UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LX-MCAPI: BIBLIOTECA DE COMUNICAÇÃO PARA SUPORTE A PROGRAMAÇÃO PARALELA EM SISTEMAS MULTI-CORE

ANTONIO DIOGO HIDEE IDEGUCHI

ORIENTADOR: PROF. DR. MÁRCIO MERINO FERNANDES
CO-ORIENTADOR: PROF. DR. CELIO ESTEVAM MORON

São Carlos – SP

Março/2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LX-MCAPI: BIBLIOTECA DE COMUNICAÇÃO PARA SUPORTE A PROGRAMAÇÃO PARALELA EM SISTEMAS MULTI-CORE

ANTONIO DIOGO HIDEE IDEGUCHI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Metodologias e Técnicas de Computação
Orientador: Prof. Dr. Márcio Merino Fernandes

São Carlos – SP

Março/2016

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Ideguchi, Antonio Diogo Hidee
I19l LX-MCAPI : biblioteca de comunicação para suporte a programação paralela em sistemas multi-core / Antonio Diogo Hidee Ideguchi. -- São Carlos : UFSCar, 2016.
139 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2016.

1. Multi-core. 2. Paralelismo. 3. Metodologia. 4. Programação. 5. Desempenho. I. Título.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Antonio Diogo Hidee Ideguchi, realizada em 12/05/2016:

Prof. Dr. Márcio Merino Fernandes
UFSCar

Prof. Dr. Célio Estevan Moron
UFSCar

Prof. Dr. Alexandro José Baldassin
UNESP

RESUMO

Os processadores *multi-core* representaram a resposta da indústria às barreiras físicas encontradas no desenvolvimento de processadores computacionais nas últimas décadas, e trouxeram novo fôlego ao avanço do desempenho de sistemas computacionais. Os complexos processadores superescalares de núcleo único com frequências de *clock* relativamente altas deram espaço a unidades de processamento com dois ou mais núcleos em um mesmo encapsulamento, geralmente mais "lentos", possibilitando uma ou mais *threads* por núcleo. Nesse contexto, os modelos de programação existentes utilizando os paradigmas sequencial e concorrente não permitiam a exploração do potencial real proporcionado pelos novos elementos de hardware introduzidos, gerando uma necessidade de criação de novas metodologias de programação que permitissem tirar proveito do paralelismo agregado à utilização dos processadores *multi-core*. Este trabalho apresenta a **LX-MCAPI**, biblioteca baseada em mecanismos modernos de IPC (*Inter-Process Communication*) e compartilhamento de memória, desenvolvida sobre a hipótese em que a passagem de mensagens é uma abstração viável, flexível e escalável, quando comparada a métodos de programação convencionais utilizando memória-compartilhada em sistemas *multi-core*. LX-MCAPI oferece um mecanismo de passagem de mensagem e compartilhamento *zero-copy* de memória entre processos, além de padrões de programação paralela prontos para uso, que facilitam o processo de abstração e construção de aplicações. Além disso, apresentando bom desempenho em termos de latências de transmissão e taxas de transferência em ambientes x86-64 e ARM.

Palavras-chave: *multi-core*; paralelismo; metodologia; programação; desempenho; mensagens; memória-compartilhada

ABSTRACT

The multi-core processors represent the industry response for the physical barriers encountered during the development of computing processors during the last decades, and brought new advances on computing system performance. The complex superscalar unicore processors with high frequency clocks gave way to processing units with two or more cores in just one encapsulation, generally with low clock frequencies, allowing one or more execution threads per core. On this context, the existing programming models using serial and concurrent paradigms do not allow exploring the real potential provided by the new hardware elements incorporated, generating a necessity of new programming methodologies that does allow exploring parallelism aggregated by the use of multi-core processors. This work presents **LX-MCAPI**, a library based on modern IPC (*Inter-Process Communication*) and memory sharing mechanisms, developed over the hypothesis that message passing is a viable, flexible and scalable abstraction, compared to conventional programming methods using shared-memory on multi-core systems. LX-MCAPI offers a message-passing, zero-copy memory sharing mechanism between processes and ready to use scalability patterns to facilitate the process of abstraction and construction of applications. It has performed well in therms of transmission latency and transfer rate on x86-64 and ARM environments.

Keywords: multi-core, parallelism, methodology, programming, performance, message passing; shared-memory

LISTA DE FIGURAS

2.1 Estrutura de um Processador de Núcleo Único 23

2.2 Estrutura de um Processador de Múltiplos Núcleos 25

2.3 Arquitetura Homogênea (Simétrica) (esq.) e Arquitetura Heterogênea (Assimétrica) (dir.) 36

3.1 Arquitetura do Procolo LINX (CHRISTOFFERSON, 2006) 57

4.1 LX-MCAPI - Diagrama conceitual de Módulos 76

4.2 LX-MCAPI - Representação em camadas 77

4.3 LX-MCAPI - Elementos de conexão e de mensagem 77

4.4 LX-MCAPI - Transmissão de Escalares e Pacotes (até 16KiB) (v1.0) 81

4.5 LX-MCAPI - Transmissão de Pacotes (a partir de 16KiB) (v1.0) 82

4.6 LX-MCAPI Zero - Transmissão de Pacotes Zero-Copy (Modo SHARED) . . . 83

4.7 LX-MCAPI Zero - Transmissão de Pacotes Zero-Copy (Modo COPY) 84

4.8 Padrão de Mensagens - PAIR 85

4.9 Padrão de Mensagens - Request/Reply 86

4.10 Padrão de Mensagens - Publish/Subscribe 87

4.11 Padrão de Mensagens - Push/Pull 89

5.1 Teste de Taxa de Transferência 92

5.2 Taxa de Transferência para Escalares (ARM e x86-64) 92

5.3 Taxa de Transferência para Pacotes (ARM e x86-64) 93

5.4 Taxa de Transferência para Mensagens (ARM e x86-64) 93

5.5 Teste de Latência Roundtrip 94

5.6 Tempos de Roundtrip para Escalares (ARM e x86-64) 95
5.7 Tempos de Roundtrip para Pacotes (ARM e x86-64) 95
5.8 Tempos de Roundtrip para Mensagens (Plataforma x86-64) 96
5.9 Pacotes x Mensagens (ARM e x86-64) 97
5.10 Taxa de Transferência - Pacotes (LX-MCAPI) x MPI (ARM e x86-64) 98
5.11 RTT - Pacotes (LX-MCAPI) x MPI (ARM e x86-64) 99
5.12 Taxa de Transferência - Pacotes (LX-MCAPI) x nanomsg (ARM e x86-64) . . 100
5.13 RTT - Pacotes (LX-MCAPI) x nanomsg (ARM e x86-64) 100
5.14 LX-MCAPI Patterns - Exemplo PAIR 102
5.15 LX-MCAPI Patterns - Exemplo Request/Reply 104
5.16 LX-MCAPI Patterns - Exemplo Publish/Subscribe 107
5.17 LX-MCAPI Patterns - Exemplo Push/Pull 109
5.18 Representação visual do Conjunto de Mandelbrot 111
5.19 Geração paralela do Conjunto de Mandelbrot 113
5.20 Regiões de Interesse do Olho Humano para Reconhecimento de Íris 117
5.21 Modelo Rubbersheet - Extraído de: (MASEK et al., 2003) 120
5.22 Resultado de aplicação do modelo *rubbersheet* 120
5.23 Problema de Oclusão no processo de Normalização 121

LISTA DE TABELAS

2.1 Mecanismos de IPC 30

3.1 Desafios enfrentados pelos programadores paralelos [Adapt. (MEADE; BUCKLEY; COLLINS, 2011)] 49

4.1 Principais funções da especificação MCAPI [Adaptada de: (VIRTANEN et al., 2014)] 64

4.2 Codificação de Assinaturas do D-Bus 72

4.3 Correspondência de denominações: MCAPI x D-Bus 79

4.4 Principais funções do módulo LX-MCAPI Zero 84

4.5 Principais funções do padrão PAIR 86

4.6 Principais funções do padrão Request/Reply 87

4.7 Principais funções do padrão Publish/Subscribe 88

4.8 Principais funções do padrão Push/Pull 89

5.1 Ferramentas e Recursos Utilizados 91

5.2 Conjunto de Mandelbrot - Compilação otimizada x não-otimizada (SHM) . . . 115

5.3 Conjunto de Mandelbrot - Tempo de Geração (μs) e *Speedup* 116

5.4 Processo de Detecção da Pupila 118

5.5 Processo de Detecção da Iris 119

5.6 Tempo de Processamento Sequencial: Pupila 125

5.7 Tempo de Processamento Sequencial: Iris 125

5.8 Transformada de Hough (Pupila) - Tempo de Processamento (μs) e *Speedup* . . 126

A.1 TR e RTT para LX-MCAPI (Pacotes) e MPICH2 (Mensagens) - x86-64 132

A.2 TR e RTT para LX-MCAPI (Pacotes) e MPICH2 (Mensagens) - ARM 133
A.3 TR e RTT para LX-MCAPI (Pacotes) e nanomsg - x86-64 133
A.4 TR e RTT para LX-MCAPI (Pacotes) e nanomsg - ARM 134

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO **12**

1.1 Contexto 12

1.2 Motivação e Justificativa 13

1.3 Objetivos 14

1.3.1 Objetivos Gerais 14

1.3.2 Objetivos Específicos 14

1.4 Publicações 15

1.5 Organização do Trabalho 16

CAPÍTULO 2 – CONCEITOS BÁSICOS **17**

2.1 Histórico da Computação Paralela 17

2.2 Classificação de Sistemas Computacionais 21

2.3 Os Processadores de Núcleo Único 22

2.4 O Advento dos Processadores Multi-core 24

2.5 A Programação Paralela 27

2.5.1 Elementos, Condições e Mecanismos de IPC 28

2.5.2 Memória Compartilhada 31

2.5.3 Memória Distribuída e Passagem de Mensagem 32

2.5.4 Concorrência x Paralelismo Real 33

2.5.5 Pensamento Paralelo 33

2.6 Arquiteturas Multi-core 35
2.6.1 Arquitetura de Hardware 35
2.6.2 Modelos de Programação para Multi-core 37
2.7 O Futuro dos Processadores Multi-core 44
2.7.1 O Problema do Dark Silicon 44
2.7.2 Outros Problemas de Escalabilidade 46
2.8 Considerações 47

CAPÍTULO 3 – TRABALHOS RELACIONADOS **48**

3.1 Programação Paralela Multi-core 48
3.2 Conceitos Emergentes 52
3.2.1 Multitask para Multi-core 52
3.2.2 Mecanismos Modernos de IPC baseados em Mensagens 53
3.2.3 Otimização do MPI e Modelos Híbridos 57
3.2.4 Memória Transacional 58
3.3 Tendências 60
3.4 Considerações 60

CAPÍTULO 4 – PROPOSTA DE BIBLIOTECA DE COMUNICAÇÃO **62**

4.1 Especificação MCAPI 62
4.1.1 Detalhes da Especificação 63
4.1.2 Implementações Existentes 66
4.2 Protocolo D-Bus 67
4.2.1 Arquitetura do Protocolo 68
4.2.2 *Buses*, Endereços e Conexões 68
4.2.3 Modelo de Objetos 70
4.2.4 Propostas e Otimizações 73

4.3 LX-MCAPI - Projeto e Implementação 75
4.3.1 Ambiente de Desenvolvimento 76
4.3.2 LX-MCAPI 76
4.3.3 LX-MCAPI Patterns 84

CAPÍTULO 5 – RESULTADOS EXPERIMENTAIS **90**

5.1 LX-MCAPI 90
5.1.1 Análise de Resultados 90
5.1.2 Ferramentas e Recursos 91
5.1.3 Taxa de Transferência (TR) 91
5.1.4 Latência Roundtrip (RTT) 94
5.1.5 Pacotes vs. Mensagens 96
5.1.6 LX-MCAPI vs. MPICH2 97
5.1.7 LX-MCAPI vs. nanomsg 99
5.1.8 Resultados Obtidos 101
5.2 LX-MCAPI Patterns 102
5.2.1 Padrão Pair 102
5.2.2 Padrão Request/Reply 104
5.2.3 Padrão Publish/Subscribe 107
5.2.4 Padrão Push/Pull 108
5.3 Aplicações 111
5.3.1 Conjunto de Mandelbrot 111
5.3.2 Detecção, Segmentação e Normalização de Iris 117
5.3.3 Considerações 126

CAPÍTULO 6 – CONCLUSÃO **127**

GLOSSÁRIO **130**

APÊNDICE A – LX-MCAPI - TABELAS DE RESULTADOS **132**

REFERÊNCIAS BIBLIOGRÁFICAS **135**

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Nos últimos anos, processadores com mais de um núcleo (denominados multi-core) ganharam espaço em vários segmentos da indústria, e permitiram a continuidade no avanço tecnológico relacionado ao desempenho regido pela intitulada Lei de Moore (AMDAHL, 1967), onde é esperado que, a cada aproximadamente 18 meses, o número de transistores utilizados em um processador dobre, permitindo assim um gradual aumento no desempenho dos mesmos.

De maneira simples, um processador *multi-core* é uma unidade computacional com duas ou mais unidades de processamento central (denominadas *cores* ou núcleos) dentro de si, que executam instruções simultaneamente, aumentando o *throughput* de instruções executadas quando comparado a somente um núcleo executando instruções, assim possibilitando a exploração do paralelismo.

O paralelismo é um conceito relativamente comum: efetuar mais de uma tarefa simultaneamente. Contudo, nem todas as tarefas são compatíveis com o conceito de paralelização, pelo menos não totalmente: não podemos colorir um desenho antes de desenhar seus contornos, porém duas ou mais pessoas pode colorir um mesmo desenho ao mesmo tempo, assim como mais de uma pessoa pode fazer os contornos.

Computacionalmente, não é trivial efetuar várias tarefas em paralelo, pois tal como as tarefas cotidianas, nem todas são paralelizáveis e algumas apesar de paralelizáveis, não há maneira simples de fazê-lo. Cálculos que dependem de resultados anteriores, alguns tipos de filtros para imagens, são exemplos de tarefas com paralelização não-trivial.

Os próximos itens abordarão os conceitos básicos que proporcionarão o entendimento e a justificativa para a exploração das arquiteturas *multi-core* e quais são os pontos de interesse

e as principais limitações a serem compreendidas neste trabalho para o desenvolvimento de aplicações paralelas que desfrutam dos benefícios proporcionados por tais arquiteturas.

1.2 Motivação e Justificativa

A programação concorrente traz dificuldades características para o desenvolvimento de aplicações. Ao explorar a concorrência, condições como os *deadlocks*, *livelocks*, condições de corrida (*race conditions*), compartilhamento de recursos via regiões críticas, são cruciais e, em muitos casos, pouco triviais e não-escaláveis.

O processo de desenvolvimento de aplicações paralelas para arquiteturas *multi-core* é um tópico recente e há novas metodologias e tecnologias emergindo, mas nenhuma pode ser definida como consolidada e geralmente envolvem uma curva de aprendizado longa para o aprendizado de uma nova linguagem, recursos diferenciados ou paradigmas mais complexos.

Inicialmente, no contexto do presente trabalho, procurou-se explorar métodos intuitivos para paralelização de aplicações através da abstração de passagem de mensagens, visando principalmente aplicações que apresentavam *loops* que consumiam grandes quantidades de tempo de processamento. Esta pesquisa data de trabalhos anteriores voltados à conversão de aplicações *multitask* para multi-core, que culminaram na publicação de (MORON et al., 2014), formalizando um processo que utilizava o *kernel de tempo-real* QuadrosRTXC e sua ferramenta de programação VisualRTXC. No processo de paralelização proposto utilizavam-se *Linux pipes* como o principal método de transferência de dados, substituindo as filas de mensagem da aplicação *multitask*, permitindo assim o desacoplamento entre processos.

A metodologia provou-se efetiva para algumas situações, porém exigia grande esforço por parte do programador, pois este era responsável por gerir todas as chamadas de sistema para criação de *pipes* entre os processos, incluindo a definição e compartilhamento de todos os identificadores e descritores de arquivos, abrindo espaço para erros lógicos e de sintaxe. Além disso, a utilização de *pipes* para transmissão de elementos de mensagem relativamente grandes e/ou com alta frequência não proporcionava bom desempenho, e poderia até mesmo eliminar o *speedup* obtido com o processo de paralelização.

É de grande interesse o desenvolvimento de novas metodologias que facilitem o uso de arquiteturas multi-core para a aceleração de aplicações que necessitem de alto desempenho. A utilização de ferramentas e metodologias simplificadas que permitam extrair tal desempenho pode reduzir a curva de aprendizado e permitir a utilização mais efetiva do potencial oferecido pelos múltiplos núcleos de maneira intuitiva. Neste cenário, concebeu-se a LX-MCAPI, bibli-

oteca de comunicação entre processos que objetiva a simplicidade e provimento de abstrações para o programador paralelo, ainda oferecendo razoáveis tempos de latência de transmissão quando comparada à ferramentas existentes.

1.3 Objetivos

1.3.1 Objetivos Gerais

O desenvolvimento de processadores de arquitetura *multi-core*, como abordado nas seções anteriores, permitiu a continuidade no aumento de desempenho dos computadores e trouxe a possibilidade de utilização de estruturas de paralelização de alto desempenho até mesmo a usuários domésticos com a popularização das linhas de até oito núcleos comercializadas principalmente pela IBM, ARM, Intel, AMD, Qualcomm, dentre outras.

As novas estruturas apresentadas por essa arquitetura permitem melhor acoplamento entre os núcleos de processamento, favorecendo a utilização de mecanismos de comunicação entre núcleos e utilização eficiente dos módulos de memória *cache*, comparados aos modelos de multiprocessadores anteriores.

O objetivo primário deste trabalho foi a investigação da hipótese em que, com os avanços tecnológicos proporcionados por arquiteturas *multi-core*, o modelo de programação baseado em passagem de mensagens pode ser uma alternativa viável comparado ao modelo baseado em memória compartilhada, proporcionando vantagens conhecidas quanto a escalabilidade da aplicação conforme disponibilidade de recursos.

As pesquisas e desenvolvimentos realizados em torno desta hipótese culminou no desenvolvimento da biblioteca LX-MCAPI, alvo principal das discussões realizadas no presente trabalho.

1.3.2 Objetivos Específicos

No intuito de avaliar a hipótese apresentada, objetivos específicos foram definidos para o desenvolvimento do trabalho proposto. Devido à variedade de técnicas, paradigmas e arquiteturas envolvidas nos conceitos de paralelização de aplicações, foi possível avaliar os diversos parâmetros envolvidos em cada uma delas.

Os seguintes itens foram abordados:

- Definição de modelos e características da arquitetura multi-core
- Definição de aplicações e *benchmarks* de interesse - conjunto de aplicações para avaliação de desempenho
- Definição de políticas e módulos a serem paralelizados
- Implementação, testes e comparações considerando as seguintes alternativas:
  - Versão sequencial
  - Versão concorrente em memória compartilhada (POSIX)
  - Versão paralela utilizando passagem de mensagem
  - Versão paralela utilizando mecanismo zero-copy
- *Profiling* de aplicações utilizando a ferramenta TAU (*Tuning and Analysis Utilities*, (SHENDE; MALONY, 2006)) para geração de relatórios sobre desempenho
- Análise comparativa considerando:
  - Desempenho
  - Facilidade de Programação
  - Escalabilidade

1.4 Publicações

As pesquisas, implementações e experimentos relacionados ao desenvolvimento deste trabalho geraram as seguintes publicações:

- *From MultiTask to MultiCore: Design and Implementation Using an RTOS.* IEEE 13th International Symposium on Parallel and Distributed Computing, França, 2014.
- *CHAOS-MCAPI: An Optimized Mechanism to Support Multicore Parallel Programming.* 27th International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), Florianópolis, Brasil, 2015.

1.5 Organização do Trabalho

Este trabalho está organizado na seguinte forma:

- No Capítulo 2 apresentam-se o contexto e atuais arquiteturas e paradigmas paralelos.
- No Capítulo 3 encontra-se a revisão bibliográfica referente a área de programação voltada a *multi-core*.
- No Capítulo 4 constam os conceitos específicos para a elaboração e concepção da LX-MCAPI, analisada em nível de implementação, incluindo detalhes e estratégias.
- No Capítulo 5 os resultados e a discussão sobre os dados obtidos através de testes e algoritmos são apresentados.
- No Capítulo 6 constam as conclusões sobre o desenvolvimento do trabalho em questão.

Capítulo 2

CONCEITOS BÁSICOS

2.1 Histórico da Computação Paralela

De acordo com a descrição sobre o histórico da computação paralela em (HOCKNEY R.W. E JESSHOPE, 1983), a primeira geração de computadores nos anos 1950 utilizava válvulas eletrônicas como principais componentes de chaveamento e trabalhavam com tempos da ordem de microssegundos por instrução, ocupavam grandes salões e consumiam grandes quantidades de energia. Na década de 1960, com o advento do transistor de germânio, reduziu-se a escala de tempo para décimos de microssegundo. Fizeram parte da segunda geração de computadores máquinas como o IBM 7090. Em meados da década de 1960 ainda, os circuitos integrados bipolares planares impressos em silício reduziram a escala de tempo novamente, agora para a casa dos nanossegundos e foram melhorados até meados da década de 1970. Até a década de 1980, processadores com o desempenho e capacidade equivalentes aos fisicamente grandes computadores da primeira geração estavam disponíveis em encapsulamentos únicos, medindo poucos milímetros quadrados de silício.

Em termos gerais, a arquitetura da primeira geração de computadores é descrita como sequencial e segue as ideias fundamentais de programas armazenados em memória, usualmente referida como organização *Von Neumann*. Computadores nessa categoria são compostos por unidades de entrada e saída, uma única memória para instruções e dados, uma única unidade de processamento para controle e interpretação de instruções, e uma unidade lógica e aritmética para processamento de dados. As duas últimas unidades são referidas conjuntamente como CPU, ou Unidade de Processamento Central.

Avanços na engenharia de semicondutores e processadores proporcionaram a possibilidade de implementação de arquiteturas paralelas que até então só existiam em teoria. O paralelismo

em si se refere a capacidade de sobrepor ou executar simultaneamente mais de uma tarefa (no caso, mais de uma instrução por intervalo de tempo). As principais maneiras de introduzir paralelismo na arquitetura de computadores são sumarizadas através de quatro itens, citados abaixo, considerando que estes podem ser combinados, associando as estratégias de paralelismo e buscando melhor aproveitamento de recursos computacionais.

- *Pipelining* – aplicação de técnicas de "linha de montagem" para aumentar o desempenho de unidades aritméticas ou de controle. Aumento do *throughput* por unidade de tempo com a introdução de estruturas de *pipeline* de instruções – amplamente utilizada.
- Funcional – provimento de várias unidades independentes que executam diferentes funções, sejam elas aritméticas ou lógicas, permitindo que operem simultaneamente sobre dados diversos. Confunde-se com o conceito de Multiprocessamento.
- *Array* (Vetorial) – provimento de diversos elementos de processamento (PE, do inglês, *processing elements*) sob um controlador comum, todos executando simultaneamente o mesmo tipo de instrução porém em diferentes dados armazenados em suas memórias privadas. Conceito que definiu a arquitetura das GPU nas últimas décadas.
- *Multiprocessing* (Multiprocessamento) – provimento de diversos processadores independentes, obedecendo suas próprias instruções e usualmente comunicando-se através de uma memória comum. Conceito que rege a arquitetura de sistemas distribuídos e *multi-core*.

A década de 1950 foi um período onde surgiram diversos estudos e discussões sobre arquiteturas de computadores, passando por máquinas analíticas, modelos de computação para resolução de problemas matemáticos, princípios como o de espacialidade proposto por Von Neumann em 1952, que podem ser considerados precursores dos supercomputadores SOLOMON, ILLIAC IV e ICL DAP na década de 1970. Apesar dos conceitos de paralelismo datarem desde o primeiro motor analítico proposto por Babbage em 1842, devido a impedimentos tecnológicos e de complexidade, a tecnologia de computação manteve-se com os conceitos de operação sequencial até que os avanços permitissem dar um passo à frente.

Entre a década de 1960 e 1970 começou o desenvolvimento de processadores escalares rápidos, que consistiram na introdução de mais estruturas de paralelismo aos processadores SISD existentes. Um computador escalar é aquele que provê instruções para manipulação de dados compreendendo um único número, em contraste com os computadores vetoriais que operam sobre uma quantidade ordenada de números.

Um dos computadores icônicos da década de 1960 foi o ATLAS, concebido na Universidade de Manchester por volta de 1956, com seu protótipo em funcionamento em 1961 e posteriormente produzido comercialmente pela Ferranti Ltd (posteriormente ICL – International Computers Ltd) em 1963. Era conhecido principalmente por ser pioneiro no uso de um sistema operacional com multiprogramação complexa, introduzindo sistemas de interrupção, mapeamento e paginação de memória. Além disso, contava com uma unidade aritmética que efetuava certas operações em paralelo e a memória principal do núcleo era dividida em quatro bancos de memória independentes que aceleravam operações em condições favoráveis. Já implementava princípios de *pipelining* para aceleração de *throughput* de instruções.

No intuito de fazer uso efetivo de atributos paralelos como múltiplas unidades aritméticas, registradores e memórias, era necessário um mecanismo que pudesse prever quais instruções do fluxo sequencial poderiam ser executadas concorrentemente sem que a lógica do programa fosse corrompida, e ao detectar possibilidade de paralelização, seria necessário "agendar" as unidades de processamento para execução ótima das instruções. Estes aspectos foram explorados e incluídos nos computadores escalares CDC 6600 e IBM360/91.

Até o fim da década de 1970, empresas como a IBM, CDC, Texas Instruments, Cray e outras, avançaram na incorporação de tecnologias cada vez mais avançadas em seus supercomputadores, explorando diversos paradigmas, aumentado cada vez mais a velocidade de processamento, complexidade de circuitos e estágios de *pipeline* (quando aplicáveis). Grandes laboratórios de pesquisa acadêmicos, privados e militares nos EUA encomendaram computadores a estas fabricantes. No auge da exploração espacial e da guerra-fria ocorreram altos investimentos em tecnologia, permitindo que projetos cada vez mais audaciosos fossem implementados. Até então, o paralelismo era explorado a nível de instruções e visava reduzir cada vez mais o tempo ocioso do processador. O grande salto que permitiu o avanço rápido dos computadores na década de 1970 foi a invenção dos primeiros microprocessadores, que encapsulavam todas as unidades funcionais necessárias para cômputo de instruções.

Os primeiros anos da década de 1980 foram testemunhas de uma mudança de grande relevância na arquitetura de computadores, passando das máquinas sequenciais, contando com um único processador rápido, para modelos que empregavam mais de um processador "não tão rápido" trabalhando em conjunto. Programar computadores paralelos era mais complexo, mas os benefícios projetados valiam os esforços, e com o tempo, pesquisadores encontraram maneiras de utilizar máquinas paralelas para essencialmente todas as aplicações científicas.

A partir daquele momento, formas reais de paralelismo eram possíveis, pois havia mais de uma unidade funcional independente disponível para processamento, permitindo a execução de

tarefas, literalmente ao mesmo tempo, não mais competindo por fatias de tempo ou necessariamente utilizando mecanismos extremamente complexos de *pipelining* e previsão de desvios no fluxo de instruções.

Enquanto os supercomputadores de 1980 usavam apenas alguns processadores, na década de 1990 máquinas com milhares de processadores começaram a aparecer nos EUA e Japão, definindo novos recordes de desempenho computacional. Computadores como o *Fujitsu Numerical Wind Tunnel* contava com 166 processadores vetoriais e alcançava 1.7 gigaFLOPS por processador; o Intel Paragon contava com 1000 a 4000 processadores Intel i860 em várias configurações, conectados por uma malha bidimensional, permitindo que os processos fossem executados em nós independentes, comunicando-se através de passagem de mensagens (MPI). Até o final da década de 1990, uma grande quantidade de supercomputadores competia pelo posto de mais rápido do mundo. Simultaneamente, os computadores pessoais se tornaram parte da vida de grande parte da população, e aplicações demandavam cada vez mais desempenho com o passar dos anos.

O aumento na demanda de desempenho é um fenômeno intrínseco do uso dos computadores atualmente: não importa quão rápidos fiquem os processadores, os softwares consistentemente encontram novas maneiras de consumir o processamento extra disponibilizado.

No início da década de 2000, a abordagem usada pelos principais fabricantes de processadores era desenvolver um processador superescalar extremamente complexo com técnicas para operações simultâneas acopladas e usando tecnologias estado-da-arte para obter encapsulamentos de alta densidade e alta frequência de *clock*. Contudo, essa abordagem se encerrou.

Limitações físicas impediram a continuidade no desenvolvimento dos processadores de núcleo único. Frequências de *clock* próximas a 4 GHz foram até então o limite factível, pois a tecnologia não conseguiria prover muito mais devido as leis da física e o consumo excessivo de energia associados ao aumento da velocidade do *clock* e contagem de transistores. Essa limitação foi denominada ***power wall*** (PANKRATIUS; ADL-TABATABAI; TICHY, 2011).

Ainda na década de 1990, identificou-se o que seria conhecido como ***memory wall***, causada pela diferença crescente entre a velocidade dos processadores e a de acesso a memória. A velocidade dos semicondutores que compõem a memória não acompanhou o rápido aumento de velocidade dos processadores. Tecnologias de *cache* foram introduzidas para compensar essa velocidade, mas ainda assim continua sendo um obstáculo relevante. Além disso, a ***instruction-level parallelism wall*** é causada pela crescente dificuldade de explorar o paralelismo numa sequência de instruções.

Essas três "paredes" levaram a cunhagem do termo ***brick wall*** ou "parede de tijolos" por David Patterson (ASANOVIC et al., 2009), ilustrada pela equação 2.1:

$$PowerWall + MemoryWall + Instruction - levelWall = BrickWall \tag{2.1}$$

Um termo que definiria os obstáculos que os processadores de núcleo único enfrentariam. A alternativa que a indústria encontrou para driblar este obstáculo e acompanhar a demanda por desempenho foi adotar uma mudança radical na arquitetura dos processadores.

Em 2001, os primeiros processadores de propósito geral contando com múltiplos núcleos de processamento no mesmo encapsulamento CMOS foram lançados: os processadores POWER4 da IBM. Desde então, os processadores *multi-core* se tornaram norma. Começaram com dois núcleos, aumentando para quatro, seis e até oito núcleos. Os processadores mais atuais possuem até centenas de núcleos.

No ano de 2014, foi desenvolvido pela IBM o processador POWER8, que introduziu a produção de componentes na escala de 22nm, permitindo velocidades de clock superiores, preservando os níveis de conservação de energia. Esta nova arquitetura da família POWER possibilitou exceder o "teto" de 4GHz, operando a frequências de 5.5GHz, contando ainda com 12 núcleos, cada um possibilitando até 8 threadlines (IBM...,).

Atualmente a arquitetura *multi-core* é o método em voga para aumento de desempenho para processadores, adicionando suporte para mais *threadlines*, tanto no número de núcleos quanto na utilização de *multithreading* para cada núcleo disponível, porém, ainda virão obstáculos de desenvolvimento tal como a geração anterior, e já se pode perceber a desaceleração no melhoramento de processadores nos últimos anos.

2.2 Classificação de Sistemas Computacionais

De acordo com a classificação proposta por (FLYNN, 1972), utilizada até hoje para o projeto de processadores modernos, e a partir do desenvolvimento do multiprocessamento como uma extensão dos sistemas de classificação, é possível classificar um sistema computacional em quatro categorias principais:

- SISD (*Single Instruction, Single Data stream*): Corresponde a um computador sequencial que não explora nenhum paralelismo tanto nas instruções quanto no fluxo de dados.

Segue o ciclo tradicional de busca, decodificação e execução de instruções. Exemplos de arquiteturas SISD são os conhecidos processadores de núcleo único utilizados em computadores pessoais e *mainframes*.

- SIMD (*Single Instruction, Multiple Data stream*): Corresponde a um computador que explora múltiplos fluxos de dados através de um único fluxo de instruções para executar operações naturalmente paralelizáveis. Exemplos de arquiteturas SIMD são encontradas nos processadores vetoriais e GPUs, onde há uma grande quantidade de processadores especializados que executam as mesmas instruções para vários conjuntos de dados distintos.

- MIMD (*Multiple Instruction, Multiple Data stream*): Corresponde a um computador que possui múltiplos processadores autônomos executando simultaneamente diferentes instruções em diferentes fluxos de dados. Sistemas distribuídos (com memória compartilhada ou distribuída) e processadores multi-core são exemplos de arquiteturas MIMD.

- MISD (*Multiple Instruction, Single Data stream*): Corresponde a um computador que opera múltiplos fluxos de instruções sobre um mesmo fluxo de dados. Não é uma arquitetura convencional, mas é utilizada em sistemas com requisitos críticos de tolerância a falha.

A partir destas classificações, pode-se desenvolver a noção da diferença entre as abordagens que os processadores são desenvolvidos e as aplicações mais adequadas para cada política de construção.

A proposta de Flynn contribuiu para a categorização de processadores por muitas décadas, contudo os processadores atuais não mais pertencem a somente uma categoria. Processadores de múltiplos núcleos são categorizados como processadores MIMD, pois podem operar sobre diferentes dados em diferentes contextos, mas cada núcleo pode implementar instruções SIMD voltadas geralmente a aritmética e processamento multimídia e utilizá-las em seu próprio contexto.

2.3 Os Processadores de Núcleo Único

O desenvolvimento de processadores de núcleo único (ilustrados através do diagrama na Figura 2.1) floresceu até a década de 90 em ritmo cada vez mais rápido, permitindo que aplicações fossem programadas utilizando o paradigma sequencial, e em relação ao desempenho, quanto

mais "rápido" fosse o processador, mais rápido seria a execução da aplicação em questão. A velocidade do processador consiste na frequência de *clock* – quanto maior fosse a frequência de *clock* do processador, mais rapidamente a sequência de instruções poderia ser executada, e portanto, mais rápida seria a execução de aplicações.

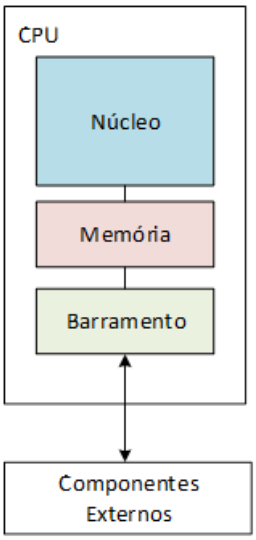


Figura 2.1: Estrutura de um Processador de Núcleo Único

O que se denomina como ILP (do inglês, *Instruction Level Parallelism*, ou Paralelismo a Nível de Instrução), permitiu o rápido aumento na velocidade dos processadores. O processador poderia reordenar, utilizar *pipelines* e executar políticas agressivas de predição de desvio (*branch prediction*) de instruções, dentre outras técnicas para tratar diversos tipos de eventos. Trata-se de uma forma de paralelização em nível de linguagem de máquina (RAUBER; RÜNGER, 2013).

Para um melhor aproveitamento do tempo de processamento, a paralelização em um nível mais alto foi definida pelo termo TLP (do inglês, *Thread Level Parallelism*, ou Paralelismo a Nível de "Linha"/*Thread*), onde técnicas foram desenvolvidas para a execução de mais de um *thread* (por exemplo, mais de uma aplicação) ao mesmo tempo, sem a necessidade de possuir mais de um núcleo de processamento para isso (FRANKLIN, 2012).

ILP e TLP são ferramentais complementares e contribuíram na definição da computação moderna, viabilizando a execução de várias tarefas pelo sistema operacional pelo modelo de *multitasking*. Visto a necessidade não só de executar vários *threads* mas também que estas linhas precisavam muitas vezes compartilhar dados entre si, veio a criação dos mecanismos de comunicação entre processos, categorizados como mecanismos de IPC (*Inter Process Communication*).

Reunindo vários processadores de núcleo único em uma estrutura que permitisse que se

comunicassem possibilitou a criação de computadores multiprocessados, ou seja, computadores que possuíam mais de um processador ou núcleo de processamento. Muitos supercomputadores e servidores foram construídos baseados na junção de vários processadores de núcleo único em placas montadas em *racks*, organizadas de maneira a utilizar o poder de processamento e comunicação entre os núcleos através de barramentos, em circuito impresso ou via cabeamento externo através de protocolos como o Ethernet, em aplicações que exigiam alto desempenho.

Após a exploração durante vários anos das arquiteturas de núcleo único, passando por avanços na tecnologia de memórias *cache*, desenvolvimento de técnicas de predição de desvios, aumento dos níveis de *pipeline*, e gradual aumento na frequência de *clock*, os processadores de núcleo único enfim chegaram a um limite físico. O aumento na frequência de *clock* de um processador passou a trazer restrições proibitivas. Processadores de núcleo único chegaram a frequências de 3 a 4 GHz – e se a frequência fosse elevada acima dessa faixa o consumo de energia se tornaria inviável, além do surgimento de problemas relacionados a superaquecimento e complexidade de projeto dos circuitos envolvidos (PANKRATIUS et al., 2008).

Sob os olhos da Lei de Moore, uma grande parede havia se consolidado no contínuo desenvolvimento e aumento de desempenho dos processadores. Os desenvolvedores necessitavam de uma alternativa para driblar as limitações físicas impostas sobre os processadores de núcleo único. Dessa necessidade, surgiram os primeiros processadores de múltiplos núcleos.

2.4 O Advento dos Processadores Multi-core

As limitações apresentadas na seção anterior levaram ao desenvolvimento de processadores que possuíam mais de um núcleo em um mesmo encapsulamento. Ao invés de criar processadores de núcleo único com um conjunto de circuitos cada vez mais complexos e caros, reunir em um mesmo espaço físico vários núcleos de menor capacidade de processamento possibilitou alcançar níveis reais de paralelismo, até então nunca alcançados utilizando as políticas de compartilhamento de tempo dos processadores de núcleo único (OLUKOTUN et al., 1996).

Exauridas todas as alternativas para ganho de desempenho através de tecnologias como a abordagem superescalar e o *pipelining*, se iniciou uma era onde os programadores são obrigados a adotar modelos de programação paralela para que seja possível explorar as estruturas dos multiprocessadores efetivamente. De acordo com (OLUKOTUN; HAMMOND, 2005), há somente três "dimensões" para aumento no desempenho dos processadores: a frequência do *clock*, instruções superescalares e o multiprocessamento. Os dois primeiros itens foram levados ao limite, restando aos programadores a alternativa de exploração dos paradigmas paralelos

para alcançar melhores níveis de desempenho.

Os processadores *multi-core* possuem em um mesmo encapsulamento dois ou mais núcleos idênticos de "uso-geral" ou núcleos especializados para certos tipos de operações, como as GPUs que possuem grande quantidade de processadores específicos para efetuar cálculos ou os processadores de arquitetura ARM, equipados com núcleos especializados para determinadas tarefas. Devido à proximidade entre os núcleos, os sinais elétricos correspondentes às transmissões de dados levam menos tempo e sofrem menor interferência e deterioração, permitindo economia de energia e maior confiabilidade ao efetuar comunicação entre núcleos. Pode-se visualizar um diagrama simplificado de um processador *multi-core* na Figura 2.2.

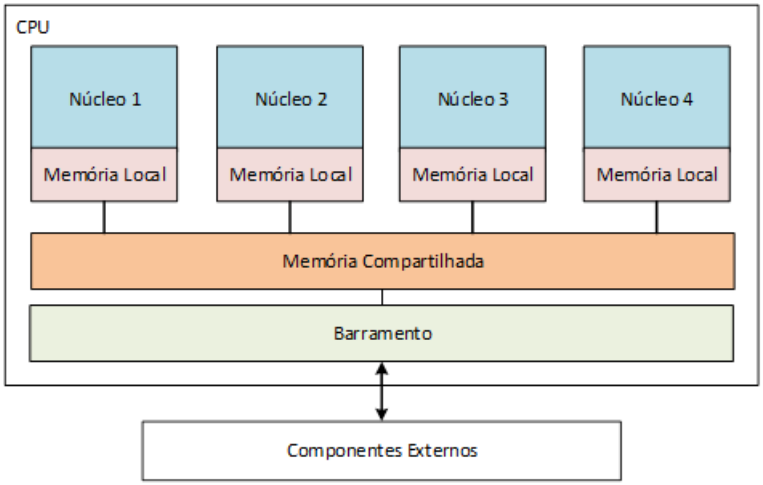


Figura 2.2: Estrutura de um Processador de Múltiplos Núcleos

De acordo com o supracitado sobre a Taxonomia de Flynn, os processadores *multi-core* podem ser classificados como MIMD, ou seja, núcleos diferentes podem executar *threads* diferentes, operando em partes diferentes da memória, sendo que os processadores *multi-core* compartilham o mesmo espaço de memória (paradigma de memória compartilhada).

Visto que várias aplicações que possuíam potencial de paralelização já tiravam proveito dos ferramentais de *multithreading*, a criação das arquiteturas *multi-core* beneficiou àquelas em que as operações poderiam ser executadas paralelamente, tais como bancos de dados, servidores web, compiladores, aplicações multimídia como editores de imagem, vídeo e áudio, aplicações científicas CAD/CAM, em outras palavras, aplicações com TLP predominante.

O paralelismo não se trata de uma solução genérica e simples para aumento de desempenho. Tal como qualquer técnica, há leis e limitações que regem tais conceitos. Um modo de verificar o possível aumento de desempenho esperado ao paralelizar uma aplicação é utilizando a Lei de Amdahl (AMDAHL, 1967), apresentada pelo arquiteto de computadores Gene Amdahl em

1967. A Lei de Amdahl propõe que a possível melhora no desempenho (*speedup*) de uma aplicação utilizando-se múltiplos processadores na computação paralela é limitada pelo tempo consumido pela parte sequencial de tal programa, admitindo que praticamente todo programa paralelo possui uma fração necessariamente sequencial.

O *speedup* pode ser expressado pela seguinte fórmula:

$$Speedup = \frac{1}{(1-P) + \frac{P}{S}} \tag{2.2}$$

Onde P corresponde a porcentagem de melhora, S à quantidade de melhoramento (quão mais rápido a fração paralelizável é acelerada) e $(1–P)$ à fração não paralelizável. Percebemos pela fórmula que o *speedup* não depende somente do que é paralelizado, mas também da parte que não sofre melhorias com a paralelização. A equação permite visualizar dois corolários muito importantes:

- "Melhore o caso mais comum": Quando P é pequeno, as otimizações terão pouco efeito sobre o *speedup*.
- Os aspectos ignorados também limitam o *speedup*. Quando S se aproxima de infinito, o *speedup* é limitado por $\frac{1}{(1-P)}$.

Amdahl também definiu uma analogia para o caso especial onde n processadores em paralelo são utilizados, permitindo o cálculo do *speedup* paralelo, usando um argumento que assumia que uma fração f do tempo de execução de um programa era infinitamente paralelizável, enquanto a fração restante, $1–f$, era totalmente sequencial. Argumentou por fim que valores típicos de $1–f$ eram grandes o suficiente para favorecer processadores *unicore*.

No trabalho de (HILL; MARTY, 2008) são propostas variações da Lei de Amdahl que se adequam melhor às características intrínsecas dos processadores *multi-core*, dependendo de sua arquitetura: simétrica, assimétrica ou dinâmica. Cada uma das arquiteturas permite descrever um processo de paralelização distinto, onde os limites no ganho de desempenho são previstos por equações modeladas por parâmetros como fração sequencial e paralelizável de código, quantidade e tipos de processadores (de uso geral ou especializados).

Além das limitações levantadas por Amdahl, a complexidade de modelar a paralelização de aplicações sequenciais e modelar aplicações paralelas em si é um grande desafio. Alguns padrões de desenvolvimento foram propostos durante os anos visando a paralelização de aplicações sequenciais e exploração de possíveis novas áreas de aplicações que teriam potencial para execução paralela.

2.5 A Programação Paralela

A programação paralela é um modelo de desenvolvimento de aplicações paralelas que tiram proveito de mecanismos de paralelização. Para avaliar as técnicas que fazem parte desse modelo, pode-se considerar dois pontos chave: sua generalidade, ou seja, o quão bem um conjunto de problemas diferentes podem ser expressados para diferentes arquiteturas; e seu desempenho, ou o quão eficiente é a execução de uma aplicação desenvolvida utilizando tal técnica, geralmente avaliando o tempo de execução entre instâncias. Além disso, requisitos fundamentais para a programação paralela são, a citar: concorrência (regiões com possibilidade de execuções simultâneas), escalabilidade (custo-benefício da paralelização ao aumentar a quantidade de elementos de processamento), localidade (em termos de memória) e modularidade (no que se refere ao reuso de código e dependência entre *threads*).

Uma máquina que segue o modelo de Von Neumann assume que o processador é capaz de executar uma sequência de instruções. Uma instrução pode especificar operações aritméticas, endereços de dados a serem lidos ou escritos na memória e/ou o próximo endereço de instrução a ser executada. A programação de um computador em termos deste modelo básico utilizando linguagem de máquina é possível, mas é de certa forma proibitivo pois é necessário manter em vista milhões de localizações de memória e organizar a execução de milhares de instruções de máquina.

Nas arquiteturas de programação sequencial, o modelo de Von Neumann facilitou o desenvolvimento de aplicações, pois provê uma conexão eficiente entre hardware e software, ou seja, linguagens de alto-nível podem ser eficientemente compiladas e implementadas em hardware. Abstrações através de linguagens como C/C++, Java, permitem o desenvolvimento mais rápido de aplicações e tradução automática em código executável, poupando o programador dos detalhes de implementação das instruções do computador.

As informações a serem exploradas nesta seção referem-se dominantemente a sistemas que se encaixam na categoria MIMD da Taxonomia de Flynn, onde diferentes processadores podem executar diferentes instruções em dados distintos. Utilizar o modelo de programação paralela introduz alguns fatores de maior complexidade. Programar aplicações paralelas diretamente em linguagem de máquina não só aumentaria o número de instruções a serem executadas como também tornaria impraticável o mapeamento de memória e o gerenciamento de interações entre processadores. Nesse ponto, os modelos de programação são importantes para abstrair detalhes muito “profundos” com relação a linguagem de máquina e possibilitar uma eficiente interação entre processos.

A seguir, serão abordados os elementos e mecanismos mais comuns que viabilizam a paralelização e a comunicação entre processos.

2.5.1 Elementos, Condições e Mecanismos de IPC

Um programa de computador se torna um processo quando é carregado de algum dispositivo de armazenamento para a memória do computador e se torna um programa em execução, podendo ser processado por um ou mais processadores. A descrição de um processo em memória contém suas informações vitais como o contador de programa (PC, ou Program Counter), registradores, variáveis, ponteiros de arquivo, sinais e assim por diante (BUTENHOF, 1997).

Um *thread* é uma sequência de instruções de um programa que pode ser executada independentemente de outras sequências de código. *Threads* contém somente informações essenciais como a pilha (para variáveis locais, argumentos de funções e valores de retorno), uma cópia dos registradores, do contador de programa e outros dados específicos do contexto de execução do *thread*, o que permite que elas sejam escalonadas individualmente. Os outros dados são compartilhados entre todos os *threads* do processo em questão. Um processo pode conter vários *threads* executando "simultaneamente", caracterizando-o como *multithreaded*. *Threads* são associados diretamente ao conceito de LWP (*Light-Weight Process*, ou em tradução livre "Processo Leve") por serem caracterizadas como "pequenos processos" que são executados no espaço do usuário e compartilham o espaço de endereçamento de um processo convencional com outros "pequenos processos".

Os elementos acima citados permitem aos desenvolvedores criarem aplicações que particionem a carga de trabalho, os recursos utilizados, controlem a interface com o hardware e o acesso paralelo aos recursos compartilhados pelos processos que compõem a aplicação.

É importante visualizar que existem meios distintos de se paralelizar uma aplicação. No caso da aplicação em questão ser composta por vários processos independentes, como é o caso dos sistemas operacionais que executam em sistemas móveis, e.g. *smartphones* e *tablets*, a paralelização consistiria basicamente em separar estes processos em núcleos diferentes de um processador com dois ou mais núcleos, permitindo sua execução simultânea.

Em certos casos onde há dependências de dados entre processos e havendo necessidade de implementar meios de comunicação que permitam que os processos compartilhem tais dados necessários à sua execução, situações conhecidas como condições de corrida (*race conditions* ou *race hazards*) podem ocorrer – situações onde o cômputo correto de uma aplicação depende

diretamente da sequência e tempo em que os processos/*threads* são executados – possibilitando a geração de erros ou comportamentos inesperados.

Além das condições de corrida, situações comuns em sistemas concorrentes são: os *deadlocks*, onde dois ou mais processos/*threads* executando em ações concorrentes ficam esperando até que um deles termine o cômputo, e nenhum deles conseguirá terminar; os *livelocks*, similares aos *deadlocks*, porém o estado dos elementos de processamento continuam a mudar – em analogia, um *livelock* é similar ao encontro de duas pessoas em um corredor estreito, e cada uma delas tenta se mover educadamente para que a outra passe, porém terminam em uma situação onde uma sempre se mexe na mesma direção que a outra; e *starvation*, que caracteriza um problema onde um processo é perpetuamente negado de recursos necessários para seu cômputo (SILBERSCHATZ; GALVIN; GAGNE, 2013).

Elementos de sincronização são essenciais para a resolução de condições de corrida e permitem alcançar cômputos corretos ao gerenciar a ordem em que eventos acontecerão no sistema. Os elementos de sincronização (e.g. mutex e semáforos), também conhecidos como *locks*, permitem delimitar regiões de código denominadas regiões críticas, geralmente aplicadas para proteger os recursos compartilhados de um processo ou *thread*, permitindo que somente determinado número de tais unidades possa acessar ou modificar um recurso. A utilização de *locks* no entanto é conhecidamente complexa, passando a responsabilidade da política de gerenciamento de acesso para o programador, e pouco escalável, as regiões críticas podem se tornar um gargalo no desempenho da aplicação com muitos processos ativos, além de limitar os níveis de paralelismo ao serializar o acesso pela região crítica (MCKENNEY, 2011).

No intuito de efetuar a coordenação de todos os fatores anteriormente citados, interfaces de programação (algumas APIs como Pthreads, OpenMP e linguagens especializadas) foram criadas, implementando mecanismos e protocolos de comunicação para interação entre as unidades via IPC (sejam elas processos ou *threads*).

A IPC pode ser descrita como o ato de compartilhar dados entre múltiplos processos usando algum tipo de protocolo de comunicação. Existem diversos mecanismos que permitem administrar a comunicação interprocessual. A Tabela 2.1 descreve alguns mecanismos de IPC.

Método	Descrição
Arquivo	Um registro armazenado em disco ou sintetizado por um servidor de arquivos que pode ser acessado por vários processos.
Sinal	Uma mensagem do sistema, enviada de um processo ao outro, usualmente utilizado não como meio de transferência de dados mas para comando remoto entre processos parceiros.
Socket	Fluxo de dados enviado por interface de rede, tanto para processos diferentes em um mesmo computador quanto para outro computador na rede.
Fila de Mensagens	Um fluxo de dados anônimo, usualmente implementado pelo sistema operacional, permitindo que múltiplos processos leiam e escrevam nafila de mensagens sem que estejam conectados entre si.
Pipe	Um fluxo de dados bidirecional entre processos conectados via entrada e saída padrão, efetuando leitura de um caractere por vez.
Named Pipe	Um pipe implementado através de um arquivo em vez de utilizar o sistema de entrada e saída padrão. Múltiplos processos podem ler e escrever no arquivo, funcionando como um *buffer* para os dados de IPC.
Semáforo / Mutex	Estrutura simples que sincroniza múltiplos processos atuando em recursos compartilhados (podendo ser implementada através de variáveis globais).
Arquivo Mapeado em Memória	Um arquivo mapeado em memória RAM que pode ser modificado ao mudar os endereços de memória ao invés de processar uma saída de dados. Compartilha dos mesmos benefícios que um arquivo convencional, mas tira proveito da velocidade superior da memória RAM.

Tabela 2.1: Mecanismos de IPC

2.5.2 Memória Compartilhada

A memória compartilhada, de maneira geral, é uma região de memória que pode ser acessada simultaneamente por múltiplos programas, promovendo um meio de comunicação entre os processos em execução e/ou evitando redundância de dados. É um meio eficiente de troca de dados, permitindo execução em processadores *unicore* ou *multi-core*, de acordo com o contexto.

Do ponto de vista de hardware, memória compartilhada se refere a um bloco de memória (tipicamente grande) de RAM (*Random Access Memory*) que pode ser acessada por diferentes CPUs em um sistema computacional de multiprocessamento. As arquiteturas de memória compartilhada podem utilizar as seguintes políticas de acesso:

- *Uniform Memory Access* – UMA (Acesso Uniforme à Memória): todos os processadores compartilham a memória física uniformemente.
- *Non-Uniform Memory Access* – NUMA (Acesso Não-Uniforme à Memória): o tempo de acesso a memória dependerá da localização relativa entre a posição de memória e o processador.
- *Cache-only Memory Architecture* – COMA (Arquitetura de Memória em Cache): as memórias locais para os processadores são utilizadas como cache.

Um sistema de memória compartilhada é relativamente fácil de se programar, considerando que todos os processadores compartilham uma única visualização de dados e a comunicação entre os mesmos podem ser tão rápidas como um acesso a mesma localização de memória. O problema com esses sistemas é que muitas CPUs precisarão de acesso rápido a memória através do cache, o que leva a duas complicações principais:

- A conexão entre CPU e memória se tornará um gargalo (*bottleneck*). Os computadores que utilizam memória compartilhada não conseguem escalar bem e a maioria chega a apenas dez processadores devido a essa limitação (muitos núcleos competindo pela memória pode causar grande *overhead*).
- Coerência de Cache: Quando um cache é atualizado com informações que deverão ser usadas por outro processador, as mudanças devem refletir nos outros processadores, caso contrário os outros estarão trabalhando com dados incoerentes. Os protocolos de coerência podem, quando trabalham corretamente, prover extremo desempenho no acesso a dados compartilhados entre múltiplos processadores. Em contrapartida, são responsáveis em

algumas ocasiões por grande parte da sobrecarga do sistema e se tornando um gargalo de desempenho.

Do ponto de vista de software, a memória compartilhada é um método de IPC, provendo uma maneira de trocar dados entre programas em execução simultânea.

Considerando que ambos os processos em execução conseguem acessar a memória compartilhada tal como uma região de memória comum, consiste em um método muito rápido de comunicação em comparação aos mecanismos de IPC como *named pipes* e *UNIX domain sockets*. Em compensação, é menos escalável pois os processos precisam ser executados na mesma máquina.

2.5.3 Memória Distribuída e Passagem de Mensagem

Em contraste aos sistemas de memória compartilhada, existem os sistemas de memória distribuída, sistemas multiprocessamento onde cada processador possui sua memória privada. Os processos computacionais só podem operar sobre os dados locais das memórias individuais, e se dados remotos forem necessários, os processos devem se comunicar com processadores remotos através da infraestrutura, geralmente uma rede implementada através de, por exemplo, protocolo Ethernet. O modelo de passagem de mensagem (abordado na seção seguinte) é o mais adotado, senão o mais adequado em sistemas de memória distribuída).

O modelo de passagem de mensagem é baseado na premissa de que uma computação paralela consiste de um número de processos, cada um trabalhando sobre um conjunto de dados local. Cada processo possui puramente variáveis locais e não há mecanismos para qualquer processo se comunicar diretamente com outra região de memória de outro processo.

A utilidade deste modelo vem de sua extrema generalidade. Essencialmente, qualquer tipo de computação paralela pode ser construído nos moldes da passagem de mensagem. Além disso, esse modelo pode ser implementado numa grande variedade de plataformas, desde multiprocessadores de memória compartilhada a redes de workstations e até mesmo máquinas unicore. Geralmente permite o controle sobre a localidade dos dados e o fluxo da aplicação paralela, podendo alcançar seu desempenho usando a passagem de mensagens explícita (GROPP; LUSK; SKJELLUM, 2014).

A passagem de mensagem distribuída fornece aos desenvolvedores uma camada de arquitetura que provê serviços comuns para construção de sistemas feitos de subsistemas executando em computadores distintos em diferentes locais e em tempos diferentes.

O modelo de passagem de mensagem foi normatizado na década de 1990, com a criação do MPI (*Message Passing Interface*), se tornando o padrão mais utilizado no desenvolvimento de aplicações distribuídas. O MPI será abordado brevemente na Seção 2.6.2.

2.5.4 Concorrência x Paralelismo Real

Concorrência e paralelismo são termos que causam grande confusão, muitas vezes sendo utilizados erroneamente e utilizados para as mesmas situações.

O termo Concorrência se refere a um conjunto de técnicas que permitem a utilização dos recursos computacionais de maneira mais eficiente, reduzindo o tempo ocioso dos elementos de processamento. Em outras palavras, concorrência se refere ao gerenciamento de mais de uma tarefa ao mesmo tempo. O elemento que permite gerenciar mais de uma tarefa ao mesmo tempo é o que se denomina *interleaving* (intercalação), ou seja, deve-se interromper uma das tarefas antes de iniciar a outra.

O termo Paralelismo se refere as técnicas de execução paralela de processos, requerendo hardware que suporte as unidades de processamento, ou seja, modos de executar simultaneamente mais de uma tarefa em mais de um processador/núcleo (OSHANA, 2015).

Concorrência e paralelismo são conceitos distintos. A concorrência está preocupada com o gerenciamento de estados compartilhados para diferentes *threads*, enquanto o paralelismo com a utilização de múltiplos processadores/núcleos para melhorar o desempenho da computação.

2.5.5 Pensamento Paralelo

De acordo com James Reinders, chefe e diretor de desenvolvimento de software da Intel, em seu artigo intitulado “Think Parallel or Perish – Where are we now?” de 2009 (INTEL. . . ,) (em tradução livre “Pense Paralelo ou Pereça – Onde nós estamos?”) relata que estamos saindo da era “não paralela”, que no futuro parecerá uma era muito primitiva da computação, e prevê que em menos de uma década um programador que não souber de antemão “Pensar Paralelo” não será um programador de fato. Isso retrata a real necessidade da exploração de paradigmas, linguagens e tecnologias que explorem de fato as oportunidades de utilização do paralelismo.

Em outro artigo (INTEL. . . ,), James Reinders introduz um conjunto de oito regras específicas para o desenvolvimento bem sucedido de aplicações para ambientes *multi-core* e que são consideradas tendências na área (defendendo a utilização do paradigma de memória compartilhada e em ferramentas que abstraem os mecanismos de paralelismo). As regras são resu-

midas abaixo:

1. Pense paralelo: Em todos os problemas, procure por paralelismo e formas de como organizá-lo e explorá-lo.

2. Utilize abstrações: Utilizar bibliotecas nativas como Pthreads, Windows Threads ou BOOST Threads, e MPI são equivalentes à linguagem Assembly do paralelismo. Procure usar bibliotecas como OpenMP, Intel TBB dentre outras.

3. Programe em tarefas e não *threads*: Crie uma quantidade abundante de tarefas para que sejam espalhadas automaticamente pelos núcleos disponíveis.

4. Desenvolva com a possibilidade de desligar a concorrência: depurar programas sem estruturas de concorrência e dependência de dados é menos complexo e permite detectar com mais facilidade problemas, diferenciando entre problemas da lógica do programa e problemas relacionados à introdução de concorrência/paralelismo.

5. Evite usar *locks*: regiões críticas são conhecidos limitantes com relação à escalabilidade de aplicações concorrentes. Simplesmente evite sua utilização. Quando necessitar de estruturas de sincronização, dê preferência a operações atômicas.

6. Use ferramentas e bibliotecas para ajudar na concorrência: seja crítico ao escolhê-las (dê preferência às mais atuais) e conte com elas para desenvolver.

7. Use alocação de memória escalável: a utilização de alocadores escaláveis de memória aumenta o desempenho das aplicações reduzindo gargalos globais, reusando memória para otimizar e regularizar a utilização de *caches*.

8. Desenvolva tendo em mente o aumento da carga de trabalho: as cargas de trabalho das aplicações aumentam com o tempo. O projeto das aplicações paralelas devem favorecer a escalabilidade prevendo esse aumento.

Obviamente as regras levam em conta o ciclo de desenvolvimento de software comercial e os conceitos explorados pela cadeia de ferramentas disponibilizada pela própria Intel, o Intel Parallel Studio, para desenvolvimento de aplicações que exploram paralelismo em ambientes *multi-core*. São regras práticas que podem ser seguidas para reduzir a complexidade na modelagem e codificação, facilitando a vida do programador em várias camadas, tornando o desenvolvimento para sistemas *multi-core* menos penoso.

2.6 Arquiteturas Multi-core

Um processador *multi-core* consiste em um componente computacional único que conta com duas ou mais unidades de processamento independentes, também denominadas *cores* ou núcleos, capazes de ler e executar instruções de programa ordinárias como adição, movimentação de dados, desvios, dentre outras, de maneira simultânea, aumentando a velocidade de execução de programas que suportam certo nível de paralelismo computacional.

Os atuais processadores *multi-core* podem contar com dois núcleos, denominados dual-core, por exemplo as arquiteturas das linhas Intel Core 2 Duo, AMD Phenom II X2 e Nvidia Tegra 2, quatro núcleos, denominados quad-core, como as arquitetura das linhas Intel i5 e Intel i7, oito núcleos, denominados octa-core, como o Intel Xeon E7 e o AMD FX-8350, dez núcleos ou mais. Algumas placas de desenvolvimento e prototipagem como a Adapteva Parallela Board conta com um processador com 16 a 64 núcleos, utilizando a arquitetura Epiphany baseada em processadores ARM. As arquiteturas *multi-core* são a atual tendência para a arquitetura de processadores modernos e é a forma que a indústria adotou para que o crescimento do desempenho não fosse interrompido pelas barreiras físicas impostas no desenvolvimento de processadores de núcleo único.

2.6.1 Arquitetura de Hardware

O conceito de múltiplos núcleos inicialmente pode parecer trivial, contudo há inúmeros pontos a serem considerados em sua utilização, por exemplo, a homogeneidade dos núcleos que compõem o processador. A maioria dos processadores multi-core de propósito geral são homogêneos, denominados simétricos ou SMP – *Symmetric Multiprocessor*), ou seja, os núcleos são baseados no mesmo conjunto de instruções (ISA) e tem o mesmo desempenho, o que significa que todo núcleo pode executar os mesmos binários, e portanto, de um ponto de vista funcional, não importa em qual núcleo exatamente o binário será executado. As arquiteturas mais recentes permitem que o sistema controle a frequência de *clock* para cada núcleo individualmente no intuito de otimizar o consumo de energia ou aumentar temporariamente o desempenho de determinada *threadline*, estratégia denominada *soft overclock*. Historicamente, o primeiro processador *multi-core* com arquitetura homogênea disponível para comercialização foi o IBM POWER4, com dois núcleos operando em frequências de 1.1 a 1.9GHz.

Já as arquiteturas heterogêneas, denominadas assimétricas ou AMP – *Assymmetric Multiprocessor*, contam com pelo menos dois tipos diferentes de núcleos que podem diferir com relação ao conjunto de instruções, funcionalidade e desempenho. Uma representante ampla-

mente conhecida das arquiteturas *multi-core* heterogêneas é a arquitetura Cell BE (*Cell Broadband Engine*), desenvolvida por IBM, Sony e Toshiba e usada em videogames, como o Playstation 3, e computadores voltados a HPC (do inglês, *High Performance Computing*, ou Computação de Alto Desempenho). A Figura 2.3 esquematiza a diferença essencial entre as arquiteturas homogêneas e heterogêneas.

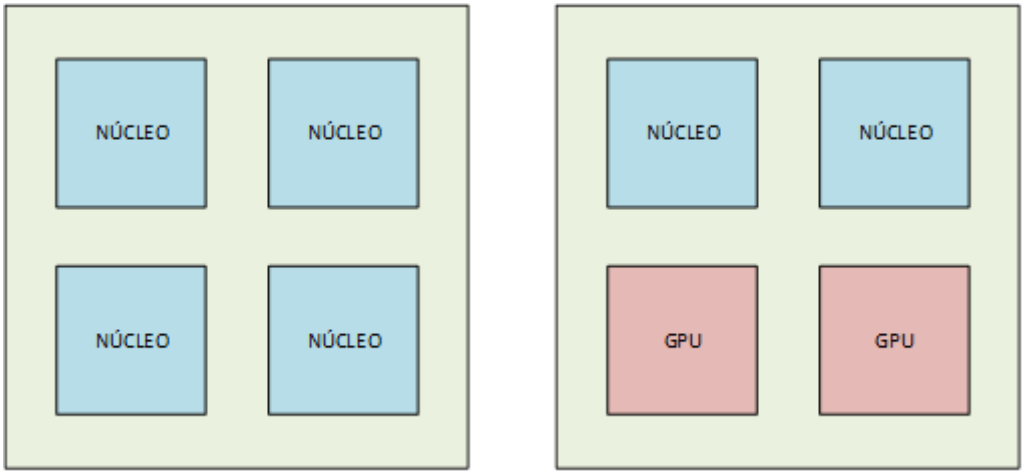


Figura 2.3: Arquitetura Homogênea (Simétrica) (esq.) e Arquitetura Heterogênea (Assimétrica) (dir.)

A maioria das arquiteturas homogêneas implementam um espaço de endereçamento global com coerência total de *cache*, e portanto, pela perspectiva de software, não se pode distinguir um núcleo do outro mesmo se o processo migrar entre núcleos durante o processamento, oferecendo transparência com relação a memória. Nas arquiteturas heterogêneas, por outro lado, implementar protocolos de coerência de cache agregaria grande complexidade ao projeto dos processadores e consiste em uma grande dificuldade para obtenção de bom desempenho – problema frequente em implementações como as GPUs, que serão abordadas na seção seguinte.

Os processadores modernos são tipicamente compostos por um número pequeno de núcleos operando em frequências altas com atributos avançados como execução de instruções fora-de-ordem e predição de desvios. São elementos de propósito geral de para uma grande variedade de aplicações, incluindo cargas de trabalho sequenciais sensíveis a latência ou cargas de trabalho com paralelismo a nível de tarefa ou de dados. São tipicamente utilizados para processamento com paralelismo mínimo, usam extensivamente *caches* relativamente grandes para compensar a latência de memória. Vários processadores incorporam também instruções SIMD para aumentar o desempenho de cargas de trabalho de aritmética densa e multimídia, porém não são diretamente expostas via linguagens de programação convencional como o C e o Fortran, requerendo bibliotecas e diretrizes de compilação especializadas (STONE; GOHARA; SHI, 2010).

2.6.2 Modelos de Programação para Multi-core

De acordo com o documento publicado (KELLY; PEDRETTI; LEVENHAGEN, 2008) pela divisão Sandia National Laboratories da Sandia Corporation (instalação governamental estadunidense com foco principal em pesquisa nuclear), podemos citar uma grande variedade de modelos de programação voltados às aplicações *multi-core*. Alguns dos modelos se encaixam não só para programação *multi-core* mas para sistemas multiprocessamento, como os *clusters* e outros sistemas distribuídos, porém, considerando o escopo deste trabalho, não foram apresentadas nesta seção (caso do MPI e PGAS).

Instruções Vector-like ou Vetoriais

Em geral, de maneira simples, uma aplicação sequencial pode ser paralelizada pelo compilador sem quaisquer instruções explícitas, de maneira limitada e em condições específicas. O compilador pode tirar vantagem de características do processador, por exemplo, usando extensões de instruções SIMD como SSE (*Streaming SIMD*) e MMX, no intuito de acelerar o processamento de instruções.

Além disso, processadores massivamente paralelos podem conter processadores vetoriais, que por sua vez podem ser operados concorrentemente em múltiplos elementos de dados. O compilador pode utilizar as instruções vetoriais desses processadores sem necessidade de diretivas explícitas vindas da aplicação, porém, o uso das capacidades vetoriais é dependente da estrutura do código da aplicação em questão.

Ambos os recursos são implicitamente ou explicitamente explorados. Neste caso, como já citado, a qualidade do código no que se refere ao melhor aproveitamento dos recursos dependerá diretamente da estrutura do código e da experiência do programador. Devido à esses fatores e dificuldades, não consiste em uma solução para todos os problemas, e por essa razão, diversos modelos e ferramentas para auxiliar o programador de aplicações paralelas foram criados.

Threading Implícito

Na perspectiva da aplicação, outro nível de paralelismo "transparente" é executado pelas bibliotecas do sistema operacional, que podem permitir a execução de *threads* para paralelizar determinadas seções de código. No âmbito da programação concorrente, *threads* podem ser implementadas em processadores *unicore* para permitir melhor aproveitamento dos recursos computacionais, reduzindo o tempo ocioso e permitindo a execução de mais de uma aplicação, o que definiu muitos dos fatores da computação moderna. No entanto, para alcançar o paralelismo

real, o hardware deve prover múltiplos *threadlines* para computação. Possíveis formas de prover esses recursos são:

- Múltiplos *threadlines* em hardware dentro do núcleo (HTT – *Hyper-Threading Technology*) como nos processadores Xeon e Pentium
- Utilizando SMT (*Simultaneous Multithreading*)
- Múltiplos núcleos em um único encapsulamento (abordagem *Multi-core*)
- Múltiplos processadores acoplados (sistemas distribuídos)

Ou uma combinação dos itens anteriores.

Multithreading

O *multithreading* é encontrado principalmente em sistemas operacionais multitarefa, sendo um modelo de programação e execução consolidado que permite que múltiplos *threads* existam no contexto de um único processo. Esses *threads* compartilham recursos do processo "pai", mas não estão aptas a executar de maneira independente (como um processo independente - o tempo de processamento do processo "pai" é divido entre seus *threads*).

Alguns fatores vantajosos na utilização do modelo de *multithreading* são:

- Capacidade de Resposta: *Multithreading* permite uma aplicação manter-se responsiva a entradas. Em um programa com um único *thread*, se o principal *thread* bloquear em uma tarefa longa, a aplicação inteira congelará até que seja possível tratar outras interações com o programa. Movendo esse *thread* para um *thread* "trabalhador" que executa concorrentemente com o *thread* principal, é possível que a aplicação continue responsiva às entradas do usuário enquanto executa outras tarefas em segundo plano (não é a única maneira de manter um programa responsivo – entrada e saída não-bloqueante e sinais podem ser utilizados para alcançar resultados similares).
- Execução mais rápida: Permite operar mais rapidamente em sistemas computacionais que possuem múltiplas CPUs ou processadores *multi-core* ou ainda em *clusters*, pois os *threads* do programa naturalmente se comportam em execução concorrente.
- Menor consumo de recursos: Uma aplicação pode servir múltiplos clientes concorrentemente usando uma quantidade de recursos menor ao criar um *thread pool*, se comparada

a abordagem de utilização de cópias de processos (*threads* são conhecidos como LWP, do inglês, *Lightweight Processes*, ou em tradução livre "Processos Leves").

- Melhor utilização do sistema: Aplicações *multithread* alcançam maior *throughput* e menor latência no acesso a memória *cache*, ao permitir vários *threads*.

- Compartilhamento e comunicação simplificados: Diferente dos processos, que requerem passagem de mensagens ou memória compartilhada para executar comunicação entre processos (operações de IPC), a comunicação entre *threads* é mais simples. *Threads*, por pertencerem a um mesmo processo, compartilham dados e código, tornando a comunicação mais simples.

- Paralelização: Aplicações que podem utilizar sistemas *multi-core* e multiprocessamento podem usar o *multithreading* para dividir os dados e tarefas em subtarefas paralelas e deixar que a arquitetura gerencie como os *threads* serão executados, seja concorrentemente ou paralelamente. Os modelos CUDA e OpenCL voltados a GPUs (que serão abordados mais à frente) usam o modelo *multithread* onde centenas de *threads* executam em paralelo numa grande quantidade de núcleos.

Por outro lado, algumas desvantagens do modelo a serem citadas:

- Sincronização: *Threads* compartilham o mesmo espaço de endereçamento de memória, portanto o programador é responsável por evitar condições de corrida e comportamentos pouco intuitivos por parte da aplicação. No intuito de manipular corretamente os dados, *threads* necessitam efetuar *rendez-vous* em tempo correto para processar os dados na ordem correta. Podem requerer ainda operações mutuamente exclusivas (implementadas por regiões críticas usando semáforos) para prevenir modificação ou leitura simultânea de dados, comprometendo a integridade do cômputo. O conjunto dessas primitivas é implementado através dos mecanismos de IPC disponíveis a nível de linguagem de programação e a nível de sistema operacional. O uso desregrado dessas primitivas podem levar a condições de *deadlock*.

- Comprometimento do processo: Uma operação ilegal executada por um *thread* pode danificar o processo responsável por sua execução e interrompendo o processamento de todos os elementos associados.

Este trabalho descreverá duas ferramentas para *multithreading*: Pthreads e OpenMP.

POSIX Threads (Pthreads)

Pthreads ou *Portable Operating System Interface* (POSIX) *threads* é um conjunto de tipos e chamadas de função da linguagem C, implementado através de um arquivo cabeçalho `pthreads.h` e uma biblioteca para criação e manipulação de *threads*. O primeiro modelo de *threads* POSIX foi criado pelo IEEE Computer Society em 1988, e é mantido atualmente pelo Austin Common Standards Revision Group.

Não depende diretamente da transferência de dados mas utiliza o *cache* bidirecionalmente entre CPU e memória, tornando o cômputo de processamento muito rápido. Além disso, a memória *heap* dinamicamente alocada e as variáveis globais são compartilhadas entre todas as instâncias de *threads* de um processo, permitindo acesso rápido, desde que respeitadas as condições de acesso para evitar condições de corrida. A exploração de todo o potencial computacional provido pelo Pthreads necessita da estruturação do programa a ser desenvolvido em unidades autônomas de processamento.

O gerenciamento de *threads* requer que o programador explicitamente crie e destrua os threads utilizando as funções `pthread_create` e `pthread_exit` respectivamente. O *thread* criado irá executar a rotina de instruções até que encontre a chamada de função `pthread_exit`. O particionamento de carga de trabalho e mapeamento de tarefas também são explicitamente especificados pelo programador como argumentos da função `pthread_create`.

A utilização de Pthreads é relativamente mais complexa do que alguns outros modelos de programação como o OpenMP, abordado na próxima seção, pois responsabiliza o programador por grande parte das políticas de acesso, concorrência e paralelização da aplicação. Caso o programador não atente aos detalhes de funcionamento dos mecanismos, condições indesejadas podem ocorrer. Quando múltiplos *threads* acessam dados compartilhados, os programadores devem estar atentos a essas condições de corrida e *deadlocks*. Para proteger as denominadas regiões críticas, onde há acesso a dados compartilhados, Pthreads provê mecanismos de exclusão mútua (*mutex*) e semáforos, permitindo o controle de acesso à região crítica.

O modelo Pthreads, apesar de possuir especificações padronizadas e implementado em vários sistemas operacionais, não é um modelo bem estruturado no âmbito de planejamento e desenvolvimento de aplicações, e portanto, não é recomendado para propósitos de desenvolvimento de programas que utilizam tecnologia paralela (MEENAL; PRASHANT, 2014), porém, a utilização de modelos híbridos, associando o modelo Pthreads com outras ferramentas e metodologias de desenvolvimento podem trazer novos paradigmas para a programação paralela, um dos objetivos deste trabalho.

OpenMP

O OpenMP é uma API para desenvolvimento e especificação de aplicações de padrão industrial, desenvolvida por fabricantes de computadores e softwares como Oracle, HP, IBM e Intel. Suportada pela maioria das linguagens de programação nativas como Fortran, C e C++, oferece uma especificação comum que permite aos programadores criarem novas aplicações paralelas ou paralelizar aplicações sequenciais existentes, tirando vantagens de sistemas *multi-core* configurados com memória compartilhada.

O gerenciamento dos *threads* é implícito. Diretivas de compilação especiais são usadas para especificar seções de código que devem ser executadas em paralelo. O número de *threads* a serem usadas é especificado usando variáveis de ambiente. Diferente da abordagem de Pthreads, não há necessidade do programador gerenciar o tempo de vida de cada *thread*.

O particionamento da carga de trabalho e o mapeamento de tarefas para cada thread exige um pouco de esforço para programação por parte do programador, inserindo as diretivas de compilação nos locais corretos (por exemplo, `#pragma omp parallel` {} para C/C++). O OpenMP abstrai como a carga de trabalho (por exemplo, um vetor) é dividido entre tarefas, e quais tarefas são atribuídas a cada thread. Além disso, suporta sincronização implícita entre processos, onde os programadores só especificam onde as situações de sincronização ocorrem, tirando o gerenciamento do mecanismo de sincronização da responsabilidade do programador(KASIM et al., 2008).

Uma conhecida vantagem da utilização do OpenMP é a visão global de endereçamento do espaço de memória da aplicação, permitindo o desenvolvimento relativamente rápido de aplicações paralelas. A abordagem a partir de diretivas de compilação possibilita a escrita de código sequencial para manutenção facilitada. O foco de paralelização é através dos *loops* (diretivas de compilação que afetam a execução de estruturas de *loop* e automaticamente distribuem as iterações entre os núcleos).

Por outro lado, uma dificuldade em utilizar a API é a dificuldade de se conseguir bom desempenho especialmente em larga escala, devido ao acesso *fine-grained* de memória governado pelo modelo de memória, que não percebe as características de acesso de memória não-uniforme do espaço de endereçamento compartilhado.

Passagem de Mensagens Explícita (MPI)

MPI (*Message Passing Interface*) é uma especificação para operações de passagem de mensagem, e é o atual padrão para desenvolvimento de aplicações HPC (*High Performance Com-*

puting) em sistemas de memória distribuída. Provê *linkers* para linguagens C, C++ e Fortran. Algumas das implementações MPI mais conhecidas incluem OpenMPI, MVAPICH, MPICH, GridMPI e LAM/MPI.

O gerenciamento de *threads* é implícito, não sendo necessário codificar a criação, escalonamento e destruição dos processos. Em vez disso, é necessário utilizar uma ferramenta em linha de comando (`mpirun`) para informar ao *runtime* do MPI sobre quantos processos serão necessários e opcionalmente mapear os processos para os processadores. Baseada nessa informação, a estrutura do *runtime* gerenciará as instâncias.

O particionamento de cargas de trabalho e mapeamento de tarefas é feito pelos programadores, similar ao modelo Pthreads. Os programadores devem gerenciar quais tarefas serão computadas por quais processos. A comunicação entre processos adota o paradigma de passagem de mensagens onde os dados são compartilhados através do envio de mensagens entre processos.

As principais vantagens do modelo são: desempenho em questão de escalabilidade e portabilidade. O desempenho é resultado direto das bibliotecas MPI otimizadas disponíveis e controle total do usuário durante o ciclo de desenvolvimento do programa. A portabilidade vem da padronização da API entre as bibliotecas desenvolvidas baseadas em sua especificação. Em geral, um programa MPI pode ser executado tanto em sistemas distribuídos quanto em sistemas de memória compartilhada (porém, é assumido que nestes sistemas, modelos de passagem de mensagens tem desempenho inferior com relação aos modelos de memória compartilhada).

As principais dificuldades do modelo MPI consistem na visão discreta da memória durante a programação, o que torna difícil abstrair certas estruturas e aumenta o tempo de desenvolvimento consideravelmente; um programa que utiliza eficientemente os recursos MPI requer cuidado especial e estratégias de partição de dados e gerenciamento de comunicação; devido à natureza distribuída, muitos dados globais podem ser duplicados por questões de desempenho, resultando em um consumo de memória elevado comparado à outras estratégias; a ausência de mecanismos de tolerância a falha dificulta manter os processos MPI rodando em sistemas instáveis; requer recursos substanciais de memória e rede; por fim, as bibliotecas MPI possuem atributos não triviais a serem ajustados e otimizados.

Linguagens PGAS

O modelo PGAS (*Partitioned Global Address Space*, ou em tradução livre, Espaço de Endereçamento Global Particionado) é um modelo de programação que visa aumentar a produ-

tividade e alto desempenho para desenvolvimento de aplicações paralelas. A premissa principal do modelo PGAS é que o endereçamento de memória global compartilhado aumenta a produtividade de codificação, porém é preciso uma distinção entre acesso a dados locais e remotos para que otimizações de desempenho e escalabilidade em larga escala sejam possíveis em arquiteturas paralelas (WAEL et al., 2015). Em outras palavras, provê um modelo de programação paralela que une o melhor entre os mundos de memória compartilhada e memória distribuída.

Dois conceitos principais do modelo são:

- Múltiplos contextos de execução com espaços de endereçamento separados, permitindo que qualquer instância de execução explore propriedades de afinidade de memória da arquitetura para bom desempenho da aplicação.
- Acesso de memórias entre instâncias de execução para exploração do princípio de localidade.

Similarmente ao MPI, aplicações usando o modelo PGAS assumem múltiplos elementos de processamento independentes equipados com mecanismos de comunicação unidirecionais entre eles, causando menor *overhead* em contrapartida aos mecanismos bidirecionais do MPI.

As linguagens PGAS iniciam sua história no final da década de 1990 com a proposição de extensões para as linguagens Fortran (Co-Array Fortran), Java (Titanium) e C (Unified Parallel C) que seguissem o modelo PGAS, com modelos de execução SPMD (*Single Program Multiple Data*), estruturas de dados e mecanismos de comunicação.

Na década de 2000, o projeto HPCS (*High Productivity Computing Systems*) administrado pela DARPA financiou a criação de três linguagens PGAS de alta produtividade: Chapel, X10 e Fortress. O objetivo era criar sistemas de desempenho altíssimo, facilidade de programação, portabilidade e robustez. Essas linguagens não somente estendiam os conceitos de linguagens existentes, tal como suas antecessoras, mas foram desenvolvidas como novas linguagens baseadas nos princípios do modelo PGAS.

No entanto, apesar de algumas dessas linguagens conseguirem atender um nicho específico de aplicações (por exemplo, aplicações que utilizam arranjos globais), as promessas das linguagens PGAS não se materializaram. Falharam em sua popularização e em trazer grandes mudanças no estado da arte da computação de alto desempenho. Apesar disso, no que se denomina “Exascale Era”, onde computadores de alto desempenho necessitarão de uma mudança de paradigmas em sua programação devido novamente a limitações físicas (como o consumo de energia), e com a utilização de estruturas cada vez mais complexas com o advento da

Warehouse-scale Computing, é possível que, com algum aperfeiçoamento, este modelo ainda traga grandes benefícios às aplicações HPC (KHAN; ZOMAYA; WANG, 2013).

2.7 O Futuro dos Processadores Multi-core

De acordo com (VAJDA, 2011), espera-se que, ao nível de tecnologia de fabricação, os fabricantes estejam aptos, com a tecnologia CMOS disponível, a continuar a produzir transistores de até 6 nm (nanômetros), porém ao chegar nessa escala, dois desafios serão introduzidos:

- Quando os transistores medirem o equivalente a algumas dezenas de átomos, os efeitos quânticos da matéria terão de ser contabilizados e consequentemente trarão incertezas com relação a confiabilidade do hardware, com componentes falhando mais frequentemente, e mais importante, intermitentemente. A falta de confiabilidade provinda do hardware no futuro pode levar à implementação de mecanismos redundantes de execução com o objetivo de aumentar a probabilidade de que ao menos um cômputo ocorra adequadamente ou que os resultados sejam avaliados por um sistema de votação para garantir a corretude do cômputo.
- Haverá tantos transistores em um único chip que será, do ponto de vista energético, impossível chaveá-los todos ao mesmo tempo; esse fenômeno – denominado o problema de *Dark Silicon* – terá impacto significante em como serão construídos os processadores num futuro próximo.

2.7.1 O Problema do Dark Silicon

Na década de 1970, Robert Dennard, pesquisador da IBM, postulou o que posteriormente foi denominada "Escalabilidade de Dennard", prevendo que à medida que o tamanho dos transistores diminuía, a densidade de energia era preservada. Em outras palavras, se o tamanho linear de um transistor fosse reduzido pela metade, a energia que seria utilizada seria reduzida por um quarto, o que posteriormente não seria constatado. As perdas por energia estática aumentaram desproporcionalmente devido aos efeitos quânticos da matéria à medida que o tamanho e constituição dos componentes mudavam. Essas perdas geram calor, que aumentam ainda mais a perda por energia estática, causando uma reação conhecida como *thermal runaway*, levando à destruição dos circuitos por superaquecimento.

De acordo com o estudo sobre escalabilidade efetuado por (ESMAEILZADEH et al., 2012), dado o baixo retorno de desempenho, adicionar mais núcleos aos processadores não proverá

benefícios suficientes para justificar o processo de escalabilidade contínuo, ou seja, o custo/benefício de se adicionar mais núcleos a um processador não justificará continuar aumentando o número de núcleos. As previsões feitas através dos modelos matemáticos no estudo demonstram que, se as tecnologias *multi-core* se tornarem as fontes primárias de ganho de desempenho ao chegar em escalas de transistores de 16nm, a "era *multi-core*" terá se encerrado em pouco mais de uma década, uma tentativa de curta durabilidade para derrotar as consequências da falha da Escalabilidade de Dennard.

O termo *Dark Silicon* não se refere a áreas inúteis, não utilizadas ou simplesmente "em branco" de um circuito integrado, mas uma área deste que não pode ser utilizada o tempo todo ou em sua frequência máxima (*dark* no sentido de apagado ou desligado). Não é um fenômeno recente, vindo desde os primórdios da evolução da tecnologia CMOS, onde diversas unidades de circuito (como os caches e FPUs) operavam em *dark logic*, sendo acionadas apenas esporadicamente (TAYLOR, 2013). Essas áreas inativas nos encapsulamentos têm crescido a cada redução no tamanho dos transistores e prevê-se que ao chegar aos 8 nm, mais de 50% dessa área corresponderá à *dark silicon*. Se este problema não for combatido, prevê-se que em um futuro próximo, as microarquiteturas dos processadores serão compostas por pouco mais de 90% de circuitos funcionando a frequências significativamente menores ou totalmente ociosos.

O problema do dark silicon afeta diretamente a escalabilidade dos sistemas multi-core devido ao déficit causado no desempenho. Em um cenário onde dobra-se o número de núcleos em um processador, sendo que cada núcleo executaria 1,4 vezes (frequência de *clock*) mais rápido, o orçamento energético disponível para melhoria aceitaria somente 1,4 vezes em melhoria (TAYLOR, 2013). Em outras palavras, a energia máxima disponível para utilização em melhorias só conseguiria "abastecer" um sistema 1,4 vezes mais rápido, limitando a escalabilidade do processador *multi-core* em questão, o que obrigaria aos arquitetos do processador balancear entre o aumento da frequência dos núcleos do processador e a quantidade dos mesmos.

Problemas como esse vem sendo abordados pela indústria através de tecnologias como as arquiteturas NVIDIA Fermi, Intel Gulftown e Tilera TileGX, que combinam quantidade de núcleos e frequências de clock que operam sob as projeções da "barreira de utilização" – para que certa porção do sistema opere a todo potencial (sob a frequência nominal de *clock*), significativa porção dos circuitos tem de operar a frequências muito reduzidas ou totalmente ociosas devido às limitações físicas. A arquitetura Intel Nehalem prevê o desligamento de alguns núcleos para que outros possam operar à frequência total de clock em determinados regimes de trabalho.

Outra abordagem para combater o efeito do *dark silicon* é a modelagem de novas arquite-

turas que agregam núcleos de processamento genéricos associados a coprocessadores especializados de alta eficiência energética. A arquitetura GreenDroid (SWANSON; TAYLOR, 2011), voltada diretamente a dispositivos móveis Android, procura implementar uma arquitetura heterogênea, associando centenas do que denominaram *c-cores* (do inglês, *conservation cores*), especialmente desenvolvidos para otimizar a pilha de execução do sistema operacional Android de maneira transparente, mapeando as instruções diretamente aos *c-cores* responsáveis por cada tipo de processamento. Detectou-se que os *c-cores* foram responsáveis por aproximadamente 90% do tempo de execução, sendo que cada *c-core* consome em média 18 vezes menos energia que um processador de propósito geral.

A indústria entrou em uma nova era de desenvolvimento, onde um problema emergente impõe restrições no redimensionamento de componentes e em seu balanço energético. Estudos nos últimos anos mostram sua relevância e iminentes consequências para o futuro das arquiteturas de processadores, e se ouvirá cada vez mais no meio científico a respeito de técnicas para combater e se utilizar o fenômeno denominado *dark silicon*.

2.7.2 Outros Problemas de Escalabilidade

Ainda é possível citar dois problemas de escalabilidade que causarão significativo impacto nas futuras arquiteturas de processadores:

- Protocolos de coerência de *cache*: a sincronização do uso de uma mesma região de memória por muitos núcleos aumentará a complexidade das diretrizes de coerência, levando ao aumento da latência de acesso em áreas de memória frequentemente acessadas. Uma possível solução seria a utilização das instruções a nível de hardware que implementassem os conceitos de memória transacional, vistos anteriormente.

- Largura de banda de memória: o nivelamento das frequências dos núcleos reduzirá a latência, porém o aumento no número de núcleos irá multiplicar a quantidade de memória necessária por núcleo, assim como a banda de memória agregada, nos processadores futuros. Se de algum modo for possível aumentar o número de núcleos seguindo as previsões adaptadas da Lei de Moore, medidas de evolução dos sistemas de memória também serão necessários, e é um requerimento que a indústria falhou em entregar no passado (*gap* entre a evolução na velocidade dos processadores e a velocidade dos sistemas de memória DRAM) (VAJDA, 2011).

2.8 Considerações

Neste capítulo foram abordados os principais conceitos sobre o paralelismo no que se refere à arquitetura e organização de sistemas computacionais, os paradigmas de programação disponíveis para desenvolvimento de aplicações voltadas à exploração de sistemas paralelos multi-core, seu histórico, especificidades, vantagens, desvantagens e "público alvo".

A partir do estudo acima desenvolvido é possível verificar que a programação paralela é um desafio emergente devido à grande heterogeneidade do campo que trata as aplicações paralelas e os diversos modos como estes podem ser abordados e abstraídos. A exploração do paralelismo em si é pouco natural e possui uma aparência complexa e pouco intuitiva, tornando o processo de desenvolvimento ainda mais penoso aos desenvolvedores.

Neste trabalho serão trabalhadas as metodologias direcionadas a programação de sistemas *multi-core*, avaliando a aplicabilidade dos modelos de programação via passagem de mensagens, confrontando seus resultados com os modelos de programação por memória compartilhada. No capítulo seguinte consta a revisão bibliográfica voltada à abordagem dos paradigmas voltados à programação em sistemas *multi-core*.

Capítulo 3

TRABALHOS RELACIONADOS

3.1 Programação Paralela Multi-core

A indústria de semicondutores se apoiou nos preceitos da Lei de Moore por muitos anos para proporcionar ganhos consistentes de desempenho através do aumento na quantidade de transistores e frequência de *clock* dos processadores (SHEKHAR et al., 2011). Esse crescimento proporcionava ainda um aumento na velocidade de execução das aplicações que executavam sobre as plataformas cada vez mais velozes, fenômeno que ficou conhecido como *Free Lunch*. Ao longo desses anos de "prosperidade", os programadores contavam com o avanço das estruturas de hardware para acelerar suas aplicações sequenciais, utilizando estruturas cada vez mais complexas de *pipelining* e mecanismos de ILP (*Instruction Level Parallelism*).

A abordagem do processamento paralelo ainda era considerada uma via extremamente especializada, associada diretamente à computação de alto desempenho e sistemas de grande porte. Contudo, quando as barreiras físicas limitantes ao contínuo avanço dos processadores de núcleo único surgiram e a exploração dos mecanismos de ILP se exauriam cada vez mais, a era dos *Free Lunches* se encerrou (SANDERS, 2014), dando lugar ao que foi denominado "era *multi-core*" com o advento dos primeiros processadores com mais de um núcleo de processamento.

Aplicações que anteriormente poderiam tirar proveito das melhorias efetuadas em um processador de núcleo único, não mais o fariam, pois o que se encontra nas arquiteturas de processadores *multi-core* são dois ou mais elementos de processamento encapsulados em um único dispositivo, e esses elementos geralmente são mais "lentos" se comparados aos processadores de núcleo único. A grande vantagem dos processadores com mais núcleos é a possibilidade de explorar o real paralelismo, a execução de várias tarefas simultaneamente.

Neste cenário fica clara a necessidade de modelos de programação que atendam as expectativas dos paradigmas que os processadores *multi-core* introduziram, e não só isso, a necessidade de treinamento dos profissionais e mudança sobre como são concebidas as aplicações e como elas podem tirar melhor proveito do hardware atualmente disponível.

Um estudo (MEADE; BUCKLEY; COLLINS, 2011), relacionado aos desafios da evolução de código sequencial para os moldes paralelos, identificou um total de dez desafios que os desenvolvedores enfrentam no processo de migração entre paradigmas. A Tabela 3.1 os apresenta.

Categoria	Desafio	Descrição
Desenvolvedor	Compreensão Paralela	Código paralelo é mais difícil de entender que o código sequencial
	Falta de Treinamento	Programadores não estão recebendo treinamento no campo de programação paralela
	Falta de Conhecimento de Domínio	Programadores paralelos não tem conhecimento de domínio para paralelizar completamente uma aplicação
Métodos de Desenvolvimento e Ferramentas	Testes	Testar programas paralelos não é trivial
	Falta de Ferramentas	Não há ferramentas suficientes disponíveis para ajudar os programadores paralelos
	Depuração Complexa	Depurar um programa paralelo não é trivial
Especificidade de Aplicação	Dependências de Dados	Gerenciar dependências de dados ao escrever código paralelo não é trivial
Contenções	Interdependência de Níveis de Abstração	Manter a consistência entre níveis de abstração não é trivial
	Variabilidade de Hardware	A configuração de conexão entre núcleos em um processador pode variar amplamente e afetar o software
	Limitações de Tempo	Reescrever código legado sequencial consome muito tempo

Tabela 3.1: Desafios enfrentados pelos programadores paralelos [Adapt. (MEADE; BUCKLEY; COLLINS, 2011)]

Da parte do desenvolvedor, detecta-se a falta de compreensão de maneira geral sobre o que é um programa paralelo. O paradigma sequencial descreve de uma maneira mais "huma-

nizada" como resolver um problema, passo-a-passo, possibilitando visualizar o fluxo de eventos que envolve um programa, porém não é mais uma solução adequada para ambientes que em sua essência consistem na exploração do paralelismo. A falta de treinamento em técnicas de concorrência e conhecimento do domínio da aplicação também influenciam diretamente na qualidade do código gerado pelos programadores, limitando o aproveitamento dos recursos computacionais e prevalecendo a visão simplista de paralelização.

A programação paralela é conhecidamente complexa. Em ambientes *desktop* o que prevalece ainda são aplicações baseadas em paralelismo via *threads* em sistemas de memória-compartilhada, o que consiste na "arte de balancear forças conflitantes". Tornar um código *thread-safe* requer proteger variáveis compartilhadas através de *locks* (definindo regiões críticas), o que pode comprometer a escalabilidade do código (OKUR; DIG, 2012). Além disso, o paralelismo pode ofuscar a real intenção do código, inserindo um elemento de complexidade na interpretação e reuso. Por muito tempo a abordagem de desenvolvimento *multithread* possibilitou ganhos com o melhor aproveitamento do tempo de processamento e é um conceito que deve ser continuamente reaproveitado em ambientes *multi-core*.

Do ponto de vista das metodologias de desenvolvimento e ferramentas, percebe-se a necessidade de uma variedade de ferramentas mais extensiva para suportar as necessidades específicas dos programadores paralelos. Apesar da grande coleção de ferramentas para modelos de programação paralela tradicionais (Pthreads, OpenMP, por exemplo), linguagens e compiladores, a comunidade de software não acompanha o desenvolvimento paralelo: aplicações, sistemas operacionais e ambientes de programação ainda não cooperam com os requerimentos propostos pela nova geração de hardware. Essa disparidade entre a evolução do hardware e a falta da contraparte em software é denominada "*multi-core programmability gap*" (em tradução livre, o abismo de programabilidade *multi-core*), e vem se tornando uma grande preocupação (VARBANESCU, 2010).

O *multi-core programmability gap* leva diretamente à análise feita em (MEADE; BUCKLEY; COLLINS, 2011) no que diz respeito aos desafios de **Especificidades de Aplicação e Contenções**. Manter a consistência entre todos os níveis de abstração (linguagem de programação, compiladores, bibliotecas, *middleware* e sistema operacional) é um processo extremamente complexo. A concorrência precisa ser expressada de maneira sistemática e propagada entre todos os níveis de abstração para execução eficiente da aplicação e manutenção da consistência de dependência de dados entre *threads*. Essa consistência ainda é dificultada pela variabilidade um tanto excessiva na arquitetura de hardware *multi-core* disponível (quantidade de núcleos, tamanho do *cache*), pois as aplicações uma vez dimensionadas, por exemplo, para um proces-

sador de 4 núcleos deverão ser calibradas novamente caso o número de núcleos aumente ou diminua.

Há duas possíveis ideias que abordam a exploração do paralelismo: implicitamente e explicitamente. Visões são expostas por August e Pingali em (ARVIND et al., 2010), onde defendem respectivamente cada uma das abordagens.

A abordagem de paralelismo implícito remove parcialmente a responsabilidade do programador quanto a paralelização do código, deixando a cargo do compilador e de outras ferramentas detectar seções de código onde há possibilidade de paralelização e assim, inserir estruturas que possibilitem a execução paralela de tal código. A complexidade deste processo é diretamente associada à dificuldade de prever dependências de dados e consistência na ordem de execução das operações. O paralelismo implícito é considerado por muitos uma abordagem exaurida. August é mais específico, propondo um híbrido, "modelo de programação paralela implícito e paralelização dinâmica", baseado em anotações (*annotations*). Diferente de abordagens como o OpenMP onde a anotação descreve como a seção de código deverá ser paralelizada, a abordagem de August propõe anotações que comunicam o tipo de dependência agregado à fração do programa em questão, criando limitações que permitirão que o compilador julgue a melhor estratégia de paralelização, conservada a consistência do código.

Por outro lado, a abordagem de paralelismo explícito é defendida por Pingali, definida por ele como um "mal necessário" aos programadores que desejam tirar proveito efetivo do paralelismo. A grande quantidade de especificidades relacionadas ao domínio de cada problema a ser solucionado, como definição e construção de estruturas de dados específicas para cada aplicação, a exploração do não-determinismo para desempenho, dependência de dados gerados durante o tempo de execução, dentre outras, tem como consequência a necessidade inerente de escrever código explicitamente paralelo e aparentemente é a solução mais viável para a situação atual de software e hardware.

Explorar o paralelismo explícito é a via mais econômica de explorar a grande quantidade de recursos disponível em processadores. Todas as aplicações de desempenho crítico precisam usar o hardware paralelo de maneira eficiente. Apesar dos grandes esforços em linguagens de programação paralela e paralelização de compiladores, a implementação de algoritmos paralelos ainda é um dos maiores desafios no ciclo de engenharia algorítmica, consequência do citado multi-core programmability gap, além da inerente dificuldade de construir, depurar e otimizar algoritmos paralelos (SANDERS, 2014).

3.2 Conceitos Emergentes

3.2.1 Multitask para Multi-core

A complexidade em programar aplicações que utilizem o paralelismo é clara após as discussões anteriores. Há uma falta evidente de cadeias de ferramentas e processos de desenvolvimento de aplicações paralelas que atendam às expectativas dos programadores. O ciclo de paralelização de código legado é longo e trabalhoso, o que muitas vezes desestimula e encarece o processo de adesão aos conceitos do paralelismo em sistemas *multi-core*.

Uma abordagem de metodologia de desenvolvimento, focada em sistemas embarcados e agregada a uma ferramenta visual de construção, mas que pode ser generalizada para ambientes multi-core homogêneos (caso da maioria dos processadores disponíveis no mercado) é a apresentada em (MORON et al., 2014).

Em linhas gerais o ciclo de desenvolvimento proposto pelos autores consiste em iniciar o desenvolvimento de aplicações a partir do modelo de *multitask* e convertê-lo em código para *multi-core* mapeando as tarefas para núcleos de processamento distintos, finalizando com a calibração da aplicação para a plataforma em que será executada.

O *multitasking* é definido como um método pelo qual diversas tarefas ou processos compartilham recursos de processamento comum. Um sistema operacional *multitask* é capaz de executar diversos processos, trocando rapidamente o contexto entre as mesmos, dando a impressão de simultaneidade. O *multitasking* é um conceito antigo e bem estabelecido, e viabilizou a utilização dos sistemas computacionais como são utilizados atualmente.

Modelar uma aplicação *multitask* consiste em fracionar o trabalho que seria realizado por um único processo em processos menores, geralmente necessitando de mecanismos de IPC para comunicação. Pode-se modelar de maneira intuitiva diversas aplicações utilizando modelos como *master-slave* e cliente-servidor. Para preservar escalabilidade, preferencialmente utilizam-se mecanismos de comunicação assíncronos, como filas, para troca de dados entre tarefas, eliminando ao máximo a necessidade de utilização de qualquer forma de região crítica implementada através de *locks* (uma das oito regras apresentadas por James Reinders abomina a utilização de *locks* e os caracteriza como último recurso).

Ao modelar a aplicação *multitask* e efetuar testes para afirmar a consistência do código (definir se o código funciona de maneira adequada aos fins que foi projetado), avalia-se o desempenho da aplicação através de ferramenta de *tuning* (TAU – Tuning and Analysis Utilities, Universidade de Oregon - (SHENDE; MALONY, 2006)), para fins de detecção de gargalos de

desempenho, aperfeiçoamento do código e balanceamento de carga entre as tarefas. Após esta etapa, a aplicação convencional para processadores *unicore* é funcional.

O próximo passo é mapear as tarefas para núcleos distintos, visando o balanceamento de carga ou o atendimento de requisitos de tempo-real no caso de sistemas embarcados críticos. Uma ou mais tarefas são mapeadas para cada núcleo e compiladas em um executável. Cada executável é processado em seu respectivo núcleo e as filas são substituídas por *pipes* que permitem a comunicação entre núcleos. Por fim, novos testes e análise através de ferramenta de *tuning* são realizados para calibração de desempenho e depuração.

A abordagem apresenta um modelo genérico de desenvolvimento de aplicações *multi-core* a partir de um modelo já conhecido de *multitask*. A refatoração de código sequencial para *multithread*, como apresentado em (VANDIERENDONCK; MENS, 2011), abre portas para a paralelização de código legado, porém o *multithreading* não possibilita um processo de paralelização produtivo em quesitos temporais. De outro ponto de vista, refatorar código sequencial para o paradigma *multitask* também possui vantagens, pois a partir deste, através de uma sequência de passos, pode-se convertê-lo para execução paralela em ambientes *multi-core* de maneira prática e eficiente (INTEL. . . ,).

3.2.2 Mecanismos Modernos de IPC baseados em Mensagens

Além dos mecanismos de IPC convencionais citados na seção 2.5.1, com a contínua busca por mais eficiência e flexibilidade e a constante evolução de arquiteturas de sistemas, cada vez mais orientadas à exploração do processamento paralelo e programação baseada em serviços, uma miríade de dificuldades que antes não existiam precisam ser tratadas, como tolerância à falhas, disponibilidade e escalabilidade. Nesse contexto surgem novos mecanismos para atender tais necessidades e há uma tendência clara, na última década, de criação de mecanismos de IPC baseados em filas e passagem de mensagens, o que suporta a hipótese de que a abstração de mensagens pode realmente facilitar o desenvolvimento de aplicações paralelas. Nesta seção serão abordados brevemente alguns dos mecanismos mais recentes nessa área de IPC.

ZeroMQ

ZeroMQ é uma biblioteca de passagem de mensagens assíncrona de alta performance, desenvolvida em C++, provinda do expertise adquirido pelos desenvolvedores em aplicações do mercado financeiro. Conta com uma grande comunidade de desenvolvimento e suporte pago para soluções corporativas. Trata-se de um *framework* de concorrência que suporta conexões

diretas e *fan-out* sobre uma diversidade de protocolos de transporte.

Conhecida pelo termo "*sockets on steroids*", a biblioteca ZeroMQ oferece uma API que lembra os *Berkeley (BSD) sockets*, porém se comportam como uma espécie de generalização entre os *TCP sockets* e os *UNIX domain sockets*. Implementa padrões de programação *PAIR*, *Request/Reply*, *Publish/Subscribe* e *Push/Pull* via canais TCP, intra-processo e inter-processos, garantindo o recebimento de mensagens completas e ordenadas (o que explica a ausência de suporte para UDP) (HINTJENS, 2013).

Consegue superar o *throughput* de uma conexão em *TCP sockets* por utilizar um mecanismo de *batching* inteligente, minimizando a passagem de mensagens pela pilha de rede e desabilitando o algoritmo de Nagle (responsável por minimizar a quantidade de pacotes enviados pela rede, controlando casos de congestionamento). Emprega também um padrão denominado *"Suicidal Snail"*, que força o encerramento de processos clientes muito lentos - a ideia desse padrão consiste em "falhar mais rápido", abrindo espaço para resoluções rápidas do que potencialmente permitir que os processos operem de maneira inadequada ou com baixo desempenho.

ZeroMQ permite alcançar comunicação confiável, rápida e escalável, tanto em ambientes distribuídos como em ambientes de computação paralela, facilitando as comunicações intraprocessuais e interprocessuais utilizando os mesmos padrões de programação. Também garante escalabilidade em ambientes multi-core sem grandes alterações no código-fonte, trabalhando como um *middleware* tradicional orientado à mensagens (TREAT, 2014a).

nanomsg

A biblioteca de *sockets* nanomsg objetiva tornar a camada de rede rápida, escalável e fácil de usar. É implementada em C e trabalha com uma ampla variedade de sistemas operacionais sem muitas dependências externas (SUSTRIK, 2015).

Ao invés de atuar como uma biblioteca de comunicação genérica, provê blocos semi-prontos para construção de sistemas distribuídos escaláveis e com bom desempenho, implementando o que denomina-se *protocolos de escalabilidade*. Os protocolos de escalabilidade são padrões de programação, abstrações sobre a camada de transporte da rede individualmente definidas, implementando um algoritmo distribuído específico.

Atualmente define seis protocolos de escalabilidade diferentes:

- PAIR (Comunicação bidirecional): Implementa comunicação bidirecional um-para-um entre dois processos. Processos podem enviar e receber mensagens entre si através dessa

conexão exclusiva.

- REQREP (Request/Reply): Define o padrão comumente conhecido como Cliente/Servidor, criando serviços *stateless* para processar requisições.
- PIPELINE (*One-Way Dataflow*): Provê fluxo unidirecional entre processos, útil para implementação de *pipelines* de processamento balanceados. Um processo produtor submete trabalho para um processo consumidor.
- BUS (Comunicação de Muitos-para-muitos): Permite o envio de mensagens de um processo para todos os outros processos na rede de conexões.
- PUBSUB (*Transmissão baseada em Tópicos*): Permite que processos *publisher* envie mensagens em modo *multicast* categorizadas por tópicos para zero ou mais processos *subscribers*. Os processos *subscriber* podem declarar interesse em um ou mais tópicos, permitindo receber somente as mensagens relevantes para seus objetivos.
- SURVEY (Requisição para Grupo): Similar ao padrão PUBSUB, onde um processo pode enviar uma mensagem para um grupo inteiro, porém, diferente do PUBSUB, todos os processos respondem à mensagem, o que permite avaliar o estado de um grande número de sistemas de maneira fácil e rápida.

Várias melhorias e correções foram propostas pela nanomsg quando comparada com a supracitada ZeroMQ. É totalmente *POSIX-compliant*, o que oferece uma interface mais limpa e compatível, e é *thread-safe*, permitindo sua utilização em ambientes *multithread* de maneira segura e transparente ao programador. Oferece ainda um mecanismo real de transmissão *zero-copy*, o que melhora consideravelmente o desempenho permitindo que memória seja copiada de máquina para máquina evitando a passagem pela CPU.

A grande quantidade de recursos e melhorias providas torna a biblioteca nanomsg promissora. Contudo, por ser um ferramental muito recente (ainda se encontra em versão Beta), não possui maturidade como a ZeroMQ e não é uma ferramenta adotada em ambientes de produção. Encontra-se em desenvolvimento ativo e pode oferecer em um futuro próximo um conjunto de soluções robusto e eficiente (TREAT, 2014b).

LINX

LINX é um protocolo aberto de IPC, desenvolvido pela *ENEA Software AB* para ser independente de plataformas, permitindo que aplicações se comuniquem transparentemente em um

mesmo hospedeiro ou em diferentes hospedeiros, no caso de sistemas distribuídos, por exemplo. LINX é baseado no conceito de passagem de mensagens e opera em conjunto com sistemas de tempo real proprietários.

Consiste em uma série de módulos do *kernel* do Linux, uma biblioteca para dar suporte à aplicações e algumas utilidades de linha de comando para monitoramento, gerenciamento e estatísticas (ENEA, 2009).

A transparência é essencial para sistemas distribuídos, pois permite que processos em máquinas diferentes ou aplicações distintas em múltiplos sistemas operacionais se comuniquem de maneira indistinta como se fossem executadas em um mesmo processador e em um mesmo sistema operacional. LINX permite que processos se localizem e se comuniquem de maneira transparente, sem fazer distinção se um processo é executado na mesma máquina, ou em uma máquina remota, ou mesmo via internet.

Um elemento chave na implementação do protocolo é o uso de um modelo de mapeamento de endereços únicos, no qual os nós comunicantes guardam somente os endereços necessários para conexões locais, e como resultado necessitam de recursos mínimos de memória para código e armazenamento de dados, permitindo reconfigurações rápidas e fáceis. Esse conceito permite que o LINX escale para grandes redes com complexas topologias de *clusters*, incluindo também nós minimalistas como DSPs e microcontroladores.

A comunicação é feita através de sinais, enviados para os *peers* conectados à rede. Como a camada de transporte do LINX não oferece segurança, o tráfego pode ser observado por qualquer processo conectado.

LINX pode ser visto como uma camada entre os clientes e uma interconexão de dados para comunicação, como *Ethernet*, memória compartilhada ou mesmo um protocolo de transmissão de pacotes como TCP ou UDP, provendo transporte de mensagens entre clientes. Um cliente, no contexto do protocolo LINX é denominado ”Comunicador”.

A arquitetura do protocolo LINX pode ser analogamente comparada com as camadas de Sessão, Transporte e Enlace do modelo OSI, como visto na Figura 3.1. Contudo, qualquer protocolo de transporte que se desejar utilizar deverá ser anexado ao conjunto do protocolo LINX, portanto, LINX não substitui nenhum mecanismo, mas agrega funcionalidades ao mesmo (CHRISTOFFERSON, 2006).

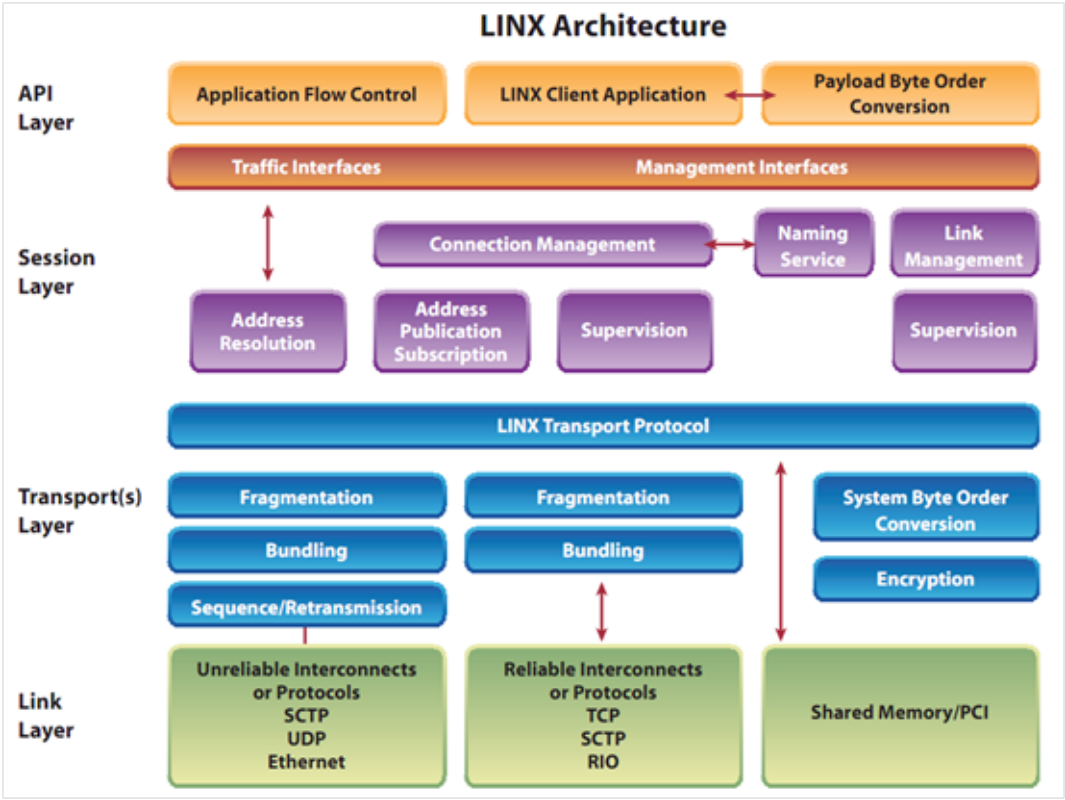


Figura 3.1: Arquitetura do Procolo LINX (CHRISTOFFERSON, 2006)

3.2.3 Otimização do MPI e Modelos Híbridos

MPI é um modelo consolidado para computação distribuída em larga-escala, computação de alto desempenho e programação paralela baseada em mensagens, porém com os avanços de arquiteturas *multi-core* e a crescente inserção de mais núcleos em um único encapsulamento, e ainda com o surgimento de arquiteturas de coprocessadores many-core como o Intel Xeon Phi, oferecendo massivas quantidades de processadores em um sistema de coprocessamento modular, programadores optam cada vez mais por modelos de programação que possam extrair o máximo de desempenho e escalabilidade de seus sistemas.

As abordagens mais comuns encontradas atualmente em sistemas distribuídos são modelos híbridos que associam OpenMP e MPI. Em geral, as tarefas são paralelizadas utilizando múltiplas *threads* criadas e gerenciadas pelo OpenMP e um *thread* ou processo é responsável por operações relacionadas à trocas de mensagens através do MPI. Somente a utilização do OpenMP é possível, porém pode gerar a necessidade frequente de sincronização de páginas de memória para evitar problemas de consistência de informação, e esse processo pode potencialmente ser custoso e complexo quando comparada à utilização de passagem de mensagem puramente por MPI (RABENSEIFNER; HAGER; JOST, 2009).

Algumas propostas criam ou modificam implementações do MPI para torná-las mais adequadas a ambientes multi-core. Propostas como MT-MPI, apresentada em (SI et al., 2014), propõe a modificação do *runtime* do OpenMP para exposição de informações de recursos ociosos, assim o MT-MPI pode escalonar livremente a paralelização baseada nos dados expostos. Abordagens como a do Hybrid MPI (FRIEDLEY et al., 2013), propõe a inserção de uma camada "híbrida", entre a aplicação e as funções do padrão MPI, responsável por gerenciar a alocação e acesso à memória para processos executando em um mesmo hospedeiro, proporcionando uma execução transparente similar a ambientes *multithread*.

3.2.4 Memória Transacional

A memória transacional (do inglês *Transactional Memory*), termo cunhado por (HERLIHY; MOSS, 1993), é um modelo de programação que oferece um alto-nível de abstração para a escrita de programas que exploram paralelismo. Nos últimos anos tem atraído novo interesse devido aos preceitos serem aplicados com sucesso há muito tempo em bancos de dados para isolamento de atividades concorrentes. O modelo disponibiliza mecanismos que permitem programas executarem em isolamento, sem se preocuparem com o contexto de outros executando concorrentemente, oferecendo um promissor, porém não provado, mecanismo para aperfeiçoar a programação paralela (LARUS; KOZYRAKIS, 2008).

Uma transação é uma sequência de ações que aparenta indivisibilidade e instantaneidade para um observador externo. No contexto dos bancos de dados, os quatro principais atributos de uma transação são conhecidos como propriedades ACID (HARRIS; LARUS; RAJWAR, 2010):

- Atomicidade: numa transação, todas as operações devem ser executadas (*commit*) ou caso alguma das operações falhe esta deve ser abortada (*abort*), o que introduz o conceito de *rollback* (desfazer todas as operações e restaurar o estado do sistema para antes da transação).
- Consistência: definição dependente da aplicação, mas em linhas gerais, após uma transação ser efetuada, o resultado da transação deve ser consistente com o resultado esperado.
- Isolamento: transações não devem interferir uma com a outra, independente se estão ou não executando em simultaneidade.
- Durabilidade: quando uma transação é concluída com sucesso, seus resultados devem ser tornados permanentes.

No contexto da Memória Transacional, são utilizadas apenas as propriedades de Atomicidade, Consistência e Isolamento. A ideia básica do modelo de memória transacional é muito simples: as propriedades de transações proveem abstração para coordenar leituras e escritas concorrentes em memória-compartilhada em um sistema concorrente ou paralelo, poupando o programador da utilização de mecanismos de baixo nível, como *locks*, semáforos e *mutexes*, que são difíceis de se utilizar corretamente e não são combináveis.

Os conceitos de memória transacional compreendem tanto implementações em hardware como em software:

- Sistemas HTM (*Hardware Transactional Memory*) envolvem modificações nos processadores, *cache* e protocolos de barramento para suportar transações. Operações *Load-Link/Store-Conditional* (LL/SC), sequência de leitura e escrita se houver atualização em posição de memória, pode ser considerada uma forma muito básica de suporte à TM implementado por processadores RISC. Os processadores da arquitetura Intel Haswell implementam um conjunto de instruções denominado Intel TSX (*Transactional Synchronization Extensions*), permitindo que o programador especifique regiões de código para execução transacional e alcançar desempenho de aplicações que usam *fine-grained locks* utilizando *coarse-grained locks*.

- Bibliotecas de STM (*Software Transactional Memory*) proveem a semântica de TM para linguagens de programação e requerem mínimo suporte do hardware, geralmente necessitando de poucas operações de comparação e troca atômicas. São mais facilmente implementadas, porém inserem uma penalidade devido ao *overhead* introduzido pelo *runtime* que gerencia as transações, e requerem grande esforço de calibração para tirar completo benefício da abstração. Há diversas implementações de bibliotecas de memória transacional para linguagens como o C/C++ (e.g. Rochester STM, TinySTM, e o compilador GCC), Java (e.g. DeuceSTM, TribuSTM), Python (e.g. PyPy STM). Haskell possui suporte nativo para operações atômicas, Scala (e.g. ScalaSTM), entre outras.

A utilização de memória transacional apresenta algumas vantagens, principalmente do ponto de vista do programador, onde os blocos que devem ser executados em transações só precisam ser marcados como atômicos ou isolados, tornando o código mais claro e passando a responsabilidade de sincronizar e garantir as propriedades ACI para o *runtime*.

As principais desvantagens são ligadas em maioria ao desempenho dos programas, pois a execução não é sempre ideal, pois é o *runtime* que decide a melhor maneira de executar; quando sucessivas colisões onde várias operações atômicas tentam executar ao mesmo tempo, o *runtime*

efetua *rollback* de todas, e a situação se repete, tendo como consequência *livelocks*; transações de memória não podem conter certos tipos de operações, como entrada e saída, limitando a sua utilização; incompatibilidade com código legado.

A memória transacional é um resgate de um conceito relativamente antigo visando uma maneira de alcançar melhores modelos de abstração para construção e programação paralela de algoritmos em resposta à evolução dos dispositivos de hardware. É improvável que somente a concepção de mecanismos de memória transacional resolvam todos os problemas de programação paralela, pois muitos outros aperfeiçoamentos de linguagens de programação, ferramentas, sistemas de *runtime* e arquitetura de computadores também são necessários. No entanto, se esses aperfeiçoamentos forem concebidos, a memória transacional pode se tornar o "pilar central" da programação paralela (LARUS; KOZYRAKIS, 2008).

3.3 Tendências

Dado o atual cenário do desenvolvimento de processadores *multi-core* e visto que diversos problemas e gargalos de desempenho estão presentes e podem causar efeitos catastróficos em sua escalabilidade, a tendência é que as arquiteturas explorarão cada vez mais sistemas que otimizem e driblem os efeitos causados pelos fenômenos físicos e barreiras impostas.

Processadores heterogêneos especializados e otimizados surgirão, agregando processadores menores, mais lentos, e menos complexos, sendo um passo natural para a otimização e ganho de desempenho. As tecnologias *power-aware* passarão a ser usadas como regra, tornando-se extremamente necessárias para o contínuo aperfeiçoamento e aumento de desempenho. Os aceleradores em hardware se tornarão abundantes, seguindo as otimizações do uso de energia.

O foco dos sistemas de memória com maior largura de banda para acesso é uma necessidade iminente. Os memristores, utilização do empilhamento de transistores em três dimensões e conexão ótica entre os elementos de memória ganharão destaque e poderão trazer grandes mudanças, reduzindo a pressão sobre como os circuitos serão organizados em futuras arquiteturas.

3.4 Considerações

Atualmente as metodologias dominantes na área de programação *multi-core* utilizam o paradigma de memória compartilhada através de APIs de programação concorrente, de mecanismos projetados para sistemas distribuídos (OpenMP e MPI), ou mesmo contando com o sis-

tema operacional para que os processos sejam distribuídos de maneira a aproveitar a estrutura de hardware.

Conceitos como a especificação MCAPI, que será apresentada na Seção 4.1, uma das primeiras especificações oficiais voltadas diretamente a programação em sistemas *multi-core*, e o mecanismo de memória transacional, voltando a ganhar atenção devido aos avanços tecnológicos na arquitetura e organização de processadores, são alguns exemplos dos conceitos emergentes e possivelmente farão parte do conjunto de técnicas estado-da-arte em programação *multi-core*.

Apesar de todos os fatores e da problemática envolvida com o futuro das arquiteturas *multi-core*, é fato que estas são a alternativa mais viável para a contínua evolução dos sistemas computacionais e é válido que seu estudo seja aprofundado e aperfeiçoado, trazendo novas metodologias visando melhor proveito dos recursos disponibilizados.

Capítulo 4

PROPOSTA DE BIBLIOTECA DE COMUNICAÇÃO

Este capítulo apresenta primeiramente alguns conceitos específicos que foram diretamente utilizados como base teórica e inspiração para o desenvolvimento do presente trabalho (seções 4.1 e 4.2). Tais conceitos são essenciais para o entendimento da estrutura e funcionamento da biblioteca LX-MCAPI, apresentada na seção 4.3, resultado das pesquisas efetuadas na área de mecanismos de comunicação interprocessual.

4.1 Especificação MCAPI

Em ambientes multi-core heterogêneos, qualquer modelo de programação que assuma implicitamente homogeneidade do sistema é impraticável. Por exemplo, Pthreads é insuficiente para interações onde os núcleos não compartilham o mesmo domínio de memória ou que não executam a mesma instância de sistema operacional. Além disso, muitas implementações de outros padrões, como o OpenMP, usam Pthreads como uma API em uma das camadas de abstração, permeando as mesmas dificuldades. Sistemas com núcleos, ISAs e arquiteturas de memória heterogêneas se assemelham muito aos sistemas distribuídos ou de computação científica. Para estes, existem diversos modelos de programação, incluíndo *sockets*, CORBA e MPI, porém há diferenças com relação a computadores interconectados e computadores multi-core que não são bem descritos por tais modelos (HOLT et al., 2009).

No intuito de atender a demanda dos desenvolvedores por um modelo de programação que atenda as plataformas de hardware *multi-core*, as empresas que fazem parte da Multicore Association (MCA) publicaram um modelo generalizado para programação *multi-core* endereçada especificamente aos sistemas embarcados. A prioridade do modelo foi estabelecer os mecanismos de comunicação entre núcleos, uma das grandes características positivas dos processadores multi-core, provendo melhor acoplamento.

Consiste em uma especificação que define uma API e uma semântica de comunicação e sincronização entre núcleos de processamento, primariamente voltada à sistemas embarcados. Não define quais políticas de gerenciamento de conexão, modelos de dispositivos ou protocolos de conexão são utilizados como base de construção.

MCAPI provê uma quantidade limitada de chamadas com funcionalidades de comunicação básicas, mantendo sua utilização simples para permitir implementações eficientes. Funcionalidades adicionais podem ser adicionadas em camadas sobre a API principal.

Baseado nos conceitos de trocas de mensagens, MCAPI é um protocolo de comunicação independente de linguagem, processador e sistema operacional usado para programação de dispositivos *multi-core*. Provê três modos de comunicação: mensagens, pacotes e escalares

Os principais objetivos são portabilidade de código-fonte e escalabilidade de comunicação e memória conforme o número de núcleos dos processadores aumente nas gerações futuras de processadores. Além desses objetivos, a especificação satisfaz requisitos mais amplos de processamento multi-core, tais como (WALLS, 2012):

- Adequação a ambientes *multi-core* homogêneos e heterogêneos
- Adequação a arquiteturas uniformes, não-uniformes ou de memória distribuída
- Compatibilidade com aceleração de hardware dedicado
- Compatibilidade com sistemas operacionais SMP e Não-SMP
- Compatibilidade com padrões de programação existentes
- Suporte para comunicação bloqueante e não-bloqueante

4.1.1 Detalhes da Especificação

MCAPI usa uma hierarquia que organiza as entidades comunicantes e os elementos de mensagens. Define três tipos de elementos de mensagens (mensagens, escalares e pacotes) e chamadas síncronas e assíncronas de envio e recebimento. Os elementos que compõem a hierarquia são descritos a seguir.

Os domínios são coleções de nós, onde cada nó pode ser um processo, *thread*, ou outro tipo de programa em execução. Em um sentido mais amplo pode denotar separações entre núcleos ou entre processadores. Fica a cargo da implementação decidir qual a função dos nós.

Nós devem ser inicializados com um identificador único (geralmente um número inteiro não-sinalizado) que permitirá identificá-lo perante outros nós.

Cada um dos nós pode ter um ou mais terminais usados para comunicação de mensagens não-orientadas-à-conexão e canais orientados-à-conexão de pacotes e escalares. Ao utilizar canais, conexões ponto-a-ponto unidirecionais são estabelecidas entre nós utilizando um mecanismo de transporte definido pela implementação. Um terminal só poderá se comunicar com outro terminal caso estejam conectados ao mesmo canal. Mensagens, podem ser enviadas e recebidas por qualquer terminal e o mesmo pode tanto enviar quanto receber mensagens - a única exceção é um terminal previamente conectado a um canal não pode enviar nem receber mensagens.

Função	**Propósito**
Geral (Inicialização, Criação e Sincronização)	
`mcapi_initialize(6)`	Inicializa o ambiente MCAPI
`mcapi_endpoint_create(2)`	Cria um terminal local
`mcapi_endpoint_get(5)`	Recupera a identificação de um terminal
`mcapi_wait(4)`	Aguarda o retorno de uma função não-bloqueante
Mensagens	
`mcapi_msg_send(6)`	Envia uma mensagem para um terminal
`mcapi_msg_recv(5)`	Recebe uma mensagem de um terminal
Pacotes	
`mcapi_pktchan_connect_i(4)`	Conecta dois terminais, criando um canal de pacotes
`mcapi_pktchan_recv_open_i(4)`	Abre o terminal para recebimento de pacotes
`mcapi_pktchan_send_open_i(4)`	Abre o terminal para envio de pacotes
`mcapi_pktchan_send(4)`	Envia um pacote pelo canal
`mcapi_pktchan_recv(4)`	Recebe um pacote pelo canal
`mcapi_pktchan_release(2)`	Libera o *buffer* de um pacote recebido
Escalares	
`mcapi_sclchan_connect_i(4)`	Conecta dois terminais, criando um canal de escalares
`mcapi_sclchan_recv_open_i(4)`	Abre o terminal para recebimento de escalares
`mcapi_sclchan_send_open_i(4)`	Abre o terminal para envio de escalares
`mcapi_sclchan_send_uintn(3)`	Envia um escalar de n-bits (8, 16, 32 ou 64)
`mcapi_sclchan_recv_uintn(2)`	Recebe um escalar de n-bits (8, 16, 32 ou 64)

Tabela 4.1: Principais funções da especificação MCAPI [Adaptada de: (VIRTANEN et al., 2014)]

As principais funções da especificação MCAPI são listadas na Tabela 4.1, adaptada de (VIRTANEN et al., 2014), separando-as em categorias baseadas em tipos de comunicação. O número de parâmetros é informado entre parênteses.

A comunicação pode ser síncrona ou assíncrona, também denominada bloqueante e não-bloqueante, respectivamente. As chamadas síncronas trabalham de maneira muito similar às chamadas bloqueantes de outros sistemas, literalmente bloqueando o processamento até que a transferência de dados seja completada ou que retorne algum erro. O código 4.1 ilustra a utilização das funções de envio e recebimento de mensagens síncronas.

```
// Include unique header mcapi.h
#include "mcapi.h"
// Include libraries and dependencies
...
int main(int argc, char** argv)
{
  // Declare buffers and variables
  ...
  // Initialize node
  mcapi_initialize_node(1, &error)

  // Create terminals and initialize handles
  ...
  // Synchronous send and receive operations
  mcapi_pktchan_send(send_handle, sbuffer, sbufsz, &error);
  mcapi_pktchan_recv(recv_handle, &rbuffer, rbufsz, &error);
  ...
  // Release packet buffer
  mcapi_pktchan_release(&rbuffer);
}
```

Código 4.1: Envio e Recebimento de Pacotes Síncronos

As chamadas assíncronas por outro lado, executam em segundo plano, e outra função da API deve ser chamada para verificar se a operação foi completada. O sufixo "_i"caracteriza chamadas não-bloqueantes na especificação MCAPI. O Código 4.2 ilustra a utilização das chamadas assíncronas para pacotes.

```
// Include unique header mcapi.h
#include "mcapi.h"
// Include libraries and dependencies
...
int main(int argc, char** argv)
```

```
{
  // Declare buffers and variables
  ...
  // Initialize node
  mcapi_initialize_node(1, &error)

  // Create terminals and initialize handles
  ...
  // Asynchronous send operation
  mcapi_pktchan_send_i(s_handle , &s_buf , &srequest , &status );
  // Wait until completion or timeout
  mcapi_wait (&srequest , &size , TIME_OUT , &status );
  ...
  // Asynchronous receive operation
  mcapi_pktchan_recv(recv_handle, &rbuffer, rbufsz, &error);
  // Wait until completion or timeout
  mcapi_wait (&rrequest , &size , TIME_OUT , &status );
  ...
}
```

Código 4.2: Envio e Recebimento de Pacotes Assíncronos

4.1.2 Implementações Existentes

MCAPI é uma especificação relativamente recente - sua última revisão foi publicada em 2011 - e já possui algumas implementações, na maioria delas proprietárias. As implementações *opensource*, apesar de existentes, são limitadas, incompletas, ou pouco estáveis. Abaixo encontram-se algumas descrições sobre as implementações existentes.

Implementações Proprietárias

- **PolyMessenger/MCAPI**: desenvolvido pela Polycore Software, fornece um *framework* para *closely distributed computing*, como placas com múltiplos processadores ou processadores de múltiplos núcleos. Oferece níveis de abstração e ferramentas de elaboração de topologia para facilitar o processo de desenvolvimento.
- **ThreadX RTOS with MCAPI Support**: sistema operacional de tempo-real, também desenvolvido pela Polycore Software, principalmente voltado à sistemas embarcados de núcleo único ou *multi-core*, trabalha diretamente com o produto PolyMessenger, criando um ambiente de desenvolvimento coeso.

- **Dual-core Blackfin Processors Connected by MCAPI**: produzido pela Analog Devices, o CrossCore Embedded Studio, *toolchain* para programação de processadores Blackfin conta com uma implementação da MCAPI para gerenciamento de conexão e sincronização entre núcleos de um processador multi-core, ou gerenciamento *inter-threads* em processadores de núcleo único.

Implementações OpenSource

- **OpenMCAPI**: criada pela Mentor Graphics, é uma implementação da MCAPI baseada em memória compartilhada, que utiliza a abstração da passagem de mensagens para tarefas de comunicação e sincronização à nível de processos. Desenvolvida para atender sistemas dual-core, limita-se à utilização de somente dois processos e apresenta comportamento instável ao estressar o mecanismo de troca de mensagens.

- **Kactus2**: ferramenta gráfica desenvolvida na *Tampere University of Technology*, voltada principalmente a sistemas baseados em FPGAs e MP-SoCs, provê abstração de hardware e software com suporte ao conjunto de ferramentas IP-XACT e MCAPI como gerenciamento e sincronização de processos em alto nível.(KAMPPI et al., 2011)

- **PMQ-MCAPI**: desenvolvida na *Tampere University of Technology*, PMQ-MCAPI (*Posix Message Queue*-MCAPI) é baseada na utilização de filas de mensagens do padrão POSIX como camada de transporte. Não implementa chamadas assíncronas. (VIRTANEN et al., 2014)

4.2 Protocolo D-Bus

D-Bus é um mecanismo de IPC e RPC (*Remote Procedure Call*) que permite a comunicação concorrente entre múltiplos processos em um mesmo *host*. Faz parte do projeto *freedesktop.org* na tentativa de padronizar os serviços providos por ambientes de Desktop como Gnome e KDE. Concebido como um mecanismo leve e rápido, foi inicialmente desenvolvido para substituir CORBA e DCOP como uma camada de *middleware* unificada, que opera sob os ambientes de desktop oferecendo suporte à comunicação.

É baseado em conexões *stateful* - informações sobre a conexão entre dois processos são mantidas - tornando-o mais ”inteligente”que outros mecanismos de passagem de mensagem em baixo nível como *UNIX domain sockets*. Trabalha com mensagens como elementos discretos, não *streams* contínuos de bytes, tratando os dados de forma estruturada e em forma binária (não

há tradução para modos textuais de representação como XML ou JSON). Por essa razão, como a semântica dos dados é mantida, mensagens podem facilmente serem validadas e mensagens mal-formadas, rejeitadas.

4.2.1 Arquitetura do Protocolo

A arquitetura do protocolo D-Bus pode ser estratificada em três camadas principais (D-BUS. . . ,):

- Uma biblioteca de baixo-nível, `libdbus`, que permite a conexão e troca de mensagens entre dois processos.
- Um *message bus daemon* executável, construído a partir da `libdbus`, que permita a conexão de múltiplas aplicações. O *daemon* poderá rotear as mensagens para uma ou mais aplicações, ou até para nenhuma.
- *Bindings* ou *Wrappers* baseados em *frameworks* de aplicação, como os já existentes para `glib` e `Qt`. Há também *bindings* para linguagens como Python e Java. Recomenda-se a utilização destas abstrações para desenvolvimento de aplicações, pois a `libdbus` tem como objetivo ser o *backend* para abstrações de alto nível.

A biblioteca de baixo-nível (denominará-se BBN nas próximas referências) suporta conexões um-para-um, analogamente aos *UNIX domain sockets*, porém, ao invés de enviar *bytestreams*, enviam-se mensagens. Mensagens possuem um cabeçalho com informações relacionadas ao tipo de mensagem, e um corpo contendo o *payload*. A BBN também abstrai informações relacionadas à camada de transporte e cuida de detalhes de autenticação.

4.2.2 *Buses*, Endereços e Conexões

Em nível de protocolo, o D-Bus pode ser usado como transporte ponto-a-ponto de mensagens entre processos que se comunicam diretamente entre si. O verdadeiro potencial do D-Bus vem da utilização de *bus daemons*, que atuam como roteadores para mensagens. O conceito de *bus* refere-se a uma espécie de barramento onde entidades podem ser ligadas e se comunicar.

A implementação do D-Bus oferece de antemão dois *buses* padrão, além de outros que podem ser criados pelo desenvolvedor através de arquivos de configuração:

- **System bus**: um *daemon* global que qualquer aplicação em qualquer contexto do sistema poderá utilizar como transporte. É um ponto centralizado onde os processos podem exportar serviços para que outros processos possam utilizá-los. Geralmente utilizado para comunicação entre aplicações do sistema e sessões de usuário. Somente uma instância de *system bus* pode executar por vez.

- **Session bus**: um *bus* local à sessão de um usuário. Para cada sessão de usuário uma nova instância de *session bus* é criada. Geralmente é utilizado para comunicação entre aplicativos de sessão e o ambiente de gerenciamento de *desktop*.

Aplicações usando D-Bus podem ser clientes ou servidores. Servidores escutam um *bus* por novas conexões enquanto clientes se conectam a servidores através do *bus*. Uma vez que a conexão é estabelecida, um fluxo simétrico de mensagens se inicia. A distinção entre cliente e servidor só é relevante ao estabelecer uma conexão.

Um endereço do protocolo D-Bus especifica o local onde um servidor escutará e onde um cliente irá se conectar. Endereços do D-Bus podem especificar um *UNIX domain socket*, um socket TCP/IP ou qualquer outro endereço definido em futuras iterações da especificação.

Quando da utilização de um *bus daemon*, a BBN automaticamente descobre o endereço do *session bus* através de uma variável de ambiente e do *system bus* através do caminho de um *UNIX domain socket*, ou por uma variável de ambiente que sobrescreva o valor padrão. Sem a utilização de um *bus daemon*, fica a cargo do desenvolvedor definir os papéis de cliente e servidor e especificar um mecanismo de transporte entre os processos - um caso muito específico e pouco comum.

Toda conexão a um *bus* pode ser endereçada através de um ou mais nomes, denominados *connection bus names*. *Bus names* consistem em uma série de identificadores separados por pontos, por exemplo: `com.exemplo.nome`, e os identificadores em si podem conter letras, dígitos, traços e *underscores*. Quando uma conexão é estabelecida, ela passa a possuir um *bus name*, e o *bus* imediatamente define um *bus name* imutável e único que permanecerá enquanto o *bus* existir.

Uma conexão poderá requisitar nomes adicionais baseados em convenções de desenvolvedor para oferecer serviços a outros processos. Os nomes devem consistir em um dois ou mais elementos separados por pontos, tal como `com.exemplo.nome`. Somente uma conexão pode possuir um dado nome em um *bus*, mas nomes podem ser liberados e posteriormente associados a outros clientes.

4.2.3 Modelo de Objetos

A concepção original do D-Bus visava substituir sistemas de comunicação orientados à componentes. Por essa razão, o protocolo compartilha com seus predecessores um modelo de objetos para expressar a semântica de comunicação entre clientes e servidores. Os termos utilizados no modelo de objetos do D-Bus imitam os usados em paradigmas de orientação à objeto, porém isso não limita a utilização do protocolo somente em linguagens orientadas à objeto. De fato, a implementação mais utilizada do D-Bus é descrita em linguagem C procedural.

Objetos (*Objects*)

O termo 'objeto' em si adquire um sentido próprio dentro do contexto do protocolo, representando uma entidade que pode ser exposta, oferecendo métodos que podem ser chamados e sinais que podem ser captados.

Um objeto é criado por um processo cliente e existe dentro do contexto da conexão deste com o *bus*, sendo uma maneira do processo cliente oferecer serviços ao *bus*. Um processo cliente pode criar um ou mais objetos e oferecê-los.

Objetos no D-Bus são unicamente identificados por uma combinação de um *bus name* e um *caminho de objeto*. O *bus name* identifica unicamente o objeto dentro da aplicação e cada aplicação automaticamente atribui um *bus name* único e randômico a cada conexão. Aplicações que desejam exportar objetos pelo D-Bus podem requisitar a posse de um *bus name* adicional "bem conhecido", para prover uma localização simples e consistente para outras aplicações acessarem em tempo de execução.

O **caminho de objeto** intencionalmente se parece com um caminho do sistema de arquivos padrão UNIX. A diferença principal é que o caminho pode conter somente números, letras, *underscores* e o caractere /. De um ponto de vista funcional, o principal propósito do caminho de objeto é simplesmente ser um identificador único do objeto em questão. A hierarquia implícita na estrutura do caminho é baseada puramente em convenções, sendo que aplicações que naturalmente são organizadas em hierarquias podem tirar vantagem deste recurso enquanto outras podem ignorá-la sem quaisquer perdas.

Todo *bus* tem pelo menos um objeto, que representa o próprio *bus* (padronizado com a interface `org.freedesktop.DBus`). Em um nível maior de abstração, um *bus* suporta duas formas de comunicação:

- **Request/Reply (1 para 1)**: modo para invocação de métodos. O cliente envia uma men-

sagem para que o servidor exporte o objeto, e o servidor em resposta retorna uma mensagem ao processo cliente. A mensagem deve conter o caminho do objeto, o nome do método a ser invocado e os valores dos argumentos. A resposta deve conter os resultados da requisição, incluindo valores de parâmetros retornados pela invocação do método, ou informações de exceções por ocorrência de erro.

- **Publish/Subscribe**: modo como um objeto anuncia a ocorrência de um sinal pelas partes interessadas. O processo *publisher* transmite uma mensagem para o *bus* que por sua vez encaminha a mensagem somente aos *subscribers* conectados que estão inscritos para a recepção do tipo de sinal transmitido. A comunicação é unidirecional, considerando que o *publisher* não conhece a identidade nem o número de *subscribers*.

Mensagens

Toda mensagem consiste de um cabeçalho e um corpo. O cabeçalho é formado por diversos campos que identificam o tipo da mensagem, o remetente, assim como informações requeridas para entrega da mensagem ao destinatário (*bus name*, caminho de objeto, nome de método ou sinal, nome de interface, etc). O corpo contém o *payload* de dados a ser interpretado pelo processo, por exemplo os parâmetros ou resultados. Todos os dados são codificados em um formato binário denominado *wire format*, que suporta a serialização de vários tipos como inteiros, ponto-flutuantes, strings, tipos compostos, dentre outros.

A especificação do D-Bus define o *wire protocol*, a maneira como as mensagens são construídas para serem transportadas entre processos, porém não define o método de transporte para entrega dessas mensagens, deixando em aberto a estratégia de transporte como um elemento definido por implementação.

Assinaturas e Tipos

O D-Bus usa um mecanismo de codificação de tipos denominado Assinaturas (*Signatures*) para descrever o número e os tipos de argumentos necessários para métodos e sinais. As assinatures são usadas para declarar/documentar interfaces, *marshalling* de dados e validação. A codificação em *strings* usa um formato simples e essencial para o uso efetivo do protocolo. A Tabela 4.2 adaptada de (MASTERS; BLUM, 2007) apresenta os tipos fundamentais e suas respectivas representações. Além dos citados, ainda é possível codificar *arrays*, *structs* e dicionários.

Caractere	Tipo de Dados
y	Inteiro não-sinalizado de 8-bits
b	Booleano
n	Inteiro sinalizado de 16-bits
q	Inteiro não-sinalizado de 16-bits
i	Inteiro sinalizado de 32-bits
u	Inteiro não-sinalizado de 32-bits
x	Inteiro sinalizado de 64-bits
t	Inteiro não-sinalizado de 64-bits
d	Ponto-flutuante de precisão dupla (IEEE 754)
s	String em formato UTF-8
h	Descritor de Arquivo UNIX

Tabela 4.2: Codificação de Assinaturas do D-Bus

Métodos (*Methods*)

Métodos são o principal modo de implementação do mecanismo de Request/Reply. Métodos podem aceitar qualquer número de argumentos e podem retornar zero ou mais valores. Quando chamadas de método não especificam valores de retorno, uma mensagem "*method return*"é enviada para o processo requisitante. Isso permite que processos detectem que o método invocado remotamente foi concluído, mesmo que nenhum resultado efetivamente seja retornado.

O único caso no qual a mensagem de retorno não é enviada por uma chamada de método é na ocasião onde o parâmetro "*no reply expected*"é enviado como parte da invocação do método, suprimindo a mensagem de retorno.

As assinaturas para a lista de argumentos e para o retorno do método podem conter múltiplos tipos. Para cada argumento aceito e cada valor de retorno, o tipo do valor de argumento ou retorno é simplesmente anexado, em ordem, à assinatura da mensagem. Se o método não aceita argumentos ou não retorna valores, as assinaturas para tais atributos são vazias.

Sinais (*Signals*)

Sinais são a base para o mecanismo de Publish/Subscribe, fornecendo capacidade de envio de mensagens de um-para-muitos. Similar ao retorno de valores de métodos, os sinais podem conter uma quantidade arbitrária de dados, porém são inteiramente assíncronos e podem ser emitidos por objetos do D-Bus a qualquer momento. Por padrão, sinais emitidos por objetos não serão enviados para nenhum cliente. Para receber os sinais, os processos devem explicitamente registrar o interesse em receber determinados sinais emitidos.

Interfaces

As interfaces definem métodos e sinais suportados por objetos. Para usar uma interface, a mesma deve ser conhecida pelos processos remotos. A definição da interface pode ser *hard coded* na aplicação ou pode ser consultada em tempo de execução pelo mecanismo de introspecção do D-Bus - apesar de opcional, o suporte à introspecção é geralmente automático nas diferentes implementações do protocolo. A convenção de nomes para interfaces é similar a dos *bus names*.

Proxies

Um objeto *proxy* é um objeto criado para representar um objeto remoto em outro processo. Ao utilizar a biblioteca de baixo nível o processo de criação, envio e gerenciamento de uma chamada de método é completamente manual. *Bindings* de alto nível fornecem *proxies* como uma alternativa ao processo manual, sendo objetos nativos que quando chamados são convertidos em uma chamada de método que aguarda a mensagem e trata os valores de retorno de forma transparente.

4.2.4 Propostas e Otimizações

A implementação de referência do D-Bus atualmente disponível e amplamente utilizada em distribuições Linux tem alguns problemas conhecidos relacionados a desempenho e organização de código. Propostas para mudanças tem surgido na comunidade de desenvolvimento para solucionar tais problemas. Duas propostas em estágios ainda iniciais de desenvolvimento são apresentadas nesta seção, ambas desenvolvidas por engenheiros da RedHat.

KDBus

O KDBus é uma implementação do protocolo DBus que opera em espaço de *kernel*, permitindo transmissão de grandes quantidades de dados (mensagens da ordem de gigabytes). Pode trabalhar como um mecanismo de passagem de mensagens *zero-copy*, e mesmo no pior caso, uma mensagem e sua resposta são transmitidas com não mais que duas cópias, duas validações e duas mudanças de contexto. Informações de credenciais como ID de usuário e processo, grupos de controle dentre outras são enviadas anexadas a cada mensagem junto a um *timestamps*. O KDBus sempre está disponível para o sistema (não há necessidade de aguardar *daemons* como no caso do D-Bus), os módulos de segurança do Linux podem ser anexados diretamente, evitando condições de corrida, e a API para sua utilização foi simplificada (CORBET, 2014).

KDBus foi implementado como um *dispositivo de caractere* [1] no *kernel*. Processos que desejam se conectar ao dispositivo *bus*, chamam `mmap` para mapear uma área de passagem de mensagens em seus espaços de endereçamento. As mensagens são construídas nesta área de memória mapeada e então passadas ao *kernel* para transporte - é um simples procedimento de cópia de mensagens entre áreas de memória de processos. Mensagens podem incluir *timeouts* para recebimento de respostas. Há um registro de nomes muito similar ao registro tradicional do D-Bus.

O mecanismo *memfd* permite a passagem de mensagens *zero-copy* pelo KDBus. O *memfd* é simplesmente uma região de memória mapeada associada a um descritor de arquivos, operando similarmente a um arquivo mapeado temporário. Um processo que deseja enviar uma mensagem construirá uma área de memória anônima e então transmitirá pelo KDBus. Dependendo do tamanho da mensagem, as páginas de memória relevantes podem ser diretamente mapeadas para o espaço de endereçamento do processo remoto, evitando qualquer cópia. De acordo com os desenvolvedores, esse mecanismo não é tão efetivo para quantidades de dados até 512KiB, pois o *overhead* inserido pelo mapeamento de memória excede as melhorias por evitar o procedimento de cópia.

O mecanismo de transmissão de sinais foi reescrito para utilizar a teoria de *Bloom Filters* para selecionar os destinos das mensagens. *Bloom filters* usam um mapeamento *hash* que permite rápida eliminação de candidatos, tornando a transmissão relativamente eficiente.

Há também um servidor *proxy* em espaço de usuário que pode ser usado com código legado que não foi reescrito para utilizar a nova API, portanto é esperado que aplicações anteriores funcionem em sistemas com KDBus sem mudanças de código.

O código do KDBus já está disponível em repositório GitHub e pode ser testado em sistemas operacionais Linux compatíveis. Houveram tentativas para inserir o módulo no Fedora Rawhide - versão de desenvolvimento da distribuição Fedora - e tentativas de *push* na *mainline* do *kernel* do Linux, porém foram recusadas por Linus Torvalds por hora, considerando que há muitas questões que devem ser resolvidas antes de disponibilizá-lo como recurso do *kernel* (TORVALDS, 2015). O desenvolvedor Greg Kroah-Hartman recentemente divulgou que o KDBus voltará para a fase de planejamento buscando resolver essas questões.

[1] Dispositivos de caractere recebem essa denominação por terem sua comunicação feita através de envio e recebimento de um fluxo de caracteres, priorizando a comunicação eficiente ao invés de volume, sendo feita de forma "não-*bufferizada*", sendo assim cada caractere é lido/escrito no dispositivo imediatamente.

BUS1

BUS1 será um mecanismo de IPC de baixo nível que também proverá uma interface de alto nível para aplicações. Comparada com o KDBus, BUS1 não aparenta ser baseado diretamente na implementação do D-Bus. Ainda encontra-se em estado inicial de desenvolvimento, e funções de envio e recebimento de comandos já está disponíveis na data de escrita deste texto. Poucas informações estão disponíveis no momento, mas parece ser um projeto oficial da Red Hat, licenciado pela LGPL v2.1.

4.3 LX-MCAPI - Projeto e Implementação

LX-MCAPI visou a concepção de uma biblioteca padronizada, eficiente e de fácil utilização para comunicação entre núcleos de um processador *multi-core*. Fornece chamadas de função para conexão, sincronismo, envio e recebimento de elementos de mensagem entre processos, facilitando o projeto de aplicações paralelas que necessitem de comunicação interprocessual. A passagem de mensagens, por meio de replicação e envio de dados, evita a utilização de mecanismos de exclusão mútua (também denominados *locks*), um dos obstáculos para o processo de programação eficiente de sistemas concorrentes.

A estrutura interna é dividida conceitualmente em dois subconjuntos: **comunicação**, composto pela **LX-MCAPI** - anteriormente CHAOS-MCAPI (IDEGUCHI; MORON; FERNANDES, 2015)[2] - uma instância da MCAPI, e **suporte à programação**, composto por **LX-MCAPI Patterns** oferecendo padrões de programação paralela prontos para uso. A LX-MCAPI estende o mecanismo básico da CHAOS-MCAPI, fornecendo métodos de comunicação *zero-copy*, através do sub-módulo **LX-MCAPI Zero**. A Figura 4.1 demonstra um diagrama conceitual dos módulos que compõem a estrutura da LX-MCAPI.

A "camada de transporte" que possibilita a transferência de elementos de mensagem é baseada no protocolo D-Bus, descrito na seção 4.2 associada à utilidades de IPC do *kernel* do Linux.

[2] A CHAOS-MCAPI foi rebatizada por motivos de desambiguação, pois CHAOS já se refere ao esquema de paralelização *Controlled Hogwild with Arbitrary Order of Synchronization* apresentado em (VIEBKE; PLLANA, 2015) e o banco de dados multimídia CHAOS (CHAOS. . . , 2015).

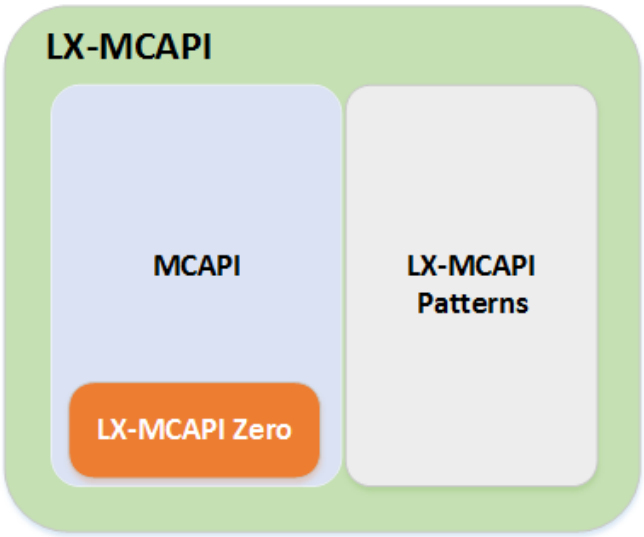


Figura 4.1: LX-MCAPI - Diagrama conceitual de Módulos

Foi implementada em linguagem C utilizando os modelos de cabeçalho fornecidos pela especificação MCAPI, visando certo grau de portabilidade. Sua utilização consiste basicamente na inserção do cabeçalho `mcapi.h`, e uma biblioteca estática nomeada `libmcapi`. A utilização dos padrões de programação paralela exige a inserção adicional do cabeçalho `lxmcapi.h`, que especifica as principais chamadas de função do *LX-MCAPI Patterns*.

4.3.1 Ambiente de Desenvolvimento

LX-MCAPI foi implementada utilizando-se a linguagem de programação C e o sistema operacional Linux como ambiente de desenvolvimento. No processo de compilação foram utilizados o compilador *GCC 5* e o utilitário *Make*. As principais dependências correspondem aos cabeçalhos e bibliotecas diretamente ligados à implementação de referência do D-Bus. Recursos específicos do *kernel* do Linux como as chamadas `pipe2()` e `memfd_create()` também foram utilizadas.

4.3.2 LX-MCAPI

A LX-MCAPI foi desenvolvida como uma interface simples entre a camada de aplicação e a camada de transporte (Figura 4.2), abstraindo o complexo conjunto de chamadas de função pertencentes à biblioteca de baixo nível do D-Bus em chamadas para operações de sincronização entre processos e envio/recebimento de elementos de mensagem.

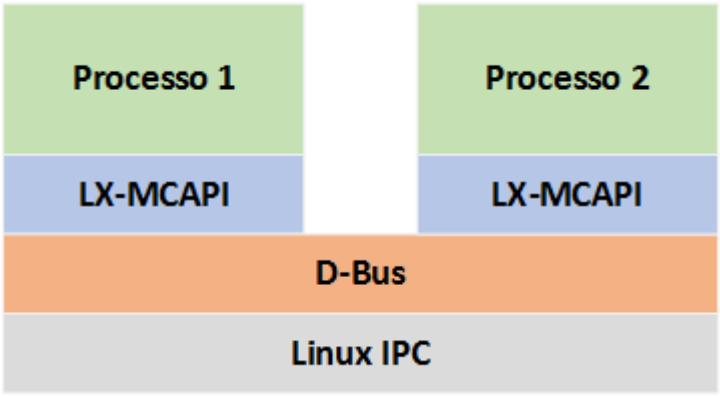


Figura 4.2: LX-MCAPI - Representação em camadas

Assim como citado na seção 4.1.1, a MCAPI define uma hierarquia de entidades comunicantes e elementos de conexão, além de políticas de envio e recebimento de dados. Todas as funções implementadas de conexão, envio e recebimento de elementos de mensagens são síncronas e bloqueantes, traduzindo o contexto da especificação MCAPI para o contexto da BBN do protocolo D-Bus. Atualmente não implementa funções auxiliares de definição e consulta de parâmetros e funções assíncronas. Os detalhes são apresentados nas próximas seções. A Figura 4.3 sintetiza os principais elementos de conexão e de mensagens utilizadas na API.

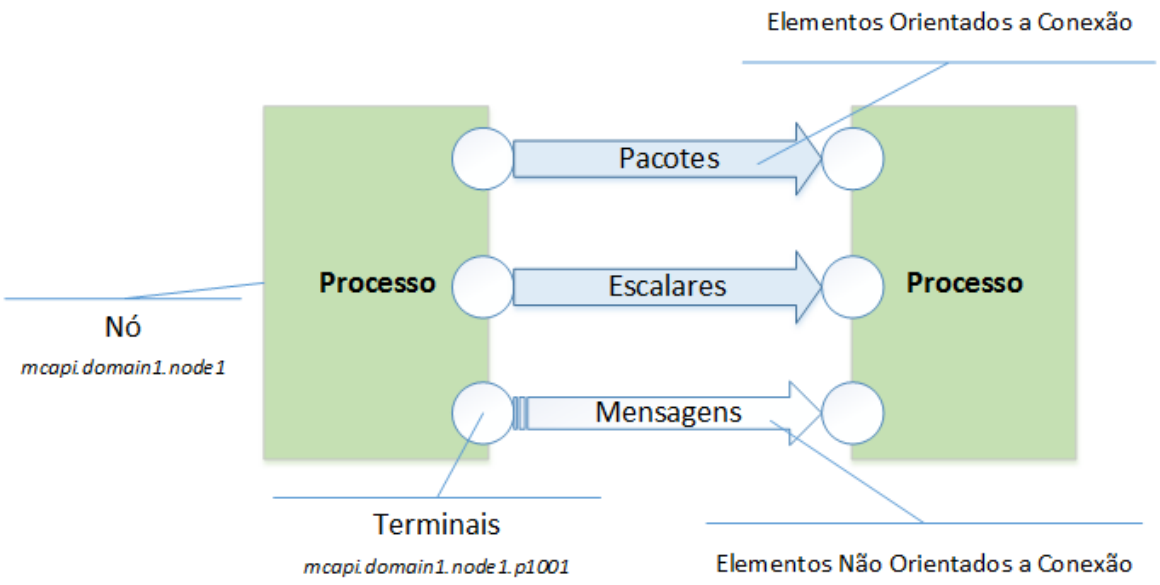


Figura 4.3: LX-MCAPI - Elementos de conexão e de mensagem

Elementos de Conexão e de Mensagem

Criar conexões entre processos exige basicamente dois **elementos de conexão** no ambiente da LX-MCAPI:

- **Nós(*Nodes*)**: Os nós são elementos identificadores de um processo e definem o *nome único* que o identificará perante outros processos comunicantes conectados ao D-Bus.

- **Terminais (*Endpoints*)**: Terminais são endereços criados para envio e recebimento de elementos de mensagem, análogos a portas no domínio TCP/IP. São identificados por números inteiros e geram um endereço de correspondência (*match*) no barramento de mensagens do D-Bus, nomeado como *interface* no contexto da BBN.

Os **elementos de mensagem** que compõem a especificação MCAPI são: Escalares, Pacotes e Mensagens. No contexto da LX-MCAPI, foram delegados objetivos específicos bem como estratégias distintas de transmissão que serão descritas nas seções seguintes.

- **Escalares (*Scalars*)**: Escalares são elementos de mensagem que podem armazenar um inteiro sem sinal de 8, 16, 32 ou 64 bits (respectivamente foram utilizados os tipos padrão `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` da linguagem C).

- **Pacotes (*Packets*)**: Pacotes são elementos genéricos que podem transportar quaisquer tipos de dados, utilizado para envio de, por exemplo, arranjos e *structs* da linguagem C. Foram concebidos para o transporte mais rápido de grandes quantidades de dados (superiores a 1KiB), sem a necessidade de serialização ou tradução para formas textuais como o XML ou JSON.

- **Mensagens (*Messages*)**: Mensagens são cadeias de caracteres (*strings*) de tamanho arbitrário, preferencialmente utilizadas para transmissão de pequenas quantidades de dados em forma textual plana.

A Tabela 4.3 demonstra a correspondência entre as denominações e suas respectivas descrições, além de convenções utilizadas para nomes internos e construção de mensagens.

MCAPI	Elemento D-Bus	Convenções e Descrições (D-Bus)
Node	Bus Name	`mcapi.domain{d}.node{x}`
Port	Interface	`mcapi.domain{d}.node{x}.p{y}`
Messages	Messages	Mensagens com corpo textual (*strings*)
Scalars	Messages	Mensagens com corpo contendo inteiro de 8, 16, 32 ou 64 bits
Packets	Messages	Mensagens com corpo contendo descritor de arquivo ou caminho para acesso ao pacote

Tabela 4.3: Correspondência de denominações: MCAPI x D-Bus

Estratégias de Transmissão de Elementos

Escalares e pacotes são elementos orientados à conexão, ou seja, necessitam de um canal virtual estabelecido para poderem transitar entre processos. Os canais virtuais são ligações ponto-a-ponto criadas entre os processos, permitindo o tráfego unidirecional dos elementos de mensagem. Para que seja possível enviar e receber escalares e pacotes, os processos devem estabelecer dois canais de comunicação, um em cada direção.

Mensagens, por sua vez, são elementos não-orientados à conexão, portanto não há necessidade de canal virtual estabelecido para envio e recebimento, bastando a criação de terminais. As mensagens da LX-MCAPI foram mantidas como mensagens do protocolo D-Bus, para compatibilidade com aplicações externas e como base de comparação para o protocolo de pacotes.

LX-MCAPI v1.0 - *Estável*

A versão 1.0 da LX-MCAPI foi desenvolvida a partir da BBN que acompanha o D-Bus em sua implementação de referência, permitindo maior compatibilidade e portabilidade entre sistemas. A BBN sofre com a falta de documentação clara e exemplos de utilização, mesmo após o amadurecimento de dez anos entre a comunidade de desenvolvimento. A utilização de *wrappers* e da *glib* ajudou na construção de aplicações baseadas no D-Bus mas restringiu a adoção da sua forma pura em linguagem C.

A implementação de referência do D-Bus opera em nível de usuário, necessitando de uma quantidade maior de cópias e trocas de contexto que mecanismos de IPC de baixo nível. É criti-

cada por seu baixo desempenho e implementação pouco eficiente, consumindo muito tempo em tarefas de alocação de memória e transmissão de elementos, quando comparada a mecanismos de IPC convencionais.

A transmissão via mensagens do protocolo D-Bus ocasiona *overhead* excessivo, prejudicando o desempenho de aplicações quando as operações de IPC ocorrem em grandes quantidades. Para contornar este problema, a LX-MCAPI utiliza a estratégia de uma segunda "camada de transporte" para rápido despacho de grandes quantidades de dados. Essa segunda "camada de transporte"utiliza funções de IPC fornecidas pelo *kernel* do Linux para transportar os dados entre processos. Os pacotes são enviados por *pipes* ou copiados diretamente para o espaço de memória de outro processo através da chamada `process_vm_readv()`. O protocolo D-Bus fica responsável por transportar as informações essenciais, como descritores de arquivos e identificadores internos da LX-MCAPI, além de possibilitar a sincronização e estabelecimento dos canais virtuais.

Os *pipes* são estruturas alocadas em memória RAM que permitem o envio de dados através de *bytestreams*. A criação de *pipes* é feita pelas chamadas de sistema `pipe()` ou `pipe2()` e a leitura e escrita respectivamente pelas chamadas `read()` e `write()`. A utilização dos pipes é exclusiva dos escalares e pacotes na LX-MCAPI, sendo que as mensagens são anexadas como conteúdo das mensagens padrão do protocolo D-Bus.

Em um cenário hipotético de comunicação entre dois processos, P1 e P2, onde P1 deseja enviar um pacote de tamanho arbitrário para P2, P1 solicitará a sincronização em modo pacote, o que criará um canal virtual unidirecional entre os dois, permitindo o envio de dados de P1 para P2.

A criação do canal virtual consiste, além de tarefas relacionadas ao gerenciamento de conexão entre os processos, na criação de *pipes* e do compartilhamento dos descritores de arquivo referentes aos mesmos entre os processos (um *pipe* possui um descritor de entrada e um descritor de saída). O processo que solicitou a sincronização se apropria dos descritores de entrada e envia os descritores de saída para o processo com o qual deseja se comunicar.

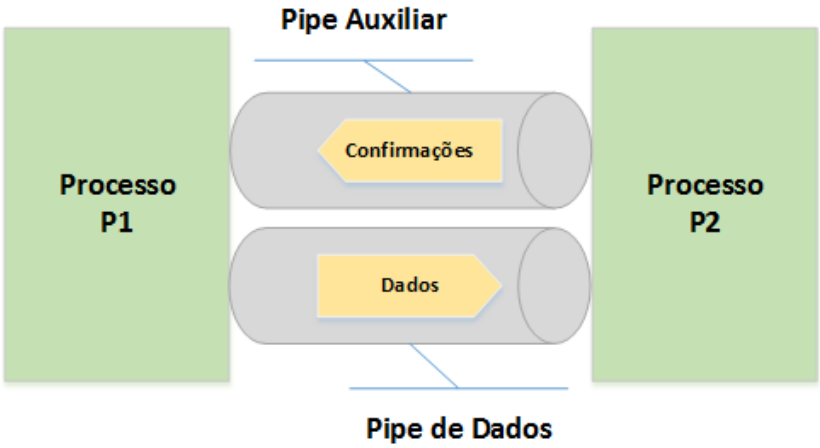


Figura 4.4: LX-MCAPI - Transmissão de Escalares e Pacotes (até 16KiB) (v1.0)

Exemplificando, no caso da transmissão de pacotes, P1 envia um pacote através da chamada `mcapi_pktchan_send()`, os *bytes* do pacote serão escritos no *pipe* pertencente ao canal virtual, estabelecido anteriormente. O processo P2 aguardará o recebimento do pacote e efetuará a leitura do *bytestream* vindo do *pipe* através da chamada `mcapi_pktchan_recv()`. A chamada de função retorna um ponteiro `void*`, assim o tratamento de tipos é de responsabilidade do programador. A Figura 4.4 ilustra a estrutura utilizada para a transferência descrita.

A partir de um limite de 16KiB, determinado através de experimentos, a estratégia de *pipes* é abandonada pois as latências de transferência se tornam muito altas. Pacotes a partir de 16KiB são transferidos através da chamada de sistema `process_vm_readv()`. Juntamente com sua contraparte `process_vm_writev()`), permitem a transferência direta de dados entre espaços de endereçamento, de um processo local para um processo remoto, sem que estes passem pelo espaço do *kernel*. Essa estratégia de transferência é denominada *kernel bypass*. Ambas as chamadas estão disponíveis a partir da versão 3.2 do *kernel* do Linux.

A LX-MCAPI utiliza um *pipe* auxiliar para transferir os dados necessários (endereço, tamanho e PID) do processo local para o remoto e aguarda a confirmação. O processo remoto recebe as informações para transferência de dados, executa `process_vm_readv()`, e ao final do processo envia a confirmação para o processo local. As confirmações são transferidas por um *pipe* auxiliar. A Figura 4.5 ilustra o processo de transferência.

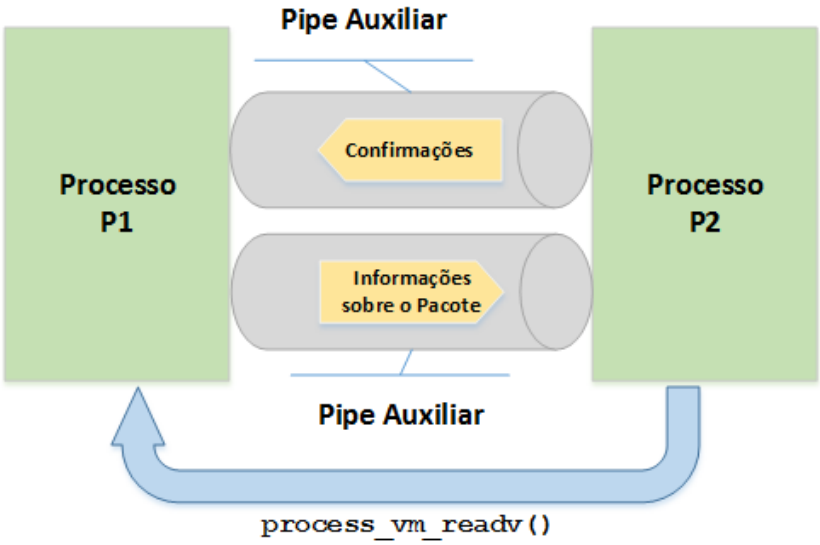


Figura 4.5: LX-MCAPI - Transmissão de Pacotes (a partir de 16KiB) (v1.0)

Todas as políticas de transmissão são executadas de forma transparente ao programador. Em nenhum momento há intervenção do programador para a escolha do método de transmissão. A LX-MCAPI se encarrega de alternar entre os modos de transmissão conforme demanda.

Além das estratégias citadas, LX-MCAPI oferece também funções para compartilhamento de memória através das funções implementadas em LX-MCAPI Zero, que permite a criação e mapeamento de áreas de memória anônimas de maneira transparente, expandindo o conjunto de chamadas da especificação MCAPI. O termo *zero-copy* é comumente utilizado para caracterizar operações onde a CPU não executa tarefas durante a transmissão de um espaço de memória para outro, contudo, no contexto da LX-MCAPI, *zero-copy* se refere especificamente a capacidade de compartilhar dados sem que hajam cópias físicas entre processos.

Basicamente, o mecanismo permite que áreas de memória compartilhada sejam utilizadas pelos processos através da abstração de passagem de mensagens. Foi concebido como parte de um esforço para criação de estratégia de programação que evitam a criação de regiões críticas (*lock-free*) e promovendo uso inteligente de recursos de memória de forma simples e escalável.

Os *buffers* de memória são compartilhados entre os processos através de descritores de arquivos, criados a partir da chamada `memfd_create()`, que cria áreas de memória anônimas, também denominadas arquivos anônimos, apontadas por um descritor de arquivos. Essas áreas de memória são mapeadas através da chamada `mmap()` e utilizadas como *buffers* para dados. As áreas de memória anônimas são endereçadas no diretório `proc` e, por residirem em memória RAM, fornecem vantagens em desempenho quando comparadas à áreas em armazenamento secundário.

Os denominados *buffers zero-copy* são áreas mapeadas que podem ser compartilhadas entre os processos através da abstração de pacotes. Um processo cria um *buffer zero-copy* através da chamada `mcapi_zc_alloc()`, passando como parâmetros um *handle* com informações internas e permitirá o acesso ao conteúdo do *buffer*, e o tamanho em *bytes* desejado do *buffer*. O espaço de memória será criado através de `memfd_create()` e mapeado para o campo *buffer* do *handle*. Um *buffer* é enviado através da chamada `mcapi_pktchan_zc_send()`, que utiliza a mesma infraestrutura do canal virtual de pacotes para transmissão.

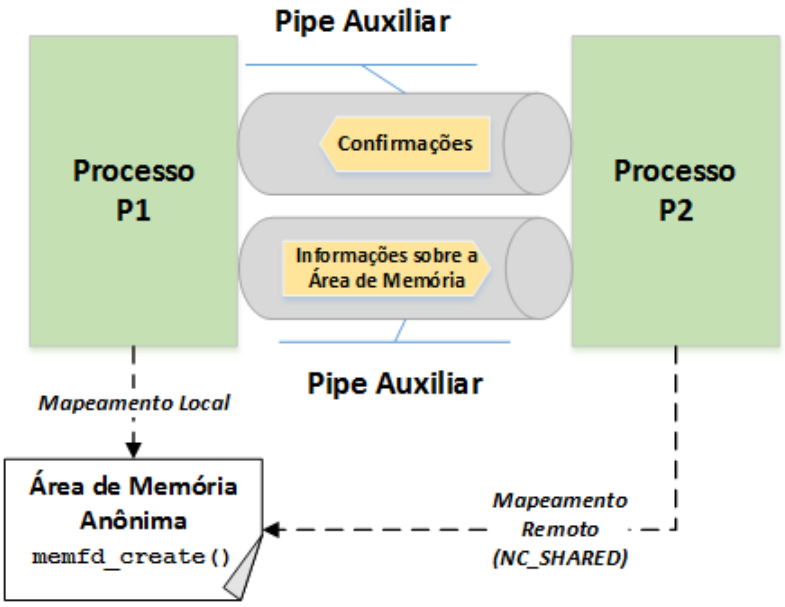


Figura 4.6: LX-MCAPI Zero - Transmissão de Pacotes Zero-Copy (Modo SHARED)

O processo remoto recebe somente o endereço da área de memória do *buffer*. A chamada `mcapi_pktchan_zc_recv()` é responsável por receber o endereço da área de memória e mapeá-la para o espaço do processo local. Dois modos de mapeamento são possíveis: o modo `NC_SHARED` mapeia a área de memória diretamente, permitindo que ambos os processos visualizem quaisquer mudanças no *buffer* em tempo de execução (Figura 4.6); o modo `NC_COPY` efetua uma cópia local do *buffer* recebido para um novo *buffer zero-copy*, preservando o conteúdo original do *buffer* inicial (Figura 4.7).

As principais funções do módulo LX-MCAPI Zero e seus respectivos propósitos são listados na Tabela 4.4. O número de parâmetros é informado entre parênteses.

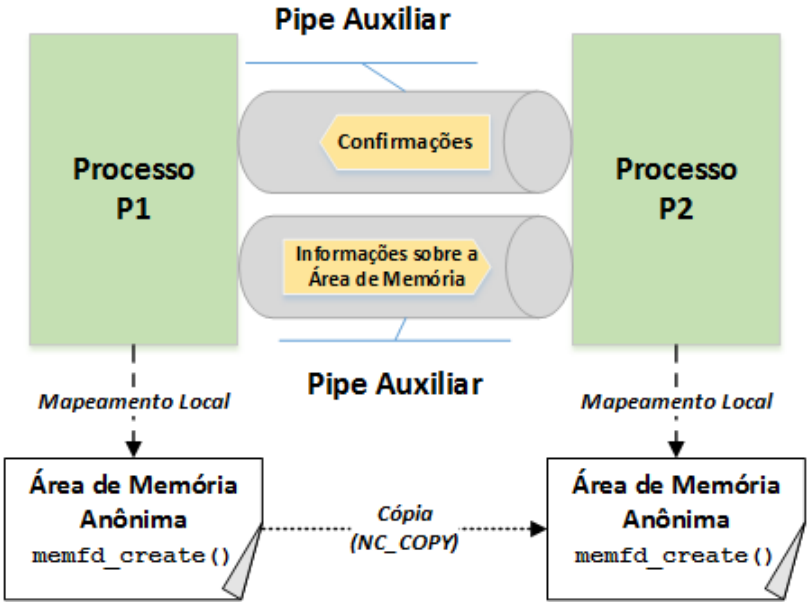


Figura 4.7: LX-MCAPI Zero - Transmissão de Pacotes Zero-Copy (Modo COPY)

Função	Propósito
`mcapi_pktchan_zc_alloc(3)`	Aloca área de memória anônima
`mcapi_pktchan_zc_free(2)`	Libera área de memória anônima
`mcapi_pktchan_zc_send(3)`	Envia um pacote zero-copy
`mcapi_pktchan_zc_recv(5)`	Recebe um pacote zero-copy

Tabela 4.4: Principais funções do módulo LX-MCAPI Zero

4.3.3 LX-MCAPI Patterns

LX-MCAPI Patterns consiste no módulo responsável pela implementação de funções que auxiliam o programador na utilização de padrões de programação para construção de aplicações, inspirado nas implementações da ZeroMQ (seção 3.2.2) e nanomsg (seção 3.2.2).

Padrões, de maneira geral, podem ser definidos como estratégias recorrentes para lidar com determinados problemas. São utilizados amplamente em áreas da computação como arquitetura, aprendizado de linguagem natural, programação orientada-a-objetos e engenharia de software. A programação paralela também contempla padrões de programação consolidados como `map`, `map-reduce`, `nesting` e `fork-join`, amplamente utilizados pelas bibliotecas de paralelismo disponíveis.

A principal ideia foi apresentar padrões de programação na forma de estratégias para desenvolver aplicações paralelas em sistemas *multi-core*, evitando, e possivelmente eliminando,

a necessidade de regiões críticas, comuns em ambientes de memória compartilhada, e que são uma das principais causas para a dificuldade de se obter melhores taxas em ganho de desempenho. As políticas *lock-free* também são adotadas nas estratégias de Memória Transacional, no intuito de facilitar a programação de aplicações concorrentes e paralelas.

Os padrões de programação foram implementados prezando por simplicidade e interoperabilidade. Diferentes padrões podem ser associados para criação de novos padrões e estratégias. A LX-MCAPI Patterns disponibiliza 4 padrões prontos para uso: PAIR, Request/Reply, Publish/Subscriber e Push/Pull. Cada um deles é abordado individualmente nas seções seguintes.

Padrão PAIR

É o padrão de mensagens mais simples, permitindo a criação de um conjunto de terminais pareados, abrindo um canal virtual de comunicação bidirecional entre processos, conectando ambos os terminais. A criação de um canal bidirecional na verdade consiste na criação de dois canais de mesma categoria em sentidos opostos, permitindo que dados fluam de um dado Processo 1 para um Processo 2, e do Processo 2 para o Processo 1.

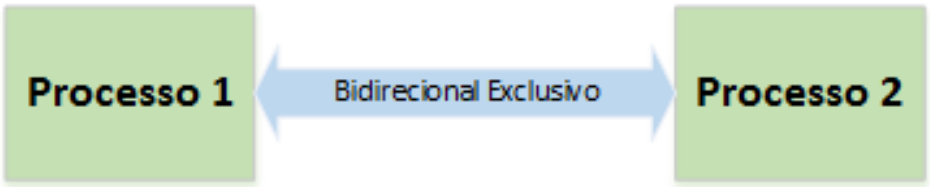


Figura 4.8: Padrão de Mensagens - PAIR

A implementação do padrão PAIR permite a criação de canais bidirecionais de pacotes e escalares (mensagens não necessitam deste tipo de pareamento por serem elementos não-orientados à conexão). As funções do padrão PAIR utilizam diretamente as chamadas da LX-MCAPI para criação, utilização e manutenção dos canais virtuais.

As principais funções do padrão PAIR e seus respectivos propósitos são listados na Tabela 4.5. O número de parâmetros é informado entre parênteses.

Função	Propósito
`mcapi_pair_sclchan_initstruct(8)`	Inicializa informações sobre o canal bidirecional de escalares
`mcapi_pair_sclchan_left(2)`	Abre um canal para envio e aguarda right
`mcapi_pair_sclchan_right(2)`	Abre um canal para recebimento e aguarda left
`mcapi_pair_pktchan_initstruct(8)`	Inicializa informações sobre o canal bidirecional de pacotes
`mcapi_pair_pktchan_left(2)`	Abre um canal para envio e aguarda *right*
`mcapi_pair_pktchan_right(2)`	Abre um canal para recebimento e aguarda *left*. Deve ser executada antes de *left*.

Tabela 4.5: Principais funções do padrão PAIR

Padrão Request/Reply

Comumente conhecido como Cliente/Servidor, o padrão Request/Reply consiste em um modelo onde o processo cliente envia uma requisição de cômputo para o processo servidor, e este responde com o resultado do cômputo.

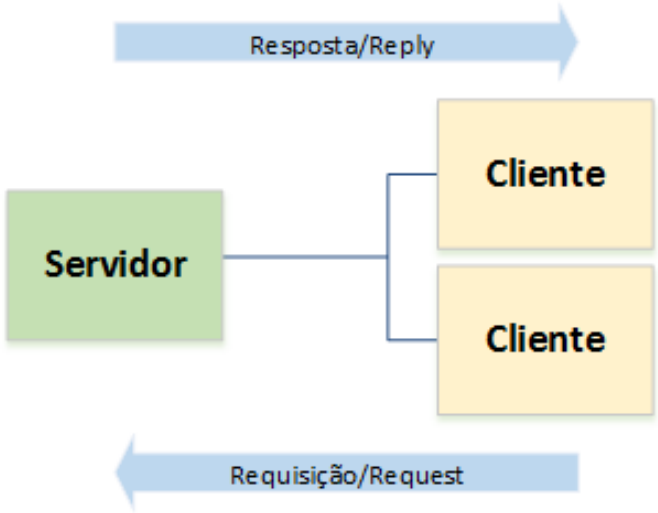


Figura 4.9: Padrão de Mensagens - Request/Reply

A implementação do padrão Request/Reply permite que um processo especifique, na criação do servidor, um ponteiro para a função que será executada ao receber uma requisição de um processo cliente e um *"nome amigável"* para identificar a operação. O processo cliente deve

informar o `node_id` do processo servidor, bem como o *"nome amigável"* da função a ser executada, além dos parâmetros de entrada e o *buffer* que receberá a saída. A Figura 4.9 ilustra o padrão Request/Reply.

As principais funções do padrão Request/Reply e seus respectivos propósitos são listados na Tabela 4.6. O número de parâmetros é informado entre parênteses.

Função	Propósito
`mcapi_rr_initstruct(5)`	Inicializa informações sobre o servidor
`mcapi_rr_create(2)`	Inicializa os elementos e infraestrutura necessários para execução do servidor
`mcapi_rr_dispatch(1)`	Executa um *loop* para aguardar requisições
`mcapi_rr_request(7)`	Executa requisição do cliente a um determinado servidor

Tabela 4.6: Principais funções do padrão Request/Reply

Padrão Publish/Subscribe

Publish/Subscribe consiste em um padrão onde os processos que enviam mensagens, denominados *publishers*, não determinam a quais processos enviarão suas mensagens. Ao invés disso, caracterizam as mensagens publicadas em classes ou tópicos, e os processos *subscribers* expressam o seu interesse por uma ou mais classes de mensagens inscrevendo-se em tópicos, e somente receberão mensagens que são de seu interesse.

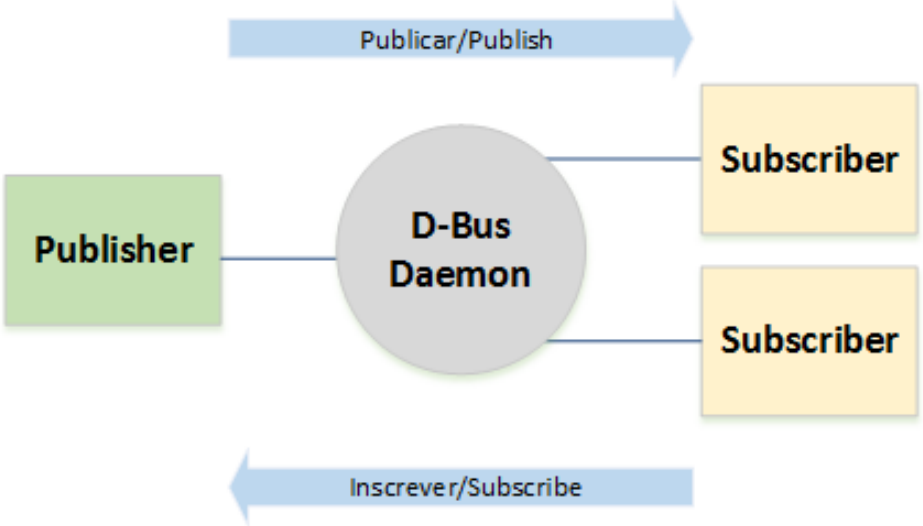


Figura 4.10: Padrão de Mensagens - Publish/Subscribe

É uma estratégia que prioriza o desacoplamento e permite a rápida dispersão de dados entre os processos. Processos *publishers* não tem ciência dos possíveis *subscribers* que monitoram as mensagens transmitidas, e os processos *subscribers* somente monitoram o(s) tópico(s) de interesse. Não há garantia de entrega pois *publishers* e *subscribers* não efetuam comunicação direta. As principais funções do padrão Publish/Subscribe e seus respectivos propósitos são listados na Tabela 4.7. O número de parâmetros é informado entre parênteses.

Função	Propósito
`mcapi_ps_publish(5)`	Publica mensagem sobre um tópico
`mcapi_ps_subscribe(4)`	Recebe mensagem pertencente a um determinado tópico

Tabela 4.7: Principais funções do padrão Publish/Subscribe

O modelo Publish/Subscribe implementado trabalha com a transmissão específica de **mensagens**. A camada de transporte do D-Bus garante que, se um *subscriber* estiver monitorando um certo tipo de *mensagem* especificada por um determinado tópico, este receberá as mensagens categorizadas com o tópico em questão em ordem temporal.

Padrão Push/Pull

O padrão Push/Pull permite distribuir mensagens para múltiplos processos *consumidores*, através de um *pipeline*. É similar ao padrão Request/Reply, porém os resultados não são enviados de volta para o processo *produtor*, mas para outro processo, denominado *coletor*, responsável por captar todos os resultados dos *consumidores* e operar sobre eles. Este padrão de comunicação foi utilizado como base dos estudos realizados no trabalho de (MORON et al., 2014).

Geralmente os processos consumidores são cópias de um mesmo processo, responsáveis por executar um trecho de código que se repete. Um *loop* oneroso em questões de tempo e recursos é um exemplo adequado para aplicação do padrão Push/Pull, permitindo que as tarefas sejam distribuídas entre processos executando de forma paralela em núcleos distintos de um ambiente *multi-core*. A Figura 4.11 ilustra o padrão Push/Pull.

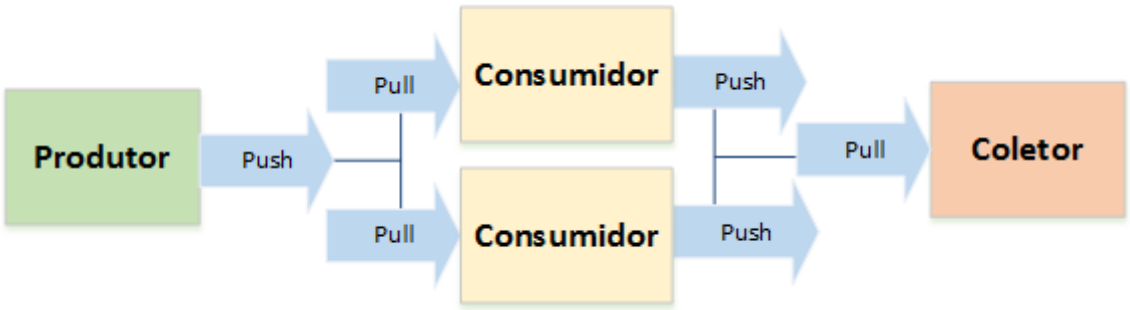


Figura 4.11: Padrão de Mensagens - Push/Pull

As principais funções do padrão Push/Pull e seus respectivos propósitos são listados na Tabela 4.8. O número de parâmetros é informado entre parênteses.

Função	Propósito
`mcapi_pp_broker(4)`	Inicializa um *broker* e aguarda a conexão de processos
`mcapi_pp_request(5)`	Requisita a conexão a um *broker*
`mcapi_pp_push(4)`	Envia um pacote
`mcapi_pp_pull(4)`	Recebe um pacote

Tabela 4.8: Principais funções do padrão Push/Pull

Especificamente na implementação deste padrão, os processos compartilham descritores de arquivos para *pipes* que permitem a distribuição de **pacotes**. Para esse compartilhamento, as chamadas `mcapi_pp_broker()` e `mcapi_pp_request()` efetuam a negociação entre os processos para abrir os canais de comunicação.

Os processos produtor e coletor devem iniciar *brokers* em modo PUSH e PULL respectivamente, através da chamada `mcapi_pp_broker()`. Os processos consumidores devem utilizar a chamada `mcapi_pp_request()` para conectarem-se previamente aos processos produtor e coletor, criando os canais de transmissão de pacotes. Ambas as chamadas retornam *handles* que permitirão as operações de despacho `mcapi_pp_push()` e recebimento `mcapi_pp_pull()`.

A política de despacho de pacotes para processos consumidores segue um simples escalonamento *round-robin*, onde o produtor envia os pacotes e efetua o *load-balancing* de maneira transparente para o programador, alternando o destino de envio entre os processos consumidores registrados. O processo coletor alterna entre os canais dos consumidores conectados, recebendo também em regime *round-robin* os pacotes despachados.

Capítulo 5

RESULTADOS EXPERIMENTAIS

5.1 LX-MCAPI

Os principais experimentos são relacionados à avaliação de velocidades de transmissão dos três tipos de elementos de mensagem. Dois conjuntos de teste foram programados para avaliar o desempenho relacionado a transmissão dos elementos: Taxa de Transferência e Roundtrip. Os resultados dos testes para as arquiteturas x86-64 e ARM são encontrados nas seções 5.1.3 e 5.1.4. O Apêndice A contém os resultados dos experimentos organizados em tabelas para melhor visualização.

5.1.1 Análise de Resultados

Foram feitas análises descritivas e estatísticas dos resultados extraídos via ferramenta de *profiling* gerados pelas execuções dos algoritmos, organizando as informações derivadas dos experimentos para possibilitar o entendimento geral e influência das metodologias abordadas de desenvolvimento de aplicações em termos de desempenho, facilidade de programação e escalabilidade.

O desempenho foi avaliado pela execução de várias instâncias de uma mesma aplicação e estabelecendo o consumo médio de tempo, dado que o ambiente de execução da aplicação pode causar interferências (processos em *background* no sistema operacional como *daemons*, por exemplo, que consomem recursos do processador).

A facilidade de programação é certamente um ponto de interesse, o que gerou um breve comparativo que poderá contribuir no aperfeiçoamento dos métodos de paralelização mais adequados para cada situação.

Quanto a escalabilidade, foi avaliada a capacidade no aumento de *throughput* das aplicações à medida que se disponibilizam mais núcleos de processamento para utilização.

5.1.2 Ferramentas e Recursos

No desenvolvimento deste trabalho, foram utilizados os seguintes recursos e ferramentas (Plataforma x86-64 à esquerda e ARM à direita):

Especificação	**x86-64**	**ARM**
Processador		
CPU	FX-8120	ARM Cortex-A7
# Núcleos	8	4
Frequência	3.1GHz	900MHz
Memória		
RAM	6GiB	1GiB
Cache L1	4 x 64 KB (Instruction) 8 x 16 KB (Data)	32KiB (Instruction) 32KiB (Data)
Cache L2	2 x 4 MiB	256 KiB
Cache L3	8 MiB	-
Plataforma		
Sistema Operacional	Fedora 23	Ubuntu MATE 15.04
Kernel	4.4.5	4.2
Compilador	GCC 5.3	GCC 5.2

Tabela 5.1: Ferramentas e Recursos Utilizados

5.1.3 Taxa de Transferência (TR)

Os testes de Taxa de Transferência, ou TR (*Transfer Rate*), estabelecem uma determinada quantidade de dados a ser transmitida entre dois processos. Essa quantidade em *bytes* é então dividida pelo tempo total da operação, gerando uma estimativa da taxa de transferência na qual os dados foram transmitidos naquele intervalo de tempo. Os processos foram atribuídos à núcleos distintos do processador utilizando a chamada `sched_setaffinity()`. A Figura 5.1 ilustra o teste de Taxa de Transferência.

Os testes foram executados para cada um dos elementos de mensagem, repetindo dez mil vezes cada operação - número mínimo para atenuar ruídos, determinado empiricamente durante

o processo de desenvolvimento dos algoritmos de teste - e avaliando a média de tempo para cada operação de envio/recebimento. Os resultados foram obtidos a partir do *profiling* utilizando a ferramenta TAU (*Tuning and Analysis Utilities*).

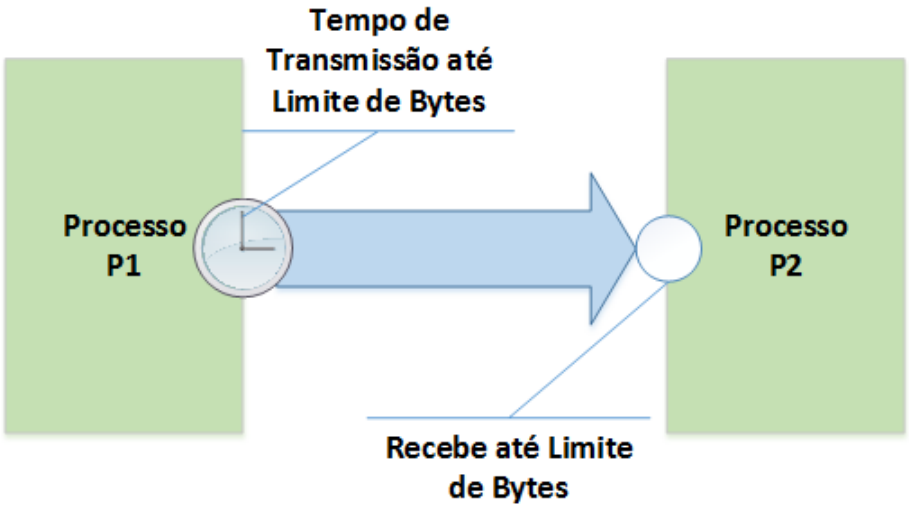


Figura 5.1: Teste de Taxa de Transferência

O gráfico da Figura 5.2 mostra os resultados dos testes TR para escalares de 8, 16, 32 e 64 bits (representados respectivamente como 1, 2, 4 e 8 bytes) nas plataformas x86-64 e ARM. Os resultados demonstram que para envio de escalares, a taxa de transferência não excede 600KB/s na plataforma x86-64 e 140KB/s na plataforma ARM, gradualmente aumentando conforme o tamanho do escalar aumenta de 1 a 8 bytes.

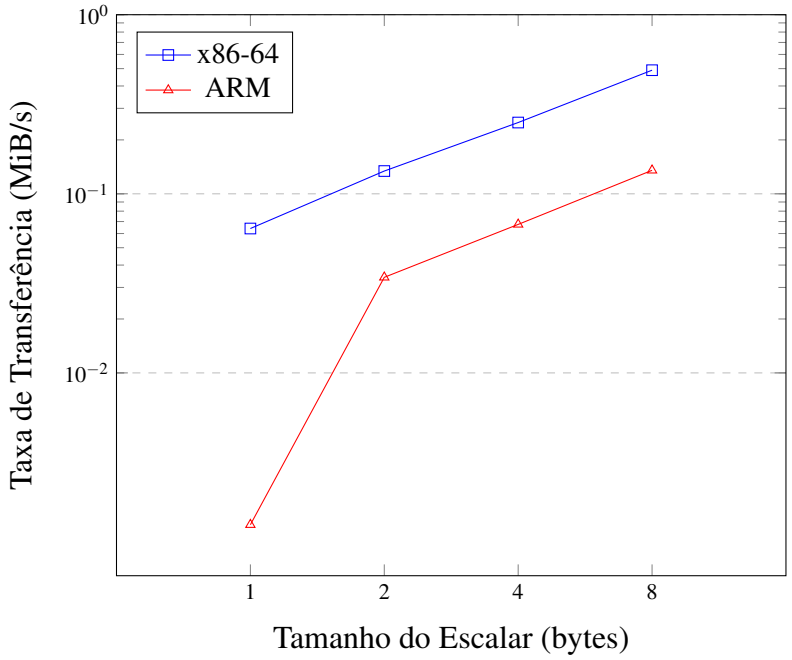


Figura 5.2: Taxa de Transferência para Escalares (ARM e x86-64)

As estatísticas visualizadas na Figura 5.3 demonstram os resultados dos testes TR para pacotes, variando-se o tamanho de 2 bytes a 1MiB. O limite máximo de 1MiB foi estabelecido por ser o máximo estabelecido pelo cabeçalho `limits.h` para o tamanho do `pipe`. Pode-se visualizar a transição da estratégia de *pipes* para a cópia *kernel bypass* na Figura 5.3 quando há um aumento na taxa de transferência entre as estatísticas dos pacotes de 16KB e 32KB.

Os mesmos testes foram efetuados na plataforma ARM. A tendência observada é similar em ambas as plataformas, com aumento progressivo da taxa de transferência conforme o tamanho dos pacotes aumenta.

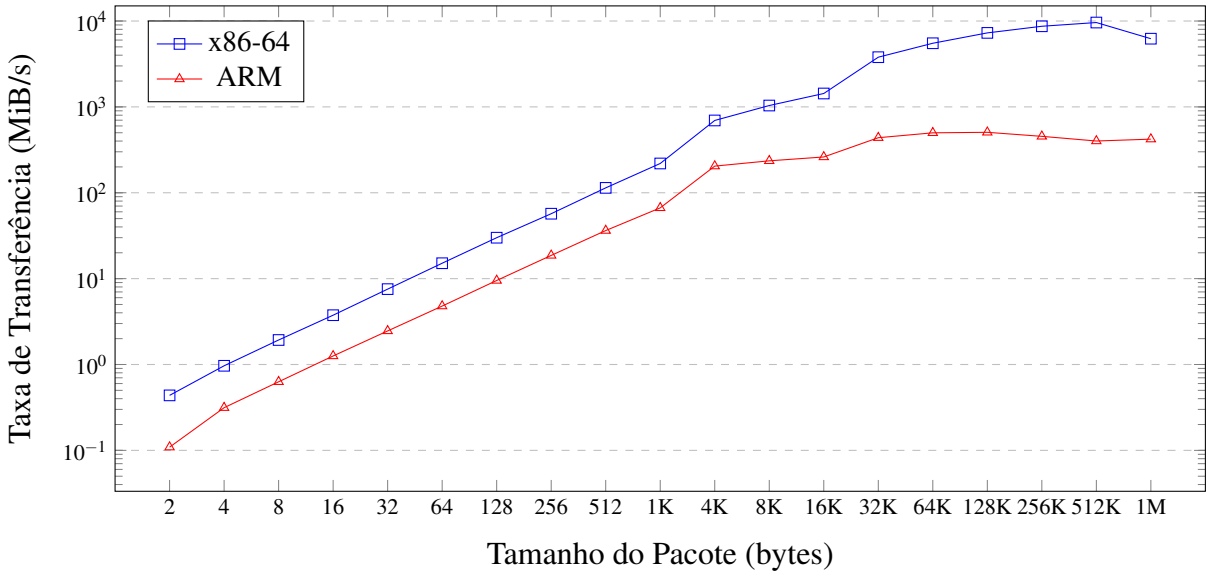


Figura 5.3: Taxa de Transferência para Pacotes (ARM e x86-64)

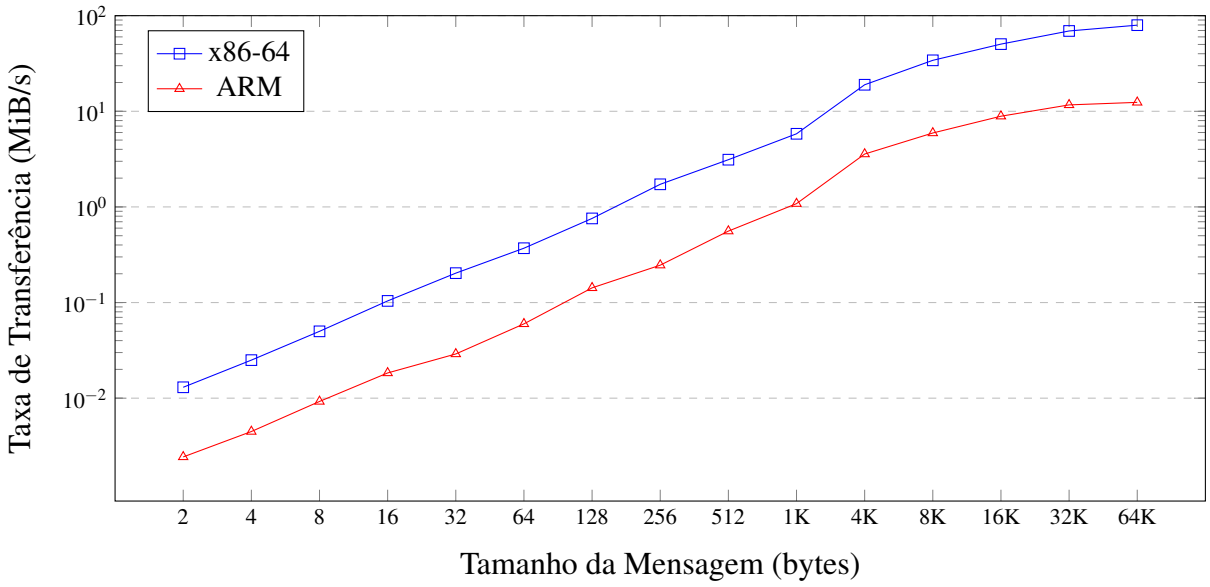


Figura 5.4: Taxa de Transferência para Mensagens (ARM e x86-64)

As mensagens foram testadas variando-se o tamanho de 2 bytes até 64KiB. O limite de 64KiB foi estabelecido pois mensagens acima deste tamanho passaram a consumir tempo em forma exponencial e inviabilizariam o funcionamento de aplicações que buscam por melhora de desempenho. O gráfico na Figura 5.4 demonstra os resultados obtidos nas plataformas x86-64 e ARM. O comportamento das curvas é semelhante, dadas as proporções das medidas de Taxa de Transferência para cada plataforma.

5.1.4 Latência Roundtrip (RTT)

Testes de RTT (*RoundTrip Time*) consistem na avaliação da latência de comunicação bidirecional entre dois processos. Cada um dos processos comunicantes envia e recebe um mesmo elemento de mensagem à cada operação, alternando a ordem das operações. O tempo total, somando-se o tempo de envio e o tempo de recebimento do elemento em questão, caracteriza uma operação de **roundtrip**. A Figura 5.5 ilustra o teste de RTT.

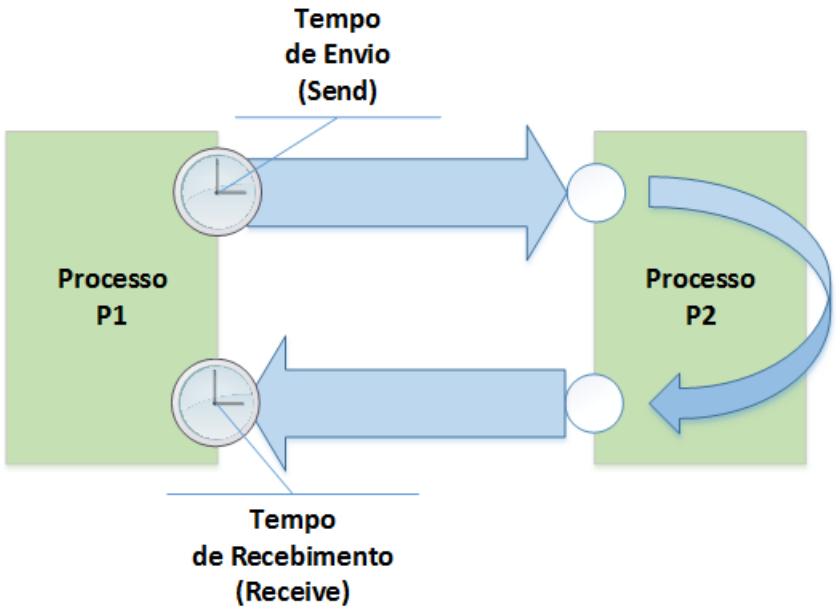


Figura 5.5: Teste de Latência Roundtrip

O tempo de *roundtrip* para escalares de 8, 16, 32 e 64 bits resultou em valores entre 14 e 19μs na plataforma x86-64 e 74 a 78 μs na plataforma ARM. Os resultados para cada plataforma estão listados na Figura 5.6.

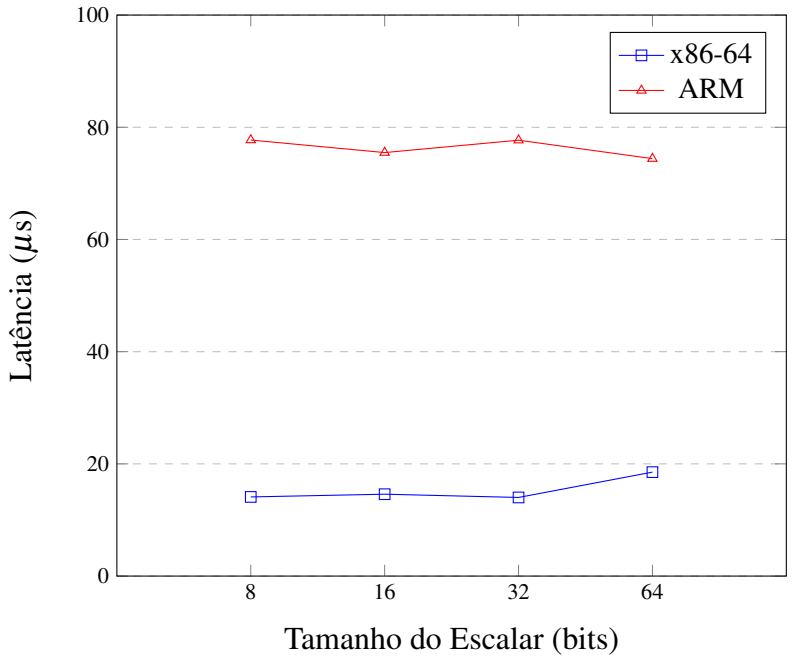


Figura 5.6: Tempos de Roundtrip para Escalares (ARM e x86-64)

A estratégia de transmissão de pacotes manteve um *roundtrip* de 6μs a 8μs na plataforma x86-64 e de 22 a 26μs na plataforma ARM, até o tamanho de 1KiB. A partir de 1KiB, o *roundtrip* passa a aumentar de forma próxima a linear, aumentando proporcionalmente conforme o aumento do tamanho dos pacotes. Os resultados para as plataformas x86-64 e ARM estão representados na Figura 5.7.

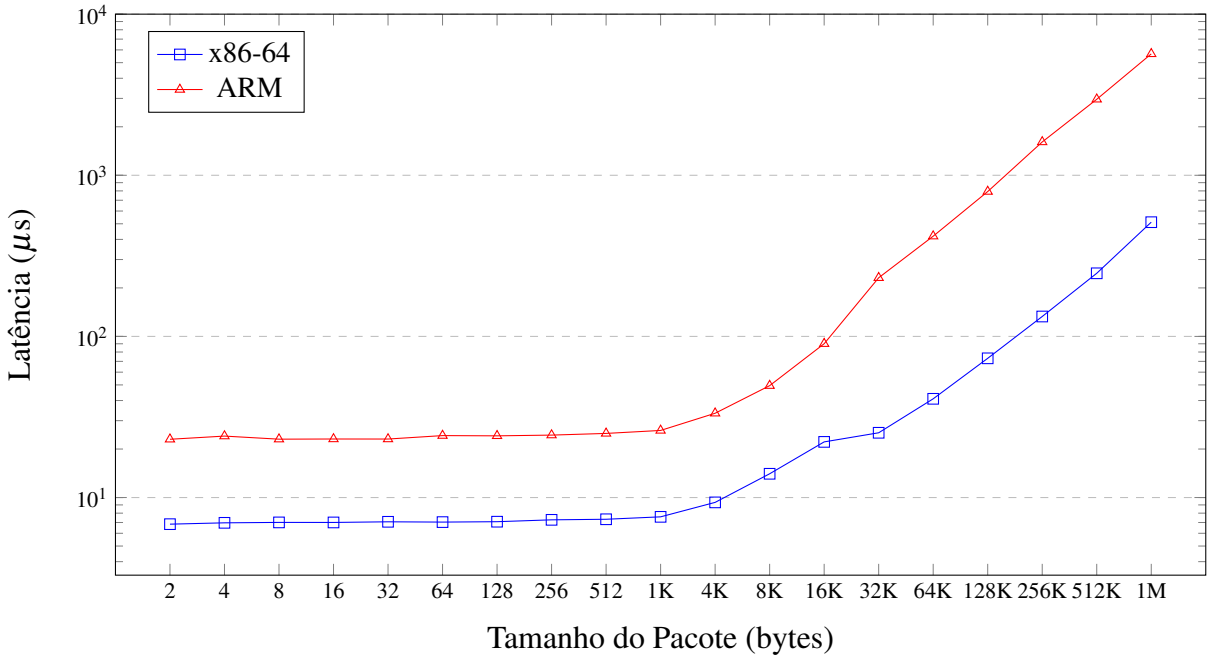


Figura 5.7: Tempos de Roundtrip para Pacotes (ARM e x86-64)

As mensagens geraram tempo de *roundtrip* entre 190 a 265μs na plataforma x86-64 e entre 895 a 1290μs na plataforma ARM, até 1KiB. A partir de 1KiB o crescimento no tempo de *roundtrip* teve aumento proporcional conforme o aumento do tamanho dos pacotes. Os resultados para as plataformas de teste estão listados na Figura 5.8.

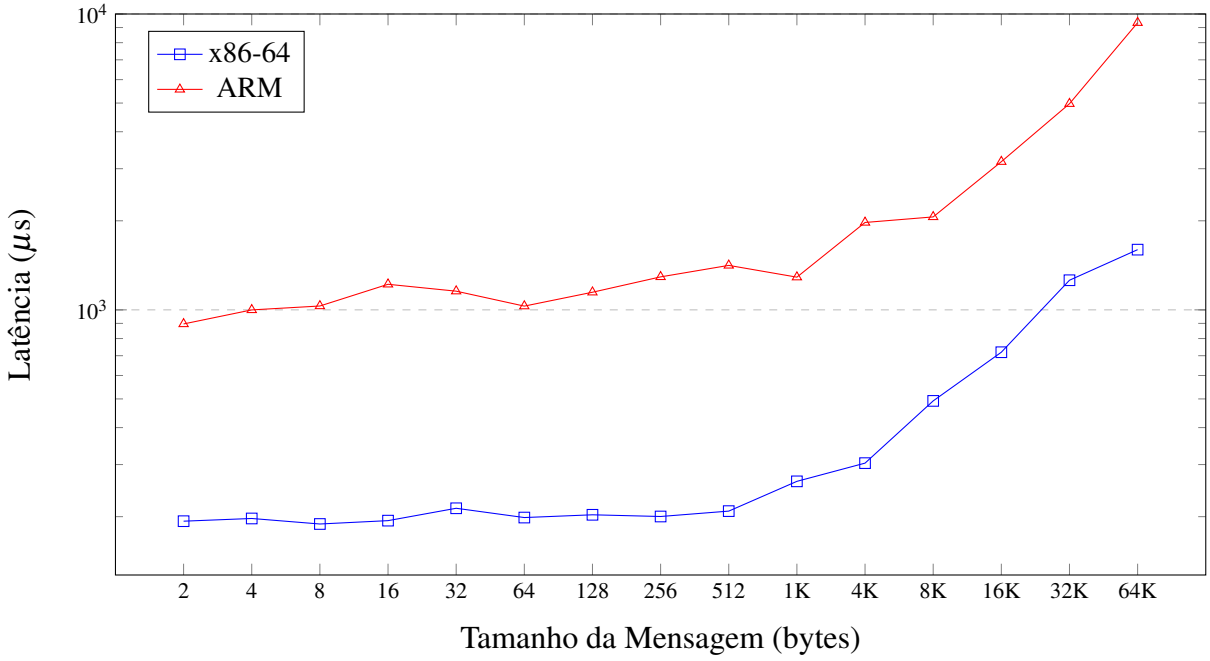


Figura 5.8: Tempos de Roundtrip para Mensagens (Plataforma x86-64)

5.1.5 Pacotes vs. Mensagens

Pode-se verificar a partir da análise dos resultados apresentados anteriormente que o protocolo D-Bus insere *overhead* elevado na transmissão de mensagens, principalmente quando comparado à estratégia empregada pelos pacotes na LX-MCAPI, que mantém um crescimento relativamente contido e com latências de transmissão sempre inferiores às mensagens.

A inserção de *overhead* ocorre pois o D-Bus não efetua simplesmente a passagem do *payload* desejado para outro espaço de endereçamento ponto-a-ponto. O processo de serialização dos dados, inserção de cabeçalho, detecção do processo destinatário por meio do serviço de nomes e a latência da fila de mensagens do *daemon* explicam o tempo de processamento superior encontrado nos resultados.

Pode-se visualizar comparativamente na Figura 5.9 o comportamento da Taxa de Transferência de pacotes e mensagens nas plataformas x86-64 e ARM.

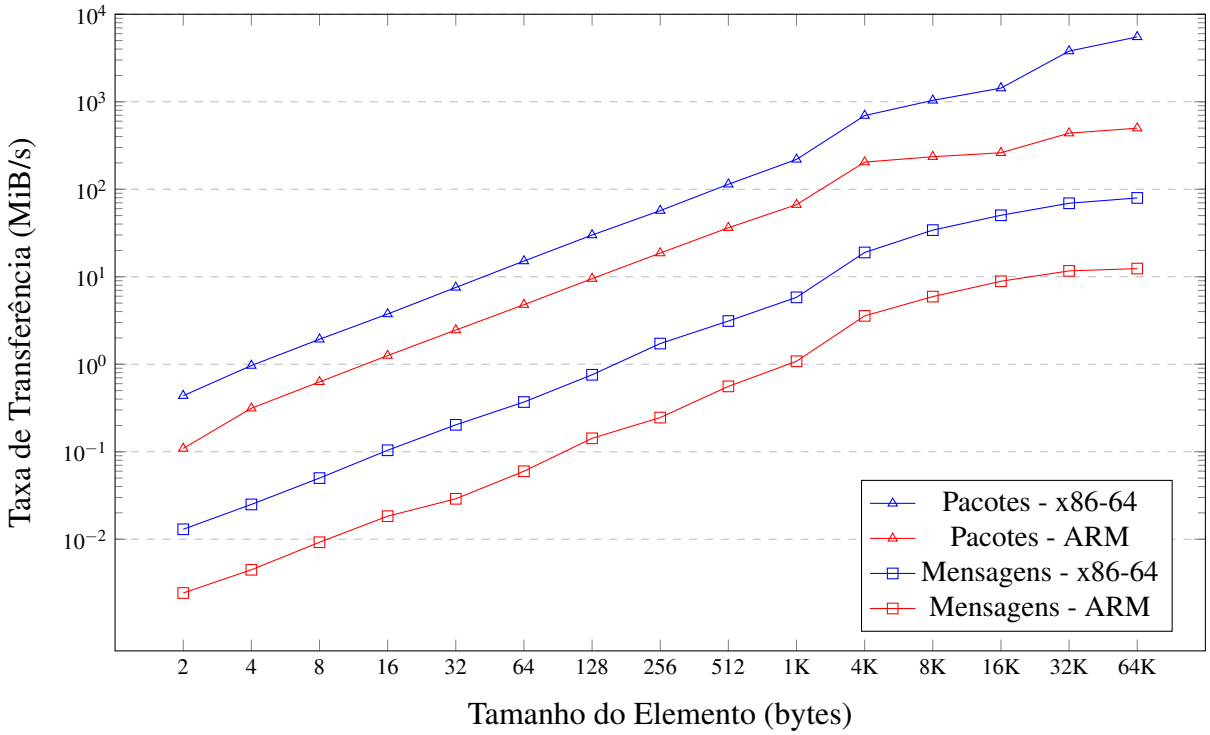


Figura 5.9: Pacotes x Mensagens (ARM e x86-64)

Os gráficos indicam que a estratégia dos pacotes evita essa inserção de *overhead* excessivo e é uma alternativa para transmissão de *payloads*. O protocolo D-Bus é utilizado somente nas funções de registro, sincronização e criação de canais de comunicação entre processos, deixando a transmissão do *payload* a cargo das utilidades e estratégias de transferência implementadas na LX-MCAPI.

5.1.6 LX-MCAPI vs. MPICH2

De modo a comparar os resultados gerados pela LX-MCAPI com uma ferramenta consolidada, escolheu-se a implementação de alto desempenho MPICH2, uma instância da MPI, para efetuar tal análise.

Visto que as estratégias de envio e recebimento de mensagens e escalares da LX-MCAPI não ofereceram desempenho adequado ou suficiente, as comparações foram feitas somente com a implementação de pacotes, que obtiveram o melhor desempenho dentre os três elementos disponíveis. As chamadas da MPICH2 utilizadas são síncronas (`MPI_Send()` e `MPI_Recv()`).

As Figuras 5.10 e 5.11 demonstram o comparativo entre os pacotes da LX-MCAPI e de mensagens da MPICH2 nas arquiteturas x86-64 e ARM, respectivamente para testes de TR e RTT.

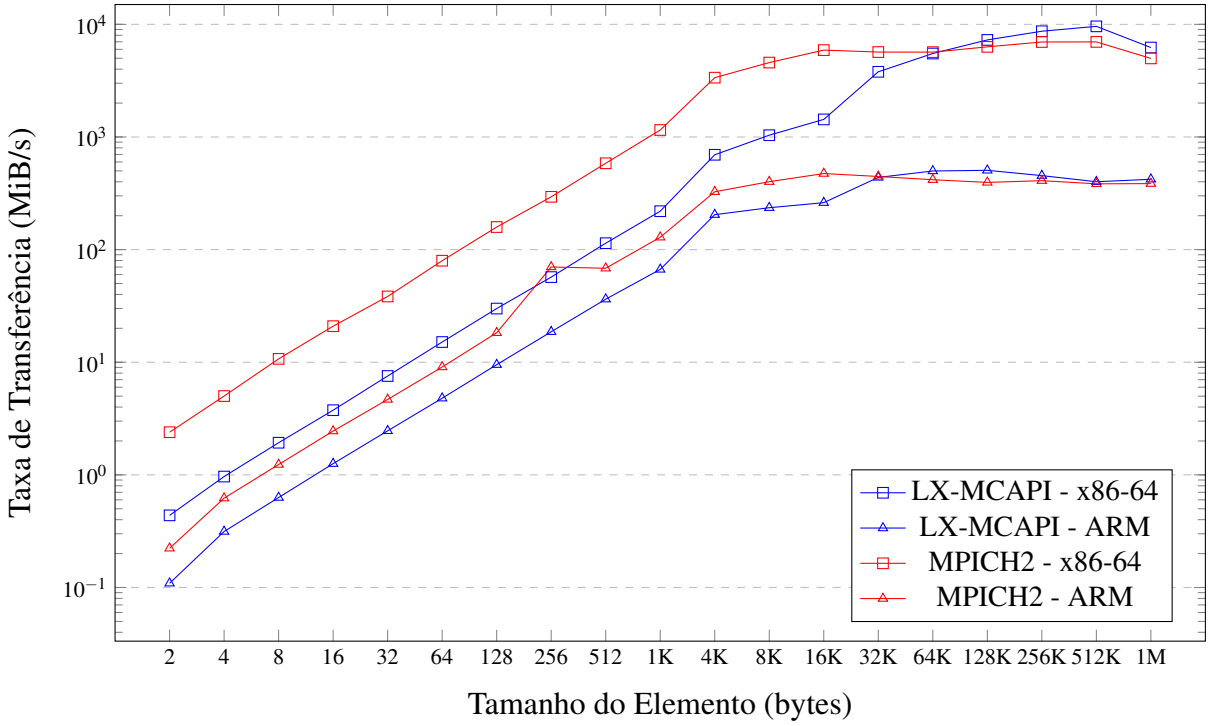


Figura 5.10: Taxa de Transferência - Pacotes (LX-MCAPI) x MPI (ARM e x86-64)

Podemos visualizar em ambos os casos que a LX-MCAPI tem desempenho razoável quando comparada a MPICH2, mantendo crescimento estável até elementos de 32KiB.

A partir de 32KiB, o desempenho da LX-MCAPI se aproxima ao da MPICH2 devido a estratégia de cópia *kernel bypass*, evitando cópias em espaço de *kernel*, migrando os dados diretamente entre espaços de endereçamento de processos. A partir de pacotes de 64KB, a taxa de transferência da LX-MCAPI supera a MPICH2 em alguns MiB/s.

A plataforma ARM apresentou comportamento similar ao da plataforma x86-64. Até elementos de 16KiB as latências da MPICH2 se mantiveram em média 75% mais baixas que as da LX-MCAPI. A partir de 16KiB, o comportamento da LX-MCAPI se aproximou da MPICH2, e demonstrou latências até 25% mais baixas que a MPICH2 em alguns casos. Comportamento semelhante foi observado para as taxas de transferência.

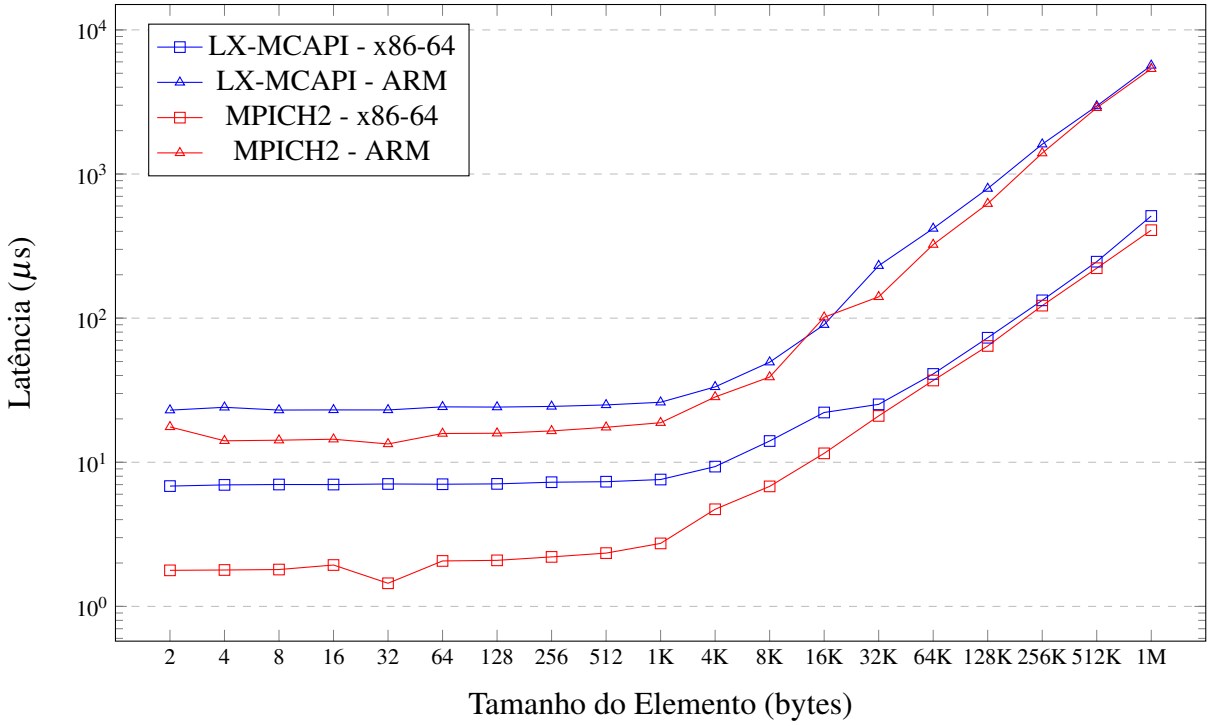


Figura 5.11: RTT - Pacotes (LX-MCAPI) x MPI (ARM e x86-64)

5.1.7 LX-MCAPI vs. nanomsg

De maneira similar aos testes comparativos entre LX-MCAPI e MPICH2, foram executados testes comparativos entre LX-MCAPI e nanomsg (seção 3.2.2), no intuito de avaliar TR e RTT de ambas as bibliotecas. As chamadas utilizadas são síncronas e bloqueantes (`nn_send()` e `nn_recv()`). Os resultados são apresentados nas Figuras 5.12 e 5.13.

Os pacotes da LX-MCAPI mantém TR aproximadamente 100% maior que as mensagens da nanomsg para *payloads* de até 1KiB. A partir de 1KiB os pacotes da LX-MCAPI passam a ter TR progressivamente superior as da nanomsg, chegando a até 600% maior para 1MiB.

No caso da plataforma ARM, os testes de TR indicam que a LX-MCAPI teve melhor desempenho geral com relação a nanomsg. Até 16KiB, as latências apresentadas pela nanomsg eram aproximadamente 60% maiores que as da LX-MCAPI. A partir de elementos maiores que 16KiB, as latências da nanomsg se tornavam em média 150% maiores que as apresentadas pela LX-MCAPI.

Em relação aos testes de RTT na plataforma ARM, as latências apresentadas pela nanomsg são de uma a duas ordens de grandeza maiores que as apresentadas pela LX-MCAPI.

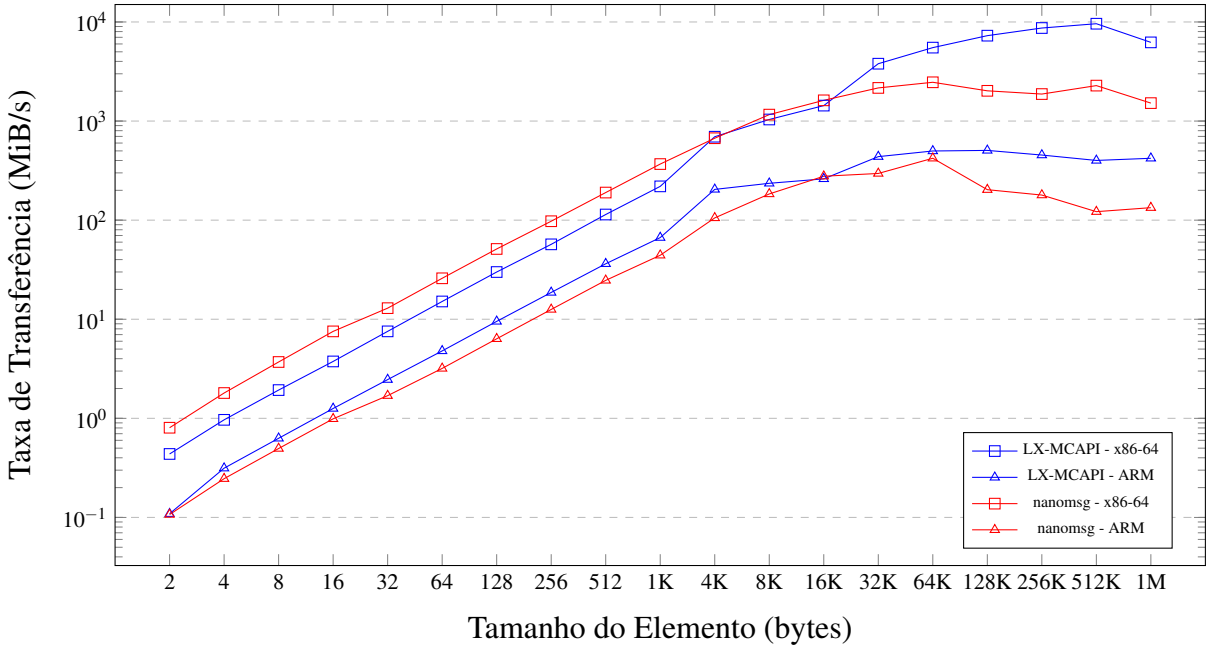


Figura 5.12: Taxa de Transferência - Pacotes (LX-MCAPI) x nanomsg (ARM e x86-64)

Apesar de apresentar Taxa de Transferência maior em testes de TR, os testes de RTT revelam que ao estressar o mecanismo da nanomsg com envios e recebimentos seguidos de pacotes, a latência de transmissão da LX-MCAPI é até 10 vezes menor que a gerada pela nanomsg dependendo do tamanho do elemento de mensagem.

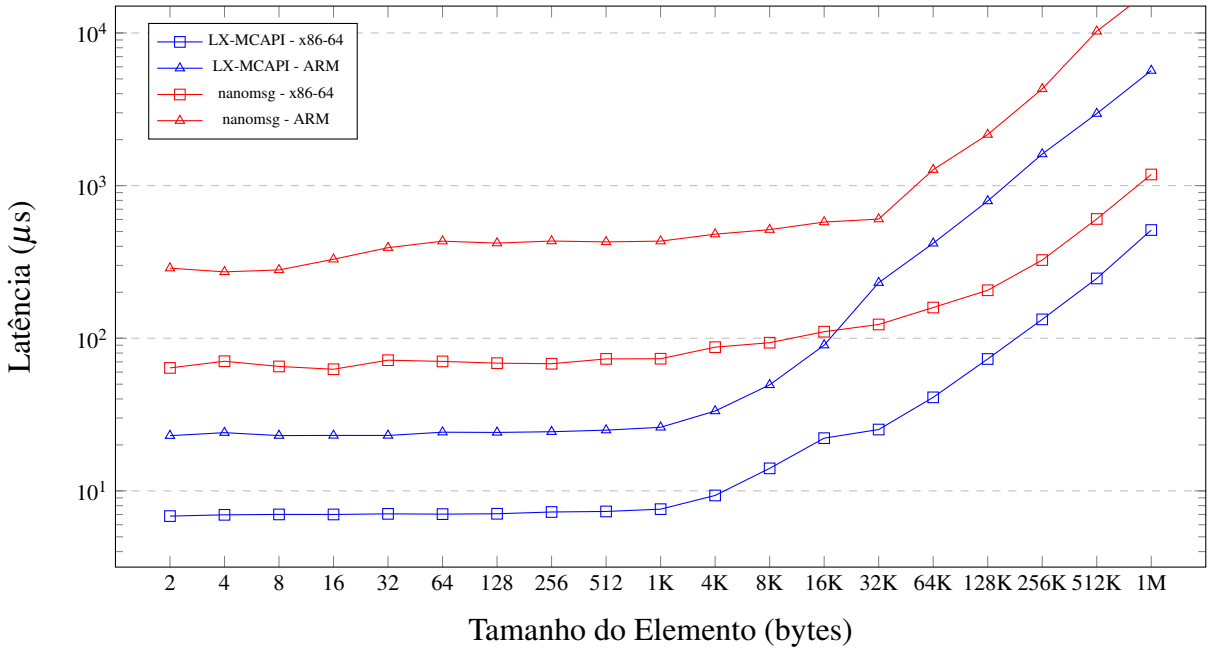


Figura 5.13: RTT - Pacotes (LX-MCAPI) x nanomsg (ARM e x86-64)

5.1.8 Resultados Obtidos

As pesquisas e experimentos realizados durante o presente trabalho resultaram no desenvolvimento da biblioteca LX-MCAPI, instância da especificação MCAPI para bibliotecas de comunicação voltadas a sistemas multi-core. Associando o protocolo D-Bus, amplamente utilizado em distribuições Linux como mecanismo de comunicação para aplicações *desktop*, e utilidades do *kernel* do Linux para comunicação entre processos, LX-MCAPI oferece uma API simplificada para passagem de mensagens e compartilhamento de memória, ocultando estruturas de código complexas do ponto de vista do programador.

Os resultados gerados utilizando a infraestrutura oferecida pela LX-MCAPI permitiram a constatação de que a passagem de mensagem é uma estratégia viável de paralelização, obtendo desempenho similar aos do modelo de memória compartilhada, proporcionando uma curva de aprendizado reduzida por utilizar um conjunto pequeno de funções relativamente simples quando comparadas a outras APIs e linguagens disponíveis.

5.2 LX-MCAPI Patterns

No intuito de avaliar e ilustrar cada um dos padrões de programação implementados em LX-MCAPI Patterns, alguns exemplos foram criados, podendo servir como base para criação de aplicações mais complexas.

5.2.1 Padrão Pair

O exemplo para o padrão PAIR consiste em dois processos que enviam e recebem pacotes de tamanho 1000 bytes, representados por *structs* que encapsulam um vetor de inteiros de 256 posições (4 bytes por posição). A Figura 5.14 ilustra a aplicação.

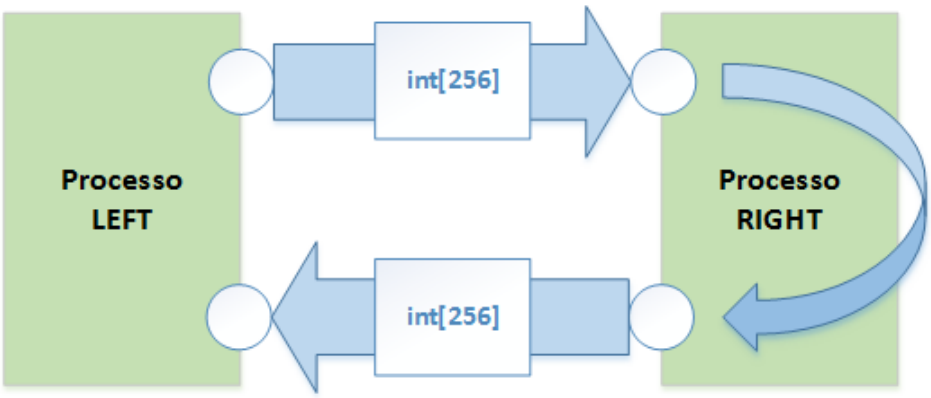


Figura 5.14: LX-MCAPI Patterns - Exemplo PAIR

Ambos os processos devem iniciar uma *struct* do tipo `mcapi_pair_pktchan_st` utilizando a chamada `mcapi_pair_pktchan_initstruct()` que recebe como parâmetros a própria *struct*, o `node_id local`, `node_id` do processo remoto, a porta de transmissão local, a porta de recebimento local, a porta de transmissão remota e a porta de recebimento remota.

O processo Right (Código 5.1) executa `mcapi_pair_pktchan_right()` aguardando a conexão do processo Left (Código 5.2) através da chamada `mcapi_pair_pktchan_left()`. A conexão entre os processos cria um canal bidirecional de pacotes e as chamadas geram dois *handles*, um para envio e um para recebimento de pacotes. A transmissão de pacotes é executada através das chamadas de envio e recebimento da LX-MCAPI, `mcapi_pktchan_send()` e `mcapi_pktchan_recv()`, respectivamente. Estruturas e chamadas equivalentes para escalares também estão disponíveis no módulo.

```
#include "lxmcapi.h"

typedef struct {
```

```
  int ar[256];
} packet1;

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  mcapi_pair_pktchan_st pktst;

  mcapi_initialize(1, 2, NULL, NULL, NULL, &mcapi_status);

  mcapi_pair_pktchan_initstruct(&pktst, 2003, 2004, 1, 1, 1003, 1004, &
   mcapi_status);

  mcapi_pair_pktchan_right(&pktst, &mcapi_status);

  int n_op = 10000;
  int len;

  packet1 p1;

  for(int i = 0; i < n_op; i++){
    mcapi_pktchan_send(pktst.send_handle, &p1, sizeof(p1), err);
    mcapi_pktchan_recv(pktst.recv_handle, (void*)&p1, sizeof(p1), &len, err
   );
  }
  mcapi_finalize(&mcapi_status);

  return 0;
}
```

Código 5.1: Exemplo PAIR - Processo *Right*

```
#include "lxmcapi.h"

typedef struct {
  int ar[256];
} packet1;

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  mcapi_pair_pktchan_st pktst;

  mcapi_initialize_node(1, 1, NULL, NULL, NULL, &mcapi_status);
```

```
  mcapi_pair_pktchan_initstruct(&pktst, 1003, 1004, 1, 2, 2003, 2004, &
    mcapi_status);

  mcapi_pair_pktchan_left(&pktst, &mcapi_status);

  int n_op = 10000;
  int len = 256 * sizeof(int);

  packet1 p1;

  for(int i = 0; i < n_op; i++){
    mcapi_pktchan_recv(pktst.recv_handle, (void*)&p1, sizeof(p1), &len, err
    );
    mcapi_pktchan_send(pktst.send_handle, &p1, sizeof(p1), err);
  }
  mcapi_finalize(&mcapi_status);
  return 0;
}
```

Código 5.2: Exemplo PAIR - Processo *Left*

5.2.2 Padrão Request/Reply

O exemplo para o padrão Request/Reply consiste em um simples servidor que efetua a adição de três inteiros e um cliente que solicita tal adição, passando como parâmetro três números inteiros, correspondentes à hora atual, e um *buffer* de memória para receber o resultado. O servidor responde com a soma dos três números. A Figura 5.15 ilustra a aplicação.

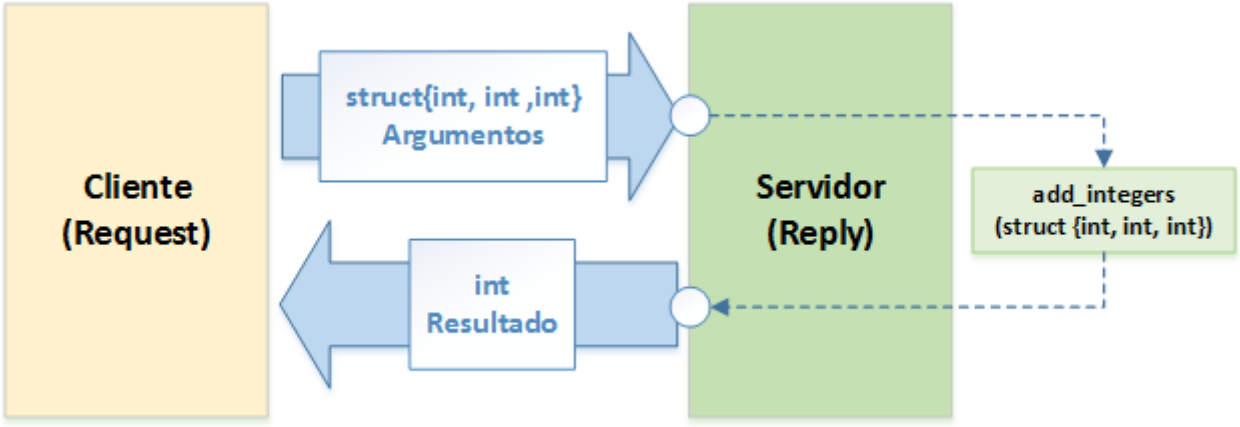


Figura 5.15: LX-MCAPI Patterns - Exemplo Request/Reply

O processo servidor (Código 5.3) inicializa uma *struct* do tipo `mcapi_server_data` através da chamada `mcapi_rr_initstruct()` contendo os dados para criação do servidor. Essa chamada recebe um "nome amigável"para que o servidor filtre as requisições vindas do cliente, um ponteiro para a função que será executada ao receber uma requisição do cliente e o tamanho do *buffer* de retorno da função especificada. No exemplo, a função `add_integers()` retorna um valor inteiro, portanto o último parâmetro de `mcapi_rr_initstruct()` é `sizeof(int)`.

Após a inicialização do processo, `mcapi_rr_create()` recebe a *struct* do passo anterior e cria a interface que monitorará as requisições. A chamada `mcapi_rr_dispatch()` corresponde à um *loop* de espera que manterá o monitoramento de requisições. Esta chamada pode ser envolvida por um outro *loop* - *while()* por exemplo - com uma condição de parada baseada em *signals* ou outro evento externo. Especificamente, no exemplo desta seção, o processo aguarda indefinidamente até que o mesmo seja encerrado por meios externos.

```
#include "lxmcapi.h"
typedef struct{
  int arg1;
  int arg2;
  int arg3;
} function_args;

void* add_integers(void* args)
{
  function_args* fa = args;
  printf("Adding %d %d %d\n", fa->arg1, fa->arg2, fa->arg3);
  return fa->arg1 + fa->arg2 + fa->arg3;
}

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  mcapi_server_data sd;
  sd = mcapi_rr_initstruct("add_integers", &add_integers, sizeof(int), &
    mcapi_status);
  mcapi_initialize(1, 1, NULL, NULL, NULL, &mcapi_status);
  mcapi_rr_create(&sd, &mcapi_status);
  mcapi_rr_dispatch(&mcapi_status);
  return 0;
}
```

Código 5.3: Exemplo Request/Reply - Servidor

O processo cliente (Código 5.4) inicializa uma *struct* com três inteiros, que será passada como parâmetro para o servidor. A função `add_integers()` espera a passagem de um *buffer* do tipo `function_args`. Após solicitar a inicialização do processo, o cliente requisita a operação através de `mcapi_rr_request()`, informando o `node_id` do servidor, o nome amigável da função desejada, o *buffer* de entrada e seu respectivo tamanho, e por fim o *buffer* que receberá o retorno do servidor.

```
#include "lxmcapi.h"

typedef struct {
  int arg1;
  int arg2;
  int arg3;
} function_args;

int main(int argc, char** argv)
{
  struct tm *cur_time;
  time_t cur_time_t;
  mcapi_status_t mcapi_status;

  cur_time_t = time(NULL);
  cur_time = localtime(&cur_time_t);

  function_args fa;
  fa.arg1 = cur_time->tm_hour;
  fa.arg2 = cur_time->tm_min;
  fa.arg3 = cur_time->tm_sec;

  void* buffer;
  void* fa_p = &fa;

  mcapi_initialize(1, 2, NULL, NULL, NULL, &mcapi_status);

  mcapi_rr_request(1, 1,"add_integers",fa_p, sizeof(function_args),&buffer,
    &mcapi_status);

  printf("Result of: add_integers (%d, %d, %d) is %d\n",
        fa.arg1, fa.arg2, fa.arg3, buffer);
  return 0;
}
```

Código 5.4: Exemplo Request/Reply - Cliente

5.2.3 Padrão Publish/Subscribe

O exemplo para o padrão Publish/Subscribe demonstra a implementação de dois processos, um no papel de Publisher e um no papel de Subscriber. A Figura 5.16 ilustra a aplicação.

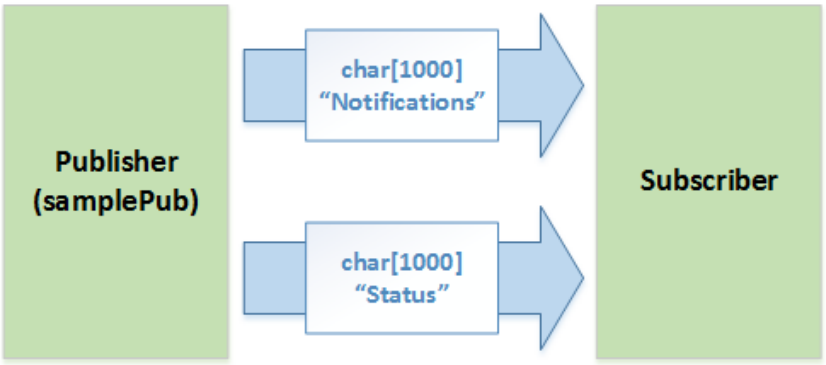


Figura 5.16: LX-MCAPI Patterns - Exemplo Publish/Subscribe

O processo *publisher* (Código 5.5) publica duas mensagens de 1000 bytes através da chamada `mcapi_ps_publish()`. Esta chamada recebe um nome que define a identidade do *publisher*, o nome do tópico, o *buffer* e o tamanho do *buffer* a ser enviado.

```
#include "lxmcapi.h"

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  char* buffer;
  buffer = malloc(1000*sizeof(char));
  memset(buffer, 'a', 999);

  mcapi_initialize(1, 1, NULL, NULL, NULL, &mcapi_status);

  mcapi_ps_publish("samplePub", "Notifications", buffer, 1000, &
    mcapi_status);
  usleep(10);

  memset(buffer, 'b', 999);
  mcapi_ps_publish("samplePub", "Status", buffer, 1000, &mcapi_status);
  free(buffer);
  return 0;
}
```

Código 5.5: Exemplo Publish/Subscribe - Publisher

No exemplo são publicadas, uma mensagem em nome de "*samplePublisher*" com o tópico "*Notifications*" e um *buffer* preenchido pelo caractere 'a', e uma mensagem também em nome de "*samplePublisher*" com o tópico "*Status*" e um *buffer* preenchido pelo caractere 'b'.

O processo *subscriber* (Código 5.6) recebe as mensagens, informando o "nome amigável"do servidor, o tópico de interesse e o *buffer* que armazenariará a mensagem recebida, através da chamada `mcapi_ps_subscribe()`.

```
#include "lxmcapi.h"

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  char* buffer;
  buf = malloc(1000*sizeof(char));

  mcapi_initialize(1, 2, NULL, NULL, NULL, err);

  mcapi_ps_subscribe("samplePub", "Notifications", &buffer, &mcapi_status);
  puts(buffer);

  mcapi_ps_subscribe("samplePub", "Status", &buffer, &mcapi_status);
  puts(buffer);

  free(buffer);
  return 0;
}
```

Código 5.6: Exemplo Publish/Subscribe - Subscriber

5.2.4 Padrão Push/Pull

O exemplo para o padrão Push/Pull consiste na inicialização de um processo produtor (Código 5.7), um processo consumidor (Código 5.8) e um processo coletor (Código 5.9). A Figura 5.17 ilustra a aplicação.

O processo produtor inicia um *broker* em modo PUSH, utilizando `mcapi_pp_broker()`, que aguarda a conexão do processo consumidor. Por sua vez, o processo coletor também inicia de maneira similar um *broker* aguardando a conexão do processo consumidor, porém em modo PULL para retirada de pacotes. O processo consumidor conecta-se ao processo produtor e ao processo consumidor utilizando `mcapi_pp_request()`, enviando requisições de conexão.

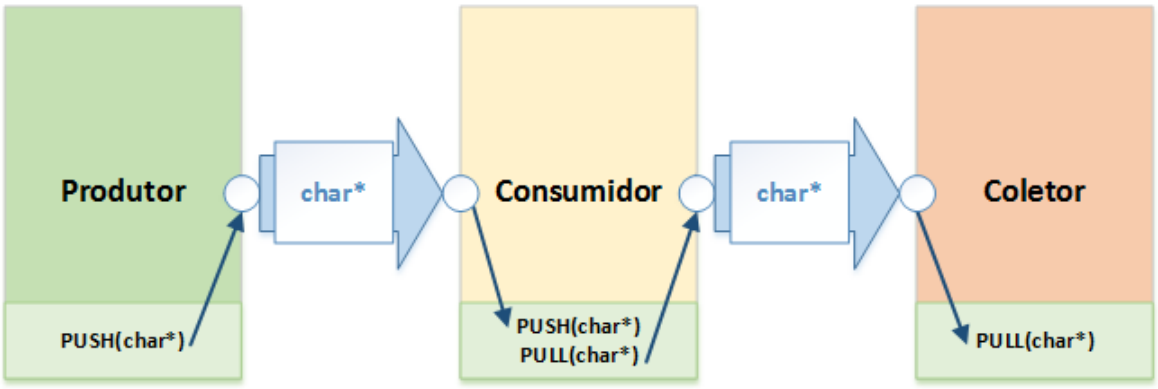


Figura 5.17: LX-MCAPI Patterns - Exemplo Push/Pull

A partir do momento em que os processos estão sincronizados, o processo produtor despacha pacotes para o processo consumidor (`mcapi_pp_push()`); o processo consumidor recebe pacotes do processo produtor (`mcapi_pp_pull()`) e despacha os mesmos para o processo coletor(`mcapi_pp_push()`); e o processo coletor recebe os pacotes (`mcapi_pp_pull()`) do consumidor e exibe seu conteúdo. Ressaltando que pacotes são quaisquer estruturas ou elementos que podem ser enviados através de *pipes* - no caso específico do exemplo correspondem a *strings*.

```
#include "lxmcapi.h"

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  mcapi_pp_handle_t handle;
  char* buffer = "Hello World";

  mcapi_initialize(1, 1, NULL, NULL, NULL, &mcapi_status);

  handle = mcapi_pp_broker(1001, 1, PUSH, &mcapi_status);

  puts("Producer pushing buffer...");
  for(int i = 0; i < 100000; i++) {
    printf("Iteration - %d\n", i);
    mcapi_pp_push(handle, buffer, strlen(buffer), &mcapi_status);
  }
  return 0;
}
```

Código 5.7: Exemplo Push/Pull - Processo Produtor

```
#include "lxmcapi.h"

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  char buffer[12];
  mcapi_pp_handle_t handle, handle2;

  mcapi_initialize(1, 2, NULL, NULL, NULL, &mcapi_status);
  handle = mcapi_pp_request(1, 1001, PULL, &mcapi_status);
  handle2 = mcapi_pp_request(3, 3001, PUSH, &mcapi_status);

  for(int i = 0; i < 100000; i++) {
    mcapi_pp_pull(handle, &buffer, 11, &mcapi_status);
    mcapi_pp_push(handle2, buffer, strlen(buffer), &mcapi_status);
  }
  return 0;
}
```

Código 5.8: Exemplo Push/Pull - Processo Consumidor

```
#include "lxmcapi.h"

int main(int argc, char** argv)
{
  mcapi_status_t mcapi_status;
  mcapi_pp_handle_t handle;
  char buffer[12];
  mcapi_initialize(1, 3, NULL, NULL, NULL, &mcapi_status);
  handle = mcapi_pp_broker(3001, 1, PULL, &mcapi_status);
  for(int i = 0; i < 100000; i++) {
    mcapi_pp_pull(handle, &buffer, 11, &mcapi_status);
    printf("%d - %s\n", i, buffer);
  }
  return 0;
}
```

Código 5.9: Exemplo Push/Pull - Processo Coletor

5.3 Aplicações

5.3.1 Conjunto de Mandelbrot

O conjunto de Mandelbrot é gerado pelo que se denomina iteração, o que significa repetir um processo diversas vezes. Matematicamente, este processo é comumente associado a aplicação de uma função matemática. No caso do conjunto de Mandelbrot, as funções envolvidas são simples: todas são polinômios quadráticos da forma $f(z) = z^2 + c$, onde c é uma constante numérica e z um número complexo.

A teoria das funções iteradas é motivada por questões reais, como o modelo de crescimento de populações. A próxima iteração dependerá da quantidade atual de indivíduos dessa população. Por essa razão, para descobrir o tamanho de uma população após vários ciclos de nascimento, é necessário iterar a função que descreve seu crescimento. Coincidentemente, as funções para os modelos-padrão para crescimento de populações também são polinômios quadráticos de forma similar a do conjunto de Mandelbrot, e foi a primeira motivação para o seu estudo.

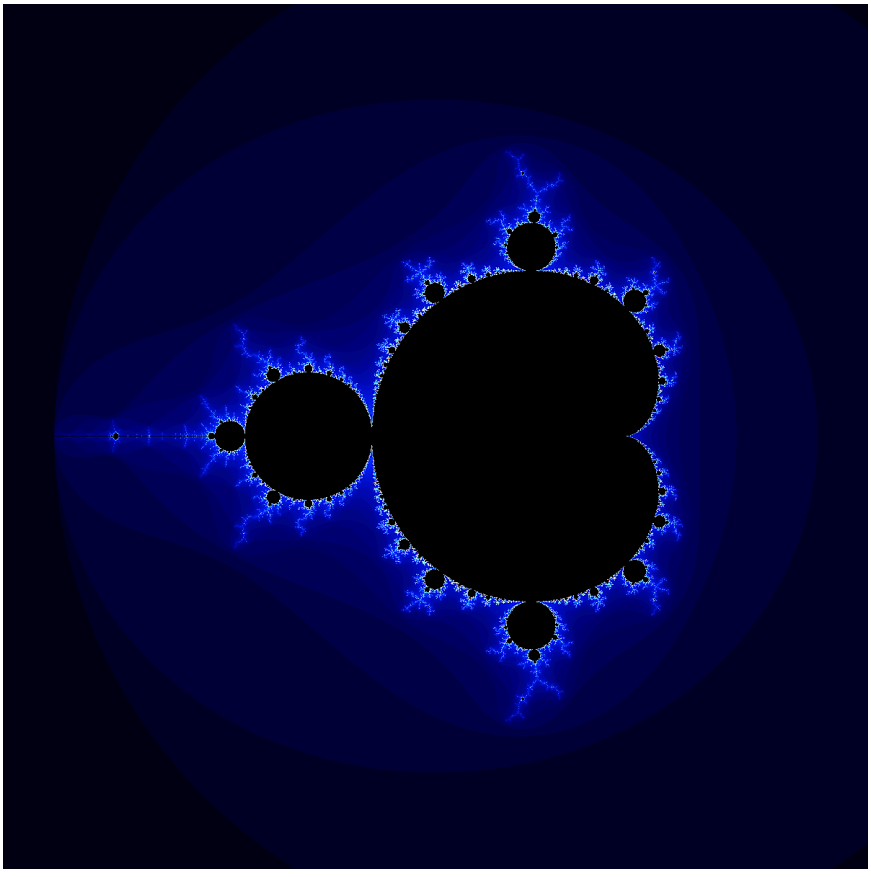


Figura 5.18: Representação visual do Conjunto de Mandelbrot

O algoritmo que gera o conjunto de Mandelbrot entra na categoria denominada *Embarassingly Parallel* (ou em tradução livre, Embaraçosamente Paralelo), ou seja, a paralelização do problema é relativamente simples, sendo possível separá-lo em um determinado número de tarefas paralelas de maneira intuitiva. É o caso onde há pouca ou nenhuma dependência direta entre os cômputos, pouca necessidade de comunicação entre as tarefas paralelas ou seus resultados.

Estratégia de Paralelização

Os principais *loops* aninhados para geração do conjunto de Mandelbrot são apresentados no Código 5.10, onde x_{min}, x_{max} e y_{min}, y_{max} correspondem as coordenadas mínimas e máximas, respectivamente nas direções do eixo x e y do plano cartesiano. Se deseja-se gerar uma imagem de 600 x 600 pixels, $x_{min} = 0, x_{max} = 600, y_{min} = 0, y_{max} = 600$.

```
for(int i = scr.y_min; i < scr.y_max; ++i) {
  for(int j = scr.x_min; j < scr.x_max; ++j) {
    c.i = i;
    c.r = j;
    c = scale(scr, fr, c);
    colors[k] = escape(c, iter_max);
    k++;
  }
}
```

Código 5.10: Processamento e Geração do Conjunto de Mandelbrot - Código Sequencial

O algoritmo admite que o **eixo y** corresponde aos valores imaginários e o **eixo x** aos valores reais. Para cada coordenada do plano são executadas duas funções, `scale()` e `escape()`. A função `scale()` é responsável por dimensionar os valores complexos para dentro de um intervalo que determinará a forma do fractal, enquanto a função `escape()` itera sobre os valores dimensionados `iter_max` vezes, e o valor resultante das iterações é mapeado para um vetor de cores, que posteriormente será plotado.

Cada iteração dos *loops* aninhados é independente de iterações passadas ou futuras, portanto podem ser paralelizadas de maneira independente. A principal modificação para paralelização foi gerar o vetor de números complexos *a priori*, e distribuir frações de tamanhos iguais desse vetor para cada um dos processos. A Figura 5.19 ilustra o processo de distribuição de *workload* entre os processos.

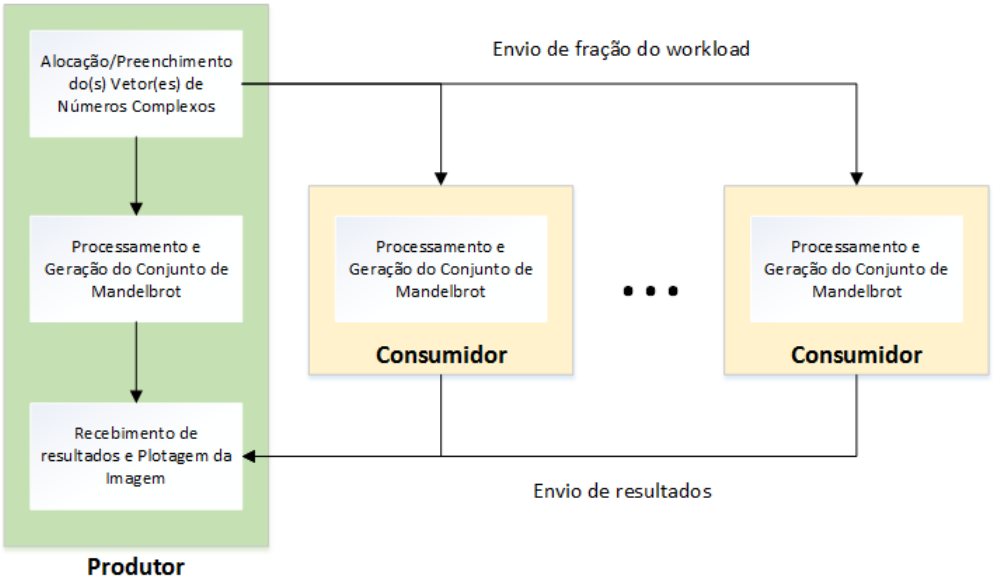


Figura 5.19: Geração paralela do Conjunto de Mandelbrot

Por exemplo, para paralelização em dois processos, o vetor de números complexos é gerado e dividido em duas partes, onde o processo Produtor gera o vetor e envia para o processo Consumidor uma das metades. Ambos os processos efetuam as mesmas operações de dimensionamento e iteração em suas respectivas metades. O processo Consumidor responde com o vetor resultante.

```
// Aloca vetor de elementos
element* elements;
elements = malloc(scr.size * sizeof(element));
// Aloca vetor resultante de cores
int* colors;
colors = malloc(scr.size * sizeof(int));
// Preenche o vetor de elementos com os numeros complexos
for(int i = scr.y_min; i < scr.y_max; ++i) {
  for(int j = scr.x_min; j < scr.x_max; ++j) {
    elements[k].c.r = j;
    elements[k].c.i = i;
    k++;
  }
}
// Aponta para a segunda parte do vetor de elementos
element* secondHalf;
secondHalf = elements + scr.size/2;
// Aponta para a segunda parte do vetor de cores
int *color_sh = colors + scr.size/2;
// Envia a segunda parte para o processo Consumidor
```

```
mcapi_pktchan_send(&send_handle, *&secondHalf, sizeof(element) * scr.size
 /2, &mcapi_status);
// Calcula sobre a primeira parte do vetor (comum ao processo Consumidor)
for(int i = 0; i < scr.size/2; i++)   {
  elements[i].c = scale(scr, fr, elements[i].c);
  colors[i] = escape(elements[i].c, iter_max);
}
// Recebe o vetor resultante
mcapi_pktchan_recv(recv_handle, *&color_sh, sizeof(int) * scr.size/2, &
 len, &mcapi_status);
```

Código 5.11: Conjunto de Mandelbrot - Paralelização em 2 processos

O processo de divisão e envio das frações do vetor e posterior recebimento das partes resultantes pode ser aplicado intuitivamente para 2, 4 e 8 processos distribuídos nos núcleos do processador. O Código 5.11 omite as funções de conexão e sincronização da LX-MCAPI e ilustra uma possível solução para o problema de dois processos.

Estatísticas e Resultados

Um extenso conjunto de testes foi executado, no intuito de verificar as capacidades de escalabilidade da infraestrutura fornecida pela LX-MCAPI. O algoritmo gerador do Conjunto de Mandelbrot foi desenvolvido em quatro variantes: sequencial, concorrente por *threads* em memória compartilhada, paralela por pacotes e paralela por *zero-copy*. Imagens de 1200x1200, 2400x2400, 4800x4800 e 9600x9600 pixels foram geradas avaliando-se para cada dimensão o seu tempo de geração (desconsiderando o tempo de saída em arquivo).

Em todos os casos de teste foi utilizada a *flag* de otimização `-O3` do compilador GCC para geração do executável. A habilitação de *flags* de otimização faz com que o compilador tente melhorar o desempenho e o tamanho do código em troca de tempo de compilação e opcionalmente adicionando capacidades de *debug*. A *flag* utilizada habilita todas as otimizações possíveis do nível `-O2`, incluindo otimizações no escalonamento de *threads*, vetorização e otimização de *loops*, que contribuem no aumento de desempenho da aplicação.

No intuito de ilustrar qual a influência das otimizações no processo de geração do código de máquina, extraiu-se somente o *loop* apresentado no Código 5.10 e analisou-se de maneira superficial as instruções em MASM geradas. Analisando os códigos ASM gerados com a flag `O3` ativada, o compilador passa a utilizar registradores e operações SSE (*Streaming SIMD Extensions*) com muito mais frequência, paralelizando a nível de instrução os *loops*, principalmente sobre operações em vetores, tirando proveito dos registradores XMM de 128 bits e as instruções

de precisão dupla como `movsd`, `mulsd`, `addsd`, dentre outras. Claramente não há uma correspondência direta um-para-um entre os códigos gerados com e sem a *flag* de otimização ativada.

A variante em memória compartilhada utilizando Pthreads foi expandida para utilização de um a oito *threads* concorrentes. Uma comparação simples foi feita, habilitando e desabilitando a *flag* de otimização no processo de compilação. Os tempos de processamento para geração do conjunto de Mandelbrot foram avaliados para as 4 dimensões de imagem. Pode-se visualizar na Tabela 5.2 uma drástica diferença de até uma ordem de magnitude no tempo de processamento (em μs) quando as otimizações são habilitadas.

		1200 x 1200	**2400 x 2400**	**4800 x 4800**	**9600 x 9600**
Sem Flags	**1 thread**	2268511	8942016	36821035	151562582
	2 threads	1168210	4544346	18717685	82512098
	4 threads	1110971	4431044	17889469	72040491
	8 threads	858744	3427390	13905873	56088489
-O3 Flags	**1 thread**	547403	2233178	8715809	34705868
	2 threads	279071	1117215	4553985	17742210
	4 threads	456279	1080512	7209518	17256818
	8 threads	279502	1023642	3536304	13843963

Tabela 5.2: Conjunto de Mandelbrot - Compilação otimizada x não-otimizada (SHM)

As variantes paralelas foram expandidas para utilização de dois, quatro e oito processos atribuídos para núcleos diferentes da plataforma de testes x86-64. A Tabela 5.3 demonstra os resultados obtidos para cada teste executado, comparando os tempos de processamento paralelo com o tempo de processamento sequencial através do cálculo de *speedup*.

Através da análise dos resultados apresentados, pode-se visualizar que ambos os modelos de programação da LX-MCAPI ofereceram desempenhos similares, muito próximos à versão em memória compartilhada utilizando *threads*. As *threads* apresentaram melhor desempenho, principalmente por tirar proveito das otimizações aplicadas em tempo de compilação e pela vantagem do acesso direto às regiões de memória compartilhada, eliminando *overheads* de transferência, presentes nas estratégias de passagem de mensagem (o tempo gasto em comunicação interprocessual é provavelmente menor).

	1200x1200	**2400x2400**	**4800x4800**	**9600x9600**	**Speedup**
Sequencial	1006371	3972431	16074128	63772398	1,000
Copy (2 cores)	582838	2163473	8119225	33365231	1,863
Copy (4 cores)	568298	1392689	6187118	22833977	2,504
Copy (8 cores)	248546	1012808	4418181	16782085	3,852
ZC (2 cores)	599535	2087412	7226940	23267991	2,137
ZC (4 cores)	559339	2252329	5576654	20117065	2,404
ZC (8 cores)	292132	1031627	4134187	19228194	3,625
SHM (1 thread)	547403	2233178	8715809	34705868	1,825
SHM (2 threads)	279071	1117215	4553985	17742210	3,571
SHM (4 threads)	456279	1080512	7209518	17256818	2,952
SHM (8 threads)	279502	1023642	3536304	13843963	4,158

Tabela 5.3: Conjunto de Mandelbrot - Tempo de Geração (μs) e *Speedup*

A utilização do modelo Copy da LX-MCAPI é similar aos modelos de passagem de mensagem tradicionais, como o MPI, inclusive em relação a estrutura de código, permitindo que um código MPI que utilize operações básicas de envio e recebimento de dados possa ser traduzido entre as duas especificações, impostas as limitações da LX-MCAPI (por exemplo, LX-MCAPI não suporta diretamente o modelo de *broadcast* disponível no MPI).

O modelo Zero-copy, por outro lado, implementa o modelo de memória compartilhada utilizando a abstração da passagem de mensagens, permitindo o compartilhamento de dados entre *threads* e processos. As funções da LX-MCAPI não foram programadas como funções *thread-safe*, mas seu uso em ambientes que utilizam *threads* não é impeditivo por este fato.

5.3.2 Detecção, Segmentação e Normalização de Iris

O Reconhecimento de Íris é um método automático de identificação biométrica que utiliza técnicas de reconhecimento de padrões sobre imagens estáticas ou dinâmicas dos olhos de um indivíduo, extraindo características únicas dos padrões encontrados na região da íris, permitindo criar métodos de autenticação de indivíduos. A Figura 5.20 ilustra as principais regiões de interesse no estudo do Reconhecimento de Íris.

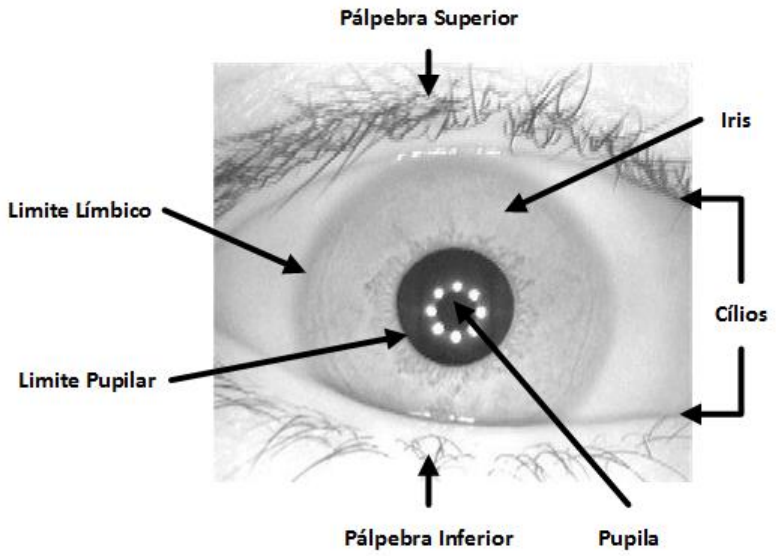


Figura 5.20: Regiões de Interesse do Olho Humano para Reconhecimento de Íris

Um dos principais autores na área de reconhecimento de íris é John Daugman, responsável pelo desenvolvimento do IrisCode (DAUGMAN, 1993), um algoritmo desenvolvido em 1993 e atualizado continuamente, aplicado em diversas soluções comerciais de autenticação biométrica.

No desenvolvimento deste trabalho, implementou-se um algoritmo de segmentação que utiliza técnicas de processamento de imagem para detecção da região dos limites pupilar e límbico, e normaliza a região da íris obtida, criando um padrão de dimensão fixa que pode ser utilizado para fins de autenticação após tratamento e extração de atributos.

O algoritmo foi desenvolvido em linguagem C, utilizando a biblioteca OpenCV para carregamento e processamento de imagens. Basicamente o processo de segmentação da íris pode ser dividido em três estágios: detecção dos limites da pupila, detecção dos limites do *limbus* e normalização da região da íris.

A detecção da pupila consiste na estimação da posição do centro e o raio do círculo que se aproxima das dimensões da pupila. O processo implementado consiste nos passos descritos na Tabela 5.4.

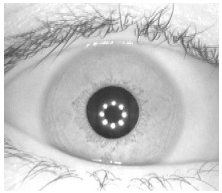
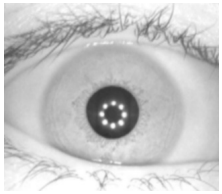
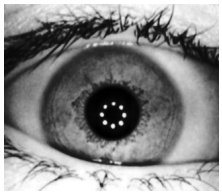
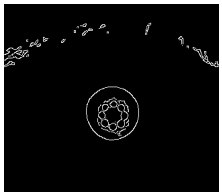
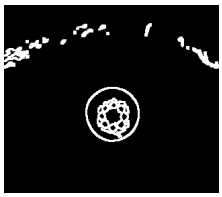
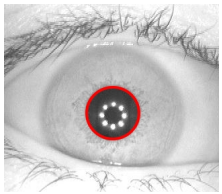
Imagem Resultante	Descrição
	Carregamento da imagem original do olho e conversão para escala de cinza
	Aplicação de *kernel* gaussiano para atenuação de ruídos
	Equalização de Histograma para uniformização de contraste
	Correção de gamma com valor negativo, evidenciando a pupila
	Detecção de bordas através do Algoritmo de Canny
	Dilatação das bordas unindo possíveis descontinuidades
	Detecção do maior círculo encontrado através da Transformada de Hough

Tabela 5.4: Processo de Detecção da Pupila

O processo de detecção dos limites do *limbus* é similar ao processo de detecção da pupila, porém o pré-processamento da imagem e os parâmetros dos algoritmos são ajustados para estimação da posição do centro e o raio do círculo que se aproxima das dimensões da região. Os passos do processo implementado são citados na Tabela 5.5.

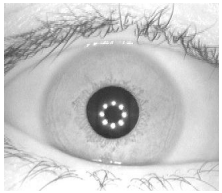
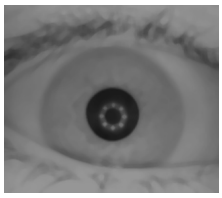
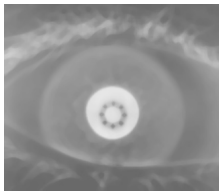
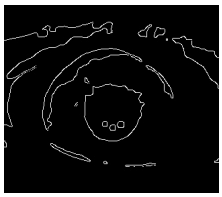
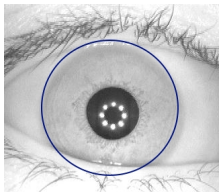
Imagem Resultante	Descrição
	Carregamento da imagem original do olho e conversão para escala de cinza
	Aplicação de filtro da mediana para atenuação de ruídos
	Inversão dos valores de intensidade dos pixels
	*Threshold* binário, evidenciando as regiões da íris e pálpebras
	Erosão das áreas reveladas pelo *threshold* binário para separação da íris
	Detecção de bordas através do Algoritmo de Canny
	Detecção do maior círculo encontrado através da Transformada de Hough

Tabela 5.5: Processo de Detecção da Iris

Por fim, o processo de normalização consiste na aplicação do modelo *rubbersheet* (DAUGMAN, 2004), fornecendo como parâmetros de entrada o centro e o raio da pupila, o centro e o raio da iris, e as dimensões fixas do padrão que será gerado. A implementação em linguagem C foi baseada na implementação em MATLAB de (MASEK; KOVESI, 2003).

O modelo *rubbersheet* mapeia cada ponto dentro da região da íris para um par de coordenadas polares (r,θ) onde r é um valor entre $[0,1]$ e θ um ângulo entre $[0,2\pi]$. A Figura 5.21 ilustra o mapeamento de coordenadas.

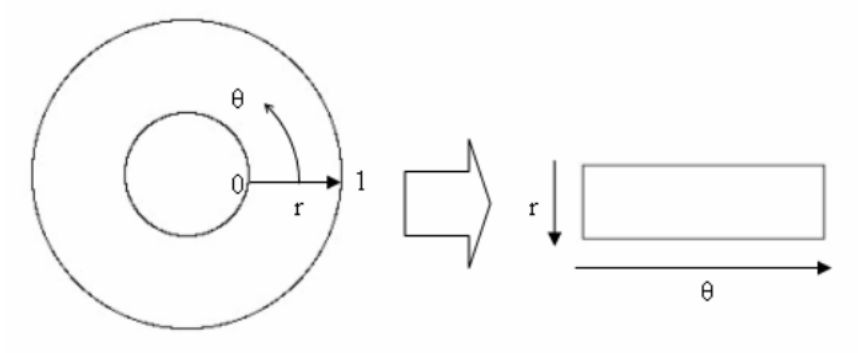


Figura 5.21: Modelo Rubbersheet - Extraído de: (MASEK et al., 2003)

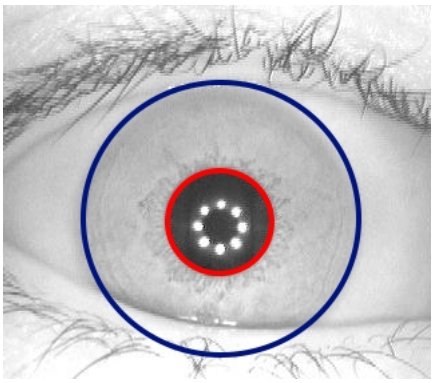
O mapeamento da região da iris de coordenadas cartesianas (x,y) para a representação polar normalizada e não-concêntrica é descrita pelas equações:

$$I(x(r,\theta),y(r,\theta) \rightarrow I(r,\theta)$$

$$x(r,\theta) = (1-r)x_p(\theta) + rx_i(\theta)$$

$$y(r,\theta) = (1-r)y_p(\theta) + ry_i(\theta)$$

Onde $I(x,y)$ consiste na região da íris na imagem, (x,y) são as coordenadas cartesianas originais e (r,θ) as coordenadas polares normalizadas correspondentes. Os valores x_p, y_p e x_i, y_i correspondem respectivamente às coordenadas dos limites da pupila e da íris na direção do ângulo θ. Um possível resultado da aplicação do modelo *rubbersheet* para a Figura 5.22a é visualizado na Figura 5.22b.



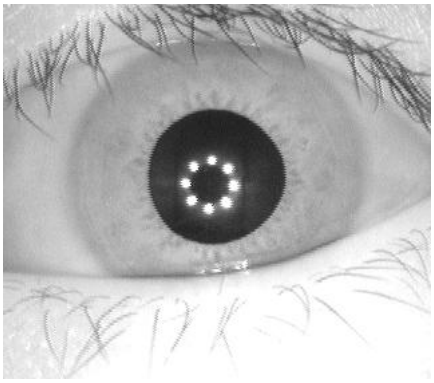
(a) Região da iris detectada

(b) Normalização da região da iris

Figura 5.22: Resultado de aplicação do modelo *rubbersheet*

O modelo *rubbersheet* leva em conta a dilatação da pupila, distância de amostragem e posicionamento não concêntrico entre pupila e íris, porém não compensa possíveis inconsistências rotacionais. Essa compensação é feita na fase de busca e *matching* na implementação de Daugman, efetuando operações de *shifting* na direção de θ até que dois padrões se alinhem.

Um problema comum para extração do padrão da íris pelo modelo *rubbersheet* é a oclusão, ocasião onde partes da região da iris encontram-se obstruídas por pálpebras ou cílios, por exemplo. Várias abordagens para atenuação de áreas ocluídas e métodos de segmentação alternativos já foram propostos, porém este trabalho limitou-se a uma breve investigação sobre o assunto e não efetuou implementações do gênero. A Figura 5.23 demonstra um caso de oclusão no processo de segmentação e normalização.



(a) Imagem original

(b) Normalização com Oclusão

Figura 5.23: Problema de Oclusão no processo de Normalização

Estratégia de Paralelização

O método que consome a maior parte do tempo de processamento dentre todas as operações realizadas para segmentação da região da íris é a Transformada de Hough (tabelas 5.6 e 5.7 da seção 5.3.2), consistindo em um método de ”força-bruta”que verifica para cada tamanho estimado de raio se há a possibilidade de se aproximar um círculo em alguma região da imagem. A imagem de entrada consiste na saída do Algoritmo de Canny para detecção de bordas. A implementação da Transformada de Hough foi baseada no trabalho de (MISEIKIS, 2012), que apresenta um estudo comparativo entre uma versão sequencial em C e uma versão em CUDA.

O código da Transformada de Hough consiste de quatro *loops* aninhados que testam a presença de bordas na imagem de entrada para um dado círculo com raio estimado. O algoritmo vasculha um intervalo (por exemplo, no caso da detecção da pupila, o valor mínimo de 30 e o valor máximo de 70) para os raios a serem buscados. O *loop* mais externo permite paralelização intuitiva e foi o alvo das otimizações.

```
for (int radius=radMin; radius <= radMax; radius++) {
  //Inicializa variaveis  ...
  for (int x=radius; x < width-radius; x++) {
    for (int y=radius; y < height-radius; y++) {
      // 'Desenha' um circulo
      for (int theta=0; theta < 360; theta+=5) {
        // Calcula coordenadas x e y
        float angle = (theta*PI) / 180;
        int tempX = (int)(x - radius*cos(angle));
        int tempY = (int)(y - radius*sin(angle));
        // Se houver uma borda no ponto (x,y), adiciona um voto
        if (img[tempY*width+tempX] == 1)
          houghSpace[y*width+x] = houghSpace[y*width+x] + 1; }}}}
```

Código 5.12: Transformada de Hough Sequencial

Neste caso, onde há grande quantidade de variáveis compartilhadas como o vetor de intensidades de pixels da imagem e os vetores de posição, a passagem de mensagens por cópia cria redundância de dados em cada processo, aumentando o *memory footprint* da aplicação. No entanto, o desempenho obtido é similar ao desempenho obtido para a memória compartilhada. O Código 5.13 demonstra um trecho do código responsável pela paralelização do *loop* mais externo.

```
int* posxArray;
posxArray = malloc(sizeof(int) * (radMax - radMin));
int* posyArray;
posyArray = malloc(sizeof(int) * (radMax - radMin));
int* maxValArray;
maxValArray = malloc(sizeof(int) * (radMax - radMin));

int* radArray;
radArray = malloc(sizeof(int) * (radMax - radMin));

int* imgPointer[N_WORKERS+1];
int* posXPointer[N_WORKERS+1];
int* posYPointer[N_WORKERS+1];
int* maxValPointer[N_WORKERS+1];
int* radPointer[N_WORKERS+1];

imgPointer[0] = imgData;
posXPointer[0] = posxArray;
posYPointer[0] = posyArray;
maxValPointer[0] = maxValArray;
```

```
radPointer[0] = radArray;

for(int i = 1; i < N_WORKERS+1; i++){
  posXPointer[i] = posXPointer[i-1] + (radMax - radMin)/(N_WORKERS+1);
  posYPointer[i] = posYPointer[i-1] + (radMax - radMin)/(N_WORKERS+1);
  maxValPointer[i] = maxValPointer[i-1] + (radMax - radMin)/(N_WORKERS+1)
 ;
  radPointer[i] = radPointer[i-1] + (radMax - radMin)/(N_WORKERS+1);
}

for(int i = 0; i < N_WORKERS; i++) {
  mcapi_pktchan_send(&pktst[i].send_handle, *&imgData, (sizeof(int)*width
 *height), &mcapi_status);
}
// Codigo comum a todos os Consumidores
int index = 0;
for (int r=radMin; r <= radMin+(radMax-radMin)/(N_WORKERS+1); r++) {
  // Transformada de Hough
  radius = r;
  Hough2D(imgPointer[0], width, height, radius, &posX, &posY, &maxVal);
  posxArray[index] = posX;
  posyArray[index] = posY;
  maxValArray[index] = maxVal;
  radArray[index] = radius;
  index++;
}

for(int i = 0; i < N_WORKERS; i++) {
  mcapi_pktchan_recv(pktst[i].recv_handle, *&posXPointer[i+1],sizeof(int)
 *(radMax - radMin)/(N_WORKERS+1), &len, &mcapi_status);
  mcapi_pktchan_recv(pktst[i].recv_handle, *&posYPointer[i+1],sizeof(int)
 *(radMax - radMin)/(N_WORKERS+1),  &len, &mcapi_status);
  mcapi_pktchan_recv(pktst[i].recv_handle, *&maxValPointer[i+1],sizeof(
 int)*(radMax - radMin)/(N_WORKERS+1), &len, &mcapi_status);
  mcapi_pktchan_recv(pktst[i].recv_handle, *&radPointer[i+1],sizeof(int)*
 (radMax - radMin)/(N_WORKERS+1), &len, &mcapi_status);
}
```

Código 5.13: Transformada de Hough - Paralelização Copy

Utilizando o mecanismo *zero-copy*, a estratégia adotada para paralelização foi compartilhar *a priori* todas as variáveis comuns entre os processos, onde o processo Produtor é responsável pela alocação dos espaços de memória, reduzindo o *memory footprint* do sistema e atribuindo

intervalos de *workload* igualmente distribuídos para cada um dos processo. O Código 5.14 demonstra um trecho do código responsável pela paralelização do *loop* mais externo utilizando os mecanismos de *zero-copy*.

```
mcapi_zc_hndl_t posxArray;
mcapi_zc_hndl_t posyArray;
mcapi_zc_hndl_t maxValArray;
mcapi_zc_hndl_t radArray;

mcapi_zc_alloc(&posxArray, sizeof(int) * (radMax - radMin), &mcapi_status
 );
mcapi_zc_alloc(&posyArray, sizeof(int) * (radMax - radMin), &mcapi_status
 );
mcapi_zc_alloc(&maxValArray, sizeof(int) * (radMax - radMin), &
 mcapi_status);
mcapi_zc_alloc(&radArray, sizeof(int) * (radMax - radMin), &mcapi_status)
 ;

for(int i = 0; i < N_WORKERS; i++) {
  mcapi_pktchan_zc_send(&pktst[i].send_handle, imgData, &mcapi_status);
  mcapi_pktchan_zc_send(&pktst[i].send_handle, posxArray, &mcapi_status);
  mcapi_pktchan_zc_send(&pktst[i].send_handle, posyArray, &mcapi_status);
  mcapi_pktchan_zc_send(&pktst[i].send_handle, maxValArray, &mcapi_status
 );
  mcapi_pktchan_zc_send(&pktst[i].send_handle, radArray, &mcapi_status);
}
int* px = (int*)posxArray.buffer;
int* py = (int*)posyArray.buffer;
int* mv = (int*)maxValArray.buffer;
int* ra = (int*)radArray.buffer;
// Codigo comum a todos os Consumidores
int index = 0;
for (int r=radMin; r <= radMin+(radMax-radMin)/(N_WORKERS+1); r++) {
  // Transformada de Hough
  radius = r;
  Hough2D(imgPointer, width, height, radius, &posX, &posY, &maxVal);
  px[index] = posX;
  py[index] = posY;
  mv[index] = maxVal;
  ra[index] = radius;
  index++;
}
for(int i = 0; i < N_WORKERS; i++) {
```

```
    mcapi_pktchan_zc_receive(pktst[i].recv_handle, &radArray, sizeof(int) *
    (radMax - radMin), NC_SHARED, &mcapi_status);
}
```

Código 5.14: Transformada de Hough - Paralelização Zero-copy

Cada processo será responsável por calcular a Transformada de Hough para uma fração dos valores dos raios no intervalo e retornará o maior valor encontrado para o processo Produtor, que comparará e decidirá pelo maior valor dentre os resultados. Em casos onde não é possível dividir o intervalo de raios igualmente (dimensões ímpares), um dos processos consumidores processará uma posição a menos que os outros. O método`Hough2D` consiste no encapsulamento dos três *loops* mais internos do código 5.12.

Estatísticas e Resultados

Testes foram aplicados sobre o banco de imagens CASIA v3, consistindo em imagens capturadas sobre iluminação NIR (Near InfraRed) dos olhos esquerdo e direito de indivíduos. Limitou-se o conjunto de testes a 50 amostras escolhidas arbitrariamente. Todas as imagens tem dimensão de 320 x 280 pixels e a dimensão dos padrões gerados pelo modelo *rubbersheet* é de 180 x 20 pixels.

Estatísticas do processo de segmentação foram extraídas para verificar o consumo de tempo de processamento para cada estágio individual do *workflow* proposto. Os resultados dos algoritmos sequenciais para a região da pupila e da íris são encontrados respectivamente nas tabelas 5.6 e 5.7.

Estágio de Processamento	Tempo (μs)
Filtro Gaussiano	228
Equalização de Histograma	158
Correção de Gamma	323
Detecção de Bordas	1200
Transformada de Hough	***9343415***

Tabela 5.6: Tempo de Processamento Sequencial: Pupila

Estágio de Processamento	Tempo (μs)
Filtro da Mediana	4900
Inversão (NOT)	10
Threshold	24
Erosão	98
Detecção de Bordas	1107
Transformada de Hough	***2775740***

Tabela 5.7: Tempo de Processamento Sequencial: Iris

Os testes demonstraram que o tempo médio da Transformada de Hough sequencial é de 10 ± 1 segundos para a região da pupila e 3 ± 1 segundos para a íris. O algoritmo foi paralelizado e expandido para utilização de 2, 4 e 8 processos mapeados um para cada núcleo do processador da plataforma x86-64. A estratégia de paralelização da Transformada de Hough proporcionou

melhora no desempenho, chegando ao valor de *speedup* 4 comparado a versão sequencial.

Os resultados para a Tranformada de Hough para a região da pupila são apresentados na Tabela 5.8. Resultados similares foram obtidos para a região da íris.

	Tempo (μs)	**Speedup**
Sequencial	9527757	1
Copy (2 cores)	5509956	1,729
Copy (4 cores)	3605120	2,649
Copy (8 cores)	2292892	4,155
ZeroCopy (2 cores)	5333989	1,786
ZeroCopy (4 cores)	3439412	2,770
ZeroCopy (8 cores)	2365835	4,027
SHM (1 thread)	8366114	1,139
SHM (2 thread)	5201159	1,832
SHM (4 thread)	3257950	2,925
SHM (8 thread)	2139970	4,452

Tabela 5.8: Transformada de Hough (Pupila) - Tempo de Processamento (μs) e *Speedup*

Similar a discussão realizada para o Conjunto de Mandelbrot, os modelos de Copy e Zero-copy apresentaram desempenho muito similar entre os dois, e muito próximo à solução de memória compartilhada. A principal vantagem na utilização do modelo Zero-copy neste caso é a economia no espaço de armazenamento, evitando redundância de dados entre os processos. Um resultado interessante é que o processo de cópia dos dados consome uma fração bem pequena do tempo de processamento, evitando interferências significativas no desempenho total da aplicação, condição favorável para adoção do método de paralelização utilizando a LX-MCAPI.

5.3.3 Considerações

Ambas as aplicações abordadas, em teoria, desfavorecem a utilização da passagem de mensagem, pois compartilham grandes quantidades de dados e possuem operações paralelizáveis que seriam facilmente implementadas através de *threads* e memória compartilhada. No entanto, pode-se visualizar pelas análises apresentadas que a passagem de mensagem pode ser um método inteligível e intuitivo para implementação de soluções e permite o projeto e compreensão das características paralelas, inserindo pouca complexidade em termos de código, e ainda favorecendo a posterior portabilidade e expansão para sistemas de execução distintos.

Capítulo 6

CONCLUSÃO

O presente trabalho propôs inicialmente uma investigação das estratégias de programação paralela voltadas aos sistemas multi-core, uma área relativamente recente que apesar da grande variedade de ferramentas disponíveis, não possui um padrão consolidado, herdando grande parte das metodologias dos modelos já existentes, como o modelo de memória compartilhada e o modelo de passagem de mensagem em sistemas distribuídos.

As técnicas atuais para sistemas paralelos tendem a utilizar abordagens híbridas, unindo ferramentais de memória compartilhada, como o OpenMP, aos ferramentais de passagem de mensagem, como o MPI. A associação e equilíbrio na coexistência dos dois paradigmas é o ponto-chave para o desenvolvimento de aplicações de alto-desempenho.

Outras ferramentas tentaram associar os paradigmas de memória compartilhada e o de passagem de mensagem, dando início ao desenvolvimento das linguagens PGAS. Essas linguagens criaram um novo paradigma que permitia o mapeamento do espaço de endereçamento de memória como uma única entidade, onde os processos teriam acesso de modo transparente. A curva de aprendizado de uma nova linguagem e a aparente dificuldade na implementação do novo paradigma inviabilizou a ampla utilização das linguagens PGAS, limitando-se somente a soluções específicas.

A busca para encontrar um método de programação que permitisse utilizar a passagem de mensagem como alternativa ao já consolidado modelo de memória compartilhada, possibilitando ainda tirar proveito das vantagens de ambos os paradigmas, resultou no desenvolvimento da biblioteca LX-MCAPI, uma instância da MCAPI, a única especificação existente para bibliotecas de comunicação dedicada a ambientes multi-core (principalmente embarcados).

A LX-MCAPI foi desenvolvida como resultado das investigações e experimentos, buscando inspiração nas tendências de ferramentas de comunicação baseadas em filas de mensagem,

como a ZeroMQ e nanomsg, além das funcionalidades dos modelos de passagem de mensagem, como o MPI. É voltada especialmente a sistemas multi-core com memória compartilhada e permite a comunicação entre processos através das estruturas especificadas na MCAPI. Utiliza a implementação de referência do D-Bus para controle, sincronização e gerenciamento de conexões e criação de canais virtuais entre os processos. É limitada a utilização em sistemas operacionais Linux devido suas dependências com as utilidades do *kernel*.

Através dos resultados gerados pelos experimentos com a infraestrutura da LX-MCAPI e as aplicações paralelizadas utilizando-se dos métodos disponibilizados, pode-se constatar que a passagem de mensagem é uma estratégia de paralelização viável, obtendo resultados em termos de desempenho similares aos do modelo de memória compartilhada, e inserindo pouca complexidade na estrutura do código. Para um programador familiarizado com a linguagem C, a curva de aprendizado para a utilização de uma API pequena e flexível é bem menor que, por exemplo, a de aprender uma nova linguagem do modelo PGAS ou modelos relativamente mais complexos utilizando APIs de grande porte como OpenMP e versões do MPI.

A LX-MCAPI oferece dois modos de compartilhamento de dados através dos mecanismos Copy e Zero-copy. O mecanismo Copy consiste no envio de *buffers* de maneira convencional, replicando informações entre processos, criando literalmente uma cópia dos dados no processo remoto. O mecanismo Zero-copy permite a criação e compartilhamento de uma área de memória anônima mapeável entre dois ou mais processos, de forma similar às regiões de memória utilizadas por *threads* no modelo de memória compartilhada.

Associando ambos os métodos de compartilhamento é possível atender as necessidades de aplicações de maneira flexível. Aplicações paralelas distintas tendem a apresentar necessidades específicas e ao oferecer em uma única biblioteca dois paradigmas é possível uniformizar o ambiente de desenvolvimento. O conjunto de chamadas apresenta notação verbosa porém intuitiva. As funções específicas para escalares, pacotes e mensagens evitam alguns erros lógicos de programação e permitem boa organização do código-fonte.

A utilização dos canais virtuais dedicados permite conectar de maneira simples os processos e estabelecer a topologia da aplicação *a priori*, favorecendo a organização e garantindo as conexões entre os processos distribuídos pelos núcleos do processador. Além disso, por utilizar canais virtuais entre os processos, congestionamentos de mensagens em filas compartilhadas não ocorrem, pois todas as transferências de *payloads* ocorrem ponto-a-ponto. No entanto, o estabelecimento de canais insere algumas linhas de código adicionais e tempo de *setup* que dependerá diretamente das decisões do programador com relação aos identificadores dos nós e terminais, exigindo um planejamento mais detalhado.

A estratégia de transmissão de pacotes, associando *pipes* e métodos *kernel-bypass* provou-se suficiente e mais eficiente que o método de transmissão oferecido pela implementação de referência do D-Bus, por eliminar *overhead* intrínseco ao funcionamento do protocolo.

A paralelização de um algoritmo é intrinsecamente ligada ao problema descrito por seu objetivo. Oferecer modelos de comunicação pré-programados visa acelerar o processo de paralelização e facilitar a abstração dos requisitos da aplicação alvo. Os padrões de programação fornecidos por LX-MCAPI Patterns permitem a implementação de aplicações de forma simplificada e rápida, facilitando o processo de construção de aplicações e acelerando o processo de abstração e particionamento de tarefas.

Quando comparada a outras APIs, LX-MCAPI apresentou latências de transmissão mais altas em alguns casos, mas o desempenho geral é satisfatório, principalmente para *payloads* de tamanho superior a 16KiB e quando comparado à implementação de referência do D-Bus. Claramente as latências de transmissão da LX-MCAPI são em média mais altas que as apresentadas pelos mecanismos de IPC de baixo-nível em vários casos, pois consiste em um agregado desses mesmos mecanismos, procurando aplicá-los em seus melhores casos.

Mostrou-se adequada também para ambientes de arquitetura ARM, apresentando comportamento estável e desempenho similar ou melhor que outras APIs disponíveis. A grande variedade de sistemas embarcados multi-core que executam o sistema operacional Linux abre espaço para a utilização da LX-MCAPI como um modo relativamente simples de explorar o potencial do processamento desse tipo de dispositivo. Nenhuma mudança no código-fonte é necessária para migrar entre as plataformas, portanto é possível programar a solução desejada uma única vez na plataforma x86-64 e recompilá-la para a plataforma ARM.

LX-MCAPI apresentou bom desempenho e flexibilidade para criação de aplicações paralelas, limitando-se a funções básicas e oferecendo mecanismos de IPC relativamente rápidos sob a abstração da passagem de mensagens. Dentro de suas capacidades e limitações, é uma solução leve e simplificada que permite a utilização de recursos de IPC de maneira transparente ao programador, oferecendo ainda facilidades e padrões de programação prontos para uso, tornando-a uma possível alternativa para desenvolvimento de aplicações paralelas com curva de aprendizado reduzida.

Devido a sua simplicidade, pretende-se futuramente incorporá-la em uma ferramenta de prototipagem de aplicações, principalmente para fins educacionais, procurando oferecer uma interface visual amigável para os novos programadores e recursos de abstração que sejam efetivamente aplicados e facilitem a utilização dos conceitos de programação paralela em soluções cotidianas e otimização de aplicações existentes.

GLOSSÁRIO

API – *Application Programming Interface*

CPU – *Computer Processing Unit*

DSP – *Digital Signal Processor*

FPGA – *Field Programmable Gate Array*

GDI – *Graphics Device Interface*

GPGPU – *General Purpose Graphics Processing Unit*

GPU – *Graphics Processing Unit*

HAL – *Hardware Abstraction Layer*

HTW – *Hardware Transactional Memory*

ILP – *Instruction Level Parallelism*

IPC – *InterProcess Communication*

LX-MCAPI – *Linux-MCAPI*

MCAPI – *MultiCore Application Programming Interface*

MIMD – *Multiple Instruction, Multiple Data*

MISD – *Multiple Instruction, Single Data*

MPI – *Message Passing Interface*

PGAS – *Partitioned Global Address Space*

RTT – *Roundtrip Time*

SDK – *Software Development Kit*

SHM – *Shared Memory*

SIMD – *Single Instruction, Multiple Data*

SISD – *Single Instruction, Single Data*

SPMD – *Single Program, Multiple Data*

STM – *Software Transactional Memory*

TR – *Transmission Rate*

ZC – *Zero-copy*

Apendice A

LX-MCAPI - TABELAS DE RESULTADOS

	LX-MCAPI		MPICH2	
Tamanho (Bytes)	**TR (MiB/s)**	**RTT (μs)**	**TR (MiB/s)**	**RTT (μs)**
2	0,437	6,97	2,392	1,779
4	0,966	7,141	5,009	1,789
8	1,929	7,172	10,710	1,802
16	3,752	7,118	20,901	1,937
32	7,548	7,254	38,369	1,447
64	15,116	7,166	79,701	2,067
128	29,963	7,259	158,612	2,087
256	57,009	7,277	294,253	2,205
512	113,993	7,464	584,141	2,343
1K	219,295	7,578	1153,153	2,736
4K	695,003	9,415	3362,890	4,72
8K	1037,225	14,191	4577,815	6,803
16K	1432,230	21,651	5905,208	11,533
32K	3791,934	25,814	5675,587	20,971
64K	5509,542	42,745	5668,714	36,931
128K	7273,494	75,08	6314,441	64,123
256K	8687,744	137,911	6972,378	122,131
512K	9600,586	254,446	6985,803	222,087
1M	6224,148	538,768	4984,923	407,711

Tabela A.1: TR e RTT para LX-MCAPI (Pacotes) e MPICH2 (Mensagens) - x86-64

	LX-MCAPI		MPICH2	
Tamanho (Bytes)	**TR (MiB/s)**	**RTT (μs)**	**TR (MiB/s)**	**RTT (μs)**
2	0,109	22,995	0,223	17,597
4	0,315	24,065	0,621	14,093
8	0,629	23,027	1,235	14,227
16	1,257	23,084	2,449	14,452
32	2,461	23,071	4,673	13,396
64	4,791	24,234	9,070	15,835
128	9,514	24,15	18,258	15,898
256	18,639	24,407	70,099	16,501
512	36,259	24,999	68,303	17,488
1K	66,695	26,085	128,716	18,814
4K	204,340	33,349	326,271	28,384
8K	235,436	49,472	401,018	39,017
16K	261,077	90,071	473,478	101,414
32K	437,610	231,136	446,017	140,531
64K	498,885	418,847	417,247	323,861
128K	505,741	792,756	395,050	622,731
256K	453,922	1609,628	409,428	1396,469
512K	400,540	2962,45	383,794	2879,854
1M	421,144	5668,577	385,562	5372,254

Tabela A.2: TR e RTT para LX-MCAPI (Pacotes) e MPICH2 (Mensagens) - ARM

	LX-MCAPI		nanomsg	
Tamanho (Bytes)	**TR (MiB/S)**	**RTT (μs)**	**TR (MiB/s)**	**RTT (μs)**
2	0,437	6,97	0,804	63,913
4	0,966	7,141	1,801	70,678
8	1,929	7,172	3,704	65,348
16	3,752	7,118	7,544	62,638
32	7,548	7,254	12,940	71,82
64	15,116	7,166	25,900	70,513
128	29,963	7,259	51,200	68,687
256	57,009	7,277	97,561	68,088
512	113,993	7,464	189,911	73,221
1K	219,295	7,578	368,213	73,424
4K	695,003	9,415	671,145	87,399
8K	1037,225	14,191	1161,492	93,5
16K	1432,230	21,651	1618,972	110,372
32K	3791,934	25,814	2160,195	123,049
64K	5509,542	42,745	2462,186	158,944
128K	7273,494	75,08	2019,413	206,19
256K	8687,744	137,911	1872,966	325,284
512K	9600,586	254,446	2279,305	604,504
1M	6224,148	538,768	1517,731	1182,77

Tabela A.3: TR e RTT para LX-MCAPI (Pacotes) e nanomsg - x86-64

	LX-MCAPI		**nanomsg**	
Tamanho (Bytes)	**TR (MiB/s)**	**RTT (μs)**	**TR (MiB/s)**	**RTT (μs)**
2	0,109	22,995	0,107	287,813
4	0,315	24,065	0,246	272,197
8	0,629	23,027	0,496	280,233
16	1,257	23,084	0,989	329,155
32	2,461	23,071	1,694	392,032
64	4,791	24,234	3,190	432,452
128	9,514	24,15	6,349	420,08
256	18,639	24,407	12,556	433,3
512	36,295	24,999	24,677	427,809
1K	66,695	26,085	44,228	432,726
4K	204,340	33,349	105,363	480,638
8K	235,436	49,472	183,830	514,54
16K	261,077	90,071	277,535	577,056
32K	437,610	231,136	295,940	603,331
64K	498,885	418,847	421,391	1271,101
128K	505,741	792,756	203,065	2160,385
256K	453,922	1609,628	179,164	4298,188
512K	400,540	2962,45	121,790	10253,979
1M	421,144	5668,577	133,652	19346,374

Tabela A.4: TR e RTT para LX-MCAPI (Pacotes) e nanomsg - ARM

REFERÊNCIAS BIBLIOGRÁFICAS

AMDAHL, G. Validity of the single processor approach to achieve large-scale computer capabilities. *AFIPS Conf. Proc.*, v. 30, p. 483–485, 1967.

ARVIND; AUGUST, D.; PINGALI, K.; CHIOU, D.; SENDAG, R.; YI, J. J. Programming multicores: Do applications programmers need to write explicitly parallel programs? *IEEE Micro*, v. 30, p. 19–32, 2010. ISSN 02721732.

ASANOVIC, K.; BODIK, R.; DEMMEL, J.; KEAVENY, T.; KEUTZER, K.; KUBIATOWICZ, J.; MORGAN, N.; PATTERSON, D.; SEN, K.; WAWRZYNEK, J.; WESSEL, D.; YELICK, K. A View of the Parallel Computing Landscape. *Commun. ACM*, v. 52, n. 10, p. 56–67, 2009. ISSN 0001-0782.

BUTENHOF, D. *Programming with POSIX Threads*. [S.l.]: Addison-Wesley, 1997. (Addison-Wesley professional computing series). ISBN 9780201633924.

CHAOS API Overview. 2015. `https://chaos-community.github.io/CHAOS-API-Documentation/current/overview.html#what-is-chaos`. Acesso em: 05/01/2016.

CHRISTOFFERSON, M. *LINX: an open source IPC for distributed, multicore embedded designs*. 2006. `http://www.embedded.com/design/connectivity/4006621/LINX-an-open-source-IPC-for-distributed-multicore-embedded-designs`. Acesso em: 05/01/2016.

CORBET, J. *The unveiling of kdbus*. 2014. `https://lwn.net/Articles/580194/`. Acesso em: 05/01/2016.

D-BUS Tutorial. `http://dbus.freedesktop.org/doc/dbus-tutorial.html`. Acesso em: 05/01/2016.

DAUGMAN, J. How iris recognition works. *Circuits and Systems for Video Technology, IEEE Transactions on*, IEEE, v. 14, n. 1, p. 21–30, 2004.

DAUGMAN, J. G. High confidence visual recognition of persons by a test of statistical independence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, IEEE, v. 15, n. 11, p. 1148–1161, 1993.

ENEA. *LINX for Linux User's Guide*. 2009. `http://linx.sourceforge.net/linxdoc/doc/usersguide/UsersGuide_LINX_for_Linux.html#introduction`. Acesso em: 05/01/2016.

ESMAEILZADEH, H.; BLEM, E.; St. Amant, R.; SANKARALINGAM, K.; BURGER, D. Dark Silicon and the End of Multicore Scaling. *IEEE Micro*, v. 32, n. 3, p. 122–134, 2012. ISSN 0272-1732.

FLYNN, M. J. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, v. 100, n. 9, p. 948–960, 1972. ISSN 0018-9340.

FRANKLIN, M. *Multiscalar Processors*. [S.l.]: Springer US, 2012. (The Springer International Series in Engineering and Computer Science). ISBN 9781461510390.

FRIEDLEY, A.; BRONEVETSKY, G.; HOEFLER, T.; LUMSDAINE, A. Hybrid MPI: efficient message passing for multi-core systems. In: ACM. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver, EUA, 2013. p. 18.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. [S.l.]: MIT Press, 2014. (Scientific and Engineering Computation). ISBN 9780262527392.

HARRIS, T.; LARUS, J.; RAJWAR, R. *Transactional Memory*. [S.l.]: Morgan & Claypool, 2010. (Synthesis lectures in computer architecture). ISBN 9781608452354.

HERLIHY, M.; MOSS, J. E. B. Transactional memory. *ACM SIGARCH Computer Architecture News*, v. 21, n. 2, p. 289–300, 1993. ISSN 01635964.

HILL, M. D.; MARTY, M. R. Amdahl 's law in the multicore era. *Computer*, v. 41, n. July, p. 33–38, 2008. ISSN 0018-9162.

HINTJENS, P. *ZeroMQ: Messaging for Many Applications*. [S.l.]: O'Reilly Media, 2013. ISBN 9781449334444.

HOCKNEY R.W. E JESSHOPE, C. *Parallel Computers: Architecture, Programming and Algorithms*. Bristol, Reino Unido: Taylor & Francis, 1983. ISBN 0852747527.

HOLT, J.; AGARWAL, A.; BREHMER, S.; DOMEIKA, M.; GRIFFIN, P.; SCHIRRMEISTER, F. Software standards for the multicore era. MIT - Institute of Electrical and Electronics Engineers, 2009.

IBM POWER8 Performance and Cost Advantages in Business Intelligence Systems. `http://goo.gl/024RPC`. Acesso em: 01/03/2015.

IDEGUCHI, A.; MORON, C. E.; FERNANDES, M. M. CHAOS-MCAPI: An Optimized Mechanism to Support Multicore Parallel Programming. In: SBC. *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2015 International Symposium on*. Florianópolis, Brasil, 2015.

INTEL Welcome to The Parallel Universe. `https://goo.gl/JsydzQ`. Acesso em: 01/03/2015.

KAMPPI, A.; MATILAINEN, L.; MÄÄTTÄ, J.-M.; SALMINEN, E.; HÄMÄLÄINEN, T. D.; HÄNNIKÄINEN, M. Kactus2: environment for embedded product development using IP-XACT and MCAPI. In: IEEE. *Digital System Design (DSD), 2011 14th Euromicro Conference on*. Oulu, Finlândia, 2011. p. 262–265.

KASIM, H.; MARCH, V.; ZHANG, R.; SEE, S. Survey on Parallel Programming Model. *International Federation For Information Processing*, Oulu, Finlândia, v. 5245, p. 266–275, 2008. ISSN 0302-9743.

KELLY, S. M.; PEDRETTI, K. T. T.; LEVENHAGEN, M. J. *Summary of multi-core hardware and programming model investigations*. [S.l.], 2008.

KHAN, S.; ZOMAYA, A.; WANG, L. *Scalable Computing and Communications: Theory and Practice*. [S.l.]: Wiley, 2013. (Wiley Series on Parallel and Distributed Computing). ISBN 9781118162651.

LARUS, J.; KOZYRAKIS, C. Transactional Memory. *Communications of the ACM*, v. 51, p. 80, 2008. ISSN 00010782.

MASEK, L.; KOVESI, P. Matlab source code for a biometric identification system based on iris patterns. *The School of Computer Science and Software Engineering, The University of Western Australia*, v. 2, n. 4, 2003.

MASEK, L. et al. Recognition of human iris patterns for biometric identification. *The University of Western Australia*, Citeseer, v. 2, 2003.

MASTERS, J.; BLUM, R. *Professional Linux Programming*. [S.l.]: Wiley, 2007. (Wrox Professional Guides). ISBN 9780470149492.

MCKENNEY, P. E. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*, 2011.

MEADE, A.; BUCKLEY, J.; COLLINS, J. Challenges of evolving sequential to parallel code: an exploratory review. In: ACM. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. Szeged, Hungária, 2011. p. 1–5.

MEENAL, D. C.; PRASHANT, H. G. Parallel Programming Models : A Systematic Survey. v. 5, n. 4, p. 5268–5271, 2014.

MISEIKIS, J. *CUDA/C source code for Iris Detection based on Hough Transform*. 2012. `https://github.com/jmiseikis/IrisSegmentation-CUDA`.

MORON, C. E.; IDEGUCHI, A.; FERNANDES, M. M.; MALONY, A. D. From Multitask to Multicore: Design and Implementation Using an RTOS. In: IEEE. *Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on*. Marseille, França, 2014. p. 111–118.

OKUR, S.; DIG, D. How do developers use parallel libraries? In: ACM. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary, EUA, 2012. p. 54.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. *Queue*, ACM, v. 3, n. 7, p. 26–29, 2005.

OLUKOTUN, K.; NAYFEH, B. A.; HAMMOND, L.; WILSON, K.; CHANG, K. The case for a single-chip multiprocessor. *ACM Sigplan Notices*, ACM, v. 31, n. 9, p. 2–11, 1996.

OSHANA, R. *Multicore Software Development Techniques: Applications, Tips, and Tricks*. [S.l.]: Elsevier Science, 2015. (Newnes Pocket Books). ISBN 9780128010372.

PANKRATIUS, V.; ADL-TABATABAI, A.-R.; TICHY, W. *Fundamentals of Multicore Software Development*. [S.l.]: CRC Press, 2011.

PANKRATIUS, V.; SCHAEFER, C.; JANNESARI, A.; TICHY, W. F. Software engineering for multicore systems: an experience report. In: ACM. *Proceedings of the 1st international workshop on Multicore software engineering*. Leipzig, Alemanha, 2008. p. 53–60.

RABENSEIFNER, R.; HAGER, G.; JOST, G. Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes. In: IEEE. *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. Weimar, Alemanha, 2009. p. 427–436.

RAUBER, T.; RÜNGER, G. *Parallel programming: For multicore and cluster systems*. [S.l.]: Springer Science & Business Media, 2013.

SANDERS, P. The multicore transformation: Engineering parallel algorithms. *Ubiquity*, n. July, p. 1–11, 2014.

SHEKHAR, T. D.; VARAGANTI, K.; SURESH, R.; GARG, R.; RAMAMOORTHY, R. Comparison of parallel programming models for multicore architectures. In: IEEE. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. Alaska, EUA, 2011. p. 1675–1682.

SHENDE, S. S.; MALONY, A. D. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, SAGE Publications, v. 20, n. 2, p. 287–311, 2006.

SI, M.; PEÑA, A. J.; BALAJI, P.; TAKAGI, M.; ISHIKAWA, Y. MT-MPI: Multithreaded MPI for many-core environments. In: ACM. *Proceedings of the 28th ACM international conference on Supercomputing*. Muenchen, Alemanha, 2014. p. 125–134.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating system concepts*. [S.l.]: John Wiley & Sons, Inc, 2013.

STONE, J. E.; GOHARA, D.; SHI, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, Institute of Electrical and Electronics Engineers, Inc., EUA, v. 12, n. 1-3, p. 66–73, 2010.

SUSTRIK, M. *nanomsg*. 2015. `http://nanomsg.org/index.html`. Acesso em: 05/01/2016.

SWANSON, S.; TAYLOR, M. B. Greendroid: Exploring the next evolution in smartphone application processors. *Communications Magazine, IEEE*, IEEE, v. 49, n. 4, p. 112–119, 2011.

TAYLOR, M. B. A landscape of the new dark silicon design regime. *Micro, IEEE*, IEEE, v. 33, n. 5, p. 8–19, 2013.

TORVALDS, L. *kdbus: to merge or not to merge? - Linux Kernel Mailing List*. 2015. `http://lkml.iu.edu/hypermail/linux/kernel/1506.2/05492.html`. Acesso em: 05/01/2016.

TREAT, T. *Distributed Messaging with ZeroMQ*. 2014. `http://bravenewgeek.com/distributed-messaging-with-zeromq/`. Acesso em: 05/01/2016.

TREAT, T. *A Look at Nanomsg and Scalability Protocols (Why ZeroMQ Shouldn't Be Your First Choice)*. 2014. `http://bravenewgeek.com/a-look-at-nanomsg-and-scalability-protocols/`. Acesso em: 05/01/2016.

VAJDA, A. *Programming many-core chips*. [S.l.]: Springer Science & Business Media, 2011.

VANDIERENDONCK, H.; MENS, T. Averting the Next Software Crisis. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 44, n. 4, p. 88–90, abr. 2011. ISSN 0018-9162.

VARBANESCU, A. L. *On the effective parallel programming of multi-core processors*. Países Baixos: TU Delft, Delft University of Technology, 2010.

VIEBKE, A.; PLLANA, S. The Potential of the Intel (R) Xeon Phi for Supervised Deep Learning. In: IEEE. *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. Nova Iorque, EUA, 2015. p. 758–765.

VIRTANEN, J.; MATILAINEN, L.; SALMINEN, E.; HAMALAINEN, T. D. Implementation of Multicore Communications API. In: IEEE. *System-on-Chip (SoC), 2014 International Symposium on*. Las Vegas, EUA, 2014. p. 1–6.

WAEL, M. D.; MARR, S.; FRAINE, B. D.; CUTSEM, T. V.; MEUTER, W. D. Partitioned Global Address Space Languages. *ACM Computing Surveys (CSUR)*, ACM, v. 47, n. 4, p. 62, 2015.

WALLS, C. *Embedded Software: The Works*. [S.l.]: Elsevier Science, 2012. ISBN 9780124159693.