Advanced Programmable Interrupt Controller* – APIC

O APIC foi desenvolvido pela Intel para fornecer a capacidade de lidar com grandes quantidades de interrupções, permitindo que cada interrupção possa ser encaminhada a um conjunto específico de processadores. O APIC fornece suporte à comunicação inter-processador e remove a necessidade de compartilhar uma única linha de interrupção com grande número de dispositivos [Brindley 2011].

Assim, o APIC representa uma série de dispositivos e tecnologias que trabalham em conjunto para gerar, encaminhar e lidar com interrupções de hardware de forma escalável e gerenciável. No presente estudo, a arquitetura alvo de implementação são os sistemas SMP usados em diversos modelos de processadores Intel.

A Figura 3.2 representa um típico esquema do sistema APIC, o qual basicamente consiste de um módulo I/O APIC que, por meio das linhas de IRQ é responsável por receber as requisições de interrupções dos dispositivos de I/O (ex. teclado, rede, discos, etc.) e

[1] Registradores – são memórias pequenas e muito rápidas, usadas para armazenar dados em tempo de execução existente dentro do processador. Exemplos: Contador de programa (PC – Program Counter), Apontador de pilha (SP – Stack Pointer), etc.

encaminhá-las ao módulo local APIC (LAPIC) que está integrado a cada processador. O LAPIC comunica-se com o I/O APIC através de um barramento *Interrupt Controller Communication* (ICC). Como pode ser observado, os processadores são divididos em duas classes: *bootstrap processor* (BSP) e *application processors* (AP). O processador BSP é determinado pelo hardware ou pelas configurações no BIOS (*Basic Input/Output System*) e é responsável por inicializar os outros processadores e carregar o sistema operacional. Desta forma, os processadores APs são ativados apenas após o sistema operacional ter sido inicializado.

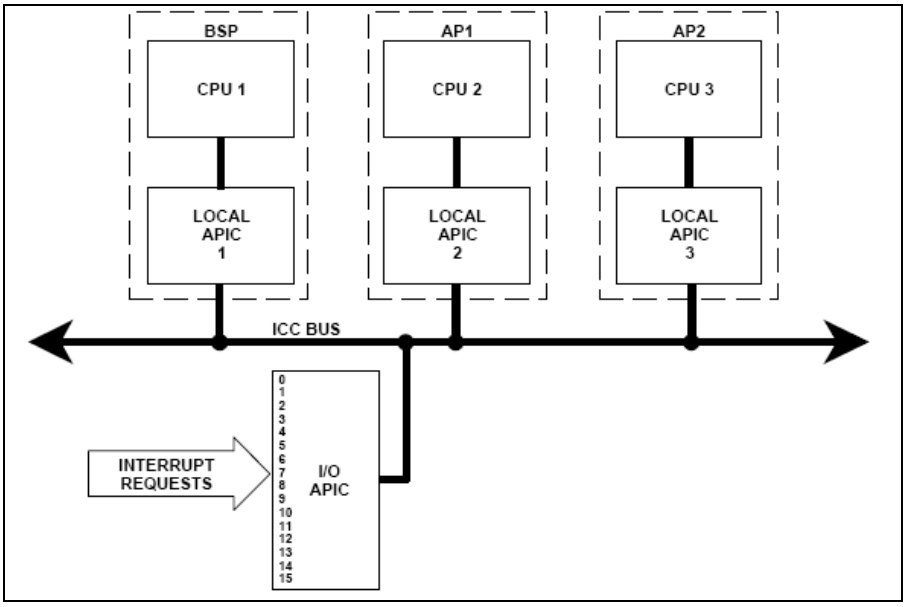


Figura 3.2 Esquema genérico de interconexão do APIC [Intel Corporation 1997]

Nesse esquema, cada I/O APIC é composto por um conjunto de linhas de IRQ, 24 entradas na IRT (*Interrupt Redirection Table*), tendo, cada uma de 64 bits, registradores programáveis e uma unidade para enviar e receber mensagens APIC sobre o barramento ICC [Intel Corporation 1996]. Quando um dispositivo gera uma interrupção, o I/O APIC seleciona a entrada correspondente dentro da IRT e com suas informações gera uma mensagem que será transmitida às unidades APIC pelo barramento. Cada entrada da IRT pode ser programada para indicar o vetor de interrupção e sua prioridade, polaridade de disparo do sinal elétrico (ativado em alto ou baixo), processador de destino da mensagem e a forma de seleção deste processador (estática ou dinamicamente).

Por sua vez, a Figura 3.3 foi elaborada com base na documentação do I/O APIC fornecida pela Intel [Intel Corporation 1996] e representa uma das 24 entradas da IRT com seus respectivos campos distribuídos ao longo de 64 bits.

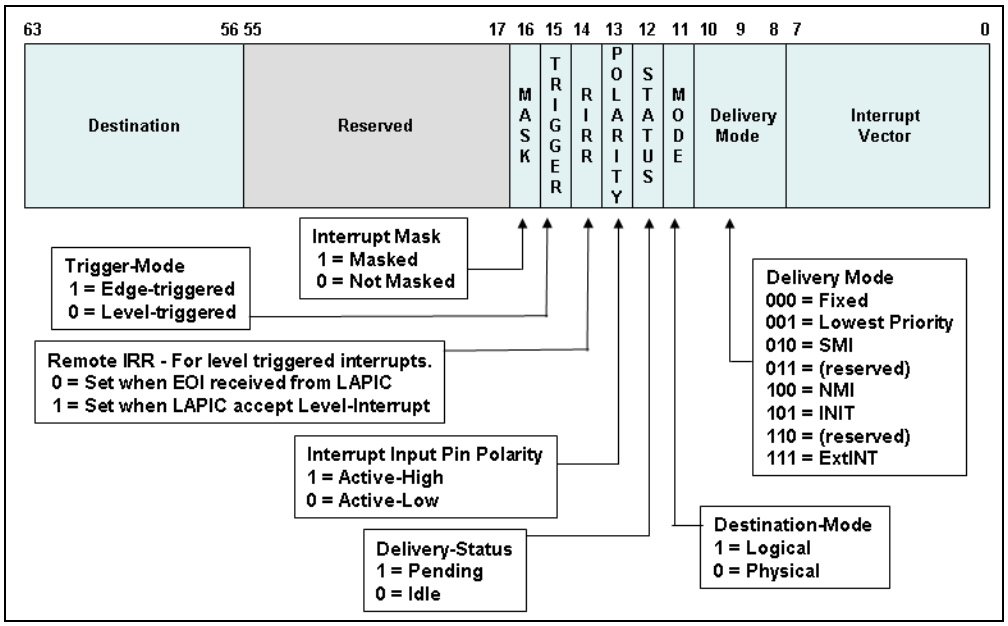


Figura 3.3 I/O APIC – Campos do *interrupt redirection table*

A programação do I/O APIC é feita por registradores que são acessados por um esquema de mapeamento indireto usando dois registradores (IOREGSEL e IOWIN). Tais registradores estão mapeados, respectivamente, para os endereços FEC0*xy*00h e FEC0*xy*10h, onde *xy* são determinados pelos campos *x* e *y* do registrador de realocação do endereçamento base do APIC e podem conter os seguintes valores: *x*=0-Fh e *y*=0,4,8,Ch. Já a Figura 3.4 mostra o mapeamento na memória do endereço base do I/O APIC quando *x* e *y* são, respectivamente, zero. Todos os registradores são acessados usando 32 bits, ou seja, para modificar um campo (bit ou byte) todos os 32 bits devem ser lidos, posteriormente, altera-se o campo e, só depois, escreve-se novamente os 32 bits no registrador.

De forma semelhante, cada LAPIC é composto por um conjunto de registradores APIC e hardware associado que controlam o fornecimento de interrupções ao processador, assim, em um sistema SMP cada processador tem o seu próprio módulo LAPIC.

Segundo [Intel Corporation 2010], o LAPIC pode receber interrupções das seguintes fontes: 1) dispositivos de I/O conectados localmente, ou seja, interrupções geradas por dispositivos de I/O que estão conectados diretamente aos pinos de interrupção local do processador (LINT0 e LINT1); 2) dispositivos de I/O conectados externamente, isto é, interrupções geradas por dispositivos de I/O que estão conectados ao módulo I/O APIC; 3) interrupção inter-processador (IPIs), ou seja, programando o *interrupt command register* (ICR), um processador pode enviar uma mensagem IPI para interromper outro processador ou grupo de processadores; 4) LAPIC que pode ser programado para enviar uma interrupção periódica; 5) interrupção gerada pelo sensor de temperatura, que é disparado quando este

detecta um valor pré-estabelecido; 6) *performance monitoring counter interrupts*, que gera uma interrupção quando ocorre um *overflow*; 7) interrupção de erro interno do APIC, que é gerada quando uma condição de erro é encontrada dentro do APIC (ex. *Illegal Register Address*, *Send Illegal Vector*, *Received Illegal Vector*, etc.).

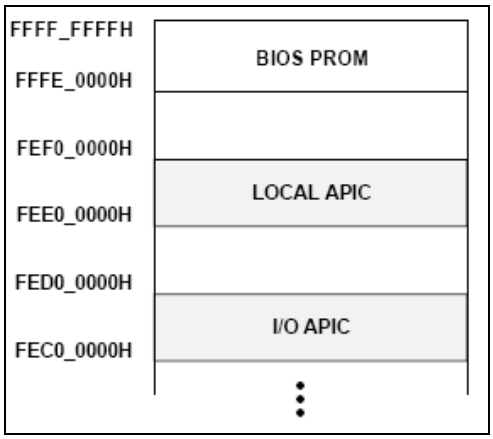


Figura 3.4 Mapeamento na memória do LAPIC e I/O APIC [Intel Corporation 1997]

As interrupções de *timer* são geradas no LAPIC. A Figura 3.5 apresenta o diagrama de bloco do LAPIC, detalhando as principais estruturas que o compõem. A programação do LAPIC é feita por meio da leitura e escrita de registradores que podem ser mapeados para endereços da memória principal. Assim, tais registradores são mapeados de memória para uma região de 4kB do espaço de endereçamento físico do processador, geralmente, com o endereçamento inicial em FEE00000H (ver Figura 3.4). Para resolver conflitos com mapeamentos de memória dos sistemas já existentes, o espaço de endereçamento inicial do LAPIC pode ser alterado. Neste caso, é possível usar o registrador IA32_APIC_BASE MSR, que é mapeado no endereçamento 1BH, para consultar o valor do novo endereçamento inicial do LAPIC.

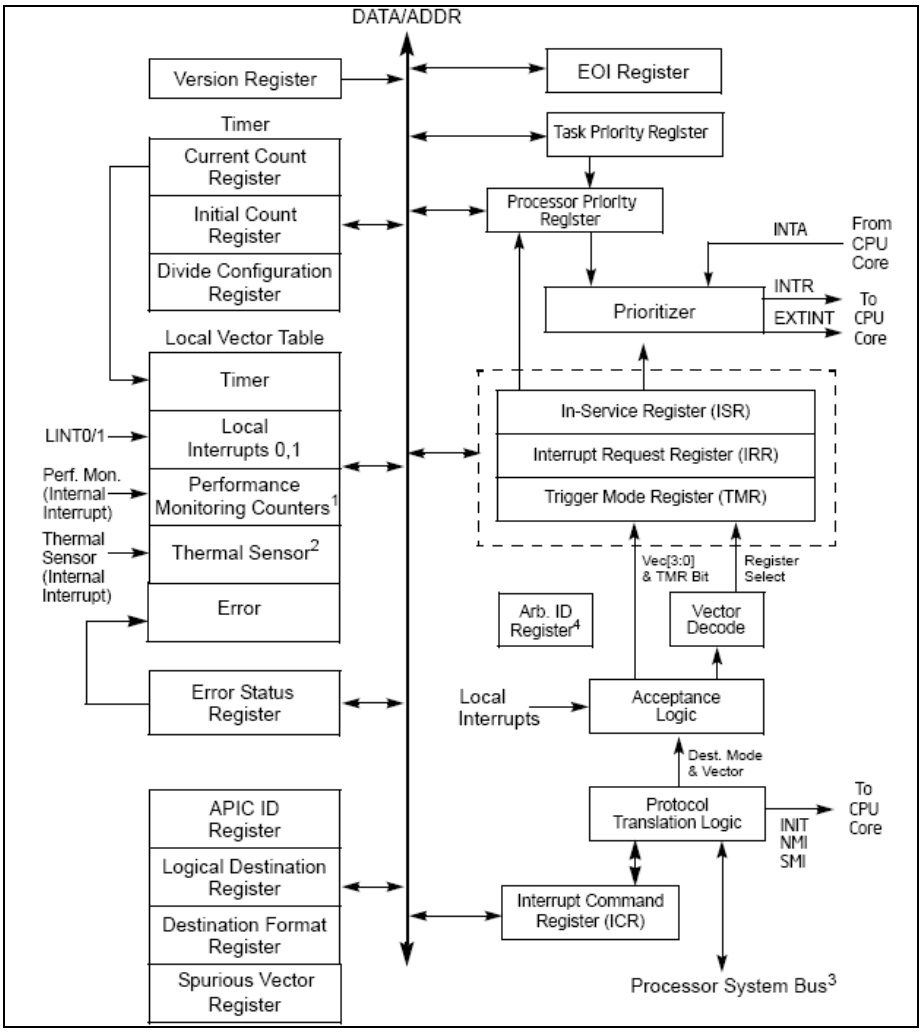


Figura 3.5 Estrutura do LAPIC [Intel Corporation 2010]

A Figura 3.6 apresenta um detalhamento dos recursos disponíveis no *Local Vector Table* (LVT) bem como o endereçamento base desses recursos. O mecanismo gerador de interrupções de *timer* do LAPIC é composto por quatro campos: 1) *Vector* (*Interrupt vector number*): é um índice único que identifica a interrupção para que um *handle* (rotina que irá manipular a interrupção) adequado possa ser executado; 2) *Delivery Status*: indica o estado de entrega da interrupção, sendo 0 (zero) para interrupção entregue e aceita pelo processador e 1 (um) para indicar que a interrupção foi entregue ao processador, mas ainda não foi aceita; 3) *Mask*: é uma *flag* que habilita ou inibe a entrega de interrupção ao processador, sendo 1 (um) para habilitar e 0 (zero) para inibir a entrega; 4) *Timer mode*: está vinculado à forma como a interrupção é gerada, podendo ser *one-shot* ou *periodic*.

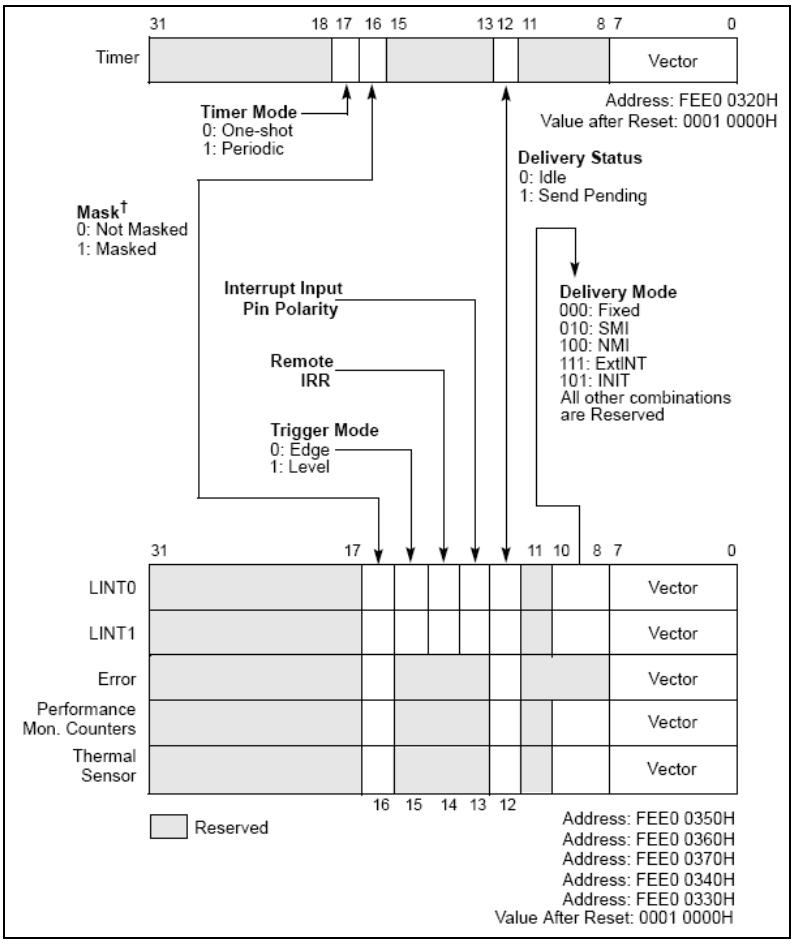


Figura 3.6 Diagrama do *local vector table* [Intel Corporation 2010]

Em modo *one-shot,* o mecanismo é iniciado programando um valor no registrador de contador inicial (*initial-count*), que é copiado para o registrador de contador atual (*current-count*) que começa a ser decrementado e, quando chega em zero, uma interrupção de *timer* é gerada no processador e seu valor permanece em zero até que ele seja reprogramado. O espaço de tempo entre cada decremento do registrador *current-count* é derivado do barramento de *clock* do processador dividido por um valor especificado no registrador de divisão (*Divide Configuration Register*). Em modo *periodic,* o funcionamento é semelhante, entretanto, o registrador *current-count* é automaticamente reprogramado com o valor do registrador *initial-count*. A Figura 3.7 apresenta um esquema com endereçamento dos registradores usados para programar o mecanismo de interrupções de *timer* do LAPIC.

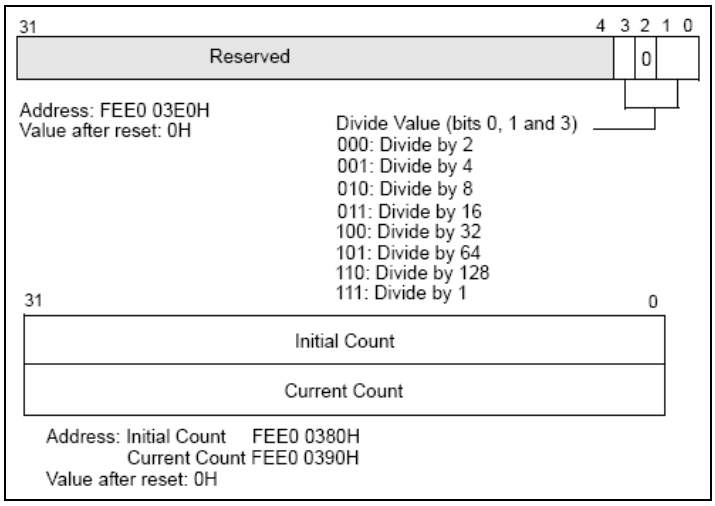


Figura 3.7 Registradores *initial count* e *current count* [Intel Corporation 2010]

Além do tratamento de interrupções, diversas outras fontes de *OS Jitter* e técnicas para reduzir os seus efeitos têm sido investigadas na literatura. A seguir, é apresentada uma revisão dos principais trabalhos na área.

3.3 – Trabalhos Correlatos

A seguir, é apresentado o estado da arte das pesquisas relacionadas ao *OS Jitter*. Os trabalhos relacionados foram agrupados por área de aplicação.

3.3.1 – Interferência dos Modos de Operação

Em um sistema operacional, as chamadas de sistema (*system calls*) são interfaces das quais os processos fazem uso para acessar serviços fornecidos pelo *kernel* do sistema operacional [Haldara e Aravind 2010]. A implementação de uma chamada de sistema ocorre por intermédio de rotinas do *kernel* [Bovet e Cesati 2005], o que significa que a execução de uma chamada de sistema deve ser feita com o processador em modo privilegiado [Haldara e Aravind 2010]. Durante a execução de um processo, diversas chamadas de sistema podem ser executadas, sendo que, para cada uma é necessário que o processador passe do modo não privilegiado (*user mode*) para o modo privilegiado (*kernel mode*) e, então, ao final da execução, retorne para o modo não privilegiado. A troca no modo de operação, geralmente, é cara e lenta, necessitando de operações (ex. salvar e restaurar os registradores) que implicam em um custo adicional de tempo na execução do programa. Outro problema relacionado, que tem sido amplamente investigado é a necessidade de mover dados do espaço de endereçamento dos processos para o espaço de endereçamento do *kernel* e vice-versa, frequentemente, utilizando-se das chamadas de sistema (ex. *send*/*receive*, *read*/*write*). Embora esses custos sejam insignificantes (nanossegundos) para chamadas individuais, eles tornam-se

relevantes à medida que o número de chamadas aumenta. Apesar disso, não é incomum a existência de servidores de aplicações (ex. *web servers*) que executem milhões de chamadas de sistemas em curtos intervalos de tempo [Appleton 2009]. Trabalhos de pesquisa em arquiteturas de sistemas operacionais estão sendo conduzidos a fim de diminuir os custos envolvidos nas chamadas de sistema.

É válido dizer que as chamadas de sistema podem criar uma sobrecarga significativa prejudicando o desempenho do programa, o que pode significar uma degradação do desempenho em até duas ordens de magnitude [Purohit et al. 2003]. De acordo com Purohit *et al*., uma forma de reduzir as cópias de dados nas chamadas de sistema é, simplesmente, minimizar o número de vezes em que acontecem as trocas no modo de operação. Por exemplo, fazer a leitura de arquivos em pedaços maiores (ao invés de pequenos pedaços) pode reduzir o número de vezes em que a chamada de sistema *read* é executada para ler o arquivo completo. Em [Purohit et al. 2003], tem-se a proposta de um *framework,* chamado *Cosy* (*Compound System Calls*), elaborado para melhorar o desempenho dos processos que fazem uso intenso de chamadas de sistema, permitindo que os processos do usuário executem segmentos de código no espaço de endereçamento do *kernel* e evitando, assim, múltiplas trocas no modo de operação e cópias de dados. Esse *framework* reúne as chamadas de sistema e códigos intermediários pertencentes a um segmento de dados do processo em um *buffer,* que os autores denominam de "*compound*". O *buffer* é passado ao *kernel* através de uma nova chamada de sistema, *cosy_run*, que irá decodificar e executar as operações, evitando trocas no modo de operação e cópias de dados. Um protótipo deste *framework* foi implementado em um sistema Linux e os testes realizados demonstraram um ganho de desempenho entre 20% e 80% para processos *IO-Bound*. De forma geral, o trabalho [Purohit et al. 2003] forneceu uma interface genérica para a aplicação da técnica de *zero-copy.*

Em [Vicente et al. 2012], os autores implementam e avaliam diferentes chamadas de sistemas compostas. Diferente de [Purohit et al. 2003], são avaliadas tanto chamadas compostas (ex. forkexec) quanto chamadas (simples e compostas) repetidas (ex. forkN e forkexecN). Também, foram investigadas chamadas de sistema executando em máquinas físicas e virtuais. A análise dos dados obtidos por meio de experimentos controlados foi realizada utilizando ANOVA (*Analysis of Variance*) e testes de Tukey. Os resultados indicam redução no tempo de execução de até 58 % e as maiores melhorias ocorrem no ambiente não virtualizado.

3.3.2 – Interferência de *Cache*

A memória *cache* é usada para amenizar as diferenças de velocidade entre o processador e a memória principal [Handy 1998]. Uma de suas vantagens é que o tempo de acesso a ela é muito menor do que o tempo de acesso à memória principal. Entretanto, por ter um alto custo de aquisição e, ainda, por não ser possível fabricar uma memória *cache* grande e rápida o suficiente para armazenar todos os dados e instruções, este tipo de memória tem um tamanho relativamente pequeno e está organizada de forma hierárquica como pode ser observado na Figura 3.8.

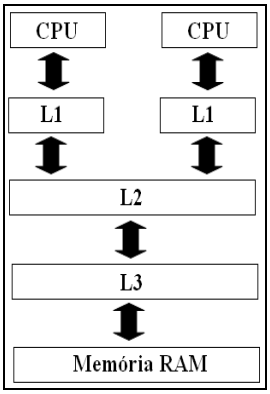


Figura 3.8 Modelo genérico de hierarquia de memória *cache*

Apesar de possuir um tamanho limitado, este tipo de memória é eficiente para amenizar as diferenças entre processador e memória principal, uma vez que os programas seguem um padrão de acesso à memória, o qual segue o princípio da "localidade de referência". Este princípio é baseado em localidade espacial e temporal [Polloni e Fedeli 2003]. A localidade espacial é definida quando há um acesso a um determinado endereço de memória, sendo muito provável que haja um novo acesso a um endereço próximo a este. Já a localidade temporal é definida quando há um acesso a determinado endereço de memória, sendo provável então que, em um futuro próximo, haja um novo acesso a este mesmo endereço. Desta forma, a *cache* mantém os últimos endereços acessados e suas vizinhanças. Com esta estratégia, mais de 95% dos acessos podem ser encontrados na memória *cache,* não sendo necessário acessar a memória principal [Polloni e Fedeli 2003]. Por isso, quando o processador necessita de algum dado, ele verifica, inicialmente, se este está presente na memória *cache*; caso positivo (*cache hit*), o dado é transferido para o processador em alta velocidade; caso contrário (*cache miss*), o dado é recuperado da memória principal, sendo que este mecanismo pode consumir vários ciclos de processamento.

Conforme [Li et al. 2007a], existe um custo direto e um custo indireto da troca de contexto entre processos compartilhando o mesmo processador. O custo direto está ligado a diversas operações como salvar e restaurar os valores dos registradores do processador, executar as rotinas necessárias para o escalonamento do próximo processo, recarregar entradas necessárias na TLB (*Translation Address Table*), limpar o *pipeline* do processador, etc. O custo indireto é associado ao compartilhamento da *cache* entre os processos, o que resulta em degradação do desempenho do sistema como um todo. Este custo pode variar conforme a arquitetura do processador e a carga do sistema, que pode ser composta por diferentes comportamentos de acesso à memória. Os experimentos realizados por [Li et al. 2007a] demonstram que, em geral, o custo indireto varia de diversos microssegundos a mais de mil microssegundos. Quando o tamanho dos dados é maior do que o tamanho da *cache*, a sobrecarga de recarga da *cache L2* tem impacto substancial sobre o custo da troca de contexto. Além disso, os autores também demonstram o potencial impacto dos tratadores de interrupção e outras rotinas do sistema operacional durante a medição do custo da troca de contexto. Tal impacto pode ser atenuado por meio de um sistema multiprocessado em que um processador é utilizado para a medição enquanto o outro executa os processos do sistema operacional.

De acordo com o estudo apresentado em [Mogul e Borg 1991], o desempenho dos processadores está intimamente dependente do desempenho da *cache*. Em princípio, um bom projeto de processador deve ser capaz de executar quase uma instrução por ciclo, mas, na prática, não é possível sustentar esta taxa de execução porque o sistema de memória nem sempre pode fornecer instruções e dados em velocidades tão altas [Mogul e Borg 1991]. Quando um sistema operacional mantém múltiplas trocas de contexto entre os processos/*threads*, o princípio de localidade pode ser violado, pois as instruções e os dados do processo escalonado podem não estar na *cache*. Isso acontece porque a memória *cache* é pequena para conter todo o conjunto de dados e/ou instruções de muitos processos ao mesmo tempo. Além disso, o custo da troca de contexto no desempenho da *cache* pode ser estimado observando-se o comportamento das taxas de *cache hit* e *cache miss* após uma operação de troca de contexto [Mogul e Borg 1991]. Modul e Borg utilizaram um simulador de *cache* para realizar diversos testes que foram divididos em grupos de *benckmarks*, assim como para diversas configurações de *cache*. Os resultados indicaram que, dependendo dos parâmetros do sistema de *cache*, foi encontrada uma média de custo entre dezenas e centenas de microssegundos para uma troca de contexto, ou seja, uma mudança de contexto gasta vários milhares de ciclos de instruções, o que é comparável ao tempo gasto para enviar ou receber

um pacote de rede. O custo da interferência de *cache* na troca de contexto pode ser maior que todos os outros custos da troca de contexto (custos diretos). Nos experimentos de Mogul e Borg, o custo de execução do código em *kernel* responsável por executar a troca de contexto não foi medido, sendo ignorada qualquer operação realizada pelo *kernel* para invalidar ou limpar a *cache*.

O artigo [Fromm e Treuhaft 1996] quantifica os efeitos da interferência de *cache* em razão da troca de contexto em ambiente multiprogramado usando um simulador, SimOS. De posse desse simulador para coleta de dados, os autores determinaram que a interferência na *cache,* devido à troca de contexto, não chega a 1% da taxa total de *cache miss*, ou seja, é muito pequena. Assim, mesmo se todas as *cache misses* resultantes da troca de contexto fossem eliminadas e cada *cache miss* custasse muitas centenas de ciclos do processador, os programas só poderiam executar, aproximadamente, 30% mais rápido. De acordo com os autores, a completa eliminação de todas as interferências na *cache* devido à troca de contexto tem um impacto mínimo no tempo total de execução. Os autores reconhecem a possibilidade de que o custo variável (custo indireto) da interferência de *cache* possa ser significativamente maior do que os custos fixos (custo direto) e que isso ocorre tipicamente quando os processos executam por longos períodos de tempo e têm oportunidade para amortizar os custos ao longo da execução de um grande número de instruções.

Segundo [Gupta et al. 1991], em um ambiente de programação paralela, as trocas de contexto podem causar não apenas a substituição parcial/total dos dados na *cache* do processador no qual o processo corrente está executando, mas pode levar o escalonador a re-escalonar o processo em um processador diferente, no qual não há dados deste processo em *cache* e, finalmente, causando um grande aumento da taxa de *cache miss* levando à degradação do desempenho.

Uma opção para reduzir os efeitos do compartilhamento da *cache* entre múltiplos processos é usar técnicas de particionamento da *cache*, de forma que as partições são criadas e atribuidas aos processos conforme as linhas de *cache*, assim processos em partições diferentes não compartilham a mesma linha de *cache* o que causa a diminuição da taxa de *cache miss*. O particionamento pode ser atingido colocando a região de código e dados dos processos em posições da memória que são mapeadas para determinadas linhas da *cache*, de tal forma que os processos são colococados em regiões distintas para reduzir o compartilhamento das linhas de *cache* [Liedtke et al. 1997]. Em [Falk e Kotthaus 2011] é apresentado um mecanismo para otimização do posicionamento de código dos processos baseado em grafo de conflito da *cache*

que reduz a taxa de *cache miss*, os resultados indicam uma redução média de 15% nas taxas de *cache miss*.

3.3.3 – Interferência do Ambiente

Como mencionado na Seção 3.1, tradicionalmente, ambientes HPC de grande escala têm reduzido o *OS Jitter* fazendo uso de sistemas operacionais especializados. Exemplos de sistemas operacionais desse tipo são Catamount, BLRTS, Puma, Cougar [Beckman et al. 2006], [Kelly e Brightwell 2005]. Porém, isso limita a utilização de tais sistemas, pois a maioria das aplicações que são escritas para sistemas operacionais comerciais não são compatíveis com estes sistemas [De et al. 2007]. Ao longo da história, isso resultou em esforços para criar versões mais leves de sistemas operacionais como o Linux para utilização em sistemas HPC de grande escala [De et al. 2007]. Essas versões leves dos sistemas operacionais necessitam de estudos detalhados para identificar as fontes de *OS Jitter* e ainda medir quantitativamente o seu impacto. Muitos dos estudos de *OS Jitter* têm se concentrado nos efeitos em escala de processos paralelos e não se aprofundaram na questão de identificar as principais causas. Uma forma de identificar as possíveis fontes de *OS Jitter* e medir seu impacto em um processo é através de um detalhado rastro (*trace*) de execução das atividades do sistema operacional [De et al. 2007] durante a execução da aplicação alvo do estudo.

No trabalho [De et al. 2007], os autores apresentam o projeto e implementação de uma ferramenta que ajuda a identificar as fontes de *OS Jitter* em sistemas operacionais Linux e que pode ser usada para medir quantitativamente a contribuição do *OS Jitter* por vários processos do sistema (*daemons*) e interrupções de hardware. A metodologia usada pelos autores foi composta por um *benchmark* rodando no espaço do usuário, onde as latências percebidas no *benchmark* são medidas. Desse modo, cada latência é associada a *daemons* ou interrupções utilizando-se de dados obtidos de rastros de execução do *kernel*. Os experimentos executados em Linux, usando *runlevel* 3, revelaram que 63% de todos os atrasos são causados pela interrupção de *timer*, sendo que o restante é proveniente de vários *daemons* do sistema e de outras interrupções, das quais, a maioria pode ser facilmente eliminada.

[Tsafrir et al. 2005] apresenta uma abordagem para quantificar o efeito do *OS Jitter* utilizando modelagem probabilística. Os autores sugerem um modelo teórico que quantifica o efeito do *OS Jitter*, independentemente da sua origem, de maneira que, em uma carga de trabalho paralela, cada processo tem alguma probabilidade de sofrer atrasos durante a fase computacional pela influência do *OS Jitter*. Além disso, eles mostram que, se essa probabilidade for suficientemente pequena, o impacto negativo do *OS Jitter* cresce

linearmente com o aumento do tamanho do *Cluster*. Dessa forma, a probabilidade de atrasos é um múltiplo do tamanho do *Cluster* e da probabilidade de um único nó sofrer com *OS Jitter*. Por meio de um *micro-benchmark* e rastros de execução relativos às interrupções do sistema operacional, os autores determinaram que a principal fonte de *OS Jitter* é a ocorrência de interrupções de *timer* e que a redução delas é uma possível solução para o problema.

O artigo [Agarwal et al. 2005] apresenta um estudo de modelagem estatística que demonstra os efeitos do *OS Jitter* na escalabilidade de aplicações paralelas. Nesse trabalho, foram estudados *OS Jitter* com três tipos de distribuições de probabilidades: exponencial, cauda pesada (*heavy-tailed*) e Bernoulli. Os estudos mostram que, na presença de *OS Jitter* com distribuição exponencial, o sistema fornece uma boa escalabilidade, entretanto, isso não acontece quando o *OS Jitter* segue uma distribuição de Bernoulli ou *heavy-tailed*. É necessário ressaltar que o modelo usado pelos autores pressupõe algumas condições: 1) a carga de trabalho das fases computacionais é igualmente dividida; 2) os processadores dos nós são idênticos (devido a homogeneidade dos nós considera-se o "*noise*" idêntico em todos os nós); 3) o *overhead* causado por processos intrínsecos (ex. interrupções de *timer*) são considerados iguais em todos os nós e o *overhead* extrínseco (processos executando apenas em um conjunto de nós) é ignorado.

Em [Gioiosa et al. 2004], foi desenvolvido um estudo do impacto do sistema operacional em aplicações paralelas. Os autores programaram um *benchmark* que executou *n* iterações computacionais que foram cuidadosamente calibradas para executar em um determinado tempo. Mediante a ocorrência de *OS Jitter,* a finalização da computação exigiu um tempo maior de execução do *benchmark*. Os resultados, usando um *benchmark* de 1000μs, indicaram atrasos que variavam entre 0,5 até 1,4μs, 8 até 12μs e alguns de 190μs, os quais por meio de monitoramento do *kernel* foram causados, respectivamente, por interrupção do "*timer* global" (PIT – *Programmable Interrupt Timer*), interrupção do "*timer* local" (LAPIC) e interrupções de rede. A principal contribuição do estudo foi fornecer e demonstrar uma metodologia para identificar e quantificar o *OS Jitter* em grandes *Clusters*.

[Garg e De 2006] buscou validar o modelo teórico estudado em [Agarwal et al. 2005] baseado em experimentos com dados coletados em *Clusters* de grande porte. A validação foi feita usando-se um *benchmark* criado pelos autores, cujo processo de computação é executado e distribuído entre múltiplos processadores. A partir dos dados coletados, foram feitas previsões de desempenho. Além das previsões, descobriu-se que as medições de distribuições do *OS Jitter* também ajudam na identificação de comportamentos anômalos em nós do *Cluster*.

Em [Jones et al. 2003], os autores investigam o impacto negativo do sistema operacional na escalabilidade e desempenho de aplicações paralelas em grandes *Clusters*. Dois tipos de interferências foram identificadas e classificadas: interferências de longa duração e interferências de curta duração. As mesmas foram causadas, respectivamente, por processos do sistema executando em *background* (ex. *daemons*) e rotinas internas do sistema operacional (ex. *timers*). Os resultados sugerem que as interferências causadas por processos em *background* podem ser minimizadas ao executar os processos em nós contendo um ou mais processadores livres que seriam dedicados para tratar os eventos do sistema operacional. Por exemplo, um nó contendo 16 processadores executa 15 processos, sendo um em cada processador, de tal forma que um processador é deixado livre para execução de processos do próprio sistema operacional. A degradação do desempenho causada pelas interferências de curta duração (ex. *timers*) depende da quantidade de sobreposições, ou seja, a interferência causada pelos *timers* torna-se menor quando todas ou a maioria delas acontece simultaneamente em um dado instante.

[Beckman et al. 2006] apresenta a implementação de um *micro-benchmark* que é usado para medir o *OS Jitter* em diversas plataformas. Os resultados indicam que as taxas de *OS Jitter* podem variar amplamente entre plataformas. Dentre as plataformas avaliadas, o IBM BG/L *compute node* é considerado virtualmente "*noiseless*", devido às baixas taxas de *OS Jitter*. Comparando o BG/L *compute node* com o BG/L I/O *node* e considerando que as duas plataformas são idênticas, algumas diferenças podem ser atribuídas diretamente ao sistema operacional. Assim, 80% do *OS Jitter* são de 1,8μs e correspondem à atualização do *timer* escalonado, 16% são ligeiramente mais longos (2,4μs) porque na sexta interrupção do *timer* o escalonador é executado e o restante é atribuído a desvios menores que 6μs. Em consequência disso, quando se compara o BG/L com outras plataformas, o *OS Jitter* do BG/L é realmente muito pequeno.

[Mann e Mittaly 2009] apresentam uma abordagem que utiliza núcleos ou *threads* adicionais em um sistema SMT (*simultaneous multithreading*) para reduzir o *OS Jitter* proveniente de diversas fontes. Basicamente, a abordagem consiste na redução de *threads* de *kernel*, gerenciamento inteligente das interrupções e administração das prioridades de hardware das *threads* em arquitetura SMT. Segundo os autores, as fontes de *OS Jitter* podem ser classificadas como interferência dos processos no espaço do usuário, *threads* de *kernel*, interrupções e interferência no código, entre *threads* do mesmo processo em execução em sistemas SMT. Assim, são apresentadas algumas técnicas que podem ser usadas para reduzir

os efeitos do *OS Jitter*. Para amenizar a influência do *OS Jitter* proveniente dos processos do usuário e *threads* de *kernel,* os autores consideram o uso de um escalonador de tempo real (*SCHED_RR*). Já os atrasos provocados por interrupções são tratados com o isolamento dos processadores, ou seja, removendo o processador das decisões de escalonamento e tratamento de interrupções. Por fim, é apresentado um recurso de prioridades para *threads* em que o hardware (neste caso, os processadores POWER5 e POWER6) implementa internamente diferentes níveis de prioridades para as *threads*. Este recurso foi usado para diminuir a interferência no código em execução pelas *threads* irmãs. Os experimentos indicaram uma redução média no "*slowdown"* de 30% para uma aplicação paralela simulada e 50% para uma aplicação real.

Os autores de [Ferreira et al. 2008] sugerem que a limitação no desempenho global das aplicações é decorrente da variação do tempo gasto pelos processos na execução de operações coletivas, como *MPI_Allreduce* [Quinn 2004]. Tais variações normalmente são vistas como efeitos causados pela influência do *OS Jitter*. O trabalho proposto por esses autores utiliza o sistema operacional Catamount (LWK) para demonstrar a importância, em termos de frequência e duração, na forma como o *OS Jitter* é gerado. Os resultados demonstram que em um *Cluster* contendo 10.000 nós e uma taxa de *OS Jitter* de 2,5%, proveniente do processamento de rede, o *OS Jitter* pode não gerar impactos ou resultar em um fator de desaceleração ("*slowdown*") do desempenho na ordem de 20%, dependendo exclusivamente da forma como o *OS Jitter* é gerado. Os resultados mostram que, conforme as operações coletivas (ex. *MPI_Bcat*, *MPI_Allreduce*), as aplicações são capazes de absorver quantidades substanciais de *OS Jitter* de alta frequência com baixa duração, mas tendem a amplificar *OS Jitter* de baixa frequência com alta duração. Em termos de comportamento das aplicações, os resultados de [Ferreira et al. 2008] confirmam a teoria generalizada de que a sensibilidade de um processo ao *OS Jitter* está fortemente relacionada com questões como a relação de comunicação/computação, quantidade de comunicações coletivas usadas e o tamanho dessas comunicações coletivas. O estudo [Ferreira et al. 2008] permite concluir que aplicações utilizando comunicações coletivas pequenas (ex. *MPI_Allreduce* de 8-byte) são mais suscetíveis ao *OS Jitter* do que em comunicações coletivas maiores (ex. 32 bytes).

Segundo [Tsafrir et al. 2005], a principal fonte de *OS Jitter* que prejudica, principalmente, aplicações de granularidade fina é a interrupção de *timer*, que, por sua vez, contribui quase exclusivamente para o *OS Jitter* de granularidade fina. Enquanto que o *overhead* direto da interrupção de *timer* é relativamente pequeno (abaixo de 1%), o *overhead* indireto pode chegar a dezenas de pontos percentuais. O estudo também apontou uma grande

correlação entre o aumento da variabilidade na duração das fases computacionais e as taxas de *cache miss*. Além disso, as taxas de *cache miss* em L1 estão fortemente correlacionadas com as interrupções de *timer*, contrastando com as taxas de *cache miss* em L2 que permanecem relativamente inalteradas.

Em [De et al. 2007], foi usado o sistema operacional Fedora Core 5, configurado em *Runlevel* 3, em que constatou-se que a principal fonte de *OS Jitter* foi a interrupção de *timer* (63%). As outras fontes de *OS Jitter* foram distribuídas entre processos do sistema (ex. hidd) e demais tipos de interrupções (ex. eth0). Algumas dessas fontes podem ser facilmente removidas (ex. hidd), contudo, a remoção de outras fontes (ex. events0, kblockd0, etc.) exigiria alterações no *kernel* do sistema operacional. Em estudo posterior, os autores de [Mann e Mittaly 2009] classificam as fontes de *OS Jitter* em quatro categorias distintas: interrupções, *threads* de *kernel workqueue*, outras *threads* de *kernel* e processos do usuário (ver Tabela 3.1).

Tabela 3.1 Fontes de *OS Jitter*. Adaptado de [Mann e Mittaly 2009]

	Fontes de OS Jitter	**Menor Interrupção (us)**	**Maior Interrupção (us)**	**Frequência Total**	**Média (us)**	**Total de OS Jitter (%)**
Interrupções	Timer	1,73	3397,14	164360	3,39	85,483
	eth0	4,6	22,55	764	9,04	1,062
Threads de Kernel (WorkQueue)	events0	1,49	166,4	3148	4,07	1,97
	rpciod0	45,08	46,8	1	46,79	0,007
Threads de Kernel (Outras)	rtasd	22,72	38,23	396	28,21	1,716
	pdflush	1,45	79,9	594	6,19	0,565
	watchdog	1,31	2,12	1384	1,54	0,326
	nfsd4	5,09	6,42	33	5,72	0,029
	nfsd	6,45	6,47	8	6,47	0,008
Processos no espaço do usuário	irqbalan	145,01	200,39	297	159,88	7,295
	init	1,49	30,85	595	16,82	1,538

[Morari et al. 2011] apresenta uma técnica capaz de fornecer uma análise descritiva de cada evento do sistema operacional que contribui para o *OS Jitter*. A técnica permite detalhar todas as fontes de *OS Jitter* objetivando fornecer a frequência e a duração de cada evento. Em [Morari et al. 2011], as atividades que mais contribuíram para o *OS Jitter* foram analisadas e classificadas em cinco categorias:

1. Periódicas: manipulador (*handler)* da interrupção de *timer* e *run_timer_softirq* responsável por executar os *timers* de software expirados;
2. *Page fault*: manipulador da exceção *page fault*;
3. Escalonamento: função *schedule* e *softirqs*;

4. Preempção: processos de *kernel* e do usuário que causam a preempção da aplicação;
5. I/O: Manipulador da interrupção de rede e *tasklets*.

Os experimentos realizados em [Morari et al. 2011] mostram que cada programa sofre um *OS Jitter* diferente, tanto em termos de *overhead,* quanto em termos de composição. Outra observação importante é que, em termos de frequência e duração, *page faults* podem ter maior impacto do que as interrupções de *timer*.

Os autores de [Gioiosa et al. 2004] comentam que para uma grande variedade de sistemas operacionais baseados em UNIX/Linux, alguns tipos de interrupções, como interrupções de rede, "*timer* global" (PIT) e "*timer* local" (LAPIC), contribuem com 95% do *OS Jitter*.

No estudo apresentado em [Jones et al. 2003], usando o sistema operacional IBM AIX para avaliar a variabilidade no tempo de execução e a baixa escalabilidade de processos que executam operações de granularidade fina, os autores identificaram dois tipos de interferências principais: 1) de curta duração associada à interrupção de *timer,* gerada a cada 20μs em média e 2) de longa duração relacionada às atividades de escalonamento do sistema (ex. "*daemons*", "*cron jobs"*).

O presente estudo difere-se dos demais trabalhos descritos anteriormente, sobretudo, na forma como os experimentos foram projetados, conduzidos e analisados, usando para isso a técnica de planejamento de experimentos estatisticamente controlados (*DOE*), apresentado em [Montgomery 2005]. Essa técnica proporciona elementos para julgar e interpretar a importância de uma variável e/ou fator sobre a variável resposta do objeto de estudo. Assim, a presente pesquisa investiga por meio de experimentos rigorosamente controlados, de forma conjunta e sistemática, diversas fontes de *OS Jitter*, quantificando-as para apontar aquelas fontes de maior influência no tempo de processamento. Além disso, não foi encontrado um estudo abordando métodos rigorosos de análise de dados, usando múltiplas replicações para avaliar de forma clara a contribuição dos fatores. Os trabalhos anteriores não consideram aspectos arquiteturais do sistema operacional como escalonamento *tickless* e regulação automática da voltagem do processador, que são características que influenciam no *OS Jitter*. Por fim, os estudos anteriores não investigam a relação entre tempo de processamento vs. número de fases vs. número de processos, que são dimensões exploradas no presente estudo.

CAPÍTULO 4 – PLANEJAMENTO E ANÁLISE DE EXPERIMENTOS

4.1 – Introdução

A técnica de projeto de experimentos estatisticamente controlados, conhecida como *DOE* (*Design of Experiments*) [Montgomery 2005] [Button 2005], fornece elementos para encontrar e interpretar a influência de um ou mais fatores sobre uma variável resposta em um estudo experimental.

Geralmente, DOE é constituído por diversos testes nos quais mudanças controladas são executadas nas variáveis de entrada (*inputs*), possibilitando observar e medir os efeitos destas mudanças nas variáveis de saída/resposta (*outputs*). Vale ressaltar que o sistema investigado pode ser uma combinação de máquinas, métodos, pessoas e outros recursos que transformam uma entrada em uma saída, de uma ou mais respostas observáveis com características ou parâmetros específicos [Montgomery 2005].

A Figura 4.1 apresenta um modelo genérico de um sistema em que diversos fatores estão sendo avaliados. Dentre os fatores, os controláveis estão representados pelas variáveis que, durante a execução do experimento, podem ter seus valores manipulados, já os fatores não controláveis (fatores de ruído) são as variáveis que influenciam nas variáveis de resposta e, devido a certas restrições, seus valores não podem ser manipulados e/ou controlados (ex. temperatura ao ar livre).

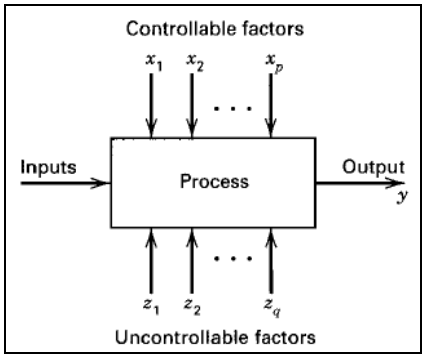


Figura 4.1 Modelo genérico de um processo e/ou sistema [Montgomery 2005]

4.2 – Planejamento de Experimentos

O principal objetivo em alterar sistematicamente os valores dos fatores controláveis é avaliar o efeito gerado na variável resposta e, com isso, fornecer elementos para determinar como e quais os principais fatores que influenciam no sistema investigado. Os fatores de controle podem ser quantitativos ou qualitativos. Em relação aos fatores não controláveis,

cuidados especiais devem ser adotados durante a execução do experimento, a fim de evitar que os efeitos gerados pelos fatores controláveis sofram influências significativas dos efeitos provocados pelos fatores não controlados.

Segundo [Montgomery 2005], técnicas de análise e projeto de experimentos são comumente aplicadas na melhoria de certas características de produtos ou processos, minimizando a quantidade de testes e aprimorando o uso dos recursos (ex. materiais, tempo, disponibilidade). De forma semelhante, [Button 2005] faz um desdobramento conforme o propósito dos experimentos:

a. Determinar quais variáveis são mais influentes nos resultados;
b. Atribuir valores às variáveis influentes de modo a otimizar os resultados;
c. Atribuir valores às variáveis influentes de modo a minimizar a variabilidade dos resultados;
d. Atribuir valores às variáveis influentes de modo a minimizar a influência de variáveis incontroláveis.

Quanto à importância de investigar o efeito provocado nas respostas dos experimentos por dois ou mais fatores, em que cada um deles possui dois ou mais níveis, [Montgomery 2005] recomenda o uso de técnicas clássicas de projeto de experimentos, como, por exemplo: projeto fatorial completo e projeto fatorial fracionado.

Em um projeto fatorial, os experimentos são constituídos pelos fatores de interesse do estudo e distribuídos em diferentes níveis, de forma que esses últimos são os valores que um fator controlável do experimento pode operar. De maneira geral, os projetos fatoriais do tipo 2^k (k fatores com dois níveis cada) são comuns em pesquisas com um número elevado de fatores controláveis sendo investigados [Montgomery 2005]. Uma questão favorável desse tipo de projeto experimental é a realização de um número relativamente limitado de experimentos fornecendo informações suficientes para efetuar a análise. Entretanto, com um número muito reduzido dos níveis torna-se impossível explorar de maneira totalmente completa o vasto universo de variáveis. Nos projetos fatoriais, onde os fatores são explorados em dois níveis, eles geralmente são representados usando um esquema de sinais, em que os símbolos (-) e (+), respectivamente, representam o nível inferior e o superior de cada fator. A representação dos fatores e níveis usando sinais ajuda na etapa de realização dos cálculos que determinam a influência dos fatores e de suas interações no sistema. Além disso, possibilita a diagramação do projeto experimental em forma de uma matriz de contraste [Almeida 2010], cuja representação é arranjada em tabelas onde os sinais (-) e (+) são organizados nas células

fazendo-se uma combinação completa ou parcial dos fatores em seus respectivos níveis. Essas tabelas são particularmente úteis quando mais de dois fatores são estudados conjuntamente.

O efeito de um fator é definido como sendo uma mudança na resposta produzida por uma alteração em seu nível. Os efeitos podem ser classificados em duas categorias: efeito principal e efeito de interação, que, respectivamente, estão relacionados à alteração do nível de um único fator ou à alteração do nível entre dois ou mais fatores simultaneamente. Assim, para o cálculo dos efeitos, além da representação relativa aos efeitos principais, também é necessário fazê-la para os efeitos de interação. O valor do sinal, referente ao efeito de uma interação, é gerado pelo produto da operação de multiplicação dos sinais relativos aos fatores envolvidos na interação.

A Tabela 4.1 apresenta um projeto em dois níveis com três fatores (2^3), ou seja, uma combinação de 8 tratamentos (combinação de fatores e níveis). Nesse exemplo, os fatores controláveis são expressos por letras (A, B, C) em que cada um pode operar nos níveis (-) ou (+), estando aptos a representar um estado ou concentração de um fator (ex. ligado/desligado). Em um projeto fatorial completo, as interações são descritas por todas as possíveis combinações dos fatores controláveis em que o sinal referente à interação é obtido pela multiplicação dos sinais referentes aos fatores em questão em um determinado tratamento. Por exemplo, o sinal da interação AB no tratamento 01 é o produto da multiplicação entre o sinal do fator A no tratamento 01 pelo sinal do fator B no tratamento 01.

Tabela 4.1 Matriz de contraste para um planejamento 2^3

	Fatores Principais			**Interação Entre Fatores**			
	A	**B**	**C**	**AB**	**AC**	**BC**	**ABC**
01	-	-	-	+	+	+	-
02	+	-	-	-	-	+	+
03	-	+	-	-	+	-	+
04	+	+	-	+	-	-	-
05	-	-	+	+	-	-	+
06	+	-	+	-	+	-	-
07	-	+	+	-	-	+	-
08	+	+	+	+	+	+	+

Uma parte importante no projeto de experimentos é a necessidade de se realizar replicações dos tratamentos para estimar o erro experimental, determinando se as diferenças observadas são estatisticamente significativas. É importante observar que as replicações devem ser repetições “autênticas”, ou seja, todas as etapas do procedimento são repetidas

igualmente de forma a representar adequadamente o espaço experimental no qual o planejamento fatorial foi desenvolvido [Montgomery 2005].

Durante a execução do experimento, é importante que todos os tratamentos e replicações previstos no projeto sejam realizados aleatoriamente, de forma que a ordem de execução deve ser obtida através de "sorteios". Esse procedimento é realizado para distribuir igualmente os efeitos produzidos pelos fatores não controláveis nas respostas analisadas e que são condições necessárias de diversos métodos estatísticos, os quais exigem que os componentes do erro experimental sejam variáveis aleatórias independentes [Montgomery 2005].

4.2.1 – Execução dos Experimentos

Para usar uma abordagem estatística em projeto e análise de experimentos necessita-se que o experimentador tenha uma idéia clara sobre o objeto alvo de estudo, como os dados serão coletados e, pelo menos, uma compreensão qualitativa de como esses dados devem ser analisados [Montgomery 2005].

Para o processo de experimentação, é importante definir o planejamento dos testes, permitindo que durante a fase de execução o processo seja meticulosamente monitorado a fim de assegurar que tudo ocorra conforme planejado, pois erros no procedimento certamente invalidarão os resultados.

Abaixo, estão enumeradas as etapas do procedimento experimental descritas por [Montgomery 2005] e utilizadas neste trabalho:

1. Identificação e definição do problema;
2. Escolha dos fatores, níveis e faixa de valores para os níveis de cada fator;
3. Seleção da variável resposta;
4. Definição do projeto experimental;
5. Execução do experimento;
6. Análise estatística dos dados;
7. Conclusão.

Durante a definição do projeto experimental, deve-se escolher o tamanho da amostra, quantidade de replicações, seleção adequada da ordem de execução, determinação da existência ou não do uso de blocos ou outras restrições de aleatorização. Segundo [Button 2005], em estudos complexos, com um grande número de variáveis, não é aconselhável iniciar com um extenso conjunto de experimentos abrangendo uma grande quantidade de variáveis em diversos níveis. Ainda segundo [Button 2005], é mais prudente trabalhar com

um número menor de variáveis/níveis para, à medida que o estudo obtenha avanços, novas variáveis e níveis sejam acrescentadas, descartando as que não se apresentam influentes. Quando inicia-se a execução do experimento, é de vital importância monitorar cuidadosamente o processo a fim de garantir o planejamento inicialmente proposto. Nesse estágio, erros no procedimento experimental, geralmente, acarretam a invalidação dos dados coletados. Antes da execução definitiva do experimento é importante aplicar algumas execuções "pilotos" para fornecer informações sobre a consistência experimental, sobre o controle do sistema de medição e, ainda, uma idéia dos erros experimentais. Com isso, tem-se a oportunidade de revisar as decisões tomadas nos passos anteriores. Sobre os métodos estatísticos para análise dos dados, estes devem ser utilizados para que os resultados e conclusões sejam verificados com rigor. Os métodos estatísticos não podem provar que um fator tem um determinado efeito particular, pois apenas fornecem orientações quanto à confiabilidade e validade dos resultados. Quando utilizados apropriadamente, métodos estatísticos não possibilitam que algo seja provado experimentalmente, mas permitem medir o provável erro em uma conclusão ou atribuir um nível de confiança a uma declaração.

Segundo [Montgomery 2005], durante todo esse procedimento, é importante considerar que a experimentação é uma parte importante do processo de aprendizagem, já que, os pesquisadores tentam formular hipóteses sobre um objeto de estudo em que experimentos são executados para investigar essas hipóteses, facilitando a formulação de novas suposições. Logo, a experimentação é um processo dinâmico. Após a análise dos resultados, o pesquisador deve obter conclusões acerca dos resultados práticos e recomendar um curso de ação. Nesse ponto, uma extensa documentação incluindo métodos gráficos e tabelas são úteis, principalmente, para apresentação dos resultados a terceiros.

4.3 – Método de Análise dos Dados

Com o objetivo de verificar se os tratamentos são considerados estatisticamente diferentes, geralmente, faz-se comparações simultâneas entre as médias de diversas amostras usando testes paramétricos como ANOVA [Montgomery 2005]. Entretanto, tais testes exigem que a distribuição de probabilidades da variável estudada seja uma distribuição normal ou, ainda, que seja possível aplicar alguma transformação (ex. Box-Cox) nos dados, a fim de atender a suposição. Contudo, a técnica de ANOVA ainda exige o cumprimento das suposições de aleatoriedade e independência entre as observações, incluindo variâncias iguais para as populações. Assim, em determinados conjuntos de dados, pode não ser possível cumprir todas as suposições necessárias pelos testes paramétricos, exigindo a utilização de

testes não paramétricos, ou seja, que não dependem de uma distribuição de probabilidades teórica. Os testes não paramétricos não especificam condições sobre os parâmetros da população cujas amostras foram retiradas. Segundo [Campos 1979], os métodos de análise estatística não paramétrica permitem estruturar os dados, de forma que os valores observados são substituídos pelas ordens (*ranks*) das observações, possibilitando métodos estatísticos equivalentes aos da análise de variância paramétrica (ex. Kruskal-Wallis).

A estatística do teste de Kruskal-Wallis é representada pela Equação 4.1 [Campos 1979]:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{K} \frac{R_i^2}{n_i} - 3(N+1) \tag{4.1}$$

onde, N é a quantidade total de observações; R_i são os valores das ordens atribuídos ao tratamento i; n_i é o tamanho da amostra no tratamento i; K é a quantidade de tratamentos.

Conforme a estrutura do teste de Kruskal-Wallis, considera-se H_0 como sendo todos os tratamentos iguais ($t_1=t_2=\ldots=t_K$) e H_1 se pelo menos dois tratamentos diferem entre si. Dessa forma, H_0 é rejeitada se o H calculado for maior ou igual ao h tabelado, ao nível α de significância. Ao rejeitar H_0, admite-se que pelo menos dois tratamentos são diferentes entre si. Para identificar quais são esses tratamentos, uma abordagem é aplicar um teste de comparação múltipla, realizado entre pares de tratamentos. A estatística do teste para o caso de grandes (maiores que 30) amostras de tamanhos iguais é dada pelas Equações 4.2 e 4.3 [Campos 1979]:

$$\left| \overline{R_i} - \overline{R_j} \right| \geq d.m.s \tag{4.2}$$

$$d.m.s = Q\sqrt{\frac{K(N+1)}{12}} \tag{4.3}$$

onde, $\overline{R_i}$ e $\overline{R_j}$ representam as médias das ordens atribuídas aos tratamentos i e j, respectivamente, na classificação conjunta das N observações referentes aos K tratamentos. Q é um valor tabelado pela distribuição q, que considera um nível de significância α para K tratamentos.

4.4 – Cálculo dos Efeitos

Para encontrar a contribuição dos fatores e/ou interações sobre a variável resposta, uma abordagem é o método conhecido como Matriz ou Tabela de Sinais, discutido em [Filho 2008]. Através do cálculo dos efeitos, é possível encontrar a fração da variação da variável

resposta que pode ser explicada por cada fator e suas interações. Segundo [Filho 2008], a matriz de contraste (ver Tabela 4.2) é equivalente aos coeficientes de um grupo de equações que são combinações lineares das respostas. O cálculo dos efeitos pode ser obtido realizando-se operações algébricas, onde o valor dos efeitos é dado pelo produto escalar do seu vetor *X* de coeficientes da matriz de contraste pelo vetor médio $\overline{R}$ de todas as respostas, dividido pela quantidade de *K* tratamentos do experimento (ex. em um planejamento 2^5 tem-se 32 tratamentos). Para realizar o cálculo, o vetor *X* é dado por cada uma das *c* colunas referentes à matriz de contraste na sua forma transposta, ou seja, um vetor linha X_c^t. Assim, a equação geral para o cálculo dos efeitos é dada pela Equação 4.4:

$$S_c = \frac{\left(X_c^t\right)\overline{R}}{K} \tag{4.4}$$

O percentual de variação atribuído a cada um dos fatores e suas interações facilita identificar o quanto um determinado fator ou interação é importante na medida do impacto que causa sobre a variável resposta [Filho 2008]. Assim, o cálculo do percentual é dado pelas Equações 4.5, 4.6 e 4.7.

$$SQE = \sum_{i=1}^{K}\sum_{j=1}^{r}\left(\overline{R_i} - R_{ij}\right)^2 \tag{4.5}$$

onde *SQE* é a parcela relativa ao erro experimental referente aos *K* tratamentos, considerando as *r* replicações. Dessa forma, $\overline{R_i}$ é a média da variável resposta no tratamento *i*, e R_{ij} é o valor da variável resposta no tratamento *i* e replicação *j*.

$$SQT = \sum_{p=1}^{c}\left(KrS_p^2\right) + SQE \tag{4.6}$$

SQT é a variação total da variável resposta, também conhecida como soma dos quadrados totais, onde, *K* é a quantidade total de tratamentos; *r* é o total de replicações no tratamento em questão; S_p é o valor calculado do efeito referente a uma determinada coluna (*p*) da matriz de contraste que representa um fator ou interação.

$$P_c = 100 \times \frac{KrS_c^2}{SQT} \tag{4.7}$$

Assim, P_c (ver Equação 4.7) representa o percentual de variação atribuído ao fator ou interação *c*.

Com o objetivo de exemplificar o uso das técnicas descritas anteriormente, suponha um projeto fatorial completo em dois níveis (-/+) usando apenas dois fatores (A, B), de forma

que cada tratamento (linhas da tabela) tenha sido replicado três vezes. Neste cenário, são obtidas três respostas em cada replicação (R_1, R_2 e R_3). Esse planejamento está representado pela Tabela 4.2.

Tabela 4.2 Matriz de contraste para um planejamento 2^2 [Filho 2008]

	Fatores / Interação			**Respostas**			**Média**
	A	**B**	**AB**	$\mathbf{R_1}$	$\mathbf{R_2}$	$\mathbf{R_3}$	$\overline{R}$
01	-	-	+	500	550	540	530
02	+	-	-	1100	1140	1090	1110
03	-	+	-	900	850	920	890
04	+	+	+	1900	1950	1910	1920

O percentual de contribuição de cada fator pode ser calculado pelas Equações 4.4 até 4.7. Iniciando pela Equação 4.4, usada para o cálculo dos efeitos dos fatores e interações, tem-se:

$$S_A = \frac{1}{4} \times \left[\begin{bmatrix} -1 & +1 & -1 & +1 \end{bmatrix} \times \begin{bmatrix} 530 \\ 1110 \\ 890 \\ 1920 \end{bmatrix} \right] = 402{,}5 \tag{4.8}$$

$$S_B = \frac{1}{4} \times \left[\begin{bmatrix} -1 & -1 & +1 & +1 \end{bmatrix} \times \begin{bmatrix} 530 \\ 1110 \\ 890 \\ 1920 \end{bmatrix} \right] = 292{,}5 \tag{4.9}$$

$$S_{AB} = \frac{1}{4} \times \left[\begin{bmatrix} +1 & -1 & -1 & +1 \end{bmatrix} \times \begin{bmatrix} 530 \\ 1110 \\ 890 \\ 1920 \end{bmatrix} \right] = 102{,}5 \tag{4.10}$$

onde S_A e S_B representam, respectivamente, os valores dos efeitos referentes aos fatores principais e S_{AB} representa o valor do efeito devido à interação entre os fatores A e B.

Seguindo o procedimento descrito anteriormente, a Equação 4.5 fornece a parcela relativa ao erro experimental (*SQE*).

$$SQE = \sum \begin{pmatrix} (530-500)^2 + (530-550)^2 + \ldots + \\ (1920-1950)^2 + (1920-1910)^2 \end{pmatrix} = 6800 \tag{4.11}$$

Pela Equação 4.6 é possível encontrar a variação total da variável resposta, referente aos fatores A e B, como também pela interação AB. Nesse projeto experimental, tem-se quatro tratamentos sendo que cada um possui três replicações, conforme:

$$SQT = \begin{pmatrix} \left(4\times 3\times 402{,}5^2\right)+\left(4\times 3\times 292{,}5^2\right) \\ +\left(4\times 3\times 112{,}5^2\right)+6800 \end{pmatrix} = 3129425 \tag{4.12}$$

Por fim, conforme a Equação 4.7, o percentual de variação atribuído a cada fator e suas interações é dado por:

$$P_A = 100\times\frac{4\times 3\times 402{,}5^2}{3129425} = 62{,}12\% \tag{4.13}$$

$$P_B = 100\times\frac{4\times 3\times 292{,}5^2}{3129425} = 32{,}81\% \tag{4.14}$$

$$P_{AB} = 100\times\frac{4\times 3\times 112{,}5^2}{3129425} = 4{,}85\% \tag{4.15}$$

Deste modo, observa-se que a variação explicada pelo fator A é de 62,12%, a variação devida ao fator B é de 32,81% e a variação devida à interação entre os fatores A e B é de 4,85%. Os 0,22% restantes não são explicáveis pelo projeto experimental desenvolvido.

CAPÍTULO 5 – ESTUDO EXPERIMENTAL SOBRE FONTES DE *OS JITTER* NO KERNEL LINUX

5.1 – Introdução

Nesse capítulo, tem-se a apresentação dos detalhes relacionados à abordagem usada nos experimentos destinados à caracterização do *OS Jitter* no *kernel* do Linux.

5.2 – Ambiente de Testes

O ambiente de testes é composto de um computador com dois processadores Intel Xeon E5620 de 2,40GHz cada, com 24GB de memória RAM e um disco rígido de um Terabyte. A Figura 5.1, produzida pelo software Portable Hardware Locality [Broquedis et al. 2010], mostra detalhes da topologia dos processadores usados nos experimentos. É possível observar que o último nível de *cache* (L3) tem um tamanho de 12MB e é compartilhado entre os quatro processadores do mesmo *socket*. Cada processador possui uma *cache* L1 de 32kB e uma *cache* L2 de 256kB, as quais não são compartilhadas. Para simplificar, cada núcleo da Figura 5.1 é referenciado como sendo PU#0 até PU#7, onde PU representa uma unidade de processamento (ou processador).

Convencionou-se, neste trabalho, que o programa de prova (ver Seção 5.3) é executado apenas na PU#1, onde as fontes de *OS Jitter* são rigorosamente controladas conforme o nível de cada fator no tratamento em questão. Os núcleos restantes são usados de acordo com a especificação de cada tratamento (ver Seção 5.3).

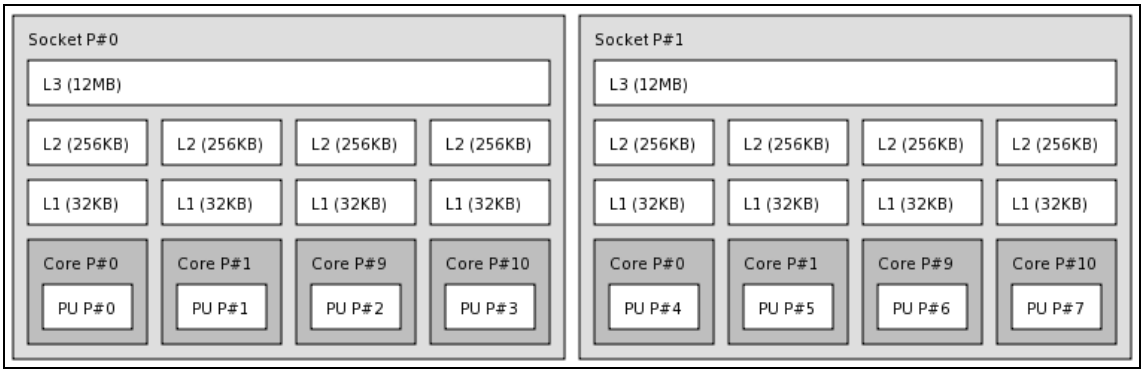


Figura 5.1 Topologia de processadores usados nos experimentos

O sistema operacional usado foi o Linux, distribuição OpenSuSe, versão 11.4, cuja instalação foi a padrão (com interface gráfica) desta distribuição. Entretanto, devido à necessidade de implementar algumas rotinas específicas para o propósito deste trabalho, o *kernel* padrão da distribuição foi trocado pelo *kernel* 2.6.32.28 obtido em www.kernel.org, o

qual foi customizado com uma chamada de sistema que é usada para configurar determinadas características, conforme o tratamento do experimento em questão.

Este *kernel* também foi compilado mantendo o recurso de *kernel tickless* habilitado. Este recurso permite que o *kernel* trabalhe com interrupções de *timer* em modo *one-shot* e não *periodic* (ver Seção 3.2.1) como acontece nas versões anteriores. Como a variável resposta é o tempo de processamento, o *kernel* customizado também foi compilado sem o recurso *CPU Frequency Scaling* (CONFIG_CPU_FREQ), o qual permite ao sistema operacional alterar a frequência de operação do processador dinamicamente. Nos experimentos, esse recurso habilitado causa um aumento na variabilidade dos tempos de execução [Mazou et al. 2010]. É importante ressaltar que, nos trabalhos anteriores, não se observa o uso de configurações específicas tanto para *kernel tickless* quanto para *CPU Frequency Scaling*.

5.3 – Plano Experimental

Nos experimentos deste trabalho, definiu-se que o tempo total (*turnaround time*) de processamento de um programa de multiplicação de matrizes é a variável resposta avaliada em cada tratamento. O programa que faz essa multiplicação de matrizes, aqui denominado programa de prova, no ambiente padrão de testes possui tempo médio de execução de aproximadamente 10 minutos e sempre ocupa 12248kB (11,96MB) de memória. O tempo de processamento do programa de prova foi medido em nanossegundos usando a rotina *clock_gettime* [Bovet e Cesati 2000].

De forma experimental, verificou-se que as primeiras replicações apresentam maior variabilidade no tempo de processamento, pelo fato dos dados deste processo não estarem na memória. Deste modo, para cada tratamento foram realizadas 53 replicações, onde as três primeiras replicações de cada tratamento foram descartadas, pois são consideradas replicações de *warm up*, ou seja, execuções realizadas para induzir um regime de maior estabilidade nas execuções subsequentes. Como essas três execuções não são utilizadas na análise estatística dos resultados, a amostra dos tempos de execução é composta de 50 valores.

Para evitar a influência da execução de um tratamento na execução do próximo tratamento, acarretando a invalidação dos dados coletados, o sistema operacional é reiniciado entre a execução de um tratamento e outro (no final da execução das 53 replicações). Com o intuito de reduzir o tempo de reinicialização, foi usado o mecanismo de *fast reboot* do Linux, o qual é implementado por meio do recurso *kexec* [Kroah-Hartman 2007], [Hariprasad 2004]. O *kexec* permite a inicialização de um novo *kernel* "sobre" o *kernel* corrente. Com isso, a

inicialização do sistema não realiza a execução do *firmware* de inicialização (POST – *Power-On Self-Test*).

Os dados dos experimentos, não satisfazem todas as suposições exigidas pelo teste ANOVA, portanto, para verificar a ocorrência de diferenças significativas entre os tratamentos realiza-se uma comparação utilizando o teste não paramétrico denominado Kruskal-Wallis, com um nível de significância de 5%.

Assim, a definição do plano experimental foi elaborada objetivando-se avaliar sistematicamente e quantitativamente a contribuição de diversas fontes de *OS Jitter*. Para tanto, utilizou-se a metodologia DOE [Montgomery 2005], o teste estatístico Kruskal-Wallis [Campos 1979] e o Método da Tabela de Sinais [Filho 2008], todos descritos no Capítulo 4. As fontes de *OS Jitter* selecionadas para avaliação foram baseadas nos principais trabalhos na área, os quais foram apresentados no Capítulo 3.

O plano experimental é composto por dois experimentos, sendo que os principais objetivos almejados são:

- Experimento #1: Caracterização dos efeitos sem carga de fundo.
- Experimento #2: Caracterização dos efeitos com compartilhamento da *cache* L3, usando carga de fundo CPU-*Bound*.

5.3.1 – Experimento #1

Esse experimento é composto por cinco fatores, os quais foram codificados usando letras (A, B, C, D, E), sendo cada uma parametrizada em dois níveis (-/+). A seguir uma descrição de cada fator usado neste experimento:

A: Representa o *runlevel* do sistema operacional [Van Vugt 2006]. O *runlevel* caracteriza o ambiente de operação, definindo quais serviços (processos e módulos de *kernel*) precisam ser carregados na inicialização do sistema operacional. O tipo de *runlevel* adotado influencia na carga e nos serviços fornecidos pelo sistema, visto que existem diferentes processos/serviços em execução de um *runlevel* para outro. Segundo [Petersen e Haddad 2003] e [Smith 2011], o padrão Linux define seis estados de *runlevel*:

- Zero (0), é um estado transitório usado para finalizar/desligar o sistema;
- Um (1), conhecido como modo mono usuário. Apenas o usuário *root* tem acesso ao sistema e *scripts* de inicialização não são executados;
- Dois (2), varia conforme a distribuição Linux. Em algumas distribuições (ex. CentOS, Red Hat, OpenSuse), é definido como modo multiusuário

onde não há serviços de rede (ex. NFS, xinetd, NIS), ou seja, é o mesmo do *runlevel 3,* porém, sem rede. Já em distribuições como Ubuntu e Debian, esse estado é definido como multiusuário completo com interface gráfica (mesmo do *runlevel 5* no OpenSuSe);

- Três (3), é um modo multisuário completo com interface de acesso via linha de comando permitindo serviços de rede como compartilhamento de arquivos. Devido à ausência de interface gráfica, também é conhecido como modo texto;
- Quatro (4), esse estado não é definido, portanto, disponível para customizações;
- Cinco (5), nas distribuições como CentOS, Red Hat, OpenSuSe, esse estado é definido como o mesmo do *runlevel* 3 contendo um complemento de uma interface gráfica (ex. KDE);
- Seis (6), é um estado transitório usado para reiniciar o sistema.

Como um dos objetivos deste trabalho é avaliar a interferência causada pela influência de processos do sistema sobre o programa de prova, então, um fator denominado *runlevel* foi usado, considerando os níveis cinco (5) e um (1). O primeiro, representado pelo nível (-) denota um sistema com maior interferência de processos do sistema operacional. O segundo, representado pelo nível (+), indica uma menor interferência dos processos do sistema.

B: Representa os *timers* de software do sistema operacional. Eles são usados para permitir a execução programada de rotinas de *kernel*, ou seja, após um dado intervalo de tempo da criação do *timer*, este expira e uma determinada rotina é executada no nível de *kernel*. Os *timers* são usados tanto por processos do usuário como pelo próprio *kernel* e, como visto no Capítulo 3 (ex. [Jones et al. 2003] e [Mann e Mittaly 2009]), são considerados uma fonte de *OS Jitter*. Neste trabalho, foram adotados dois níveis para este fator: 1) o nível (-) é definido quando não há *timers* no processador em que o programa de prova está sendo executado (PU#1). Desse modo, para implementar o nível (-) desse fator, é necessário garantir que não existam *timers* programados para execução no processador do programa de prova. Para isso, uma chamada de sistema, *sys_conf,* foi criada para mover os *timers* programados a serem executados na PU#1 para a PU#0. Portanto, ao iniciar, o programa de prova chama *sys_conf* antes de iniciar a rotina de multiplicação de

matrizes. 2) no nível (+) do fator B, os *timers* programados para executar na PU#1 não são movidos para outra PU.

C: Representa as ocorrências de interrupção de hardware (IRQ) que podem ou não ser manipuladas pelo processador onde o programa de prova está em execução. No nível (-) deste fator, o processador onde o programa de prova está sendo executado não trata requisições de interrupção (essa ação é realizada por outro processador – PU#0). Já no nível (+), as requisições de interrupção são todas manipuladas apenas pelo processador onde o programa de prova está em execução (PU#1). Para isso, empregou-se a funcionalidade de *SMP IRQ affinity* do *kernel* do Linux [Bovet e Cesati 2000].

D: Representa a afinidade de processador (*processor affinity*) dos processos do sistema. No projeto experimental, o nível (-) deste fator determina que não há afinidade, ou seja, todos os processos em execução podem ser executados em qualquer processador. Já o nível (+) define que todos os processos do sistema, com exceção do programa de prova e processos relacionados ao experimento, executem apenas na PU#0. Dessa forma, no nível superior (+) tem-se uma menor interferência por parte dos demais processos sobre o programa de prova.

E: Representa a interrupção de *timer*. Nesse caso, o nível (-) foi definido como sendo sem interrupção de *timer* no processador em que o programa de prova está sendo executado, PU#1. Já o nível (+) indica que a interrupção de *timer* ocorre normalmente em todos os processadores, como é o padrão no *kernel* do Linux.

A Tabela 5.1 apresenta um resumo dos fatores e seus níveis usados no Experimento #1. Na sequência, a Tabela 5.2 apresenta todos os tratamentos, ou seja, a combinação de todos os fatores e níveis usados na execução do Experimento #1, como também as possíveis interações entre fatores. Como resultado, tem-se 32 tratamentos (projeto fatorial completo 2^5) onde as 5 primeiras colunas referem-se aos fatores principais (A, B, C, D, E) e as 26 colunas subsequentes referem-se às interações destes fatores.

Tabela 5.1 Fatores e níveis do Experimento #1

			Níveis	
Fatores	**A**	*Runlevel*	-	+
	B	*Timers* de software	-	+
	C	IRQs	-	+
	D	Afinidade de Processador	-	+
	E	Interrupção de *timer*	-	+

Tabela 5.2 Matriz de contraste para o projeto do Experimento #1

	Fatores Principais					**Interações dos Fatores**																									
	A	**B**	**C**	**D**	**E**	**AB**	**AC**	**AD**	**AE**	**BC**	**BD**	**BE**	**CD**	**CE**	**DE**	**ABC**	**ABD**	**ACD**	**ABE**	**ACE**	**ADE**	**BCD**	**BCE**	**BDE**	**CDE**	**ABCD**	**ABCE**	**ABDE**	**ACDE**	**BCDE**	**ABCDE**
01	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	-
02	+	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	+	+
03	-	+	-	-	-	-	+	+	+	-	-	-	+	+	+	+	+	-	+	-	-	+	+	+	-	-	-	-	+	-	+
04	+	+	-	-	-	+	-	-	-	-	-	-	+	+	+	-	-	+	-	+	+	+	+	+	-	+	+	+	-	-	-
05	-	-	+	-	-	+	-	+	+	-	+	+	-	-	+	+	-	+	-	+	-	+	+	-	+	-	-	+	-	-	+
06	+	-	+	-	-	-	+	-	-	-	+	+	-	-	+	-	+	-	+	-	+	+	+	-	+	+	+	-	+	-	-
07	-	+	+	-	-	-	-	+	+	+	-	-	-	-	+	-	+	+	+	+	-	-	-	+	+	+	+	-	-	+	-
08	+	+	+	-	-	+	+	-	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	+	+	-	-	+	+	+	+
09	-	-	-	+	-	+	+	-	+	+	-	+	-	+	-	-	+	+	-	-	+	+	-	+	+	-	+	-	-	-	+
10	+	-	-	+	-	-	-	+	-	+	-	+	-	+	-	+	-	-	+	+	-	+	-	+	+	+	-	+	+	-	-
11	-	+	-	+	-	-	+	-	+	-	+	-	-	+	-	+	-	+	+	-	+	-	+	-	+	+	-	+	-	+	-
12	+	+	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	+	+	+
13	-	-	+	+	-	+	-	-	+	-	-	+	+	-	-	+	+	-	-	+	+	-	+	+	-	+	-	-	+	+	-
14	+	-	+	+	-	-	+	+	-	-	-	+	+	-	-	-	-	+	+	-	-	-	+	+	-	-	+	+	-	+	+
15	-	+	+	+	-	-	-	-	+	+	+	-	+	-	-	-	-	-	+	+	+	+	-	-	-	-	+	+	+	-	+
16	+	+	+	+	-	+	+	+	-	+	+	-	+	-	-	+	+	+	-	-	-	+	-	-	-	+	-	-	-	-	-
17	-	-	-	-	+	+	+	+	-	+	+	-	+	-	-	-	-	-	+	+	+	-	+	+	+	+	-	-	-	-	+
18	+	-	-	-	+	-	-	-	+	+	+	-	+	-	-	+	+	+	-	-	-	-	+	+	+	-	+	+	+	-	-
19	-	+	-	-	+	-	+	+	-	-	-	+	+	-	-	+	+	-	-	+	+	+	-	-	+	-	+	+	-	+	-
20	+	+	-	-	+	+	-	-	+	-	-	+	+	-	-	-	-	+	+	-	-	+	-	-	+	+	-	-	+	+	+
21	-	-	+	-	+	+	-	+	-	-	+	-	-	+	-	+	-	+	+	-	+	+	-	+	-	-	+	-	+	+	-
22	+	-	+	-	+	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	+	-	+	-	+	+
23	-	+	+	-	+	-	-	+	-	+	-	+	-	+	-	-	+	+	-	-	+	-	+	-	-	+	-	+	+	-	+
24	+	+	+	-	+	+	+	-	+	+	-	+	-	+	-	+	-	-	+	+	-	-	+	-	-	-	+	-	-	-	-
25	-	-	-	+	+	+	+	-	-	+	-	-	-	-	+	-	+	+	+	+	-	+	+	-	-	-	-	+	+	+	-
26	+	-	-	+	+	-	-	+	+	+	-	-	-	-	+	+	-	-	-	-	+	+	+	-	-	+	+	-	-	+	+
27	-	+	-	+	+	-	+	-	-	-	+	+	-	-	+	+	-	+	-	+	-	-	-	+	-	+	+	-	+	-	+
28	+	+	-	+	+	+	-	+	+	-	+	+	-	-	+	-	+	-	+	-	+	-	-	+	-	-	-	+	-	-	-
29	-	-	+	+	+	+	-	-	-	-	-	-	+	+	+	+	+	-	+	-	-	-	-	-	+	+	+	+	-	-	+
30	+	-	+	+	+	-	+	+	+	-	-	-	+	+	+	-	-	+	-	+	+	-	-	-	+	-	-	-	+	-	-
31	-	+	+	+	+	-	-	-	-	+	+	+	+	+	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	+	-
32	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

5.3.2 – Experimento #2

Este experimento é composto por seis fatores distribuídos em dois níveis cada (ver Tabela 5.3). Dentre esses fatores, os cinco primeiros são os mesmos do Experimento #1, ou seja, a diferença encontra-se no sexto fator.

O sexto fator (F) é o compartilhamento da *cache* L3 entre o programa de prova e um programa que implementa uma carga de fundo, do tipo CPU-*Bound*. Nesse fator, o nível (-) é definido como sendo a existência da carga de fundo apenas na PU#6, onde não há compartilhamento da *cache* com o programa de prova (ver Figura 5.1). No nível (+) ocorre o

compartilhamento da *cache* L3 entre o programa de prova e o programa de carga de fundo que executa na PU#2. A carga de fundo foi implementada também como uma multiplicação de matrizes de tamanho 1000x1000. Este programa executa ininterruptamente durante toda a execução do programa de prova. O programa de carga de fundo ocupa 12228kB (11,94MB) de memória. Tanto o programa de prova quanto o de carga de fundo, são grandes o suficiente para preencher a memória *cache* L3 inteira (12MB), o que significa que, ao avaliar o cenário com a *cache* compartilhada (fator F no nível +), ambos os processos competem por toda a memória *cache* L3.

Tabela 5.3 Fatores e níveis do Experimento #2

			Níveis	
Fatores	**A**	*Runlevel*	-	+
	B	*Timers* de software	-	+
	C	IRQs	-	+
	D	Afinidade de Processador	-	+
	E	Interrupção de *timer*	-	+
	F	*Shared Cache*	-	+

O Apêndice A apresenta todos os tratamentos avaliados no Experimento #2. A tabela inserida no Apêndice A possui um total de 64 tratamentos (linhas), sendo que, nas 6 primeiras colunas estão os fatores principais (A, B, C, D, E, F) e as 57 colunas subsequentes referem-se às interações destes fatores.

5.3.3 – Execução do Experimento

Para evitar a mínima intervenção humana durante a execução dos experimentos, estes foram automatizados por meio de um *script* (ver Figura 5.3). Durante a fase de *boot*, esse *script* é executado pelo sistema operacional logo após a inicialização dos serviços do sistema para o *runlevel* avaliado. Para isso, uma linha foi adicionada no final dos arquivos */etc/init.d/after.local* e */etc/init.d/single* para executar o *script* criado a fim de realizar os experimentos. Assim, a inicialização do computador sempre executa os seguintes passos: 1) carrega o *kernel* do sistema operacional, 2) inicializa os serviços do sistema operacional, 3) executa o *script* para iniciar e/ou continuar um experimento.

A lógica funcional do *script* está representada pelo fluxograma da Figura 5.3. Inicialmente, na execução do *script*, é verificado o estado/conteúdo do arquivo que armazena o ID do tratamento (*id_trat*) e, nesse caso, há três situações: 1) arquivo não existe, ou seja, já executou todos os tratamentos do experimento e o *script* finaliza a execução liberando o terminal para o usuário; 2) arquivo existe, entretanto, está vazio, implicando que durante a

execução de um determinado tratamento ocorreu algum erro (ex. queda de energia) e o computador foi reiniciado sem gravar o ID do próximo tratamento. Assim, o "terminal" fica bloqueado com uma mensagem de erro esperando a intervenção do usuário para reexecutar o tratamento em que aconteceu a falha; 3) arquivo existe e está preenchido com um valor inteiro positivo (ID) e, dessa forma, passa-se ao próximo passo que é a identificação pelo ID da configuração dos níveis de cada fator que compõem esse tratamento. Para isso, dentro da pasta de cada um dos experimentos executados, existe um arquivo (*lista_trat*) do tipo CSV (*Comma Separated Values*) que contém *n* linhas e *x* colunas. Cada linha desse arquivo representa um determinado tratamento e cada coluna representa um fator em seu respectivo nível. Os níveis (-) e (+) dos fatores estão representados, respectivamente, por zero (0) e um (1). Assim, a Figura 5.2 exemplifica o formato deste arquivo apresentando um fragmento do conteúdo usado no Experimento #1, o qual foi gerado com base na matriz de contraste do referido experimento (ver Tabela 5.2), sendo a primeira coluna representando o fator A (*runlevel*), a segunda coluna o fator B (*timers* de software) e assim sucessivamente.

01	0;0;0;0;0
02	1;0;0;0;0
03	0;1;0;0;0
04	1;1;0;0;0
	...
29	0;0;1;1;1
30	1;0;1;1;1
31	0;1;1;1;1
32	1;1;1;1;1

Figura 5.2 Fragmento do arquivo CSV do Experimento #1

Na fase de identificação dos parâmetros, existem duas opções: 1) o ID do tratamento não existe no arquivo *lista_trat*, ou seja, o valor numérico do ID é maior do que a quantidade de linhas do arquivo *lista_trat.* Assim, representando a conclusão da execução de todas as replicações do experimento em questão, permitindo então a remoção do arquivo *id_trat* e a reinicialização do computador; 2) a linha correspondente ao ID do tratamento foi encontrada, o que possibilita a leitura dos níveis de cada fator correspondente a este tratamento, prosseguindo para a próxima fase de configuração das variáveis do *script*.

Posteriormente, inicia-se a configuração dos fatores nos respectivos níveis do tratamento em questão. O primeiro a ser verificado é o *runlevel*. Caso o *runlevel* corrente seja diferente do *runlevel* em que o tratamento deve ser executado, então, o *runlevel* correspondente é configurado no arquivo */etc/inittab* e o sistema operacional é reinicializado. Caso o *runlevel* corrente esteja correto, dá-se sequência ao próximo passo. Posteriormente, conforme o nível do fator IRQ no tratamento em questão, é atribuído afinidade a todas as

IRQs do sistema (*/proc/irq/*/smp_affinity)*. A seguir, identifica-se o experimento que está em execução e, caso seja o Experimento #2, a carga de fundo é carregada em um determinado processador conforme o nível do fator F.

Seguindo os procedimentos realizados pelo *script* de automação dos experimentos, tem-se a execução de um programa denominado *ipcserver*. Esse programa é responsável por capturar (*clock_gettime*), em um vetor na memória, o tempo gasto pela fase de processamento do programa de prova em cada replicação. Com o objetivo de evitar a influência do *ipcserver* nos resultados dos experimentos, ele sempre executa no mesmo processador (*sched_setaffinity*), nesse caso, no processador PU#7, o qual por meio do recurso *isolcpus* [Kroah-Hartman 2007] foi "isolado", ou seja, o referido processador foi removido da lista de processadores disponíveis para o escalonador de processos.

A próxima etapa do *script* consiste em executar as 53 replicações do programa de prova. Este programa foi desenvolvido em linguagem C e efetua uma multiplicação de matrizes, 48 vezes por replicação, a qual em nosso ambiente de teste tem duração total aproximada de 10 minutos. As matrizes envolvidas na multiplicação são do tipo *long int* e cada uma possui tamanho de 1000x1000. A Figura 5.3 ilustra o fluxo de execução de um tratamento. Observa-se que entre as replicações existe um intervalo de 3 segundos. Esse procedimento proporciona ao sistema operacional tempo para liberar as estruturas alocadas/usadas na execução da replicação corrente.

Ao final das replicações, *o script* envia um sinal (*SIGINT*) ao processo *ipcserver*, que, ao receber, descarrega em um arquivo texto os tempos de processamento da multiplicação de matrizes que foram previamente armazenados na memória por meio de um vetor. Em seguida, o *ipcserver* configura o ID do próximo tratamento e o computador é reiniciado usando *kexec* e inicia-se um novo ciclo de execuções. Tal processo se repete até que todos os tratamentos do experimento em questão tenham sido completamente executados.

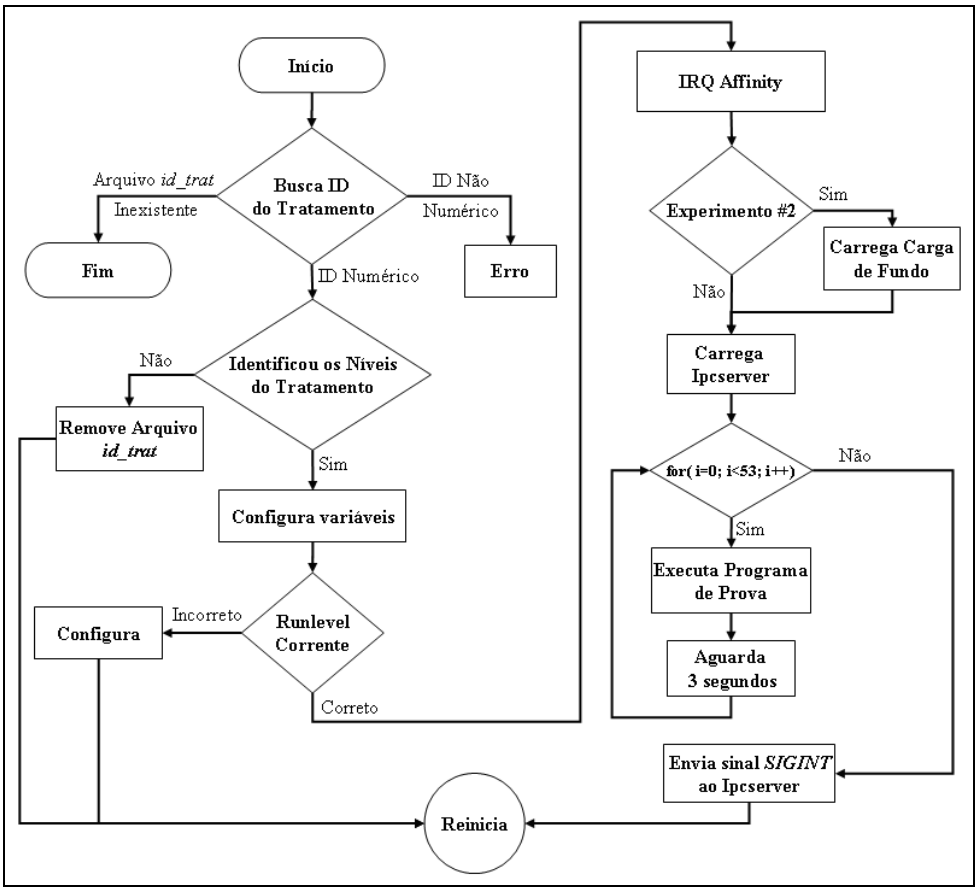


Figura 5.3 Fluxograma do *script* de automação de execução de tratamentos

5.3.4 – Programa de Prova

Ao iniciar, o programa de prova lê os parâmetros de execução que são usados para configurar os fatores *timers* de software (B), afinidade de processador (D) e interrupção de *timer* (E). Depois, através da chamada de sistema *sched_setaffinity*, esse processo se configura para sempre ser executado na PU#1. Em seguida, uma comunicação interprocessos entre o programa de prova e o *ipcserver* é configurada, sendo o *ipcserver* responsável por capturar e gravar o tempo de processamento da multiplicação de matrizes.

Posteriormente, é alocada dinamicamente 11,94MB de memória que está distribuída em três matrizes que serão usadas na operação de multiplicação de matrizes.

A seguir, conforme os parâmetros de execução, os fatores são configurados nos respectivos níveis do tratamento em questão. Quando o fator afinidade de processador (D) está setado no nível (+), aos processos/*threads* que não estão relacionados ao experimento é atribuído uma afinidade de processador para a PU#0. Se o fator *timers* de software (B) está configurado no nível (-), então, todos os *timers* de software da PU#1 são movidos para a PU#0. O último fator a ser configurado é a interrupção de *timer* (E), que quando está no nível

(-) não há interrupção de *timer* na PU#1. O ajuste desses últimos três fatores dá-se por meio da execução de uma chamada de sistema (*sys_conf*) projetada e implementada especificamente para isso.

Antes de iniciar o processamento de multiplicação de matrizes, utilizou-se a comunicação interprocessos que foi previamente estabelecida para sinalizar ao *ipcserver* o início do processamento e, assim, o *ipcserver* capturar o tempo inicial (*clock_gettime*). O processamento de multiplicação de matrizes (ver Figura 5.5) é realizado e, em seguida, uma nova sinalização é feita ao *ipcserver* fazendo com que ele grave o tempo total de processamento (T_{fim}-T_{ini}) em um vetor na memória. Como pode ser observado, o programa de prova foi estruturado sem considerar comunicação de rede, pois a intenção é medir a influência do *OS Jitter* apenas durante a fase de processamento, ou seja, em um ambiente de *Cluster* não consideramos o *OS Jitter* proveniente da distribuição das tarefas (ex. pilha do protocolo de rede).

Por fim, se a interrupção de *timer* foi desabilitada então, agora, ela é habilitada novamente; desaloca a memória alocada inicialmente e desmonta o compartilhamento de memória.

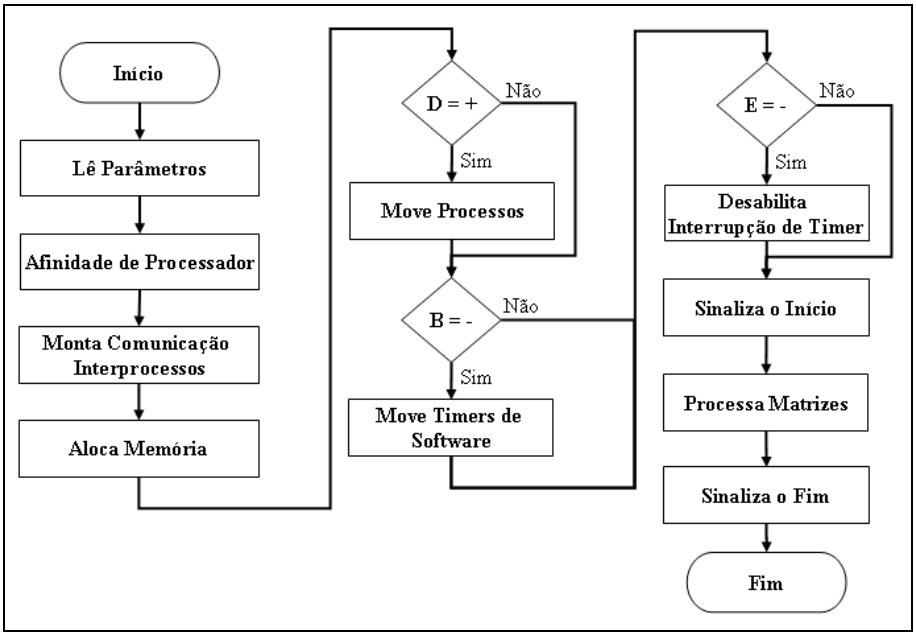


Figura 5.4 Fluxograma do programa de prova

Deste modo, as Figuras 5.4 e 5.5 representam, respectivamente, o fluxograma e as principais linhas do algoritmo usado pelo programa de prova.

```
01  Max: tamanho das matrizes
02  MultMat: numero de vezes para multiplicar as matrizes
03  M, R, T: matrizes de processamento
04  i, j, k, z: contadores
05
06  sched_setaffinity(PU#1);
07  MountIPC; monta comunicação interprocessos com o ipcserver
08  Max=1000; MultMat=48;
09  AlocMem(M,R,T); aloca [Max*4, Max*4] bytes de memória
10  SetsFactors(D,B,E); configura o nível dos fatores D, B e E
11  GetsTimeI; ipcserver pega o tempo inicial
12  for (z = 0; z < MultMat; z++)
13    for (j = 0; j < Max; j++)
14      for (i = 0; i < Max; i++)
15        for (k = 0; k < Max; k++)
16          if  (z == 0) then
17            R[j][i] = R[j][i] +  (M[j][k] * M[k][i]);
18          else
19            R[j][i] = R[j][i] + (T[j][k] * M[k][i]);
20    if  (z < (MultMat - 1)) then
21      for (j = 0; j < Max; j++)
22        for (i = 0; i < Max; i++)
23          T[j][i] = R[j][i];
24          R[j][i] = 0;
25        end for
26  end for
27  GetsTimeF; ipcserver pega o tempo final
28  if  ( o fator E esta no nível - ) then configura o fator E no nível +;
29  FreeMem(M,R,T); desaloca a memória
30  UmountIPC; Desmonta comunicação interprocessos com o ipcserver
```

Figura 5.5 Algoritmo do programa de prova

CAPÍTULO 6 – ANÁLISE DOS RESULTADOS

6.1 – Introdução

No Capítulo 5, foram descritos os experimentos realizados neste trabalho. A amostra de dados gerada em cada tratamento é de 50 tempos de execução. No total, no Experimento #1 tem-se uma amostra de 1600 valores e no Experimento #2 de 3200 valores. Com base em análises gráficas e numéricas (desvio padrão) de cada conjunto de dados dos tratamentos de ambos os experimentos (#1 e #2), foram identificados e removidos todos os valores considerados *outliers* (valores discrepantes). Esse procedimento de remoção substitui um *outlier* por outro valor que é calculado tendo como base a média dos demais valores (não-*outliers*) do conjunto de dados do tratamento em questão. Para ambos os experimentos, o número de observações *outliers* é muito pequeno. O Experimento #1 apresentou um total de 2,68% de *outliers* e não mais do que 4 *outliers* por tratamento, o que significa no máximo 8% de *outliers* em um conjunto de 50 replicações, em um dado tratamento. No Experimento #2, os resultados são bastante semelhantes aos do Experimento #1, com um total de 2,15% *outliers* e não mais do que 4 *outliers* por tratamento. O restante deste Capítulo é dedicado para a análise gráfica e numérica destes dados.

6.2 – Resultados do Experimento #1

Como descrito anteriormente, inicialmente, realizou-se a identificação e tratamento de *outliers* no conjunto de dados de cada tratamento do experimento. Por meio da representação gráfica da média (ver Figura 6.1) e dos desvios padrões (ver Tabela 6.1), dos dados coletados em cada tratamento, pode-se observar que existem tratamentos com observações de valor extremo (*outlier*s). O critério para identificar os *outliers* do Experimento #1, usando a abordagem numérica, é baseado no desvio padrão, de forma que nos tratamentos onde a interrupção de *timer* está no nível (-) foi verificado um desvio padrão entre 0,003 e 0.004 e nos tratamentos em que esse fator está no nível (+), o desvio padrão está entre 0,003 e 0,007. Em conjunto com essa abordagem numérica, também utilizou-se a análise gráfica dos dados, de forma que os "picos/vales" muito acentuados e distantes dos demais são considerados *outliers*. Como não há, na literatura, estudos estatísticos detalhados sobre *OS Jitter*, com indicativos que poderiam nortear uma decisão neste sentido, adotou-se o seguinte critério: caso a quantidade total de *outliers* em cada tratamento fosse maior do que 10%, então, os dados do tratamento seriam descartados e uma nova execução realizada. De forma oposta, se

a quantidade total de *outliers* fosse menor ou igual a 10%, então, essas observações seriam ajustadas com o valor médio das outras observações.

Nos dados apresentados na Figura 6.1, não houve tratamento com a quantidade total de *outliers* maior ou igual a 10%. Os tratamentos que não apresentaram *outliers* foram o 2º, 3º e 32º. Os tratamentos com *outliers* foram ajustados para que os valores discrepantes fossem substituídos pela média dos valores não discrepantes.

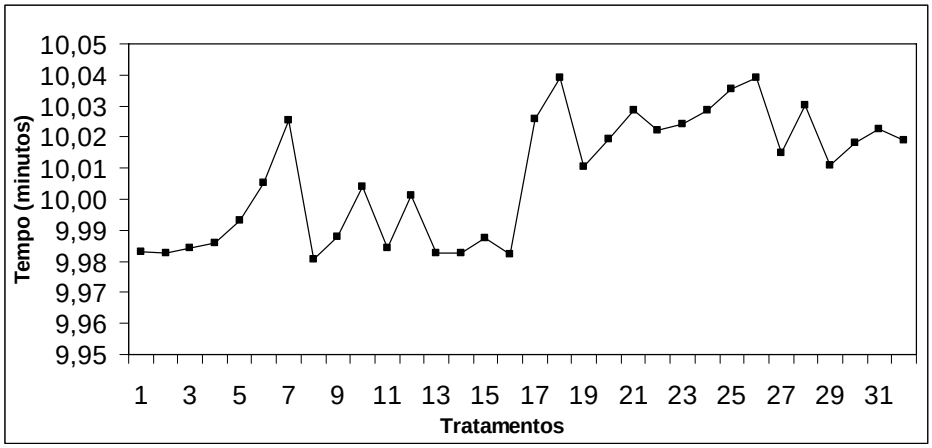


Figura 6.1 Valores médios de todos os tratamentos antes do ajuste

Um exemplo de identificação e tratamento de *outlier* é ilustrado pela Figura 6.2. Esta figura refere-se ao quinto tratamento, onde claramente as observações 39 e 48 apresentam valores discrepantes se comparados com as demais observações. Assim, o valor da observação 48 foi substituído pelo valor médio (ver Equação 6.1) das demais observações não discrepantes e, nesse caso, a média considera o valor de todas as observações exceto o valor das obervações 39 e 48. De forma análoga e subsequente, a observação 39 também foi substituída pelo valor médio de todas as observações não discrepantes, ou seja, considera-se todas as observações exceto a observação 39.

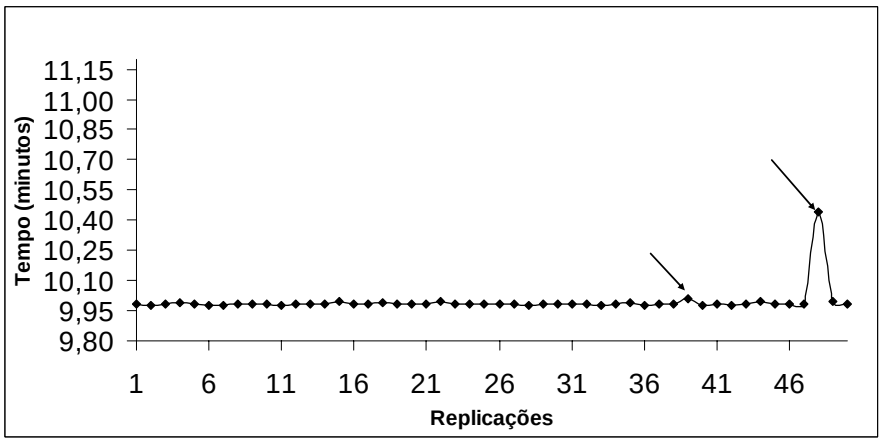


Figura 6.2 Replicações do quinto tratamento sem ajuste nos dados

$$\frac{\sum Observações\ não\ discrepantes}{Quantidade\ observações\ não\ discrepantes} \tag{6.1}$$

Outro exemplo pode ser verificado nas observações 11, 36 e 42 do sexto tratamento (ver Figura 6.3).

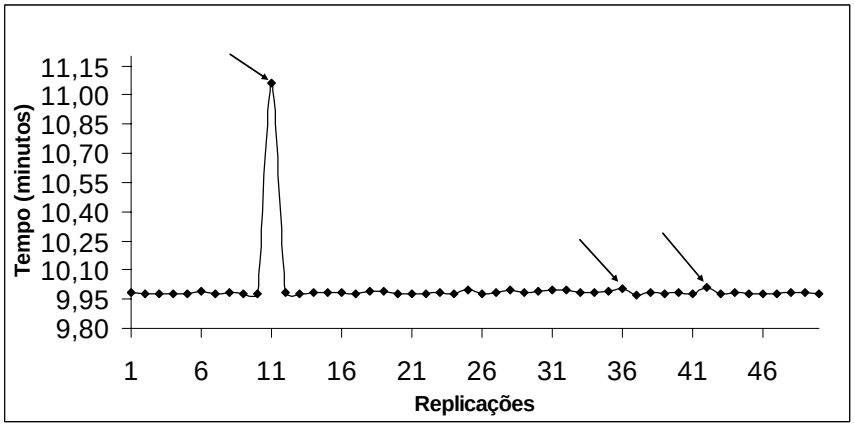


Figura 6.3 Replicações do sexto tratamento sem ajuste nos dados

As Figuras 6.4 e 6.5 apresentam como os dados do quinto e sexto tratamentos, respectivamente, ficaram dispostos após os ajustes implementados. Vale destacar que os dados não sofreram alisamento, preservando as mesmas características de variabilidade da amostra original.

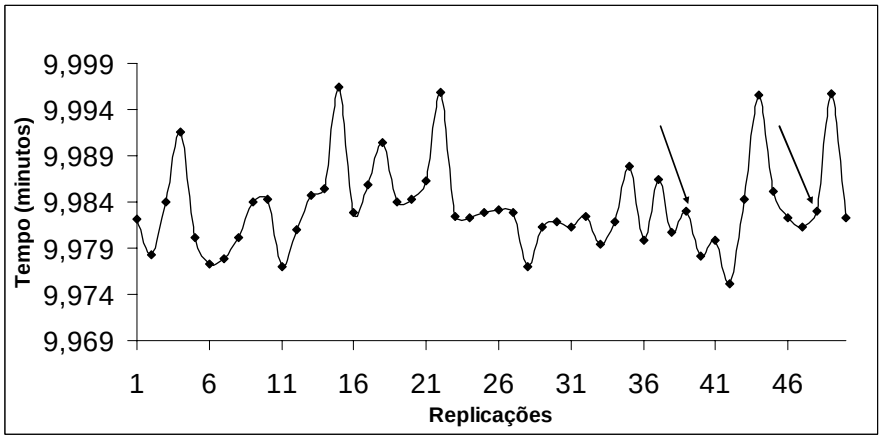


Figura 6.4 Replicações do quinto tratamento após ajustes nos dados

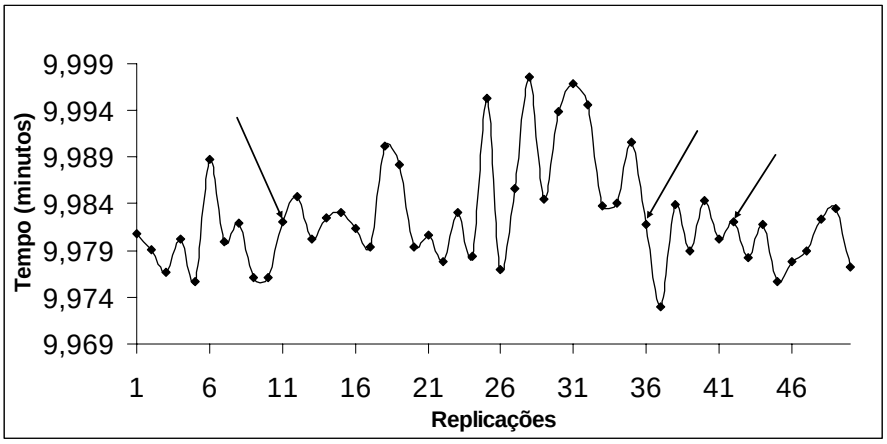


Figura 6.5 Replicações do sexto tratamento após ajustes nos dados

Com base na amostra completa de dados, sem *outliers*, pode-se observar pelo gráfico de médias (ver Figura 6.6) que a variabilidade da variável resposta, entre o primeiro e o décimo sexto tratamento, é bem menor do que entre o décimo sétimo e o trigésimo segundo tratamento.

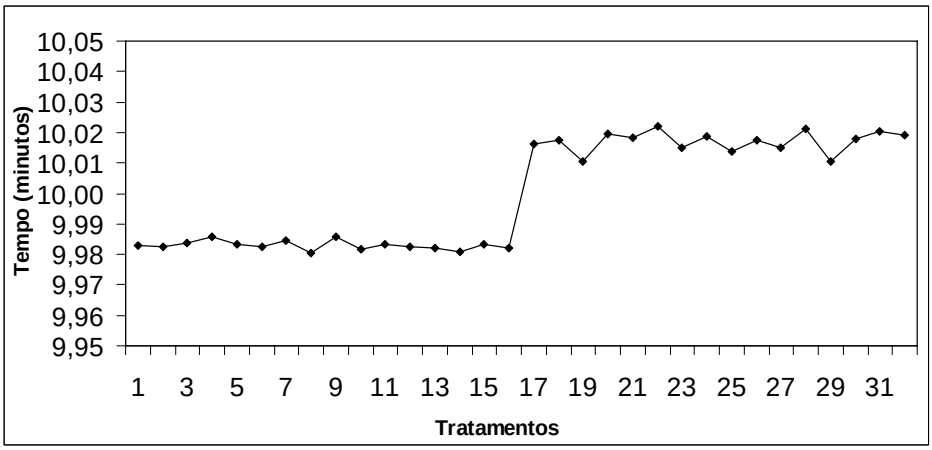


Figura 6.6 Valores médios dos tratamentos sem *outliers*

Também, pode-se observar que após o décimo sexto tratamento há um aumento significativo na média de tempo dos tratamentos. Assim, tem-se que até o décimo sexto tratamento o tempo médio encontra-se em torno de 9,983 minutos e, após o décimo sétimo tratamento, essa média passa para 10,017 minutos. Analisando o planejamento fatorial descrito na Tabela 5.2, entre o primeiro e o décimo sexto tratamento observa-se que o fator E (interrupção de *timer*) está no nível (-), indicando que não houve interrupção de *timer* na PU#1. Nos demais tratamentos deste experimento, o fator E assume o nível (+), indicando que a interrupção de *timer* estava habilitada. Assim, a influência do fator E sobre a variabilidade e sobre a média é evidente entre os dois subconjuntos de dados.

Tabela 6.1 Médias e desvios padrões dos dados ajustados e não ajustados

		Dados Não Ajustados		Dados Ajustados	
		Média	**Desvio Padrão**	**Média**	**Desvio Padrão**
Tratamentos	**01**	9,9832	0,0041	9,9829	0,0037
	02	9,9826	0,0039	9,9826	0,0039
	03	9,9841	0,0059	9,9836	0,0044
	04	9,9858	0,0038	9,9858	0,0038
	05	9,9932	0,0652	9,9835	0,0049
	06	10,0051	0,1526	9,9826	0,0058
	07	10,0252	0,1759	9,9847	0,0032
	08	9,9805	0,0039	9,9805	0,0039
	09	9,9878	0,0092	9,9859	0,0047
	10	10,0038	0,1525	9,9819	0,0033
	11	9,9842	0,0062	9,9831	0,0045
	12	10,0013	0,0916	9,9824	0,0044
	13	9,9828	0,0056	9,9823	0,0042
	14	9,9825	0,0075	9,9809	0,0030
	15	9,9875	0,0194	9,9835	0,0039
	16	9,9821	0,0038	9,9821	0,0038
	17	10,0257	0,0652	10,0165	0,0045
	18	10,0390	0,1532	10,0174	0,0031
	19	10,0106	0,0035	10,0106	0,0035
	20	10,0195	0,0041	10,0195	0,0041
	21	10,0286	0,0657	10,0184	0,0058
	22	10,0221	0,0089	10,0221	0,0089
	23	10,0241	0,0656	10,0149	0,0078
	24	10,0288	0,0650	10,0189	0,0037
	25	10,0353	0,1532	10,0137	0,0046
	26	10,0391	0,1530	10,0175	0,0042
	27	10,0149	0,0042	10,0149	0,0042
	28	10,0304	0,0650	10,0213	0,0054
	29	10,0111	0,0065	10,0103	0,0038
	30	10,0180	0,0036	10,0180	0,0036
	31	10,0225	0,0122	10,0206	0,0069
	32	10,0190	0,0035	10,0190	0,0035

Em seguida, baseando-se nos dados coletados, os efeitos dos fatores e suas interações foram calculados pelo procedimento descrito na Seção 4.4. Os resultados deste cálculo são apresentados na Tabela 6.2 que está organizada de forma decrescente listando a contribuição de cada fator/interação sobre a variação da variável resposta (tempo de processamento). Por essa tabela, verifica-se que 6,7% da variação do tempo de processamento do programa de prova não é explicada pelo planejamento fatorial proposto, o que pode ser atribuído ao erro experimental e/ou algum outro fator que não foi considerado no estudo. Observou-se que 91,23% da variação no tempo de processamento é causada pelo fator E (interrupção de *timer*). Outros fatores não mostram contribuições importantes quando comparados com a interrupção de *timer*.

Tabela 6.2 Percentual de contribuição de cada fator/interação sobre o tempo de processamento do programa de prova

Dados Ajustados	
Fator / Interação	**Contribuição**
E	91,2349%
AE	0,6046%
BDE	0,2707%
ABC	0,2512%
BD	0,1735%
A	0,1640%
CE	0,1102%
ABDE	0,1091%
BCD	0,1050%
AC	0,0376%
ACDE	0,0371%
ABD	0,0363%
CD	0,0352%
ABCE	0,0319%
CDE	0,0257%
B	0,0255%
D	0,0161%
ACD	0,0157%
ABCDE	0,0135%
AD	0,0101%
AB	0,0054%
ADE	0,0035%
BCE	0,0034%
BC	0,0034%
BE	0,0025%
C	0,0025%
ABCD	0,0006%
DE	0,0005%
ACE	0,0005%
ABE	0,0003%
BCDE	0,0000%

Legenda	
A	*Runlevel*
B	*Timers*
C	IRQ
D	Afinidade de processador
E	Interrupção de *timer*

Aplicando o teste de Kruskal-Wallis (ver Seção 4.3), tem-se que o valor de *H* calculado é igual a 1295,39. Como a amostra utilizada é relativamente grande, 32 tratamentos × 50 replicações, o valor crítico é dado por, aproximadamente, *k-1* graus de liberdade da distribuição Qui-Quadrado (X^2) ao nível de significância de 5%. Ao nível de significância de 5% e considerando 31 graus de liberdade, tem-se que o valor crítico da distribuição Qui-Quadrado é igual a 44,985 (ver Anexo A). Nesse caso, rejeita-se H_0, ou seja, ao nível de significância equivalente a 5% pode-se afirmar que pelo menos dois dos tratamentos diferem entre si. Para identificar quais tratamentos são considerados estatisticamente diferentes,

efetuou-se uma comparação estatística dois a dois entre tratamentos usando as Equações 4.2 e 4.3. O valor crítico calculado foi igual a 359,329. Assim, fazendo-se comparações dois a dois, observou-se que todos os tratamentos onde o fator E (interrupção de *timer*) estava no nível (-), correspondendo aos tratamentos 1 até 16, foram considerados estatisticamente diferentes dos tratamentos em que esse fator estava no nível (+), correspondendo aos tratamentos 17 até 32. O Apêndice B apresenta detalhadamente o resultado desta comparação dois a dois usando o teste de Kruskal-Wallis. Tais resultados numéricos corroboram a análise gráfica da Figura 6.6.

Salienta-se que, ao realizar a comparação dois a dois usando os dados não ajustados (ver Apêndice C), observou-se que em quase todas as situações onde o fator E estava no nível (-) os tempos de processamento também foram considerados estatisticamente diferentes dos tratamentos onde o nível (+) estava configurado, exceto na comparação entre o tratamento 07 e o tratamento 19. Neste caso, usando os dados ajustados (ver Apêndice B) esses tratamentos são considerados estatisticamente diferentes entre si.

6.3 – Resultados do Experimento #2

Assim como no Experimento #1, usando a representação gráfica da média (ver Figura 6.7) e dos desvios padrões (ver Tabela 6.5) de cada tratamento, observou-se valores discrepantes na variável resposta. Em relação à análise numérica usada para identificar os *outliers*, nos tratamentos onde o compartilhamento de *cache* está no nível (-) utilizou-se os mesmos critérios do Experimento #1 e no nível (+), verificou-se que o desvio padrão se encontra entre 0,02 e 0,03. Dessa forma, adotou-se os mesmos critérios de análise gráfica e substituição das observações *outliers* usadas no Experimento #1.

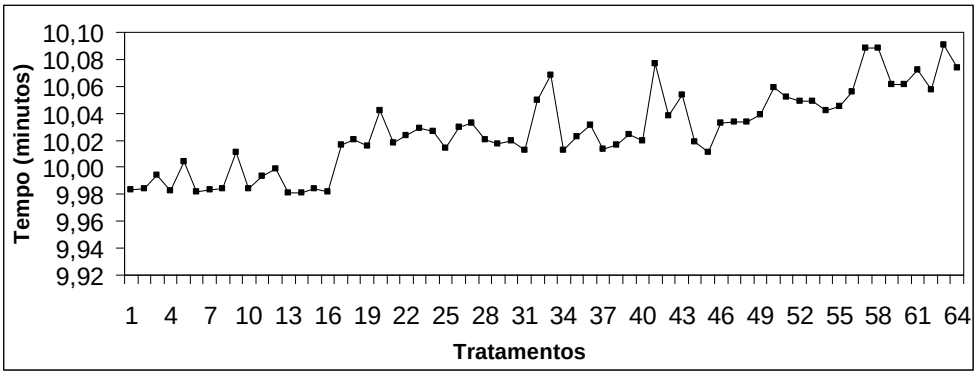


Figura 6.7 Valores médios de todos os tratamentos antes dos ajustes

No Experimento #2, também não houve nenhum tratamento com um número de *outliers* acima de 10%. Ao analisar os valores coletados em todas as 50 replicações de cada

tratamento, os *outliers* foram identificados e tratados. As Figuras 6.8 e 6.9 ilustram esse processo.

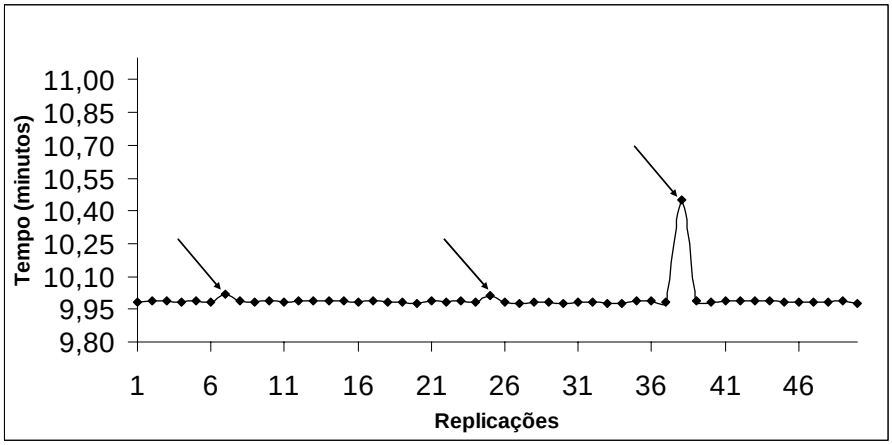


Figura 6.8 Replicações do terceiro tratamento antes dos ajustes dos dados

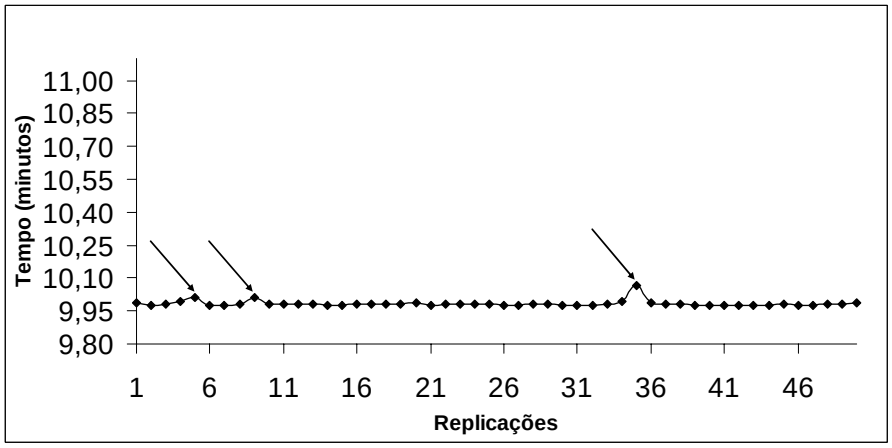


Figura 6.9 Replicações do quarto tratamento antes dos ajustes dos dados

As Figuras 6.10 e 6.11 apresentam, respectivamente, os dados do terceiro e quarto tratamentos após os devidos ajustes.

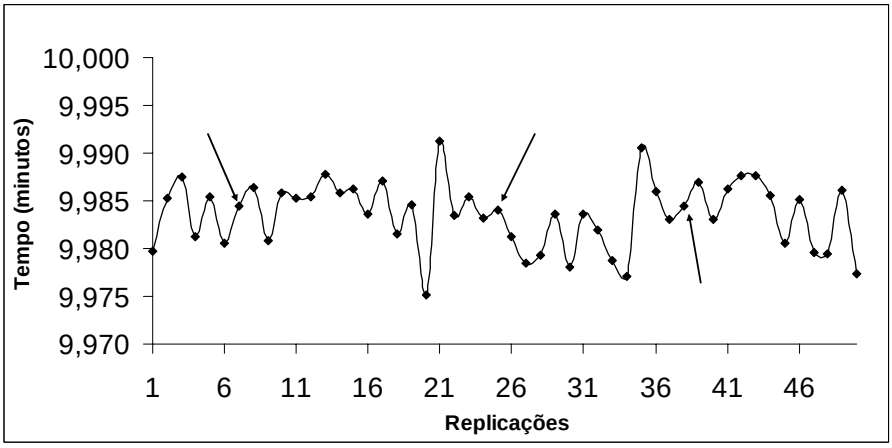


Figura 6.10 Replicações do terceiro tratamento após ajuste dos dados

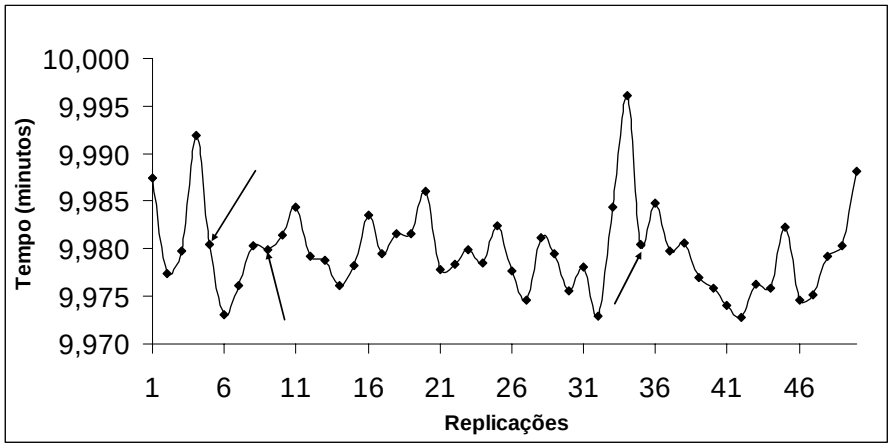


Figura 6.11 Replicações do quarto tratamento após ajuste dos dados

Um aumento na variabilidade do tempo de processamento pode ser observado no gráfico de médias (ver Figura 6.12), bem como um aumento do tempo de processamento conforme o nível de determinados fatores.

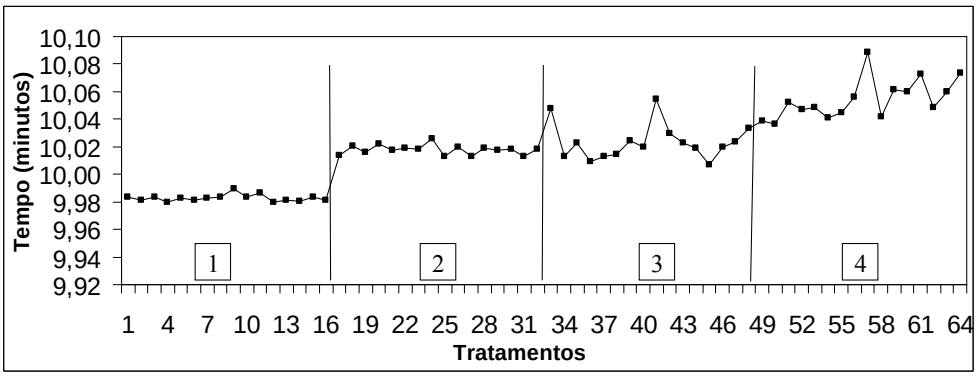


Figura 6.12 Valores médios de todos os tratamentos (separados por grupos) após ajustes

Baseando-se nessa observação, separou-se os tratamentos em quatro grupos (1º, 2º, 3º e 4º) contendo fatores em níveis semelhantes e, conforme Tabela 6.3, os tratamentos foram agrupados pelos fatores E (interrupção de *timer*) e F (*shared cache*). Assim, o 1º e o 2º grupos possuem os mesmos tratamentos avaliados no Experimento #1 e, nesse caso, os resultados obtidos são praticamente os mesmos discutidos anteriormente na Seção 6.2.

Tabela 6.3 Níveis dos fatores E e F nos grupos de tratamentos analisados

		Nível do Fator	
		E	**F**
Grupo	**1º**	-	-
	2º	+	-
	3º	-	+
	4º	+	+

Analisando graficamente (ver Figura 6.12), observa-se que o primeiro grupo de tratamentos é o que apresenta menor tempo de processamento. Ao se comparar o segundo com o terceiro grupo de tratamentos, observa-se que, em média e isoladamente, o fator interrupção de *timer* tem influência semelhante ao fator de compartilhamento de *cache* (F). Entretanto, comparando o desvio padrão destes dois grupos, o fator F aumenta em 444% a variabilidade no tempo de processamento. Ao examinar o quarto grupo de tratamentos em relação ao segundo e terceiro grupos, verifica-se que os fatores de interrupção de *timer* e compartilhamento de *cache* aumentam significativamente o tempo de processamento em comparação com os tratamentos onde ambos se encontram desligados (nível -). A Tabela 6.4 apresenta o tempo médio de processamento e o desvio padrão de cada grupo.

Tabela 6.4 Tempo médio de processamento e desvio padrão dos grupos

	1º Grupo	**2º Grupo**	**3º Grupo**	**4º Grupo**
Tempo médio (minutos)	9,9828	10,0179	10,0235	10,0544
Desvio padrão	0,0058	0,0056	0,0307	0,0285

Em seguida, a contribuição de cada fator/interação sobre a variável resposta foi calculada. Verificou-se que 33,90% da variação no tempo de processamento foi causada pelo fator F (*shared cache*). Em segundo lugar, o fator E (interrupção de *timer*) mostrou-se responsável por 24,75% da variação e, em terceiro lugar, a interação entre D e F (Afinidade de Processador e *shared cache*) correspondeu a 0,897% da variação da variável resposta. Logo, 32,84% da variabilidade total não pode ser explicada pelo projeto fatorial, provavelmente, devido a erros experimentais decorrentes da inclusão do fator F. A Tabela 6.6 apresenta todos os valores calculados.

Tabela 6.5 Média e desvio padrão dos dados ajustados e não ajustados

		Dados Não Ajustados		Dados Ajustados	
		Média	**Desvio Padrão**	**Média**	**Desvio Padrão**
Tratamentos	**1**	9,9833	0,0049	9,9833	0,0049
	2	9,9843	0,0155	9,9814	0,0040
	3	9,9941	0,0658	9,9836	0,0035
	4	9,9828	0,0142	9,9798	0,0046
	5	10,0045	0,1522	9,9830	0,0043
	6	9,9815	0,0044	9,9815	0,0044
	7	9,9836	0,0086	9,9827	0,0058
	8	9,9838	0,0053	9,9838	0,0053
	9	10,0110	0,1520	9,9895	0,0093
	10	9,9838	0,0054	9,9838	0,0054
	11	9,9931	0,0325	9,9869	0,0082
	12	9,9985	0,0922	9,9800	0,0041
	13	9,9810	0,0030	9,9810	0,0030
	14	9,9809	0,0062	9,9803	0,0044
	15	9,9844	0,0071	9,9837	0,0051
	16	9,9822	0,0059	9,9813	0,0039
	17	10,0164	0,0173	10,0134	0,0041
	18	10,0203	0,0057	10,0203	0,0057
	19	10,0158	0,0065	10,0158	0,0065
	20	10,0419	0,0920	10,0223	0,0036
	21	10,0180	0,0046	10,0177	0,0042
	22	10,0236	0,0240	10,0189	0,0040
	23	10,0293	0,0657	10,0187	0,0048
	24	10,0268	0,0056	10,0260	0,0041
	25	10,0141	0,0074	10,0132	0,0045
	26	10,0295	0,0650	10,0197	0,0043
	27	10,0326	0,0925	10,0134	0,0037
	28	10,0207	0,0144	10,0187	0,0039
	29	10,0177	0,0055	10,0177	0,0055
	30	10,0195	0,0060	10,0186	0,0035
	31	10,0128	0,0046	10,0128	0,0046
	32	10,0496	0,1648	10,0185	0,0039

Continua na próxima página

Tabela 6.5 Média e desvio padrão dos dados ajustados e não ajustados (Continuação)

		Dados Não Ajustados		Dados Ajustados	
		Média	**Desvio Padrão**	**Média**	**Desvio Padrão**
Tratamentos	**33**	10,0686	0,1488	10,0481	0,0317
	34	10,0129	0,0233	10,0129	0,0233
	35	10,0230	0,0244	10,0230	0,0244
	36	10,0310	0,1553	10,0092	0,0135
	37	10,0131	0,0208	10,0131	0,0208
	38	10,0166	0,0267	10,0144	0,0213
	39	10,0242	0,0220	10,0242	0,0220
	40	10,0197	0,0215	10,0197	0,0215
	41	10,0766	0,1661	10,0545	0,0552
	42	10,0385	0,0695	10,0300	0,0345
	43	10,0538	0,1700	10,0228	0,0267
	44	10,0191	0,0362	10,0191	0,0362
	45	10,0112	0,0301	10,0073	0,0224
	46	10,0325	0,0791	10,0202	0,0204
	47	10,0338	0,0736	10,0240	0,0248
	48	10,0335	0,0285	10,0335	0,0285
	49	10,0387	0,0150	10,0387	0,0150
	50	10,0589	0,1550	10,0361	0,0103
	51	10,0522	0,0168	10,0522	0,0168
	52	10,0488	0,0216	10,0472	0,0180
	53	10,0489	0,0261	10,0489	0,0261
	54	10,0424	0,0175	10,0412	0,0153
	55	10,0448	0,0181	10,0448	0,0181
	56	10,0559	0,0326	10,0559	0,0326
	57	10,0885	0,0343	10,0885	0,0343
	58	10,0887	0,2150	10,0417	0,0204
	59	10,0617	0,0314	10,0617	0,0314
	60	10,0617	0,0251	10,0600	0,0223
	61	10,0726	0,0348	10,0726	0,0348
	62	10,0574	0,0651	10,0484	0,0161
	63	10,0907	0,1670	10,0597	0,0302
	64	10,0735	0,0366	10,0735	0,0366

Fim

Tabela 6.6 Percentual de contribuição de cada fator/interação sobre o tempo de processamento do programa de prova

Dados Ajustados	
Fator / Interação	**Contribuição**
F	33,90%
E	24,76%
DF	0,90%
D	0,68%
AC	0,50%
ACF	0,44%
AB	0,43%
AF	0,43%
BC	0,42%
ABF	0,35%
BCE	0,35%
DEF	0,33%
BCF	0,27%
ADEF	0,26%
CE	0,25%
A	0,23%
BCEF	0,21%
ADE	0,20%
ABE	0,13%
ABDF	0,12%
ABCE	0,12%
BE	0,12%
ACE	0,12%
AEF	0,10%
DE	0,10%
ABDE	0,10%
EF	0,09%
ABDEF	0,09%
BCDF	0,09%
ABD	0,09%
BCD	0,08%
CEF	0,08%

Continua na próxima página

Legenda	
A	*Runlevel*
B	*Timers*
C	IRQ
D	Afinidade de Processador
E	Interrupção de *timer*
F	*Shared Cache*

Tabela 6.6 Percentual de contribuição de cada fator/interação sobre o tempo de processamento do programa de prova (Continuação)

Dados Ajustados	
Fator / Interação	**Contribuição**
CD	0,07%
AE	0,07%
ABCEF	0,07%
BEF	0,07%
ABEF	0,07%
ABCF	0,06%
BD	0,06%
BDE	0,05%
CF	0,05%
C	0,04%
ACD	0,03%
ABCDEF	0,03%
ABCD	0,02%
ACDF	0,02%
ABCDE	0,02%
ACDEF	0,02%
ACEF	0,01%
BCDEF	0,01%
AD	0,01%
ABC	0,01%
ABCDF	0,01%
B	0,01%
ACDE	0,01%
BDEF	0,01%
CDE	0,01%
CDF	0,00%
BDF	0,00%
BF	0,00%
BCDE	0,00%
ADF	0,00%
CDEF	0,00%

Fim

Legenda	
A	*Runlevel*
B	*Timers*
C	IRQ
D	Afinidade de Processador
E	Interrupção de *timer*
F	*Shared Cache*

Ao executar-se o teste de Kruskal-Wallis, identificou-se que o valor de *H* calculado é igual a 2473,53773. Avaliando-se o referido valor em relação ao valor crítico (82,528) obtido pela distribuição Qui-quadrado (ver Anexo A), considerando o nível de significância equivalente a 5% e 63 graus de liberdade, rejeitou-se H_0 em favor de H_1, ou seja, ao nível de significância de 5% é possível afirmar que pelo menos dois dos tratamentos diferem-se entre si.

Deste modo, efetuou-se a comparação dois a dois entre cada tratamento do referido experimento. Os resultados estão resumidamente demonstrados na Tabela 6.7. Para pares de tratamentos de um mesmo grupo não existem diferenças significativas, exceto entre os pares 33-34, 33-36 e 33-37 do terceiro grupo (7,5%). Ao comparar-se os tratamentos do segundo e terceiro grupos, apenas 4,3% das comparações são consideradas estatisticamente diferentes. As comparações entre os grupos de tratamentos 1º-2º, 1º-3º, 1º-4º, 2º-4º e 3º-4º mostraram diferenças estatisticamente significativas.

Tabela 6.7 Percentual de tratamentos considerados estatisticamente significantes – comparação entre grupos e tratamentos

		Grupos			
		1º	**2º**	**3º**	**4º**
Grupos	**1º**	0,0%	98.0%	97,3%	100,0%
	2º		0,0%	4,3%	75,4%
	3º			7,5%	66,8%
	4º				0,0%

Vale ressaltar que, confrontando-se os resultados da comparação dois a dois usando dados ajustados com os resultados dos dados não ajustados (ver Apêndice D e E), observou-se que apenas 1,7% não são iguais. Isso indica que os ajustes efetuados nos dados não descaracterizaram a amostra em relação aos dados originais.

6.4 – Simulação

Os resultados experimentais apresentados mostram os efeitos do *OS Jitter* no tempo de processamento de um programa executando em um único nó de computação. Com base nos referidos resultados, simulou-se o impacto destes efeitos em uma aplicação HPC sendo executada em vários nós de computação e composta por várias fases computacionais.

6.4.1 – Resultados da Simulação #1

Inicialmente, dois tratamentos (T10 e T23) do Experimento #1 foram selecionados. Em T10, todos os fatores de *OS Jitter* investigados estão configurados no nível de menor interferência (A(+), B(-), C(-), D(+) e E(-)) e, em T23, os fatores estão no nível de maior interferência (A(-), B(+), C(+), D(-) e E(+)). Com base nos tempos de processamento obtidos em ambos os tratamentos, melhor e pior casos, calculou-se a diferença entre T23 e T10 gerando um terceiro conjunto de dados. O novo conjunto de dados foi utilizado para obter a função de densidade de probabilidades (*pdf*) do atraso no tempo de processamento causado pelo *OS Jitter*. A *pdf* para os dados foi obtida por meio de um teste de aderência de Kolmogorov-Smirnov (K-S) [Corder e Foreman 2011], considerando um nível de confiança

de 95%. Verificou-se pelo resultado do K-S que esta amostra segue uma distribuição Gaussiana na forma: N(μ = 1,912, σ = 0,3665). A Figura 6.13 apresenta a análise gráfica do resultado do teste K-S.

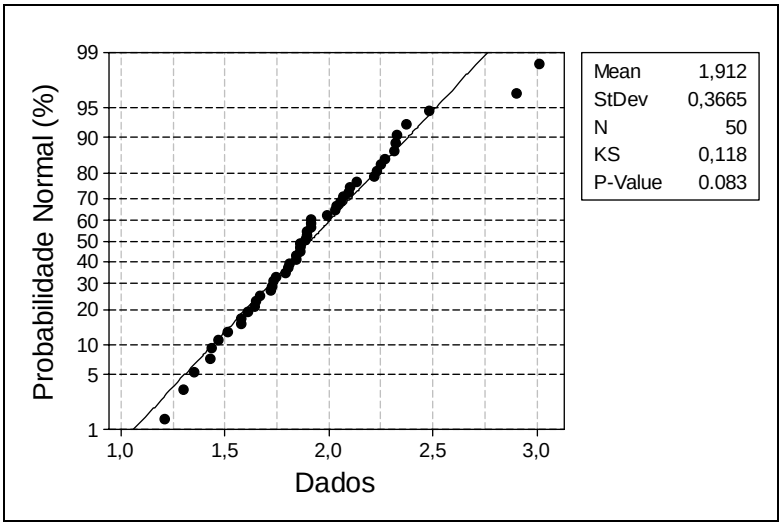


Figura 6.13 Probabilidade normal usando K-S para dados da simulação

Em seguida, utilizou-se a *pdf* estimada para simular as ocorrências de atrasos em cada fase computacional, por processo do programa de prova, em múltiplos nós de computação. Para tanto, variou-se o número de fases computacionais por processo (de 1 até 200) e o número de processos (de 1 até 500). Além disso, considerou-se que em cada nó há apenas um processo, ou seja, variando o número de processos significa variar o número de nós de computação.

Os resultados da simulação mostram que, ao variar a quantidade de processos, para qualquer quantidade de fases, o tempo de processamento da aplicação aumenta logaritmicamente. A Figura 6.14 ilustra este comportamento.

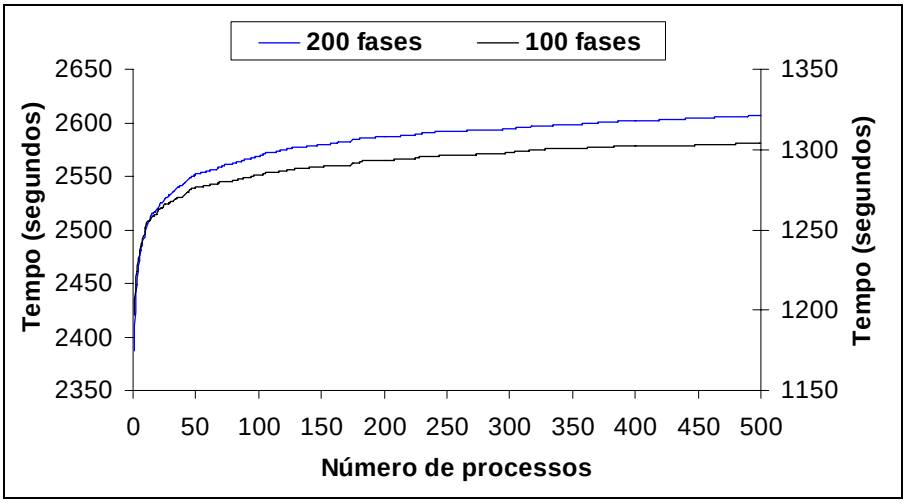


Figura 6.14 Efeitos do *OS Jitter* para 100 e 200 fases com múltiplos processos

Na Figura 6.14, para diferentes números de fases e poucos processos (ex. <20), observa-se que o crescimento da curva é bastante acentuado. Para um número maior de processos, o aumento do tempo de processamento da aplicação tende a moderar-se. Tal fato justifica-se, pois com poucos processos participando em cada fase, a chance de que em cada fase algum desses processos receba influência próximo ao máximo *OS Jitter* possível é pequena. Já quando aumenta-se a quantidade de processos em cada fase (ex. > 20), é muito provável que algum desses processos, em cada fase, sofra com os efeitos próximos ao máximo permitido pela distribuição do *OS Jitter*. Assim, à medida que a quantidade de processos (ex. 30, 40, 50, 100) aumenta, o *OS Jitter* acumulado de uma quantidade de processos para outra se encontra muito próximo do limite máximo regido pela distribuição e, isso, induz um crescimento menor e mais suave no tempo de processamento do que quando se tem poucos processos por fase.

Diferentemente das observações para múltiplos processos, quando aumenta-se o número de fases, o tempo de processamento da aplicação aumenta linearmente como ilustrado na Figura 6.15.

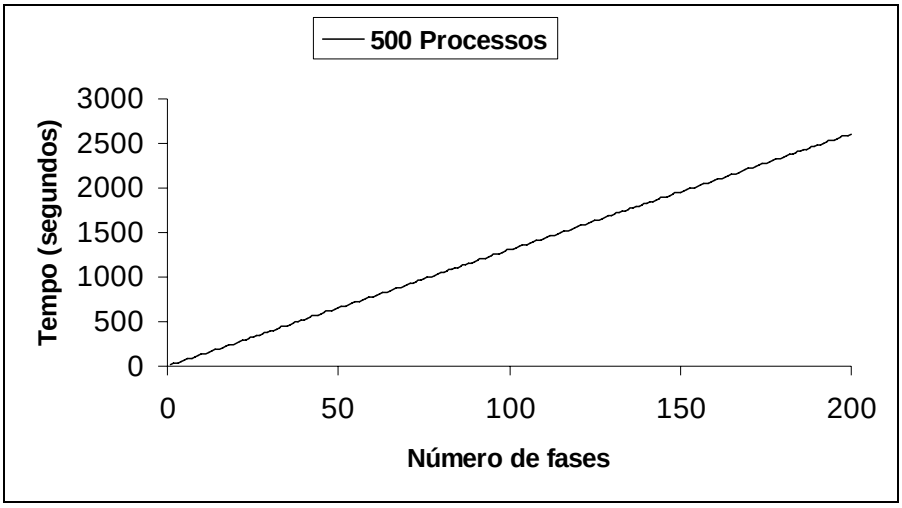


Figura 6.15 Efeitos do *OS Jitter* para 500 processos com múltiplas fases

Ao aumentar o número de fases, a probabilidade de atrasos causados pelo *OS Jitter*, dentro de cada nó do *Cluster*, também aumenta. Uma vez que os nós estão trabalhando em paralelo, a soma destes aumentos de probabilidade explica o comportamento linear.

A Figura 6.16 mostra uma representação da análise de sensibilidade do atraso no tempo de processamento em relação ao número de processos e fases, sumarizando os resultados da simulação. Com base nos resultados experimentais e na simulação, conclui-se que, a fim de reduzir os efeitos do *OS Jitter* sobre o tempo de processamento de aplicações

distribuídas, é importante reduzir o número de fases computacionais por processos, embora isso exija um aumento significativo no número de processos (ou nós de computação).

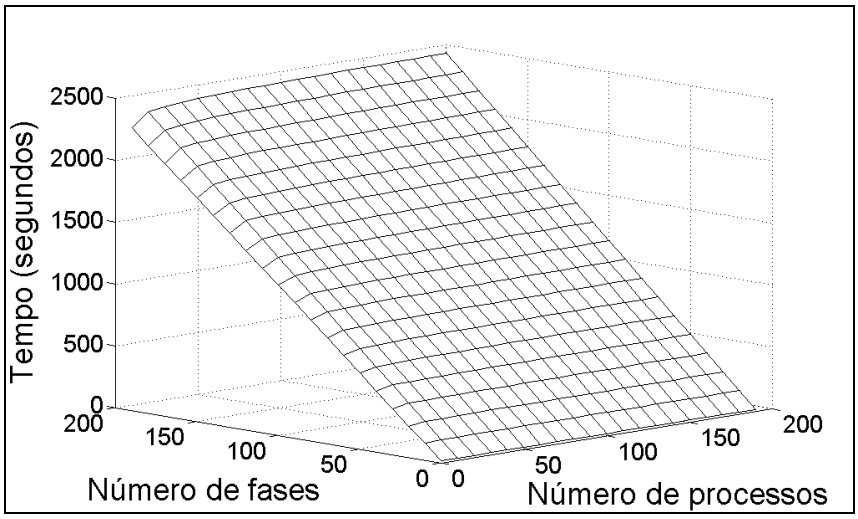


Figura 6.16 Atraso de execução em relação à quantidade de fases e processos

6.4.2 – Resultados da Simulação #2

De forma análoga à Simulação #1, selecionou-se os tratamentos do Experimento #2 em que os fatores de *OS Jitter* estão configurados nos menores e maiores níveis de influência, sendo, respectivamente, os tratamentos 10 e 55 (ver Apêndice A). Ao realizar a diferença entre os dados do tratamento 55 com os dados do tratamento 10, encontrou-se a *pdf* que descreve o comportamento do tempo de processamento para o Experimento #2, ou seja, quando há o compartilhamento de *cache*. Nestes dados, aplicando o teste Kolmogorov-Smirnov (K-S) com nível de confiança de 99%, tem-se uma distribuição normal com média 3,483 e desvio padrão 0,8760 (ver Figura 6.17).

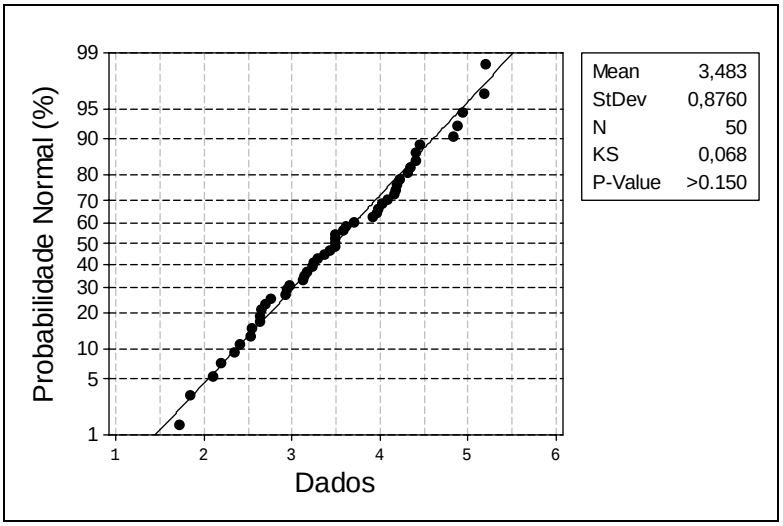


Figura 6.17 Probabilidade normal usando K-S para dados da simulação

Semelhante à Simulação #1, a *pdf* foi usada para simular os efeitos do *OS Jitter* em cada fase computacional de cada instância do programa em execução, nos vários nós de computação. Neste caso, utilizou-se uma faixa de 1 a 200 para a quantidade de fases e de 1 a 500 para a quantidade de processos. Similar ao modelo anterior, adotou-se apenas um processo de aplicação em cada nó, de forma que, ao variar a quantidade de processos acarreta a variação da quantidade de nós de computação.

A Simulação #2 apresenta resultados semelhantes aos obtidos na Simulação #1. Ao incrementar-se a quantidade de processos, para qualquer quantidade de fases, o tempo de processamento aumenta logaritmicamente (ver Figura 6.18). Até aproximadamente 20 processos, o *OS Jitter* acumulado aumenta em uma taxa muito maior do que nos cenários onde a quantidade de processos é maior (ex. 30, 40, 50, etc.).

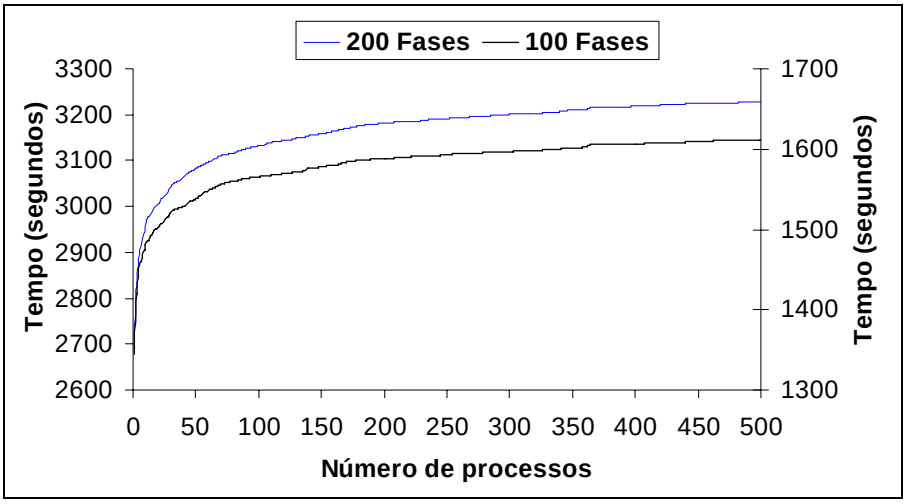


Figura 6.18 Efeitos do *OS Jitter* para 100 e 200 fases com múltiplos processos

Entretanto, nesta simulação, os efeitos do *OS Jitter* no tempo de processamento apresentaram-se maiores, de modo que, comparando os resultados das simulações #1 e #2, tem-se um aumento no tempo de processamento de até aproximadamente 23%. Tal fato é devido aos efeitos do compartilhamento de *cache* do Experimento #2. Assim como na Simulação #1, ao aumentar-se o número de fases, o tempo de processamento da aplicação aumenta linearmente (ver Figura 6.19).

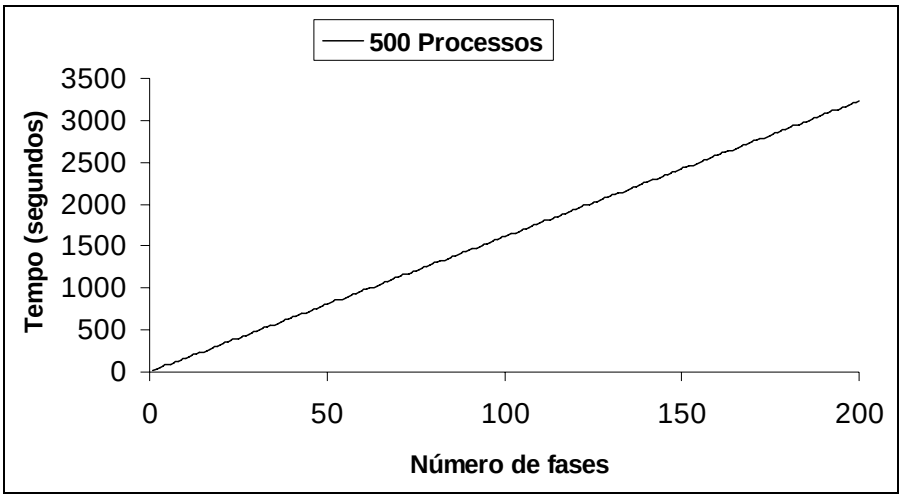


Figura 6.19 Efeitos do *OS Jitter* para 500 processos com múltiplas fases

A mesma análise de sensibilidade apresentada anteriormente foi realizada para os dados da Simulação #2. A Figura 6.20 mostra estes resultados ilustrando a sensibilidade do atraso no tempo de processamento, com respeito ao número de processos e fases computacionais por processo.

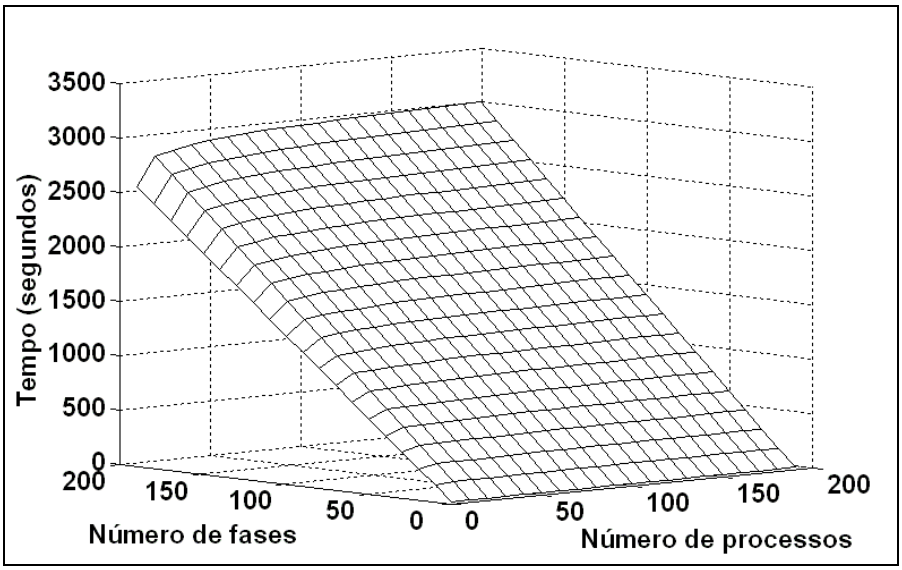


Figura 6.20 Atraso de execução em relação à quantidade de fases e processos

CAPÍTULO 7 – CONCLUSÕES DA PESQUISA

7.1 – Resultados Obtidos

Vários trabalhos anteriores investigaram os efeitos do *OS Jitter* em diferentes plataformas. No entanto, os recentes avanços em áreas relacionadas à economia de energia e topologia do processador mudaram o modo de funcionamento do *kernel* dos sistemas operacionais. Essas mudanças, consequentemente, afetam a forma como as rotinas do sistema operacional interferem nos programas do usuário. Recursos, como *CPU Frequency Scaling* e *kernel tickless,* não foram considerados em estudos anteriores, sendo necessário realizar estudos atualizados nesta área.

Assim, além de incluir estes aspectos técnicos, até então não considerados, o presente estudo também adotou uma metodologia mais rigorosa para o planejamento, execução e análise dos dados dos experimentos. A maioria dos trabalhos da literatura não esclarecem como os dados foram obtidos, em que condições ambientais e se cuidados importantes (ex. replicação dos experimentos) foram implementados.

Vários trabalhos anteriores têm indicado a interrupção de *timer* como a fonte mais influente no *OS Jitter*. Tal consideração depende dos fatores analisados no estudo. Por exemplo, no Experimento #1 também observou-se resultado similar, onde a maior contribuição foi da interrupção de *timer*. Contudo, ao introduzir um novo fator no Experimento #2, verificou-se um resultado bastante diferente, onde descobriu-se que o compartilhamento de *cache* do processador tem um impacto semelhante ao da interrupção de *timer* sobre o tempo de processamento da aplicação. E, a partir disso, verificou-se que a combinação de ambos os fatores resultam em uma influência ainda maior. Nesse contexto, a introdução deste novo fator, no presente estudo, permitiu identificar uma fonte ainda mais influente de *OS Jitter* do que até então a literatura havia reportado.

Pelos resultados da simulação, foi possível observar que o número de fases computacionais em uma aplicação distribuída tem um impacto maior sobre o atraso de tempo de processamento, em consequência do *OS Jitter*, do que o número de processos (ou nós de computação) sendo usados. Estudos anteriores haviam reportado apenas a relação número de nós vs. atraso, o que foi extrapolado neste estudo incluindo uma terceira dimensão para a análise do número de fases por processo.

Considerando que nos experimentos realizados as principais fontes de *OS Jitter* foram as interrupções de *timer* e o compartilhamento do *cache*, uma alternativa para redução do

impacto dessas fontes em ambiente de *Cluster* contendo múltiplos nós multiprocessados é particionar os processadores de cada nó de forma que um conjunto trate as tarefas/interrupções (ex. rede) do sistema e outro conjunto os processos específicos do usuário, sendo que nesse último conjunto de processadores, nas fases de processamento recomenda-se desabilitar a interrupção de *timer*. Em conjunto com o descrito anteriormente, com o intuito de minimizar a influência do compartilhamento da *cache*, pode-se aplicar técnicas de particionamento da *cache* e, ainda, aplicar um refinamento do ambiente removendo processos desnecessários (ex. *cron jobs*).

7.2 – Limitações da Pesquisa

As seguintes restrições devem ser consideradas ao analisar os resultados obtidos neste estudo:

1. A computação analisada é do tipo CPU-*bound*;
2. O trabalho de processamento é idêntico para todos os processos;
3. O número de fases é constante entre todos os processos;
4. Durante a fase de processamento não ocorre comunicação em rede;
5. O ambiente (software e hardware) é homogêneo entre os nós simulados.

7.3 – Contribuição para a Literatura

Durante o desenvolvimento desta pesquisa, foram publicados cinco trabalhos reportando resultados parciais da mesma. A Tabela 7.1 lista os veículos onde os trabalhos foram publicados.

Tabela 7.1 Publicações científicas

Conferência/Periódico	Qualis
ACM Operating Systems Review (P)	B1
IEEE SMC 2011 (C)	B1
IEEE PDCAT 2011 (C)	B2
WSO 2011 (C)	B5
WSL 2011 (C)	B5

Além dos cinco trabalhos publicados listados anteriormente, atualmente um sexto trabalho foi submetido e encontra-se em processo de avaliação em uma conferência Qualis A2.

7.4 – Dificuldades Encontradas

Pela ausência de trabalhos científicos que realizassem um estudo sistematizado e contemplando os fatores estudados, o projeto experimental deste trabalho considerou um

grande conjunto de tratamentos e replicações. Neste caso, o Experimento #1 executou por aproximadamente 11 dias, totalizando 1696 execuções. De forma análoga, o Experimento #2 obteve um total de 3392 execuções com duração de aproximadamente 23 dias. Assim, a extensa dimensão do projeto experimental, bem como a própria característica dos dados e dos experimentos, introduziram certas dificuldades:

1. Elaboração de um mecanismo para automatizar a execução dos experimentos;
2. O grande volume de dados a ser tratado inviabilizou o uso de ferramentas já conhecidas e de fácil acesso (ex. Minitab) que, a princípio, seriam usadas na etapa de análise de dados e simulação, mas não suportaram a grande quantidade de dados, exigindo a busca por ferramentas alternativas (ex. Matlab).
3. O uso de métodos estatísticos paramétricos não foi possível em virtude da característica dos dados observados, o que exigiu o uso de testes não paramétricos, acarretando na alteração do arcabouço de análise preestabelecido;
4. Devido ao longo período de execução, algumas falhas no fornecimento elétrico provocaram a interrupção dos experimentos.

7.5 – Trabalhos Futuros

Como trabalhos futuros, sugere-se a melhoria no planejamento experimental, incorporando novos fatores e níveis. Abaixo uma lista de sugestões.

1. Investigar especificamente a contribuição da interrupção de rede para os efeitos do *OS Jitter* (em andamento);
2. Estudar o comportamento do *OS Jitter* sobre aplicações *I/O-Bound* e híbridas (CPU e I/O);
3. Avaliar os efeitos do compartilhamento parcial de *cache*, em diversos níveis (10%, 20%, etc.);
4. Comparar os resultados da simulação com a execução distribuída do programa de prova em um *Cluster* real;
5. Buscar a identificação de outros fatores que possam ser fontes de *OS Jitter*.

REFERÊNCIAS BIBLIOGRÁFICAS

[Agarwal et al. 2005] Agarwal, S., Garg, R. e Vishnoi, N. K. (2005). The impact of noise on the scaling of collectives: a theoretical approach. In *Proceedings of 12th IEEE International Conference on High Performance Computing (HiPC)*, pages 280–289, Goa, India, December, 2005.

[Almeida 2010] Almeida, A. P. (2010). Etanólise do Óleo de Coco: Estudo das Veriáveis de Processo. Master's thesis, Universidade Federal de Alagoas, Programa de Pós-Graduação em Engenharia Química, Alagoas, BR-AL.

[Appleton 2009] Appleton, R. R. (2009). File System Call Accounting Measuring Different Workloads. In *CATA'09*, pp. 278-283.

[Beckman et al. 2006] Beckman, P., Iskra, K., Yoshii, K., e Coghlan, S. (2006). The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *Cluster Computing, 2006 IEEE International Conference on*, pp. 1--12.

[Bovet e Cesati 2000] Bovet, D. P e Cesati, M. (2000). *Understanding The Linux Kernel*. O' Reilly.

[Brindley 2011] Brindley L. e Young A. (2011). *Red Hat Enterprise MRG 2 Realtime Reference Guide Reference guide for the Realtime component of Red Hat Enterprise MRG Edition 1*. Red Hat Enterprise.

[Broquedis et al. 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., e Namyst, R. (2010). hwloc: a Generic Framework for Managing Hardware A-nities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pp. 180-186, Pisa, Italia. IEEE Computer Society Press.

[Button 2005] Button, S. T. (2005). Metodologia para Planejamento Experimental e Análise de Resultados. http://pessoal.utfpr.edu.br/lincolngusmao/arquivos/Planejamento%20Experimental.pdf.

[Buyya 1999] Buyya, R. (1999). In *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR.

[Canal 2000] Canal, A. P. (2000). Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em um Cluster de PCs. Master's thesis, Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS.

[Campos 1979] Campos Humberto (1979). *Estatística Experimental Não-Paramétrica, 3rd edição*. Piracicaba-SP.

[Carter et al. 2007] Carter, S., Minich, M., e Rao, N. S. V. (2007). Experimental evaluation of infiniband transport over local- and wide-area networks. In *Proceedings of the 2007 spring simulation multiconference - Volume 2*, SpringSim '07, pp. 419-426, Norfolk, Virginia, San Diego, CA, USA. Society for Computer Simulation International.

[Carvalho 2001] Carvalho, A. J. P. (2001). *Estudo de processos electroquímicos em solução*. PhD em Química, Departamento de Química Faculdade de Ciências da Universidade do Porto, Rua do Campo Alegre 687 - Portugal.

[Chakravarthi et al. 2002] Chakravarthi, S., Pillai, A., Neelamegam, J., Apte, M., e Skjellum, A. (2002). A fine-grain clock synchronization mechanism for Myrinet clusters. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pp. 708-715.

[Chapra e Canale 2006] Chapra, S. C. e Canale, R. (2006). *Numerical Methods for Engineers*. McGraw-Hill, Inc., New York, NY, USA, 5 edition.

[Chevance 2005] Chevance, R. (2005). *Server architectures: multiprocessors, clusters, parallel systems, Web servers, and storage solutions*. Elsevier/Digital Press.

[Corder e Foreman 2011] Corder, G. e Foreman, D. (2011). *Nonparametric Statistics for Non-Statisticians: A Step-By-Step Approach*. John Wiley & Sons.

[Costigan e Scott 2007] Costigan, N. e Scott, M. (2007). Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061. http://eprint.iacr.org/.

[Culler et al. 1999] Culler, D., Singh, J., e Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers.

[De et al. 2007] De, P., Kothari, R., e Mann, V. (2007). Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 331–340, Washington, DC, USA. IEEE Computer Society.

[De et al. 2008] De, P., Kothari, R., e Mann, V. (2008). A trace-driven emulation framework to predict scalability of large clusters in presence of OS Jitter. In *CLUSTER*, pp. 232-241.

[De e Mann 2010] De, P. e Mann, V. (2010). JitSim: a simulator for predicting scalability of parallel applications in presence of OS jitter. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pp. 117-130, Ischia, Italy, Berlin, Heidelberg. Springer-Verlag.

[Dongarra et al. 2003] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., e White, A. (2003). *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Falk e Kotthaus 2011] Falk, H. e Kotthaus, H. (2011). WCET-driven cache-aware code positioning. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pp. 145□154, Taipei, Taiwan, New York, NY, USA. ACM.

[Fedorova et al. 2009] Fedorova, A., Saez, J. C., Shelepov, D. e Prieto, M. (2009). Maximizing power efficiency with asymmetric multicore systems. *Commun*. ACM, 52:48-57.

[Ferreira et al. 2008] Ferreira, K. B., Bridges, P., e Brightwell, R. (2008). Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 19:1-19:12, Austin, Texas, Piscataway, NJ, USA. IEEE Press.

[Filho 2008] Paulo José de Freitas Filho (2008). *Introdução à modelagem e simulação de sistemas com aplicações em Arena, 2º Edição*. Visul Books.

[Fosdick 1996] Fosdick, L. D. (1996). *An Introduction to high performance scientific computing*. MIT Press.

[Fröhlich et al. 2011] Fröhlich, A. A., Gracioli, G., e Santos, J. F. (2011). Periodic timers revisited: The real-time embedded system perspective. *Computers and Electrical Engineering*, 37(3):365 - 375.

[Fromm e Treuhaft 1996] Fromm, R. e Treuhaft, N. (1996). Revisiting the cache interference costs of context switching. http://citeseer.ist.psu.edu/252861.html.

[Garg e De 2006] Garg, R. e De, P. (2006). The impact of noise on scaling of collectives: an empirical evaluation. In *Proceedings of 13th IEEE International Conference on High Performance Computing (HiPC)*, Bangalore, India.

[Gioiosa et al. 2004] Gioiosa, R., Petrini, F., Davis, K., Lebaillif-Delamare, F. (2004). Analysis of system overhead on parallel computers. In *Signal Processing and Information Technology, 2004. Proceedings of the Fourth IEEE International Symposium on*, pages 387–390.

[Gioiosa et al. 2010] Gioiosa, R., McKee, S., e Valero, M. (2010). Designing OS for HPC Applications: Scheduling. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 78–87.

[González et al. 2010] González, A., Latorre, F., Magklis, G., e Hill, M. (2010). *Processor Microarchitecture: An Implementation Perspective*. Synthesis Lectures on Computer Architecture. Morgan & Claypool.

[Greenhalgh 2011] Greenhalgh, P. (2011). Big.LITTLE Processing with ARM Cortex-A15 Cortex™-A7. Technical report, ARM.

[Gropp et al. 1999] Gropp, W., Lusk, E., e Skjellum, A. (1999). *Using MPI Volume 2, in Scientific and engineering computation*. MIT Press.

[Gupta et al. 1991] Gupta, A., Tucker, A., e Urushibara, S. (1991). The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems, SIGMETRICS '91*, pp. 120--132, San Diego, California, United States, New York, NY, USA. ACM.

[Haldara e Aravind 2010] Haldara, S. e Aravind, A. A. (2010). *Operating Systems*. Pearson Education, Upper Saddle River, NJ.

[Hammond 2011] Hammond J. R., Romero N. A., Lilienfeld O. A. (2011). *High-Performance Computing Advances Computationally Intensive Materials Research*. HPC Source.

[Handy 1998] Handy, J. (1998). *The cache memory book*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press.

[Happe et al. 2009] Happe, J., Groenda, H., e Reussner, R. (2009). Performance evaluation of scheduling policies in symmetric multiprocessing environments. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009*. MASCOTS '09. IEEE International Symposium on, pp. 1--10.

[Hariprasad 2004] Hariprasad N. (2004). Reboot Linux faster using kexec, IBM technical report, http://www.ibm.com/ developerworks/linux/library/l-kexec/index.html

[Hwang et al. 2000] Hwang, K., Jin, H., e Ho, R. (2000). RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing. In *proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, p. 287.

[Hwang 2010] Hwang (2010). *Advanced Computer Architecture, 2E*. McGraw-Hill Education.

[IBM 2010] IBM (2010). Identifying Jitter Sources. http://domino.watson.ibm.com/comm/research_projects.nsf/pages/osjitter.Identifying.html. Acessado em 21 de Novembro de 2010.

[Intel Corporation 1997] Intel Corporation (1997). *MultiProcessor Specification Version 1.4*.

[Intel Corporation 1996] Intel Corporation (1996). *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*.

[Intel Corporation 2010] Intel Corporation (2010). *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide*.

[Jones et al. 2003] Jones, T. R., Brenner, L. B. e Fier, J. M. (2003). Impacts of operating systems on the scalibility of parallel applications. Tech. Rep. UCRL-MI-202629, Lawrence Livermore National Laboratory, Mar. 2003.

[Jones 2011] Jones, T. (2011). Linux kernel co-scheduling for bulk synchronous parallel applications. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pp. 57-64, Tucson, Arizona, New York, NY, USA. ACM.

[Kelly e Brightwell 2005] Kelly, S. M. e Brightwell, R. (2005). Software architecture of the light weight kernel, Catamount. In *47th Cray User Group Conference*, pp. 16-19.

[Kroah-Hartman 2007] Kroah-Hartman, G. (2007). *Linux kernel in a nutshell*. In a Nutshell. O'Reilly.

[Lastovetsky 2003] Lastovetsky, A. (2003). *Parallel Computing on Heterogeneous Networks*. Wiley Series on Parallel and Distributed Computing. John Wiley.

[Leon F. de Carvalho et al. 2006] Leon F. de Carvalho, A. C. P., Brayner, A., Loureiro, A., Furtado, A. L., von Staa, A., de Lucena, C. J. P., de Souza, C. S., Medeiros, C. M. B., Lucchesi, C. L., e Silva, E. S., Wagner, F. R., Simon, I., Wainer, J., Maldonado, J. C., de Oliveira, J. P. M., Ribeiro, L., Velho, L., Gonçalves, M. A., Baranauskas, M. C. C., Mattoso, M., Ziviani, N., Navaux, P. O. A., da Silva Torres, R., Almeida, V. A. F., Jr., W. M., e Kohayakawa, Y. (2006). Grandes Desafios da Pesquisa em Computação no Brasil - 2006 - 2016. Technical report, Sociedade Brasileira de Computação (SBC).

[Liedtke et al. 1997] Liedtke, J., Haertig, H., e Hohmuth, M. (1997). OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, Washington, DC, USA. IEEE Computer Society.

[Li et al. 2007a] Li, C., Ding, C., e Shen, K. (2007a). Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, San Diego, California, New York, NY, USA. ACM.

[Li et al. 2007b] Li, T., Baumberger, D., Koufaty, D. A. e Hahn, S. (2007b). Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA. ACM.

[Mann e Mittaly 2009] Mann, P. D. V. e Mittaly, U. (2009). Handling OS Jitter on multicore multithreaded systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA. IEEE Computer Society.

[Marc Snir 1996] Marc Snir, Steve Otto, S. H. D. W. J. D. (1996). *MPI: The Complete Reference*. Massachusetts Institute of Technology.

[Mattson et al. 2005] Mattson, T. G., Sanders, B. A., e Massingill, B. (2005). *Patterns for parallel programming*. Software patterns series. Addison-Wesley.

[Mazou et al. 2010] Mazou, A., Touat, S., e Barthou, D. (2010). Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study. Technical Report HAL-inria-00514548, Université de Versailles Saint-Quentin-en-Yvelines.

[Mogul e Borg 1991] Mogul, J. C. e Borg, A. (1991). The effect of context switches on cache performance. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, volume 26, pages 75–84, New York, NY, USA. ACM Press.

[Montgomery 2005] Montgomery, D. C. (2005). *Design and Analysis of Experiments*. John Wiley, 3 edition.

[Moore 2000] Moore, G. E. (2000). Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Morari et al. 2011] Morari, A., Gioiosa, R., Wisniewski, R., Cazorla, F., e Valero, M. (2011). A Quantitative Analysis of OS Noise. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 852-863.

[Nussbaum et al. 2009] Nussbaum, L., Anhalt, F., Mornard, O., e Gelas, J.-P. (2009). Linux-based virtualization for HPC clusters. In *Proceedings of the Linux Symposium*.

[Osthoff et al. 2011] Osthoff, C. , Boito, F., Kassick, R., Pilla, L., Grunmann, P. J., Schepke, C., Maillard, N. B., Navaux, P. O., Panetta, J., Lopes, P. P., Dias, P. L. S., Walko, R., Souto, R. P., Vilasboas, F. G. (2011). *Improving Atmospheric Model Performance on a Multi-Core Cluster System*. Intech.

[Padua 2011] Padua, D. (2011). *Encyclopedia of Parallel Computing*. Número v. 4 in Springer Reference. Springer.

[Prabhu 2008] Prabhu (2008). *Grid And Cluster Computing*. Prentice-Hall Of India Pvt. Ltd.

[Petersen e Haddad 2003] Petersen, R. e Haddad, I. (2003). *Red Hat Linux Pocket Administrator*. Pocket Administrator Series. McGraw-Hill/Osborne.

[Piel et al. 2005] Piel, E., Marquet, P., Soula, J. e Dekeyser, J. (2005). Asymmetric scheduling and load balancing for real time on linux smp. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Wasniewski, J., editors, PPAM, volume 3911 of *Lecture Notes in Computer Science*, pages 896–903. Springer.

[Pitanga 2008] Pitanga, M. (2008). *Construindo supercomputadores com Linux 3ª edição*. Brasport.

[Polloni e Fedeli 2003] Polloni, E. e Fedeli, R. (2003). *Introdução à Ciência da Computação*. Thomson Pioneira.

[Purohit et al. 2003] A. Purohit, J. Spadavecchia, C. Wright e E. Zadok. Improving Application Performance Through System Call Composition. Technical Report FSL-02-01, Computer Science Department, Stony Brook University, June 2003. www.fsl.cs.sunysb.edu/docs/cosy-perf/.

[Qingbo et al. 2009] Qingbo, Y., Yungang, B., Mingyu, C. e Ninghui, S. (2009). A scalability analysis of the symmetric multiprocessing architecture in multi-core system. In *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, NAS ’09, pages 231–234, Washington, DC, USA. IEEE Computer Society.

[Quinn 2004] Quinn, M. J. (2004). *Parallel programming in C with MPI and OpenMP*. McGraw-Hill Higher Education.

[Rauber e Rünger 2010] Rauber, T. e Rünger, G. (2010). *Parallel Programming: For Multicore and Cluster Systems*. Springer.

[Saez et al. 2010] Saez, J. C., Prieto, M., Fedorova, A. e Blagodurov, S. (2010). A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, EuroSys ’10, pages 139–152, New York, NY, USA. ACM.

[Sloan 2004] Sloan, J. D. (2004). *High performance Linux clusters with OSCAR, Rocks, openMosix and MPI*. O'Reilly Series. O'Reilly.

[Smith 2011] Smith, R. (2011). *LPIC-2 Linux Professional Institute Certification Study Guide: Exams 201 and 202*. John Wiley & Sons.

[Sterling 2002] Sterling, T. (2002). *Beowulf Cluster Computing with Linux*. Scientific and Engineering Computation Series. MIT Press.

[Suleman et al. 2009] Suleman, M. A., Mutlu, O., Qureshi, M. K. e Patt, Y. N. (2009). Accelerating critical section execution with asymmetric multi-core architectures. SIGPLAN Not., 44:253–264.

[Tanenbaum 2005] Tanenbaum, A. S. (2005). *Organização Estruturada de Computadores*. Prentice Hall Brasil.

[Top 500 2011] Top 500 (2011). Top 500 Supercomputer sites. http://www.top500.org. Acessado em 21 de Novembro de 2011.

[Tsafrir et al. 2005] Tsafrir, D., Etsion, Y., Feitelson, D. G. e Kirkpatrick, S. (2005). System noise, os clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS’05, pages 303–312, New York, NY, USA. ACM.

[Vajda e Brorsson 2011] Vajda, A. e Brorsson, M. (2011). *Programming Many-Core Chips*. Springer.

[Van Vugt 2006] Van Vugt, S. (2006). *The Definitive Guide to Suse Linux Enterprise*

Server. Definitive Guide. Apress.

[Vicente et al. 2012] Vicente, E., Matias, R., Borges, L., e Macêdo, A. (2012). Evaluation of compound system calls in the Linux kernel. *SIGOPS Operating Systems Review*, 46(1):53-63.

[Wazlawick 2009] Wazlawick, R. (2009). *Metodologia de Pesquisa para Ciência da Computação*. Elsevier Editora Ltda.

APÊNDICE A. Matriz de contraste para o Experimento #2. Sinais dos tratamentos 01 ao 21.

Fatores Principais						Interação Entre Fatores																																																								
A	**B**	**C**	**D**	**E**	**F**	**AB**	**AC**	**AD**	**AE**	**AF**	**BC**	**BD**	**BE**	**BF**	**CD**	**CE**	**CF**	**DE**	**DF**	**EF**	**ABC**	**ABD**	**ACD**	**ABE**	**ACE**	**ADE**	**ABF**	**ACF**	**ADF**	**AEF**	**BCD**	**BCE**	**BDE**	**BCF**	**BDF**	**BEF**	**CDE**	**CDF**	**CEF**	**DEF**	**ABCD**	**ABCE**	**ABDE**	**ACDE**	**ABCF**	**ABDF**	**ACDF**	**ABEF**	**ACEF**	**ADEF**	**BCDE**	**BCDF**	**BCEF**	**BDEF**	**CDEF**	**ABCDE**	**ABCDF**	**ABCEF**	**ABDEF**	**ACDEF**	**BCDEF**	**ABCDEF**
-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	+
+	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	-	-
-	+	-	-	-	-	-	+	+	+	+	-	-	-	-	+	+	+	+	+	+	+	+	-	+	-	-	+	-	-	-	+	+	+	+	+	+	-	-	-	-	-	-	-	+	-	-	+	-	+	+	-	-	-	-	+	+	+	+	+	-	+	-
+	+	-	-	-	-	+	-	-	-	-	-	-	-	-	+	+	+	+	+	+	-	-	+	-	+	+	-	+	+	+	+	+	+	+	+	+	-	-	-	-	+	+	+	-	+	+	-	+	-	-	-	-	-	-	+	-	-	-	-	+	+	+
-	-	+	-	-	-	+	-	+	+	+	-	+	+	+	-	-	-	+	+	+	+	-	+	-	+	-	-	+	-	-	+	+	-	+	-	-	+	+	+	-	-	-	+	-	-	+	-	+	-	+	-	-	-	+	-	+	+	+	-	+	+	-
+	-	+	-	-	-	-	+	-	-	-	-	+	+	+	-	-	-	+	+	+	-	+	-	+	-	+	+	-	+	+	+	+	-	+	-	-	+	+	+	-	+	+	-	+	+	-	+	-	+	-	-	-	-	+	-	-	-	-	+	-	+	+
-	+	+	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	+	+	+	-	+	+	+	+	-	+	+	-	-	-	-	+	-	+	+	+	+	+	-	+	+	-	-	+	-	-	-	-	+	+	+	+	-	-	-	-	-	+	+	-	+
+	+	+	-	-	-	+	+	-	-	-	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	+	-	-	+	+	-	-	+	-	+	+	+	+	+	-	-	-	+	+	-	+	+	+	+	-	+	+	+	-	-	+	+	+	-	-	-	-
-	-	-	+	-	-	+	+	-	+	+	+	-	+	+	-	+	+	-	-	+	-	+	+	-	-	+	-	-	+	-	+	-	+	-	+	-	+	+	-	+	-	+	-	-	+	-	-	+	+	-	-	-	+	-	-	+	+	-	+	+	+	-
+	-	-	+	-	-	-	-	+	-	-	+	-	+	+	-	+	+	-	-	+	+	-	-	+	+	-	+	+	-	+	+	-	+	-	+	-	+	+	-	+	+	-	+	+	-	+	+	-	-	+	-	-	+	-	-	-	-	+	-	-	+	+
-	+	-	+	-	-	-	+	-	+	+	-	+	-	-	-	+	+	-	-	+	+	-	+	+	-	+	+	-	+	-	-	+	-	+	-	+	+	+	-	+	+	-	+	-	-	+	-	-	+	-	+	+	-	+	-	-	-	+	-	+	-	+
+	+	-	+	-	-	+	-	+	-	-	-	+	-	-	-	+	+	-	-	+	-	+	-	-	+	-	-	+	-	+	-	+	-	+	-	+	+	+	-	+	-	+	-	+	+	-	+	+	-	+	+	+	-	+	-	+	+	-	+	-	-	-
-	-	+	+	-	-	+	-	-	+	+	-	-	+	+	+	-	-	-	-	+	+	+	-	-	+	+	-	+	+	-	-	+	+	+	+	-	-	-	+	+	+	-	-	+	-	-	+	+	-	-	+	+	-	-	+	-	-	+	+	-	-	+
+	-	+	+	-	-	-	+	+	-	-	-	-	+	+	+	-	-	-	-	+	-	-	+	+	-	-	+	-	-	+	-	+	+	+	+	-	-	-	+	+	-	+	+	-	+	+	-	-	+	+	+	+	-	-	+	+	+	-	-	+	-	-
-	+	+	+	-	-	-	-	-	+	+	+	+	-	-	+	-	-	-	-	+	-	-	-	+	+	+	+	+	+	-	+	-	-	-	-	+	-	-	+	+	-	+	+	+	+	+	+	-	-	-	-	-	+	+	+	+	+	-	-	-	+	-
+	+	+	+	-	-	+	+	+	-	-	+	+	-	-	+	-	-	-	-	+	+	+	+	-	-	-	-	-	-	+	+	-	-	-	-	+	-	-	+	+	+	-	-	-	-	-	-	+	+	+	-	-	+	+	+	-	-	+	+	+	+	+
-	-	-	-	+	-	+	+	+	-	+	+	+	-	+	+	-	+	-	+	-	-	-	-	+	+	+	-	-	-	+	-	+	+	-	-	+	+	-	+	+	+	-	-	-	+	+	+	-	-	-	-	+	-	-	-	+	-	+	+	+	+	-
+	-	-	-	+	-	-	-	-	+	-	+	+	-	+	+	-	+	-	+	-	+	+	+	-	-	-	+	+	+	-	-	+	+	-	-	+	+	-	+	+	-	+	+	+	-	-	-	+	+	+	-	+	-	-	-	-	+	-	-	-	+	+
-	+	-	-	+	-	-	+	+	-	+	-	-	+	-	+	-	+	-	+	-	+	+	-	-	+	+	+	-	-	+	+	-	-	+	+	-	+	-	+	+	-	+	+	-	-	-	+	+	-	-	+	-	+	+	-	-	+	-	-	+	-	+
+	+	-	-	+	-	+	-	-	+	-	-	-	+	-	+	-	+	-	+	-	-	-	+	+	-	-	-	+	+	-	+	-	-	+	+	-	+	-	+	+	+	-	-	+	+	+	-	-	+	+	+	-	+	+	-	+	-	+	+	-	-	-
-	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	-	+	-	+	-	+	+	-	+	-	+	-	+	+	-	+	+	-	+	-	+	-	+	-	+	-	+	-	+	-	-	+	-	+	-	+	-	+	-	+	-	+	-	-	+

Continua na próxima página

APÊNDICE A. Matriz de contraste para o Experimento #2 (continuação). Sinais dos tratamentos 22 ao 43.

Fatores Principais						Interação Entre Fatores																																																								
A	**B**	**C**	**D**	**E**	**F**	**AB**	**AC**	**AD**	**AE**	**AF**	**BC**	**BD**	**BE**	**BF**	**CD**	**CE**	**CF**	**DE**	**DF**	**EF**	**ABC**	**ABD**	**ACD**	**ABE**	**ACE**	**ADE**	**ABF**	**ACF**	**ADF**	**AEF**	**BCD**	**BCE**	**BDE**	**BCF**	**BDF**	**BEF**	**CDE**	**CDF**	**CEF**	**DEF**	**ABCD**	**ABCE**	**ABDE**	**ACDE**	**ABCF**	**ABDF**	**ACDF**	**ABEF**	**ACEF**	**ADEF**	**BCDE**	**BCDF**	**BCEF**	**BDEF**	**CDEF**	**ABCDE**	**ABCDF**	**ABCEF**	**ABDEF**	**ACDEF**	**BCDEF**	**ABCDEF**
+	-	+	-	+	-	-	+	-	+	-	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	+	-	+	+	-	+	-	+	-	+	+	-	+	-	+	-	+	+	-	+	+	-	+	-	+	+	-	+	-	+	-	-
-	+	+	-	+	-	-	-	+	-	+	+	-	+	-	-	+	-	-	+	-	-	+	+	-	-	+	+	+	-	+	-	+	-	-	+	-	-	+	-	+	+	-	+	+	+	-	-	+	+	-	-	+	-	+	+	+	-	+	-	-	+	-
+	+	+	-	+	-	+	+	-	+	-	+	-	+	-	-	+	-	-	+	-	+	-	-	+	+	-	-	-	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	-	-	+	+	-	-	+	-	+	-	+	+	-	+	-	+	+	+	+
-	-	-	+	+	-	+	+	-	-	+	+	-	-	+	-	-	+	+	-	-	-	+	+	+	+	-	-	-	+	+	+	+	-	-	+	+	-	+	+	-	-	-	+	+	+	-	-	-	-	+	+	-	-	+	+	-	+	+	-	-	-	+
+	-	-	+	+	-	-	-	+	+	-	+	-	-	+	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	+	+	-	-	+	+	-	+	+	-	+	+	-	-	-	+	+	+	+	-	+	-	-	+	+	+	-	-	+	+	-	-
-	+	-	+	+	-	-	+	-	-	+	-	+	+	-	-	-	+	+	-	-	+	-	+	-	+	-	+	-	+	+	-	-	+	+	-	-	-	+	+	-	+	+	-	+	-	+	-	+	-	+	-	+	+	-	+	+	-	-	+	-	+	-
+	+	-	+	+	-	+	-	+	+	-	-	+	+	-	-	-	+	+	-	-	-	+	-	+	-	+	-	+	-	-	-	-	+	+	-	-	-	+	+	-	-	-	+	-	+	-	+	-	+	-	-	+	+	-	+	-	+	+	-	+	+	+
-	-	+	+	+	-	+	-	-	-	+	-	-	-	+	+	+	-	+	-	-	+	+	-	+	-	-	-	+	+	+	-	-	-	+	+	+	+	-	-	-	+	+	+	-	-	-	+	-	+	+	-	+	+	+	-	+	-	-	-	+	+	-
+	-	+	+	+	-	-	+	+	+	-	-	-	-	+	+	+	-	+	-	-	-	-	+	-	+	+	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	+	+	+	-	+	-	-	-	+	+	+	-	-	+	+	+	-	+	+
-	+	+	+	+	-	-	-	-	-	+	+	+	+	-	+	+	-	+	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	-	-	-	+	-	-	-	-	-	-	-	+	+	+	+	+	+	+	-	-	-	-	-	+	+	+	+	-	+
+	+	+	+	+	-	+	+	+	+	-	+	+	+	-	+	+	-	+	-	-	+	+	+	+	+	+	-	-	-	-	+	+	+	-	-	-	+	-	-	-	+	+	+	+	-	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-
-	-	-	-	-	+	+	+	+	+	-	+	+	+	-	+	+	-	+	-	-	-	-	-	-	-	-	+	+	+	+	-	-	-	+	+	+	-	+	+	+	+	+	+	+	-	-	-	-	-	-	+	-	-	-	-	-	+	+	+	+	+	-
+	-	-	-	-	+	-	-	-	-	+	+	+	+	-	+	+	-	+	-	-	+	+	+	+	+	+	-	-	-	-	-	-	-	+	+	+	-	+	+	+	-	-	-	-	+	+	+	+	+	+	+	-	-	-	-	+	-	-	-	-	+	+
-	+	-	-	-	+	-	+	+	+	-	-	-	-	+	+	+	-	+	-	-	+	+	-	+	-	-	-	+	+	+	+	+	+	-	-	-	-	+	+	+	-	-	-	+	+	+	-	+	-	-	-	+	+	+	-	+	-	-	-	+	-	+
+	+	-	-	-	+	+	-	-	-	+	-	-	-	+	+	+	-	+	-	-	-	-	+	-	+	+	+	-	-	-	+	+	+	-	-	-	-	+	+	+	+	+	+	-	-	-	+	-	+	+	-	+	+	+	-	-	+	+	+	-	-	-
-	-	+	-	-	+	+	-	+	+	-	-	+	+	-	-	-	+	+	-	-	+	-	+	-	+	-	+	-	+	+	+	+	-	-	+	+	+	-	-	+	-	-	+	-	+	-	+	-	+	-	-	+	+	-	+	+	-	-	+	-	-	+
+	-	+	-	-	+	-	+	-	-	+	-	+	+	-	-	-	+	+	-	-	-	+	-	+	-	+	-	+	-	-	+	+	-	-	+	+	+	-	-	+	+	+	-	+	-	+	-	+	-	+	-	+	+	-	+	-	+	+	-	+	-	-
-	+	+	-	-	+	-	-	+	+	-	+	-	-	+	-	-	+	+	-	-	-	+	+	+	+	-	-	-	+	+	-	-	+	+	-	-	+	-	-	+	+	+	-	-	-	+	+	+	+	-	+	-	-	+	+	-	+	+	-	-	+	-
+	+	+	-	-	+	+	+	-	-	+	+	-	-	+	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	+	+	-	-	+	-	-	+	-	-	+	+	+	-	-	-	-	+	+	-	-	+	+	+	-	-	+	+	+	+
-	-	-	+	-	+	+	+	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	+	-	-	+	+	+	-	+	+	-	+	+	-	+	+	-	+	-	-	+	-	-	-	+	+	-	-	+	-	+	-	+	+	+	-	+	-	-	-	+
+	-	-	+	-	+	-	-	+	-	+	+	-	+	-	-	+	-	-	+	-	+	-	-	+	+	-	-	-	+	-	+	-	+	+	-	+	+	-	+	-	+	-	+	+	+	-	-	+	+	-	-	+	-	+	+	-	+	-	+	+	-	-
-	+	-	+	-	+	-	+	-	+	-	-	+	-	+	-	+	-	-	+	-	+	-	+	+	-	+	-	+	-	+	-	+	-	-	+	-	+	-	+	-	+	-	+	-	+	-	+	+	-	+	+	-	+	-	+	-	+	-	+	-	+	-

Continua na próxima página

APÊNDICE A. Matriz de contraste para o Experimento #2 (continuação). Sinais dos tratamentos 44 ao 64.

Fatores Principais						Interação Entre Fatores																																																								
A	**B**	**C**	**D**	**E**	**F**	**AB**	**AC**	**AD**	**AE**	**AF**	**BC**	**BD**	**BE**	**BF**	**CD**	**CE**	**CF**	**DE**	**DF**	**EF**	**ABC**	**ABD**	**ACD**	**ABE**	**ACE**	**ADE**	**ABF**	**ACF**	**ADF**	**AEF**	**BCD**	**BCE**	**BDE**	**BCF**	**BDF**	**BEF**	**CDE**	**CDF**	**CEF**	**DEF**	**ABCD**	**ABCE**	**ABDE**	**ACDE**	**ABCF**	**ABDF**	**ACDF**	**ABEF**	**ACEF**	**ADEF**	**BCDE**	**BCDF**	**BCEF**	**BDEF**	**CDEF**	**ABCDE**	**ABCDF**	**ABCEF**	**ABDEF**	**ACDEF**	**BCDEF**	**ABCDEF**
+	+	-	+	-	+	+	-	+	-	+	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	-	+	-	-	+	-	+	-	+	-	-	+	-	+	-	+	-	-	+	-	+	-	+	-	+	+	-	+	-	+	+	+
-	-	+	+	-	+	+	-	-	+	-	-	-	+	-	+	-	+	-	+	-	+	+	-	-	+	+	+	-	-	+	-	+	+	-	-	+	-	+	-	-	+	-	-	+	+	+	-	-	+	+	+	-	+	+	-	-	+	-	-	+	+	-
+	-	+	+	-	+	-	+	+	-	+	-	-	+	-	+	-	+	-	+	-	-	-	+	+	-	-	-	+	+	-	-	+	+	-	-	+	-	+	-	-	-	+	+	-	-	-	+	+	-	-	+	-	+	+	-	+	-	+	+	-	+	+
-	+	+	+	-	+	-	-	-	+	-	+	+	-	+	+	-	+	-	+	-	-	-	-	+	+	+	-	-	-	+	+	-	-	+	+	-	-	+	-	-	-	+	+	+	-	-	-	+	+	+	-	+	-	-	-	+	-	+	+	+	-	+
+	+	+	+	-	+	+	+	+	-	+	+	+	-	+	+	-	+	-	+	-	+	+	+	-	-	-	+	+	+	-	+	-	-	+	+	-	-	+	-	-	+	-	-	-	+	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-
-	-	-	-	+	+	+	+	+	-	-	+	+	-	-	+	-	-	-	-	+	-	-	-	+	+	+	+	+	+	-	-	+	+	+	+	-	+	+	-	-	+	-	-	-	-	-	-	+	+	+	-	-	+	+	+	+	+	-	-	-	-	+
+	-	-	-	+	+	-	-	-	+	+	+	+	-	-	+	-	-	-	-	+	+	+	+	-	-	-	-	-	-	+	-	+	+	+	+	-	+	+	-	-	-	+	+	+	+	+	+	-	-	-	-	-	+	+	+	-	-	+	+	+	-	-
-	+	-	-	+	+	-	+	+	-	-	-	-	+	+	+	-	-	-	-	+	+	+	-	-	+	+	-	+	+	-	+	-	-	-	-	+	+	+	-	-	-	+	+	-	+	+	-	-	+	+	+	+	-	-	+	-	-	+	+	-	+	-
+	+	-	-	+	+	+	-	-	+	+	-	-	+	+	+	-	-	-	-	+	-	-	+	+	-	-	+	-	-	+	+	-	-	-	-	+	+	+	-	-	+	-	-	+	-	-	+	+	-	-	+	+	-	-	+	+	+	-	-	+	+	+
-	-	+	-	+	+	+	-	+	-	-	-	+	-	-	-	+	+	-	-	+	+	-	+	+	-	+	+	-	+	-	+	-	+	-	+	-	-	-	+	-	-	+	-	+	+	-	+	+	-	+	+	+	-	+	-	-	-	+	-	+	+	-
+	-	+	-	+	+	-	+	-	+	+	-	+	-	-	-	+	+	-	-	+	-	+	-	-	+	-	-	+	-	+	+	-	+	-	+	-	-	-	+	-	+	-	+	-	-	+	-	-	+	-	+	+	-	+	-	+	+	-	+	-	+	+
-	+	+	-	+	+	-	-	+	-	-	+	-	+	+	-	+	+	-	-	+	-	+	+	-	-	+	-	-	+	-	-	+	-	+	-	+	-	-	+	-	+	-	+	+	-	+	+	-	-	+	-	-	+	-	-	+	+	-	+	+	-	+
+	+	+	-	+	+	+	+	-	+	+	+	-	+	+	-	+	+	-	-	+	+	-	-	+	+	-	+	+	-	+	-	+	-	+	-	+	-	-	+	-	-	+	-	-	+	-	-	+	+	-	-	-	+	-	-	-	-	+	-	-	-	-
-	-	-	+	+	+	+	+	-	-	-	+	-	-	-	-	-	-	+	+	+	-	+	+	+	+	-	+	+	-	-	+	+	-	+	-	-	-	-	-	+	-	-	+	+	-	+	+	+	+	-	+	+	+	-	-	-	-	-	+	+	+	-
+	-	-	+	+	+	-	-	+	+	+	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	+	-	-	+	+	+	+	-	+	-	-	-	-	-	+	+	+	-	-	+	-	-	-	-	+	+	+	+	-	-	+	+	+	-	-	+	+
-	+	-	+	+	+	-	+	-	-	-	-	+	+	+	-	-	-	+	+	+	+	-	+	-	+	-	-	+	-	-	-	-	+	-	+	+	-	-	-	+	+	+	-	+	+	-	+	-	+	-	-	-	-	+	-	+	+	+	-	+	-	+
+	+	-	+	+	+	+	-	+	+	+	-	+	+	+	-	-	-	+	+	+	-	+	-	+	-	+	+	-	+	+	-	-	+	-	+	+	-	-	-	+	-	-	+	-	-	+	-	+	-	+	-	-	-	+	-	-	-	-	+	-	-	-
-	-	+	+	+	+	+	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	-	+	-	-	+	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	-	+	+	-	+	-	-	-	-	-	-	+	+	+	+	+	-	-	+
+	-	+	+	+	+	-	+	+	+	+	-	-	-	-	+	+	+	+	+	+	-	-	+	-	+	+	-	+	+	+	-	-	-	-	-	-	+	+	+	+	-	-	-	+	-	-	+	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-
-	+	+	+	+	+	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	-	-	-	-	-	+	-
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

Fim

APÊNDICE B. Resultado da comparação dois a dois dos dados ajustados do Experimento #1 usando o teste de Kruskal-Wallis

Tratamentos																															
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**1**
	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**2**
		O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**3**
			O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**4**
				O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**5**
					O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**6**
						O	O	O	O	O	O	O	O	O	X	X	**X**	X	X	X	X	X	X	X	X	X	X	X	X	X	**7**
							O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**8**
								O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**9**
									O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**10**
										O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**11**
											O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**12**
												O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**13**
													O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**14**
														O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**15**
															X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**16**
																O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**17**
																	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**18**
																		X	O	X	O	X	O	O	O	X	O	O	X	X	**19**
																			O	O	O	O	O	O	O	O	X	O	O	O	**20**
																				O	O	O	O	O	O	O	O	O	O	O	**21**
																					O	O	O	O	O	X	O	O	O	O	**22**
																						O	O	O	O	O	O	O	O	O	**23**
																							O	O	O	O	X	O	O	O	**24**
																								O	O	O	O	O	O	O	**25**
																									O	O	O	O	O	O	**26**
																										O	O	O	O	O	**27**
																											X	O	O	O	**28**
																												O	X	X	**29**
																													O	O	**30**
																														O	**31**

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados

X = Há diferença estatística entre tratamentos

O = Não há diferença estatítica entre tratamentos

APÊNDICE C. Resultado da comparação dois a dois dos dados não ajustados do Experimento #1 usando o teste de Kruskal-Wallis

Tratamentos																															
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**1**
	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**2**
		O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**3**
			O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**4**
				O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**5**
					O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**6**
						O	O	O	O	O	O	O	O	O	X	X	**O**	X	X	X	X	X	X	X	X	X	O	X	X	X	**7**
							O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**8**
								O	O	O	O	O	O	O	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	**9**
									O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**10**
										O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**11**
											O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**12**
												O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**13**
													O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**14**
														O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**15**
															X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**16**
																O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**17**
																	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**18**
																		X	O	X	O	X	O	O	O	X	O	O	X	X	**19**
																			O	O	O	O	O	O	O	O	X	O	O	O	**20**
																				O	O	O	O	O	O	O	O	O	O	O	**21**
																					O	O	O	O	O	O	X	O	O	O	**22**
																						O	O	O	O	O	O	O	O	O	**23**
																							O	O	O	O	X	O	O	O	**24**
																								O	O	O	O	O	O	O	**25**
																									O	O	O	O	O	O	**26**
																										O	O	O	O	O	**27**
																											X	O	O	O	**28**
																												O	X	X	**29**
																													O	O	**30**
																														O	**31**

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados

X = Há diferença estatística entre tratamentos

O = Não há diferença estatítica entre tratamentos

APÊNDICE D. Resultado da comparação dois a dois dos dados ajustados do Experimento #2 usando o teste de Kruskal-Wallis

02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**1**
	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**2**
		O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**X**	X	X	X	X	**X**	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**3**
			O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**4**
				O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**5**
					O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**6**
						O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**7**
							O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**8**
								O	O	O	O	O	O	O	O	X	**X**	X	X	X	X	X	O	X	O	X	X	X	O	X	X	O	X	O	O	X	X	X	X	X	**X**	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**9**
									O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**10**
										O	O	O	O	O	**X**	X	X	X	X	X	X	X	**X**	X	X	X	X	X	O	X	X	**X**	X	O	**X**	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**11**
											O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**12**
												O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**13**
													O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**14**
														O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**15**
															X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**16**
																O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	**X**	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**17**
																	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	**18**
																		O	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**19**
																			O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	X	O	O	X	X	O	O	X	**20**
																				O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	**O**	X	X	X	X	X	X	**21**
																					O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	**X**	O	**X**	X	X	O	X	X	X	X	X	X	**22**
																						O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	**X**	O	**X**	X	X	O	X	X	X	X	X	X	**23**
																							O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**24**
																								O	O	O	O	O	O	X	O	O	O	O	O	O	O	**X**	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**25**
																									O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	**X**	O	O	O	**X**	X	O	X	X	X	X	X	X	**26**
																										O	O	O	O	X	O	O	O	O	O	O	O	**X**	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**27**
																											O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**X**	O	**X**	X	X	O	X	X	X	X	X	X	**28**
																												O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**29**
																													O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**X**	O	**X**	X	X	O	X	X	X	X	X	X	**30**
																														O	X	O	O	O	O	O	O	X	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**31**

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados
X = Há diferença estatística entre tratamentos
O = Não há diferença estatítica entre tratamentos

Continua na próxima página

APÊNDICE D. Resultado da comparação dois a dois dos dados ajustados do Experimento #2 usando o teste de Kruskal-Wallis (cont.)

02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**X**	**X**	**X**	X	X	O	X	X	X	X	X	X	**32**
																														X	O	X	X	X	O	O	O	O	O	X	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**33**
																														O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**34**
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	**35**
																																O	O	O	O	X	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**36**
																																	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**37**
																																		O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**38**
																																			O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	X	O	X	X	X	O	X	X	**39**
																																				O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	O	X	X	X	X	X	X	**40**
																																					O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	**41**
																																						O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	X	O	X	X	X	**X**	X	X	**42**
																																							O	O	O	O	O	O	O	O	X	**X**	O	**X**	X	X	O	X	X	X	X	X	X	**43**
																																								O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**44**
																																									O	O	**X**	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**45**
																																										O	O	O	O	X	X	**X**	O	**X**	X	X	O	X	X	X	X	X	**46**
																																											O	O	O	X	**X**	O	O	O	**X**	X	O	X	X	X	X	X	**47**
																																												O	O	O	O	O	O	O	O	X	O	O	O	O	X	**48**
																																													O	O	O	O	O	O	O	O	O	O	O	O	O	**49**
																																														O	O	O	O	O	O	O	O	O	O	O	**50**
																																															O	O	O	O	O	O	O	O	O	O	**51**
																																																O	O	O	O	O	O	O	O	O	**52**
																																																	O	O	O	O	O	O	O	O	**53**
																																																		O	O	O	O	O	O	O	**54**
																																																			O	O	O	O	O	O	**55**
																																																				O	O	O	O	O	**56**
																																																					O	O	O	O	**57**
																																																						O	O	O	**58**
																																																							O	O	**59**
																																																								O	**60**
																																																									O	**61**
																																																										O	**62**
																																																											O	**63**

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados
X = Há diferença estatística entre tratamentos
O = Não há diferença estatítica entre tratamentos

Fim

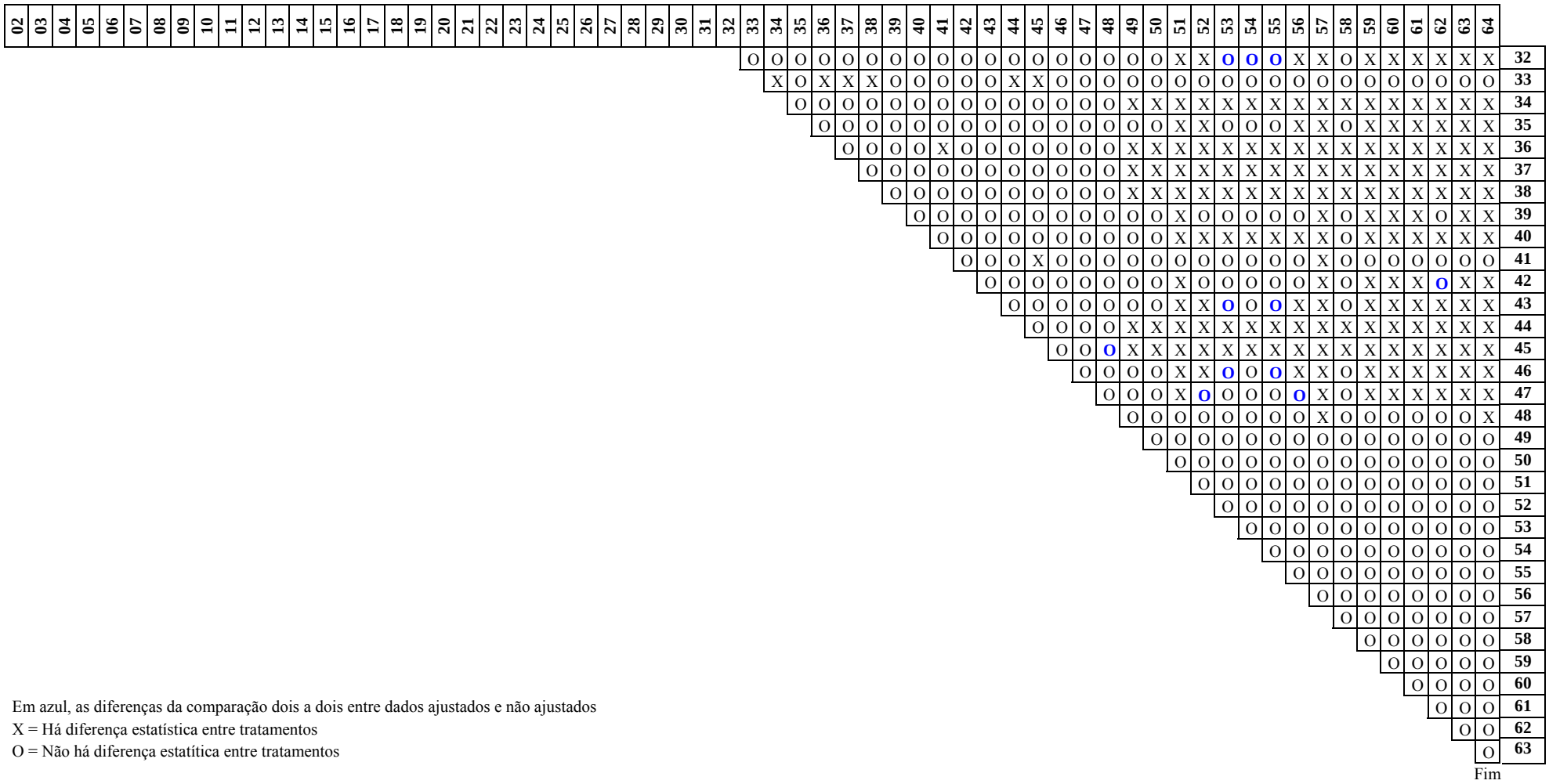
APÊNDICE E. Resultado da comparação dois a dois dos dados não ajustados do Experimento #2 usando o teste de Kruskal-Wallis

02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**1**
	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**2**
		O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**O**	X	X	X	X	**O**	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**3**
			O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**4**
				O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**5**
					O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**6**
						O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**7**
							O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**8**
								O	O	O	O	O	O	O	X	**O**	X	X	X	X	X	O	X	O	X	X	X	O	X	X	O	X	O	O	X	X	X	X	X	X	X	**O**	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**9**
									O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**10**
										O	O	O	O	O	**O**	X	X	X	X	X	X	**O**	X	X	X	X	X	O	X	X	**O**	X	O	**O**	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**11**
											O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**12**
												O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**13**
													O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**14**
														O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**15**
															X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**16**
																O	O	O	O	O	O	O	O	O	O	O	O	O	**O**	O	O	O	O	O	O	O	O	**O**	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**17**
																	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	**18**
																		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**19**
																			O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	O	X	X	X	X	X	X	X	X	**20**
																				O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	**X**	X	X	X	X	X	**21**
																					O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**O**	O	**O**	X	X	O	X	X	X	X	X	**22**
																						O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**O**	O	**O**	X	X	O	X	X	X	X	X	**23**
																							O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	**24**
																								O	O	O	O	O	O	X	O	O	O	O	**O**	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**25**
																									O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	**O**	O	O	O	**O**	X	O	X	X	X	X	X	X	**26**
																										O	O	O	O	X	O	O	O	O	**O**	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**27**
																											O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**O**	O	**O**	X	X	O	X	X	X	X	X	X	**28**
																												O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**29**
																													O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	**O**	O	**O**	X	X	O	X	X	X	X	X	X	**30**
																														O	X	O	O	O	O	O	O	X	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	**31**

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados
X = Há diferença estatística entre tratamentos
O = Não há diferença estatítica entre tratamentos

Continua na próxima página

APÊNDICE E. Resultado da comparação dois a dois dos dados não ajustados do Experimento #2 usando o teste de Kruskal-Wallis (cont.)



02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	32
																															X	O	X	X	X	O	O	O	O	O	X	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	33
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	34
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	35
																															O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	36
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	37
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	38
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	39
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	40
																															O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	41
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O	X	X	X	O	X	X	42
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	X	X	X	X	43
																															O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	44
																															O	O	O	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	45
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	X	O	O	O	X	X	O	X	X	X	X	X	X	46
																															O	O	O	O	X	O	O	O	O	O	O	O	O	O	O	O	O	X	O	O	O	O	O	X	O	X	X	X	X	X	X	47
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O	X	X	X	X	X	X	48
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	49
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	50
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	51
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	52
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	53
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	54
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	55
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	56
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	57
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	58
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	59
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	60
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	61
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	62
																															O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	63

Em azul, as diferenças da comparação dois a dois entre dados ajustados e não ajustados
X = Há diferença estatística entre tratamentos
O = Não há diferença estatítica entre tratamentos

Fim

ANEXO A. Distribuição Qui-Quadrado [Montgomery 2005]

v \ α	0,995	0,99	0,975	0,95	0,90	0,75	0,50	0,25	0,10	0,05	0,025	0,01	0,005	0,001
1	0,0004	0,002	0,001	0,004	0,016	0,102	0,455	1,323	2,706	3,841	5,024	6,635	7,879	10,828
2	0,010	0,020	0,051	0,103	0,211	0,575	1,386	2,773	4,605	5,991	7,378	9,210	10,597	13,816
3	0,072	0,115	0,216	0,352	0,584	1,213	2,366	4,108	6,251	7,815	9,348	11,345	12,838	16,266
4	0,207	0,297	0,484	0,711	1,064	1,923	3,357	5,385	7,779	9,488	11,143	13,277	14,860	18,467
5	0,412	0,554	0,831	1,145	1,610	2,675	4,351	6,626	9,236	11,071	12,833	15,086	16,750	20,515
6	0,676	0,872	1,237	1,635	2,204	3,455	5,348	7,841	10,645	12,592	14,449	16,812	18,548	22,458
7	0,989	1,239	1,690	2,167	2,833	4,255	6,346	9,037	12,017	14,067	16,013	18,475	20,278	24,322
8	1,344	1,646	2,180	2,733	3,490	5,071	7,344	10,219	13,362	15,507	17,535	20,090	21,955	26,125
9	1,735	2,088	2,700	3,325	4,168	5,899	8,343	11,389	14,684	16,919	19,023	21,666	23,589	27,877
10	2,156	2,558	3,247	3,940	4,865	6,737	9,342	12,549	15,987	18,307	20,483	23,209	25,188	29,588
11	2,603	3,053	3,816	4,575	5,578	7,584	10,341	13,701	17,275	19,675	21,920	24,725	26,757	31,264
12	3,074	3,571	4,404	5,226	6,304	8,438	11,340	14,845	18,549	21,026	23,337	26,217	28,299	32,909
13	3,565	4,107	5,009	5,892	7,042	9,299	12,340	15,984	19,812	22,362	24,736	27,688	29,819	34,528
14	4,075	4,660	5,629	6,571	7,790	10,165	13,339	17,117	21,064	23,685	26,119	29,141	31,319	36,123
15	4,601	5,229	6,262	7,261	8,547	11,036	14,339	18,245	22,307	24,996	27,488	30,578	32,801	37,697
16	5,142	5,812	6,908	7,962	9,312	11,912	15,338	19,369	23,542	26,296	28,845	32,000	34,267	39,252
17	5,697	6,408	7,564	8,672	10,085	12,792	16,338	20,489	24,769	27,587	30,191	33,409	35,718	40,790
18	6,265	7,015	8,231	9,390	10,865	13,675	17,338	21,605	25,989	28,869	31,526	34,805	37,156	43,312
19	6,844	7,633	8,907	10,117	11,651	14,562	18,338	22,718	27,204	30,144	32,852	36,191	38,582	43,820
20	7,434	8,260	9,591	10,851	12,443	15,452	19,337	23,828	28,412	31,410	34,170	37,566	39,997	45,315
21	8,034	8,897	10,283	11,591	13,240	16,344	20,337	24,935	29,615	32,671	35,479	38,932	41,401	46,797
22	8,643	9,542	10,982	12,338	14,042	17,240	21,337	26,039	30,813	33,924	36,781	40,289	42,796	48,268
23	9,260	10,196	11,689	13,091	14,848	18,137	22,337	27,141	32,007	35,172	38,076	41,638	44,181	49,728
24	9,886	10,856	12,401	13,848	15,659	19,037	22,337	28,241	33,196	36,415	39,364	42,980	45,559	51,179
25	10,520	11,524	13,120	14,611	16,473	19,939	24,337	29,339	34,382	37,652	40,646	44,314	46,928	52,620
26	11,160	12,198	13,844	15,379	17,292	20,843	25,336	30,434	35,563	38,885	41,923	45,642	48,290	54,052
27	11,808	12,879	14,573	16,151	18,114	21,749	26,336	31,528	36,741	40,113	43,194	46,963	49,645	55,476
28	12,461	13,565	15,308	16,928	18,939	22,657	27,336	32,620	37,916	41,337	44,461	48,278	50,993	56,892
29	13,121	14,257	16,047	17,708	19,768	23,567	28,336	33,711	39,087	42,557	45,722	49,588	52,336	58,302
30	13,787	14,954	16,791	18,493	20,599	24,478	29,336	34,800	40,256	43,773	46,979	50,892	53,672	59,703
31	14,458	15,655	17,539	19,281	21,434	25,390	30,336	35,887	41,422	44,985	48,232	52,191	55,003	61,098
32	15,134	16,362	18,291	20,072	22,271	26,304	31,336	36,973	42,585	46,194	49,480	53,486	56,328	62,487
33	15,815	17,074	19,047	20,867	23,110	27,219	32,336	38,058	43,745	47,400	50,725	54,776	57,648	63,870
34	16,501	17,789	19,806	21,664	23,952	28,136	33,336	39,141	44,903	48,602	51,966	56,061	58,964	65,247
35	17,192	18,509	20,569	22,465	24,797	29,054	34,336	40,223	46,059	49,802	53,203	57,342	60,275	66,619
36	17,887	19,233	21,336	23,269	25,643	29,973	35,336	41,304	47,212	50,998	54,437	58,619	61,581	67,985
37	18,586	19,960	22,106	24,075	26,492	30,893	36,336	42,383	48,363	52,192	55,668	59,892	62,883	69,346
38	19,289	20,691	22,878	24,884	27,343	31,815	37,335	43,462	49,513	53,384	56,896	61,162	64,181	70,701
39	19,996	21,426	23,654	25,695	28,196	32,737	38,335	44,539	50,660	54,572	58,120	62,428	65,476	72,055
40	20,707	22,164	24,433	26,509	29,051	33,660	39,335	45,616	51,805	55,758	59,342	63,691	66,766	73,402
41	21,421	22,906	25,215	27,326	29,907	34,585	40,335	46,692	52,949	56,942	60,561	64,950	68,053	74,745
42	22,138	23,650	25,999	28,144	30,765	35,510	41,335	47,766	54,090	58,124	61,777	66,206	69,336	76,084
43	22,859	24,398	26,785	28,965	31,625	36,436	42,335	48,840	55,230	59,304	62,990	67,459	70,616	77,419
44	23,584	25,148	27,575	29,787	32,487	37,363	43,335	49,913	56,369	60,481	64,201	68,710	71,893	78,750
45	24,311	25,901	28,366	30,612	33,350	38,291	44,335	50,985	57,505	61,656	65,410	69,957	73,166	80,077
50	27,991	29,707	32,357	34,764	37,689	42,942	49,335	56,334	63,167	67,505	71,420	76,154	79,490	86,661
60	35,534	37,485	40,482	43,188	46,459	52,294	59,335	66,981	74,397	79,082	83,298	88,379	91,952	99,607
70	43,275	45,442	48,758	51,739	55,329	61,698	69,335	77,577	85,527	90,531	95,023	100,425	104,215	112,317
80	51,172	53,540	57,153	60,391	64,278	71,145	79,335	88,130	96,578	101,879	106,629	112,329	116,321	124,839
90	59,196	61,754	65,647	69,126	73,291	80,625	89,335	98,650	107,565	113,145	118,136	124,116	128,299	137,208
100	67,328	70,065	74,222	77,929	82,358	90,133	99,335	109,141	118,498	124,342	129,561	135,807	140,169	149,449